Semester 1: Lecture notes

This semester is about core concept of algorithms:
You will learn about simple data structures, algorithms and also techniques to analyze them.
The concepts of this semester will be used multiple times across the next ones.

Summary

1 - Algorithm, Time complexity, Merge sort 1
  1.1 - What is an algorithm? 1
  1.2 - Time complexity 1
  1.3 - Big-O notation 2
  1.4 - Recursive algorithms 3
  1.5 - Sorting 5
2 - Data structures, binary heap, heap sort 10
  2.1 - Data structures 10
  2.2 - Binary heap 10
  2.3 - Heap sort 13
3 - Quick-sort, order statistics 16
  3.1 - Randomized algorithms 16
  3.2 - Quick-sort 16
  3.3 - Order statistics 18
4 - Lower bound for sorting, Radix sort, Sorting networks 20
  4.1 - Can we go faster the $O(n \log n)$? 20
  4.2 - Counting sort 21
  4.3 - Radix sort 22
  4.4 - Sorting networks 23
5 - Binary search 28
  5.1 - Naive version 28
  5.2 - Lower and upper bound 29
  5.3 - Integer problem 30
  5.4 - Double problem 31
  5.5 - Ternary search 33
6 - Stacks, Queues, Amortized costs 35
  6.1 - Stack 35
  6.2 - Queue 36
  6.3 - Dequeue 36
  6.4 - Dynamic array 36
  6.5 - Amortized analysis 38
  6.6 - Queue from 2 stacks 42
7 - Linked List, Pointer Machine 44
  7.1 - Pointer Machine 44

7.2 - Linked List 44
7.3 - Doubly Linked List 46
7.4 - Stack 48
7.5 - Queue 48
7.6 - Why pointer machine? 49
7.7 - Persistent data structures 50
8 - Disjoint sets (Union-Find) 57
8.1 - Definition 57
8.2 - Basic implementation 58
8.3 - Efficient implementation (rank & path compression) 61
9 - Fibonacci Heap 68
9.1 - Binomial trees 68
9.2 - Binomial heap 69
9.3 - Fibonacci heap 72
10 - Dynamic Programming - PART I 77
10.1 - Fibonacci sequence 77
10.2 - Grasshopper problem 79
10.3 - Number of binary vector not containing `11` 80
10.4 - Grasshopper with k jumps 80
10.5 - Grasshopper with cell cost 81
10.6 - Grasshopper with cell cost + k jumps 81
10.7 - Turtle on a $2D$ grid 83
10.8 - Grasshopper with jump restrictions 83
11 - Dynamic Programming - PART II 85
11.1 - Levenshtein distance 85
11.2 - Word alignment and text justification 88
11.3 - Run length encoding (RLE) 89
12 - Knapsack 91
12.1 - Problem statement 91
12.2 - No cost 91
12.3 - Cost, all integers 92
12.4 - Small integers costs 93
12.5 - n is very small 93
12.6 - Multi-knapsack 95
13 - Dynamic programming on subsets, on profiles 98
13.1 - Dynamic programming on subset 98
13.2 - Dynamic programming on profiles (Domino tilling) 100
13.3 - A second problem on profile (Grid coloring) 104
14 - Hash Tables 105
14.1 - Simple implementations 105
14.2 - Random hash function 107
14.3 - Open addressing 108
14.4 - Hash tables poisoning 110
14.5 - Dynamic hash table 110
15 - Perfect & Cuckoo hashing, Bloom filters 111
15.1 - Perfect hashing 111
15.2 - Cuckoo hashing 112

15.3 - Bloom filters 113
15.4 - Cuckoo filter 115

1 - Algorithm, Time complexity, Merge sort

1.1 - What is an algorithm?

Formalized way to solve some problems.

$$\text{Input data} \longrightarrow \text{Algorithm} \longrightarrow \text{Output data}$$

Example: Sum of elements of an array

$$\text{input} : a[0..n-1]$$
$$\text{output} : \sum_i a[i]$$

```
s = 0
for i = 0..n-1:
    s += a[i]
print(s)
```

1.2 - Time complexity

Our main way to measure the efficiency of an algorithm.

$$\text{Unit} = \text{number of operations}$$

To know what is an operation, we need a computational model.

A computational model is a mathematical abstraction that is a simplified version of our processor.

1.2.1 - RAM Model

Ram:

$$0 \xrightarrow{\qquad\qquad i \qquad\qquad} n-1$$

We can access an element i of the memory in constant time (not the case in all computational model).

Example: let's take our previous example back

```
s = 0                          } 1 op

for i = 0..n-1:                }
    s += a[i]                  } n times, 2 op in for declaration & 3 op in for body

print(s)                       } 1 op
```

$$T(n) = \cancel{2} + \cancel{5}n$$
$$= \mathrm{O}(n)$$

1.3 - Big-O notation

1.3.1 - Big-O

$$f(n) = \mathrm{O}(g(n))$$
$$\iff \exists\ n_0, c \mid \forall n \geq n_0,\ f(n) \leq c \cdot g(n)$$

Let's prove that: $2 + 5n = \mathrm{O}(n)$

$$\rightarrow f(n) = 2 + 5n$$
$$g(n) = n$$
$$n_0 = 2$$
$$c = 6$$

O(.) is an upper bound of our real complexity. One could say that that $n = \mathrm{O}(n^2)$ which is true with $n_0, c = 1$.

$$\mathrm{O}(f(n)) = \text{not slower}$$

1.3.2 - Big-Ω

$$f(n) = \Omega(g(n))$$
$$\iff \forall n \geq n_0,\ f(n) \geq c \cdot g(n)$$

Which mean Ω is a lower bound of our complexity.

For our sum example, we can show that $2 + 5n = \Omega(n)$ with $n_0, c = 1$

1.3.3 - Big-Θ

We have shown for our example that:

$$T(n) = \mathrm{O}(n)$$
$$T(n) = \Omega(n)$$

If we have those two properties, we say that $T(n) = \Theta(n)$.

When we invent a new algorithm we try to prove both O & Ω. Proving O is sufficient that we are fast enough. Ω is useful to show that we are not faster than.

1.3.4 - Common time complexity

For loop:

```
for i = 0..n-1
```
} $\mathrm{O}(n)$

```
for i = 0..n-1
    for j = 0..n-1
```
} $\mathrm{O}(n^2)$

```
for i = 0..n-1
    for j = 0..i-1
```
} $\mathrm{O}(n^2)$

For loop are trivial, while loop can be trickier:

```
i = 0
while i * i < n
    i += 1
```
} $O(\sqrt{n})$

```
i = 1
while i < n
  i += 2
```
} $O(\log_2 n) = O(\log n)$

Constant does not affect asymptotic time.

1.4 - Recursive algorithms

To compute the complexity of $f(n)$ we need to compute the number of time we call $f(n)$ & the number of operations in each $f(n)$ call.

1.4.1 - Example:

```
def f(n)
    if n = 0
        return
    f(n - 1)
```
} $O(1)$

The number of operation is constant. The number of call is n. Therefore complexity of f is $O(n)$.

1.4.2 - Example 2:

```
def f(n)
    if n = 0
        return
    f(n / 2)
```
} $O(1)$

Here the number of operation is still constant. But, we call:

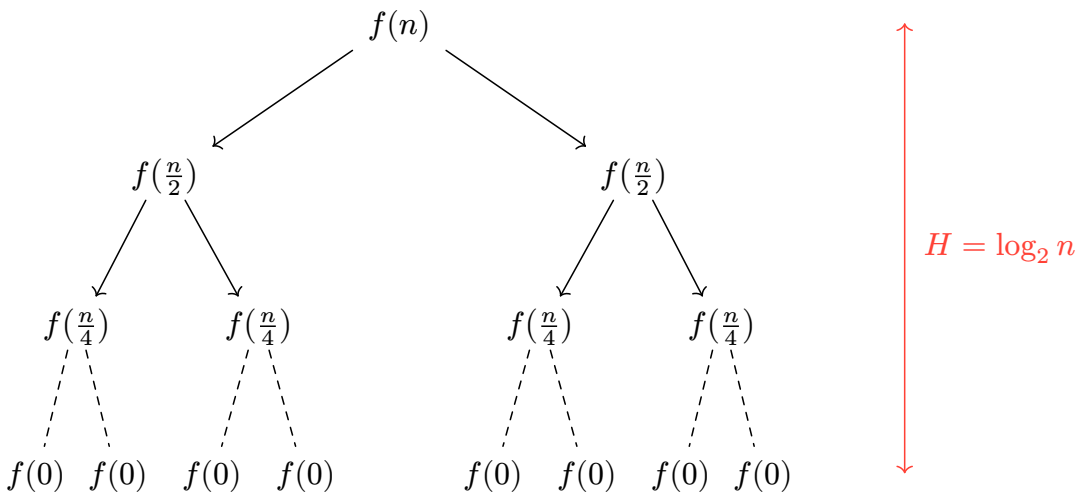
$$f(n) \longrightarrow f(\tfrac{n}{2}) \longrightarrow \ldots \longrightarrow f(0)$$

which make $\log n$ recursive call. Overall the complexity of f is $O(\log n)$.

1.4.3 - Example 3:

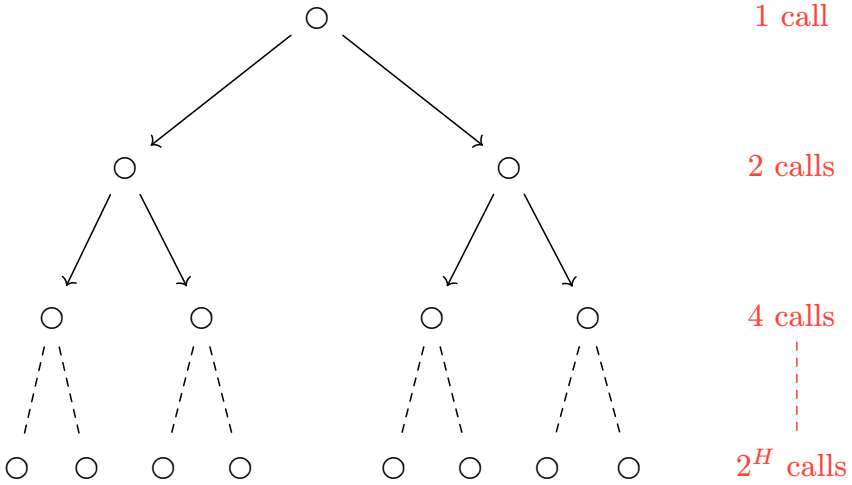
```
def f(n)
    if n = 0
        return
    f(n / 2)
    f(n / 2)
```
} $O(1)$

Let's compute the number of call to f:



The height of the calling tree is $\log_2 n$. The number of call to f is the number of nodes in the tree which is $\mathrm{O}(2^H)$.

Explanation:



Total number of nodes:

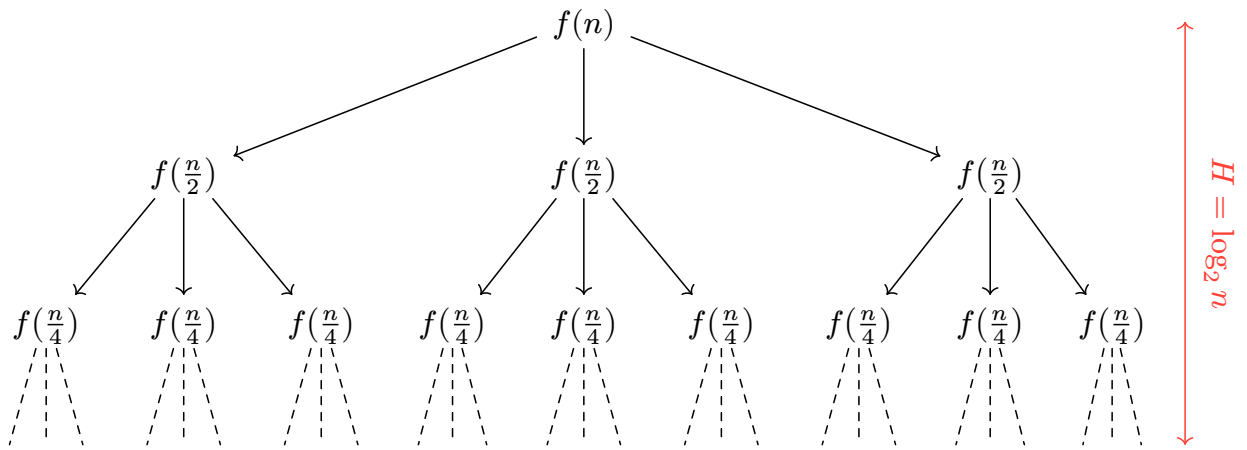
$$\begin{aligned}\text{nb_nodes} &= 1 + 2 + 4 + \ldots + 2^H \\ &= 2^{H+1} \\ &= \mathrm{O}(2^H)\end{aligned}$$

In our case

$$\begin{aligned}f(n) &= \mathrm{O}(2^{\log_2 n}) \\ &= \mathrm{O}(n)\end{aligned}$$

1.4.4 - Example 4:

```
def f(n)
    if n = 0 -- O(1)
        return
    f(n / 2)
    f(n / 2)
    f(n / 2)
```



The height of the tree is the same but the total number of call will now be $\mathrm{O}(3^H)$

In our case:

$$\begin{aligned} f(n) &= \mathrm{O}(3^{\log_2 n}) \\ &= \mathrm{O}(n^{\log_2 3}) \\ &\simeq \mathrm{O}(n^{1.7}) \end{aligned}$$

1.5 - Sorting

$$\text{input: } a[0..n-1]$$
$$\text{output: } b[0..n-1]$$

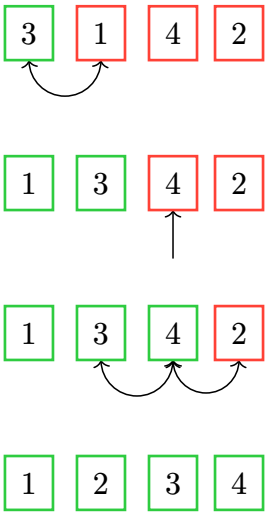
b contain the same number as a but in sorted order.

1.5.1 - Insertion sort

1.5.1.1 - Algorithm

The principle is easy, swap the i^{th} element with the previous element while the i^{th} is lesser.

Let's schematize the insertion sort with the array $a[3, 1, 4, 2]$.



Here is the pseudo-code for the insertion sort:

```
for i = 1..n-1
    j = i
    while j > 0 and a[j] < a[j-1]
        swap(a[j], a[j-1])
        j -= 1
    // sorted a[0..i]
```

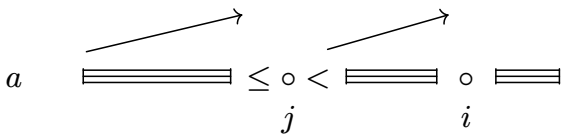
Let's prove that this algorithm works using for invariant.

Let's prove that after each iterations we have sorted $a[0..i]$.

By induction: at the i^{th}, we have $a[0..i-1]$ sorted.

We are swapping the j^{th} element until $a[j] > a[j-1]$ and from the previous while iteration we have $a[j] < a[j+1]$.

So, we finally have:



We have sorted the element in a previously sorted array.

1.5.1.2 - Complexity

Now, let's compute the algorithm complexity. What is tricky here is that the complexity may vary from an input to another.

1.5.1.2.1 - Best case

In an already sorted array, $O(n)$.

Each time, we do not enter in the while-loop so constant operation by iteration.

1.5.1.2.2 - Worst case

Inverted sorted array, $O(n^2)$.

If we do not make the difference for an algorithm, we are talking about the worst case.

1.5.2 - Merge sort

1.5.2.1 - Merge function

$$\text{input} : a \ \& \ b \text{ two sorted array}$$
$$\text{output} : c \text{ the merged sorted array}$$

Example:

$$\text{input} : a[1;5;10] \ \& \ b[2;4;6]$$
$$\text{output} : c[1;2;4;5;6;10]$$

In order to achieve that, we use two pointer technique:

$a \mid 1 \ \ 5 \ \ 10$ — $a[i] < a[j]$
$\uparrow i$

$b \mid 2 \ \ 4 \ \ 6$ — $c \mid \ 1$
$\uparrow j$

$a \mid \cancel{1} \ \ 5 \ \ 10$ — $a[i] > a[j]$
$\uparrow i$

$b \mid 2 \ \ 4 \ \ 6$ — $c \mid \ 1 \ \ 2$
$\uparrow j$

$a \mid \cancel{1} \ \ 5 \ \ 10$ — $a[i] > a[j]$
$\uparrow i$

$b \mid \cancel{2} \ \ 4 \ \ 6$ — $c \mid \ 1 \ \ 2 \ \ 4$
$\uparrow j$

$\vdots$

```
def merge(a, b)                                          } O(n+m)
    n = len(a)
    m = len(b)
    i, j, k = 0
    c = int[n+m]
    while i < n or j < m
        if j = m or (i < n and a[i] < b[j])
            c[k++] = a[i++]
        else
            c[k++] = b[j++]
    return c
```

Here, we construct a new array c containing the sorted element. It is possible to do it without but it is more complicated.

1.5.2.2 - Algorithm

To use this merge function to make the a sorting algorithm we use the divide & conquer method.

The objective here is to make recursive call dividing the array by 2 each time until is size is 1. After that we reconstruct the array with our merge function.

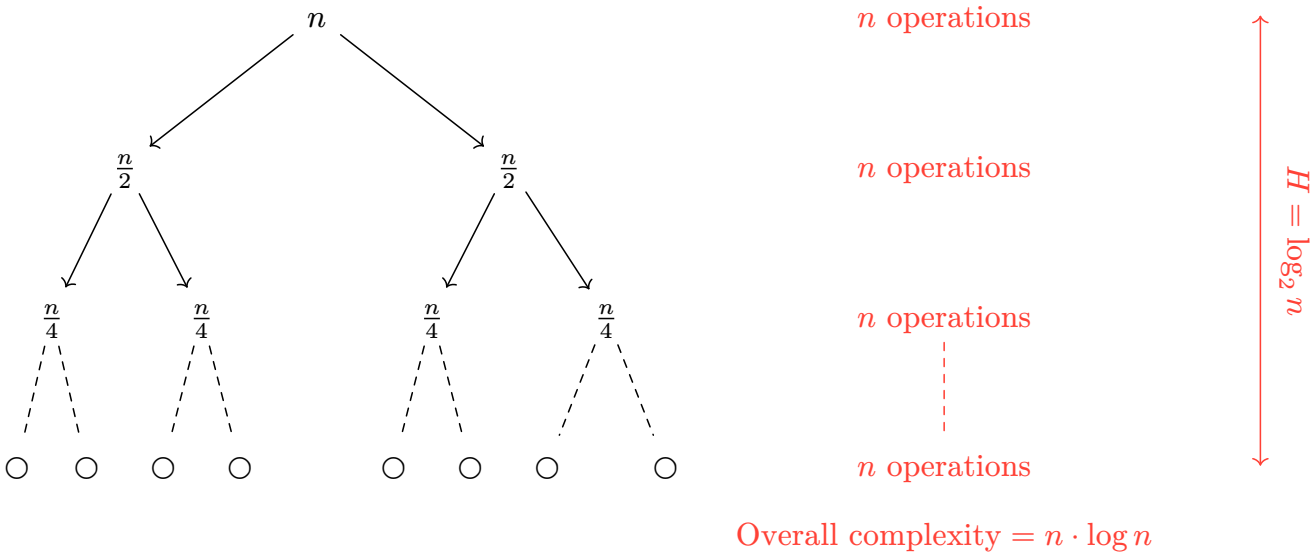
```
def sort(a)
    if len(a) < 2                          } O(n)
        return a
    b = a[0..(n/2 - 1)]
    c = a[n/2..n-1]

    sort(b)                                } T(n/2) * 2
    sort(c)

    return merge(b, c)                     } O(n)
```

1.5.2.3 - Time complexity

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n$$
$$= \mathrm{O}(n \log n)$$



Overall complexity $= n \cdot \log n$

Another proof:

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n$$
$$\leq 2c \cdot \frac{n}{2} \cdot \log \frac{n}{2} + c \cdot n$$
$$= c \cdot n \cdot \log(n-1) + c \cdot n$$
$$= \mathrm{O}(n \log n)$$

1.5.2.4 - Generalization

Let's generalize such complexity using master theorem.

We want to determine the complexity of $g(n)$. This function make a calls to $g\left(\frac{n}{b}\right)$ and 1 call to $f(n)$.

$$\begin{aligned} T(n) &= a \cdot T\left(\frac{n}{b}\right) + f(n) \\ &= a^{\log_b n} + f(n) \\ &= n^{\log_b a} + f(n) \end{aligned}$$

From there, multiple case:

$$\begin{aligned} f(n) &= \mathrm{O}\left(n^{(\log_b a)-\varepsilon}\right) \Rightarrow T(n) = \mathrm{O}\left(n^{\log_b a}\right) \\ f(n) &= \mathrm{O}\left(n^{(\log_b a)+\varepsilon}\right) \Rightarrow T(n) = \mathrm{O}\left(n^{(\log_b a)+\varepsilon}\right) \\ f(n) &= \mathrm{O}\left(n^{\log_b a}\right) \Rightarrow T(n) = \mathrm{O}\left(n^{\log_b a}\right) \end{aligned}$$

Note that we can use Θ in master theorem.

2 - Data structures, binary heap, heap sort

2.1 - Data structures

Structures that contains some data. We want specific structures because we want to make operations on our data.

When thinking about a program, we think about what operations we need to do on our data structure. When we know what operations we need, we chose our data structure.

There are multiple classes of data structure, each of them have dedicated operations.

When we analyze a data structure, we want to analyze the complexity of each operations they allow us.

Example: For an array we have the operations get(i) & put(i, v)

Data structure are closely related to algorithm because we need algorithm to implement data structures operations and algorithm run with specific data structure.

2.2 - Binary heap

- From a the data structure classes of heap (priority queue)
- contain set of elements
- two basic operations
  - insert(x)
  - remove_min()

2.2.1 - Let's try use simpler data structure to implement those operations:

2.2.1.1 - Array

```
def insert(x)
    a[n] = x
    n += 1
```

O(1)

```
def remove_min()
    j = 0
    for i = 0..n-1
        if a[i] < a[j]
            j = i
    swap(a[j], a[n-1])
    n -= 1
    return a[n]
```

O(n)

2.2.1.2 - Sorted array (desc)

```
def insert(x)
    a[n] = x
    n += 1
    i = n-1
    while i > 0 and a[i]
    > a[i-1]
        swap(a[i],
        a[i-1])
        i -= 1
```

$O(n)$

```
def remove_min()
    n -= 1
    return a[n]
```

$O(n)$

2.2.2 - Now let's do a binary heap

2.2.2.1 - Heap properties and construction.

We take a complete binary tree, each layer is complete except for the last one that can be empty on the right.

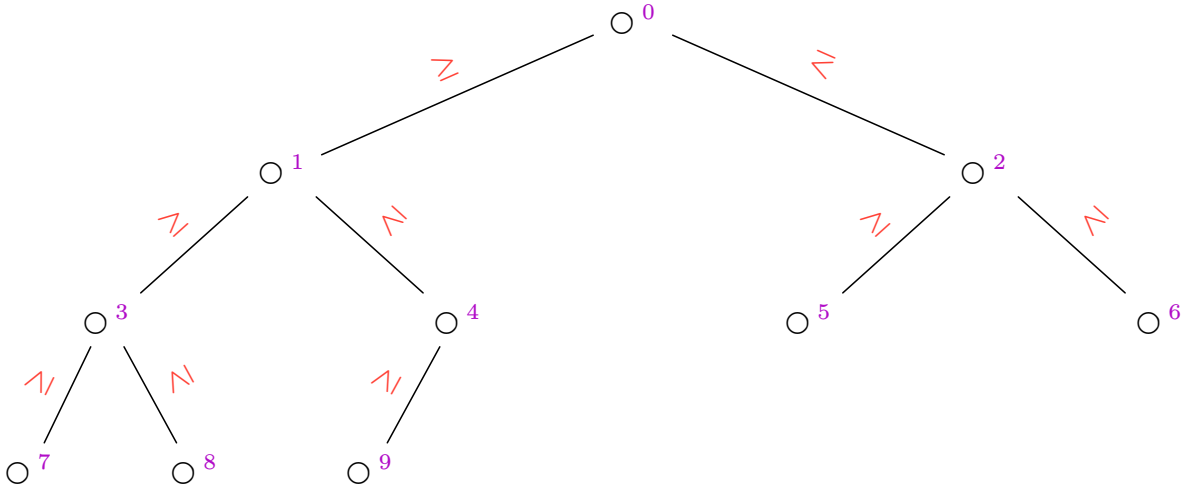
*Heap property is that each sub-layer elt is bigger than the one in the upper layer.

*Node are indexed from left to right, top to bottom.

Since the tree structure for a given number of node will always be the same, we can put the node into an array. We will know the position in the tree given the index in the array.

Indeed, if the index of a node is i, the index of its left child will be $2i + 1$ and is right child $2i + 2$. While its parent index will be $\lfloor \frac{i-1}{2} \rfloor$.

Given those construction rules, let's see the skeleton of the binary heap with ten node:

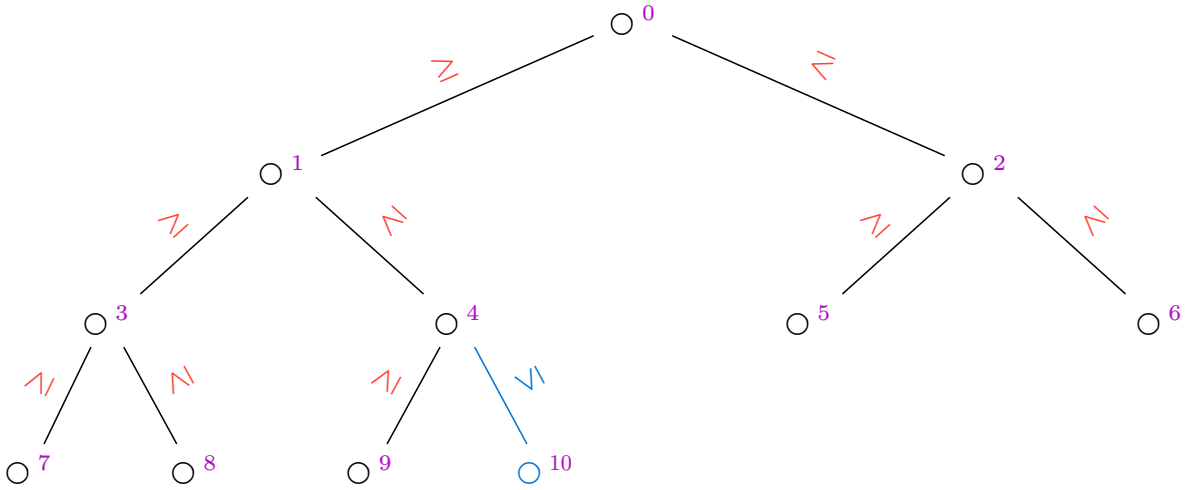


2.2.2.2 - Insertion

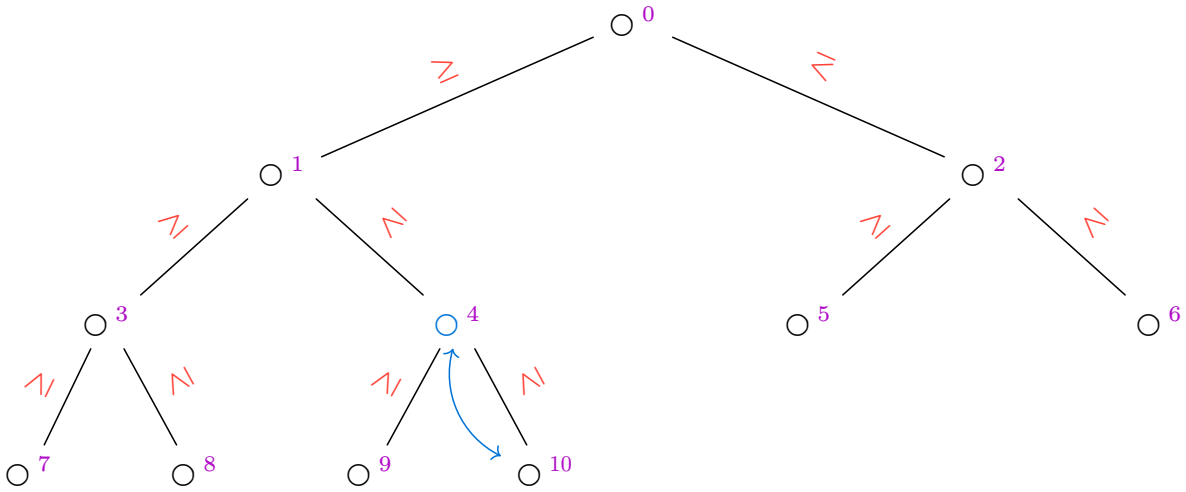
*If we want to insert a new element in the tree, we put it in the next index. Then we ensure heap property, we swap the new element with is parent until the property is satisfied everywhere.

Let's see this on our previous example:

- First, we place the new node in the next place:



- Since the condition $\texttt{node}_4 \leq \texttt{node}_{10}$ is not satisfied, we swap them:



Now the properties are all satisfied, the node is well placed. If the properties were not yet satisfied, we would swap with the node at index 1 and again with the node 0 if it was necessary.

Let's write that function insert.

```
def insert(x)                                          } O(log n)
    h[n] = x
    n += 1
    i = n-1
    while i > 0 and a[i] and a[(i-1) / 2]              }
        swap(a[i], a[(i-1) / 2])                       } O(log n), sift_up operation
        i = (i - 1) / 2                                }
```

The sift_up operation takes a node i and swap it with its parent until it respect the heap property, its complexity is $\mathrm{O}(\log n)$ because we go up at each iteration. At most, we go from

the bottom layer to the first. And since there is at most $\log_2 n$ layer in a binary the complexity is $O(\log n)$.

2.2.2.3 - Remove minimum

To remove the minimum element, we swap the root with the last element, we cut the last element. After the swap, the heap does not satisfy the property anymore.

To satisfy it:

- if the parent if bigger than both of its child, we swap it with the smallest of the two.
- if only one child is lesser, we swap the parent with this child.

Repeat while the parent as at least one smaller child.

```
def remove_min()                                    } O(log n)
    swap(h[0], h[n-1])
    n -= 1
    i = 0

    // at leat 1 child
    while 2i + 1 < n                                }
        j = 2i + 1                                  }
        if (2i+2 < n) and h[2i+2] < h[j]            }
            j = 2i + 2                              } O(log n), sift_down operation
        if h[j] >= h[i]                             }
            break                                   }
        swap(h[i], h[j])                            }
        i = j

    return h[n]
```

Here the complexity is logarithmic for the same reason as in the insert but of going from the bottom layer to the first one, we go from the first one to the bottom one.

Note: here we have implemented the binary heap using an array but can perfectly do it with a tree structure too.

2.3 - Heap sort

2.3.1 - Naive version

Let's create a basic heap sort that we will improve after. The idea of heap sort is pretty simple when we think about the two operations that we have implemented for our binary heap.

- First we need to create a binary heap using the insert function. (we insert each element in an array using insert)
- Secondly we remove the minimum with the associated function n times and store the result in an array.

There is an implementation of:

```
def sort(a)                          } O(n log n)
    for i = 0..n-1
        insert(a[i])                 } O(log n)
    for i = 0..n-1
        a[i] = remove_min()          } O(log n)
```

Here our overall complexity is simple to calculate, we do n times two operations that have a $O(\log n)$ so our complexity is $O(n \log n)$.

But, implemented like this, we need two array, a of size n and h of size n too to make the heap.

We can improve this by doing our heap operation on our sorted array.

2.3.2 - With one array

In order to achieve that, we heapify (transform an array into a heap) a and we remove the minimum element each time.

Let's write the algorithm:

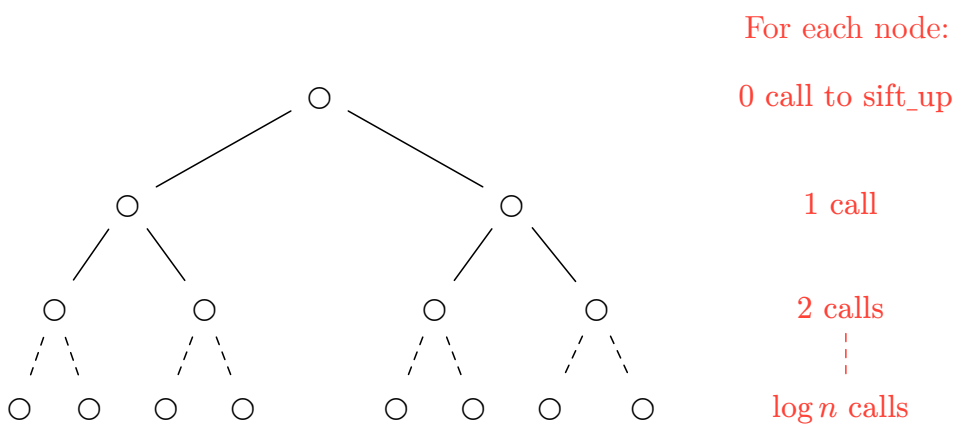
```
def sort(a)
    for i = 0..n-1
        sift_up(i)                   } O(log n)
    for i = 0..n-1
        swap(a[0], a[i])             } O(log n)
        sift_down(0)                 }
```

Our overall complexity is the same, for the same reason. But now we do not need the array h because we are doing all our operations on array a.

2.3.3 - Linear heapify

We still have room for improvement. Indeed we can heapify (the first for-loop in the previous algorithm) in linear time.

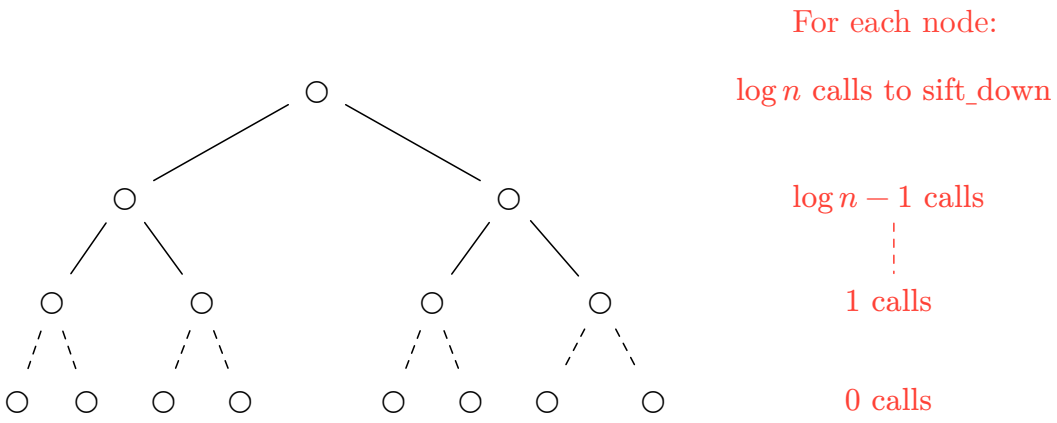
Let's decompose why the complexity of this first for-loop is $O(n \log n)$ in order to improve it.



So assuming that we are at a layer γ the number of call to sift_up is at most $2^\gamma \cdot \gamma$ which makes our total complexity:

$$\sum_{\gamma}^{\log n} 2^\gamma \cdot \gamma = \Omega(n \log n)$$

To improve this complexity, we are going to use sift_down instead of sift_up in the first for-loop. Which gives us this situation.



Which bring our overall complexity to:

$$\sum_{\gamma}^{\log n} 2^\gamma \cdot (\log n - \gamma) \tag{1}$$

Let's show that this is $O(n)$:

Let's do the following substitution:

$$h = \log n - \gamma$$
$$\gamma = \log n - h$$

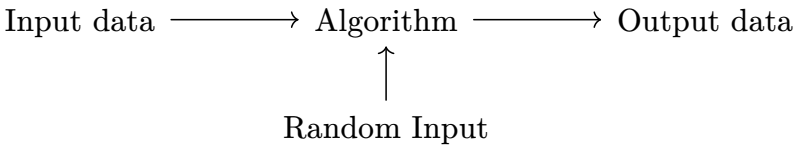
then (1) become:

$$\sum_{h}^{\log n} 2^{\log n - h} \cdot h = n \cdot \sum_{h}^{\log n} \frac{h}{2^h}$$
$$= O(n) \qquad \text{because } \sum_{k}^{\infty} \frac{k}{2^k} = 2$$

Note: if you want a more visual prove counting the number of swaps, see the lecture video of Pavel Martin (link below).

3 - Quick-sort, order statistics

3.1 - Randomized algorithms



We add a bit of random data to our input. And that random data can help us to solve the problem more efficiently.

3.2 - Quick-sort

3.2.1 - Algorithm

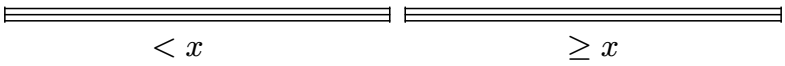
We have an array a that we want to sort:



We chose x as:

$$x = a[\text{random}(n-1)]$$

Then we split the array at x, before x all the number $< x$ after all number $\geq x$.



Then we recurse on each part until our array is completely sorted.

We do not need to make a new array at each recursion, every thing can be done in the array a.

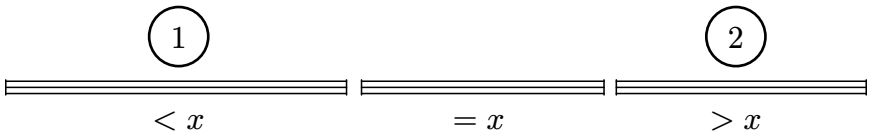
Let's write the algorithm for this sort:

```
def sort(l, r)
    if r - l <= 1
        retrun
    x = a[random(l, r-1)]
    m = l
    for i = l..r-1
        if a[i] < x
            swap(a[i], a[m])
            m += 1

    sort(l, m)
    sort(m, r)
```

This algorithm actually has a flaw. If we have multiple element that are equal to our chosen x we may enter in am infinite recursion.

One of the most simple way to solve this problem id to split the array in three instead of two. One part for the number $< x$, one part for the number $= x$ and one part for the number $> x$.



We then make the recursion on the part 1 and 2

3.2.2 - Time complexity

Since our algorithm is randomized, we will compute our complexity differently. We take the expected value of our time complexity.

$$\mathbb{E}(T(n)) = \sum_X X \cdot \mathbb{P}(T(n) = X)$$

Why it is important ?

Because if we do not do that, for example, in the worst case (when each time the random x that we chose is the current minimum) in our quick sort, the complexity is $\mathrm{O}(n^2)$ and we are going to prove that our expected value is $\mathrm{O}(n \log n)$.

$$\begin{aligned}
T(n) &= \frac{1}{n}\sum_{k=0}^{n-1}(n + T(k) + T(n-k)) \\
&\leq \underbrace{\frac{1}{3}\left(T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right)\right)}_{\substack{\text{good split} \\ x \simeq \text{ middle}}} + \underbrace{\frac{2}{3}T(n)}_{\text{bad split}} + n \\
\frac{1}{3}T(n) &\leq \frac{1}{3}\left(T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right)\right) + n \\
T(n) &\leq 3n + T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right)
\end{aligned}$$

Now let's prove that $T(n) \leq c \cdot n \log n$

$$\begin{aligned}
T(n) &\leq 3n + c \cdot \frac{n}{3}\log\frac{n}{3} + c \cdot \frac{2n}{3}\log\frac{2n}{3} \\
&\leq 3n + c \cdot \frac{n}{3}(\log n - \log 3) + c \cdot \frac{2n}{3}\left(\log n - \log\frac{3}{2}\right) \\
&\leq n\left(\frac{1}{3}c \cdot \log n + \frac{2}{3}\log n + \underbrace{3 - \frac{1}{3}c \cdot \log 3 - \frac{2}{3}c \cdot \log\frac{3}{2}}_{\mathrm{O}(1)}\right) \\
&= \mathrm{O}(n \log n)
\end{aligned}$$

3.2.3 - Improvement ?

We cannot improve asymptotically our algorithm but we can improve it by making our constant factor decrease. **Because constant factor is also important is you seek real-world performance.**

To make it decrease, we need to raise the possibility of choosing a x that is good.

One idea is, instead of picking 1 random element, we pick 3 (or 5…) random element and choose the middle one for x.

We can understand as said in the beginning of the time complexity part that in the case of a random algorithm, our worst case come from the random data that we inject and not from out input.

3.3 - Order statistics

Objective: find the k^{th} element of am array.

3.3.1 - Naive option

Sort the array and pick the k^{th} element. $O(n \log n)$

Can be done with a heap $O(n \log k)$

3.3.2 - Linear solution

Sometimes randomized algorithm are faster than non-randomized ones. But sometimes they are not.

Here we are going to present two technique that work in linear time for this problem. The objective here is not to compare the asymptotic difference but the difference in the complexity of the algorithm.

Because this is another reason to chose randomized algorithm, they can be as fast as the other ones but simpler.

3.3.2.1 - Randomized: using quick-sort

Basically the idea is to do a quick-sort but instead of recursing on all array sub-part, we do it only on the part that contain k until we find it.

```
def find(l, r, k)
    if r - l = 1
        return a[k]
    x = a[random(l, r-1)]
    m = l
    for i = l..r-1
        if a[i] < x
            swap(a[i], a[m])
            m += 1
    if k < m:
        return find(l, m, k)
    else:
        return find(m, r, k)
```

Here we are able to achieve a better complexity than $O(n \log n)$ because we make only one recursive call instead of two like in the basic quick-sort algorithm.

We have $O(n)$:

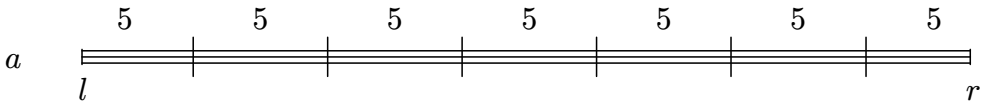
$$\begin{aligned} &\simeq \left(n + \frac{2}{3}n + \frac{4}{9}n + \cdots\right) \\ &\simeq 3n \end{aligned}$$

When we make a good split.

3.3.2.2 - Deterministic: Blum-Floyd-Pratt-Rivestt-Tasjar

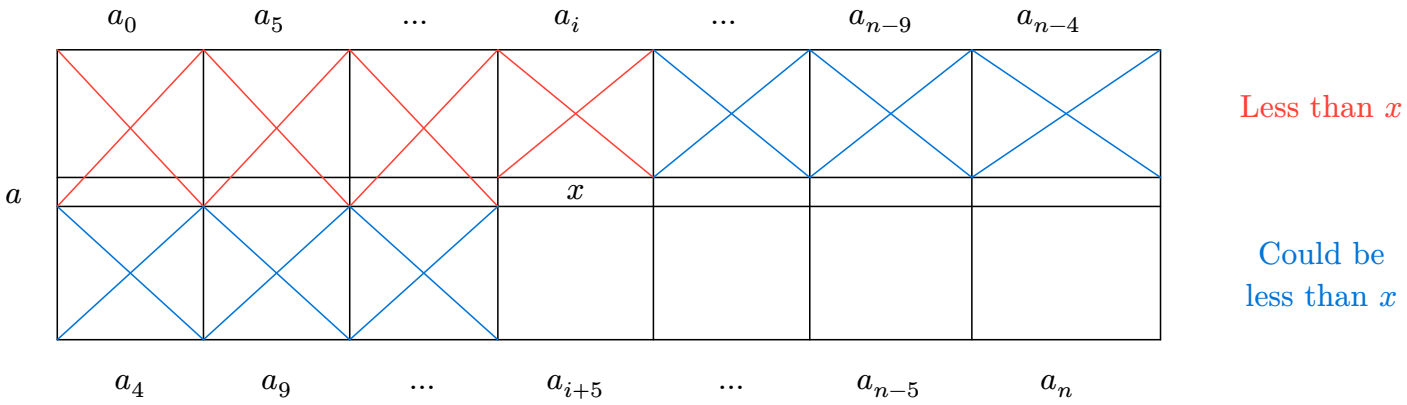
Here we are going to do the same algorithm than previously with the modified quick-sort, but instead of finding x randomly, we are going to find it deterministically.

We separate our array a in block of five elements.



$\frac{n}{5}$ blocks

We take the median of each block. After that, we take the median of those median. This is our x.



To find the median of all the median, we call recursively the algorithm that make the block in the array & take their median.

$$T(n) = \underbrace{n}_{\substack{\text{Make} \\ \text{groups}}} + \underbrace{T\left(\frac{n}{5}\right)}_{\substack{\text{Find} \\ \text{median}}} + \underbrace{T\left(\frac{7n}{10}\right)}_{\substack{\text{Recursion} \\ \text{calls}}} \text{**}$$

Let's show that $T(n) \leq c \cdot n$:

$$T(n) \leq n + c\frac{n}{5} + c\frac{7n}{10} = n\left(1 + \frac{9}{10}c\right) = O(n)$$

The algorithm is indeed working as fast as the randomized one but is far more complex put in motion.

4 - Lower bound for sorting, Radix sort, Sorting networks

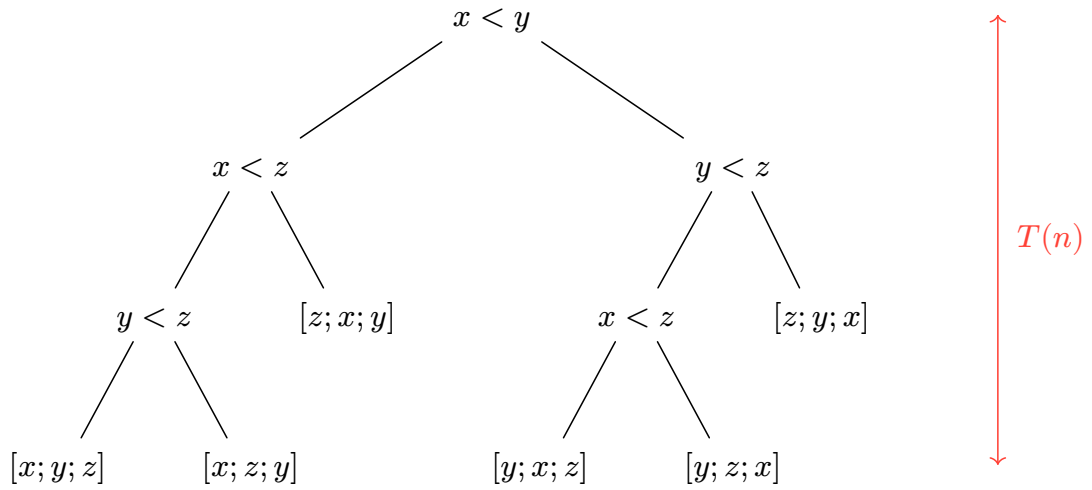
$$\left.\begin{array}{l}\text{Merge sort}\\ \text{Heap sort}\\ \text{Quick sort}\end{array}\right\} \rightarrow \mathrm{O}(n \log n)$$

4.1 - Can we go faster the $\mathrm{O}(n \log n)$?

The only operations needed in those sort are the comparisons between two elements.

Let's prove that if we only have the comparison operations the lower bound for sorting is $\mathrm{O}(n \log n)$.

Let's say we have an array of three elements $[x; y; z]$. Let's compute all the different output that the sort function can gives for this array.



We can build those kind of tree for every sorting algorithm.

If we look at the tree, we see that the height of the tree is the number of comparison needed to find our result.

The height is then therefore the complexity of our algorithm.

The number of possible output is $n!$:

$$\begin{aligned} T(n) &\geq \log_2 n! \\ &\geq \sum_{i}^{n} \log i \\ &= \Omega(n \log n) \end{aligned}$$

We have proved that in our comparison model, the lower bound for sorting is $\mathrm{O}(n \log n)$.

4.2 - Counting sort

4.2.1 - Version 1

We have an array a with:

$$a[i] \in [\![0..m-1]\!] \text{ where } m \text{ is small}$$

Let's take for example $m = 3$ and $a[1;0;1;2;2;0;1;1;2;0]$.

We count the number in a:

$$\begin{matrix} \text{cnt}[3;4;3] \\ 0\ 1\ \ 2 \end{matrix}$$

The complexity of this operation is $O(n+m)$ where $n =$ the number of elements in a and $m =$ number of elements in cnt.

After that, we create a' and put the number of elements of cnt in a'.

$$a'[0;0;0;1;1;1;1;2;2;2]$$

The cost of this operation is also $O(n+m)$.

Overall we just made an algorithm that sort in $O(n+m)$.

Usually this sort is used in situation where we know that $n \simeq m$.

4.2.2 - Version 2

In reality, we generally don't want to sort integers but key of data structures.

```
struct Item {
  int Key // 0..m-1
  Data data
}
```

$$a[(1,A);(0,B);(1,C);(2,D);(2,E);(0,F);(1,G);(1,H);(2,I);(0,J)]$$

$$f(x,y) := \begin{cases} 0\ [B;F;J] \\ 1\ [A;C;G;H] \\ 2\ [D;E;I] \end{cases}$$

$$a'\left[\underbrace{B\ F\ J}_{0}\ \underbrace{A\ C\ G\ H}_{1}\ \underbrace{D\ E\ I}_{2}\right]$$

Instead of creating m vectors, we can count like in the first version, create a big array and report the object in the array after that.

$$a[(1,A);(0,B);(1,C);(2,D);(2,E);(0,F);(1,G);(1,H);(2,I);(0,J)]$$

$$\begin{matrix} \text{cnt}[3;4;3] \\ 0\ 1\ \ 2 \end{matrix}$$

$$a'\left[\underbrace{B\ F\ J}_{0}\ \underbrace{A\ C\ G\ H}_{1}\ \underbrace{D\ E\ I}_{2}\right]$$

Complexity is still $O(n+m)$.

4.3 - Radix sort

Let's do counting sort with bigger int. We still have our array a but this time:

$$a[i] \in [\![0..m^2-1]\!] \text{ where } m \text{ is small}$$

For x an element of a we have:

$$x \in [\![0..m^2-1]\!]$$
$$x = y \cdot m + z \ \text{ such that } \ y, z \in [\![0..m-1]\!]$$

We make two array for x and y and we have:

$$a[i] = b[i] \cdot m + c[i]$$

Compare $a[i]$ and $a[j]$ is then the same as comparing $(b[i], c[i])$ and $(b[j], c[j])$.

$$a[6;2;4;1;7;2;3;4;6]$$
$$= a[(2,0);(0,2);(1,1);(0,1);(2,1);(0,2);(1,0);(1,1);(2,0)]$$

In order to sort this array we do the counting sort 2 times.

We need to begin by sorting the second element:

$$\text{cnt}[3;4;2]$$

$$a'[(2,0);(1,0);(2,0);(1,1);(0,1);(2,1);(1,1);(0,2);(0,2)]$$

Now we do the counting sort on the first element:

$$\text{cnt}[3;3;3]$$

$$a''[(0,1);(0,2);(0,2);(1,0);(1,1);(1,1);(2,0);(2,0);(2,1)]$$

Then we can reconstruct the number of the original a

$$a''[1;2;2;3;4;4;6;6;7]$$

Here, we have sorted our array because we started by sorting the second element of the pair and then the first element, since counting sort is stable the combinations of these two sort have sorted our whole array.

Since the complexity of the counting is $O(n+m)$ and we do it two times our complexity is also $O(n+m)$ here.

Now instead of being limited to m^2 let's do it for integers up to $m^k - 1$.

The reasoning is the same but here we need to do k iterations of our counting sort.

Our total complexity will be: $O(k \cdot (n+m))$.

And this is the radix sort.

Note that in practice algorithm with worse asymptotic can be better than the algorithms with better ones.

This algorithm is usually not the best for sorting an array if n, m are not that big. You'll need to check for your case if it is worth the shot. This algorithm is also a very important sort because in some algorithms that we will discuss about in the later lessons it is at the base of the algorithm.

4.4 - Sorting networks

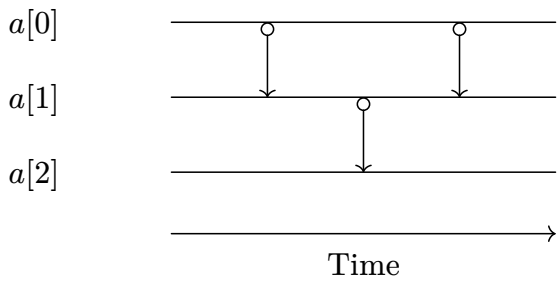
4.4.1 - Concurrent calls

We want to sort an array and the only allowed operations in our model is:

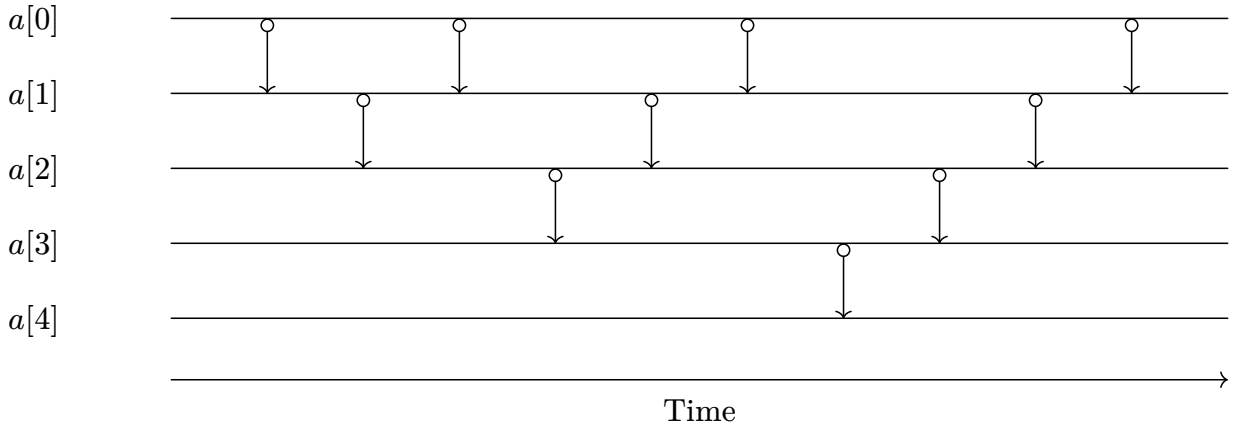
```
def cmp(i, j)
    if a[i] > a[j]
        swap(a[i], a[j])
```

Example for $n = 3$ we do: $\begin{cases} \text{cmp}(0,1) \\ \text{cmp}(1,2) \\ \text{cmp}(0,1) \end{cases}$

Let's schematize those call over the time:



Let's do it now with an array of 5 elements:

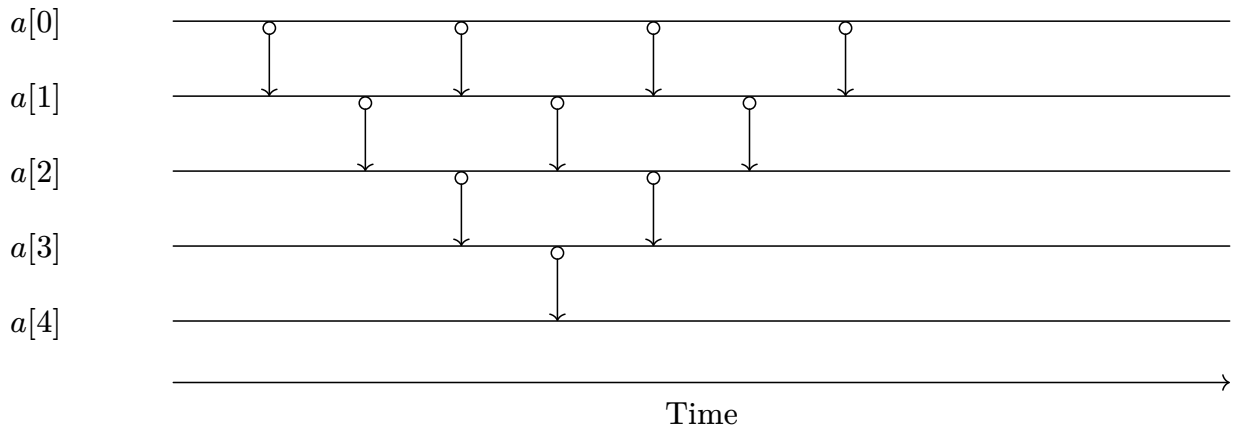


Here we make $\mathrm{O}(n^2)$ concurrent call.

We want to optimize 2 parameters here:
- The number of comparator
- The length of the network

We are allowed to call simultaneously the unrelated comparators.

If we do that the previous example become:



We now make only $O(n)$ concurrent call.

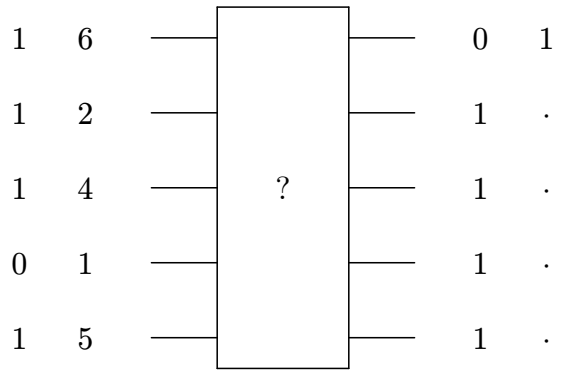
It is even possible to take an algorithm with $O(n \log n)$ comparators and reduce it to $O(\log n)$ (very hard and the constant is to big, not for practical use).

4.4.2 - Theorem

If a network sort any array of 0 and 1 it can sort any array.

Let's prove that by taking an array $a[6; 2; 4; 1; 5]$ and associate 0 and 1 to it. The minimum element is paired to a zero and the other are paired to ones.

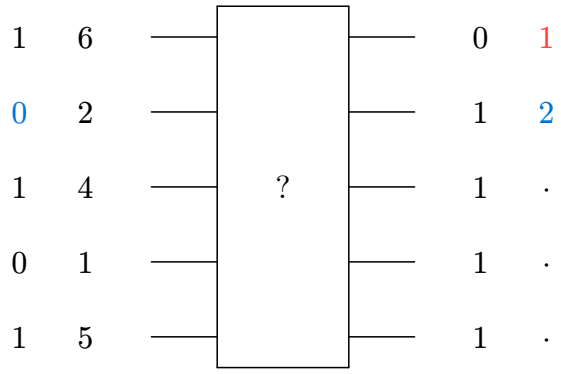
$$a[(6, 1); (2, 1); (4, 1); (1, 0); (5, 1)]$$



Since our network sort array of zeros and ones, if we sort this array with the second element of the pair, we know that the result will be:

$$a[(1, 0); (6, 1); (2, 1); (4, 1); (5, 1)]$$

In the next step, we add a zero. to the new minimum. We now have two elements paired with zeros and the other with ones. We know that our two elements paired with zeros are going to be in the 2 first positions. But since the first position is occupied by our 1 we have no choice but to put our 2 in second place because of the previous iteration.



We iterate 3 more times for $4, 5, 6$ and we have sorted our array.

4.4.3 - Bitonic sort

Bitonic sort is a bitonic sort that is made to sort bitonic sequence.

4.4.3.1 - Bitonic sequence

A bitonic sequence is a sequence made of increasing and decreasing interval that can be splited in 2 parts, 1 increasing and 1 decreasing forming a cycle.

Example:

$$a[3; 2; 4; 6; 10; 25; 17; 11; 8; 6]$$

a is a bitonic sequence because if we move the $[3; 2]$ at the end of the array we have:

$$a[4; 6; 10; 25; 17; 11; 8; 6; 3; 2]$$

which is composed of 1 increasing interval (index 0 to 3) and 1 decreasing (index 3 to 9).

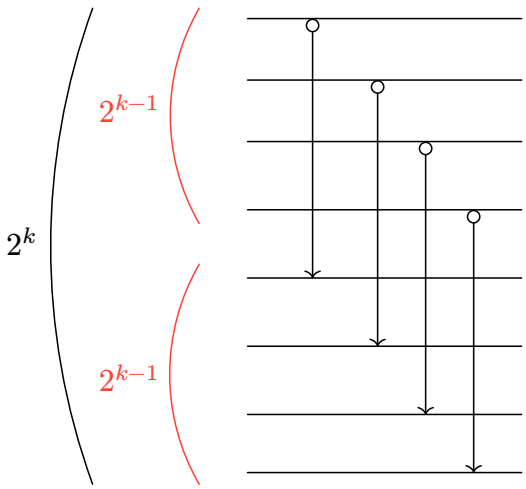
4.4.4 - Sort

Now let's build the sorting network that will sort the bitonic sequence.

To achieve this, we only need to sort a bitonic sequence of zeros and ones ($a[0; 0; 0; 0; 1; 1; 1; 0; 0; 0]$). Where the 1 represent the increasing part and the 0 represent the decreasing part of the sequence.

Note: to achieve this with the RAM-model we just need to find the minimum and merge the 2 parts. Here the model is different and we cannot do that.

4.4.4.1 - Sort a bitonic sequence:



$$[0\ 0\ 0\ 1\ 1\ 1\ 0\ 0] \to \left[\underbrace{0\ 0\ 0\ 0}_{a}\ \underbrace{1\ 1\ 0\ 1}_{b}\right]$$

Let's prove that each part a and b are bitonic sequence and all element of a are less or equal to the element of b.

we have three possibilities:

- either there were only zeros in a before the comparison:
  - in that case no element in the first half can be greater than the one of the second half, so, we do not have any swap then after the comparison no element of a have changed.

$$\left[\underbrace{0\ 0\ 0\ 0}_{a}\ \underbrace{0\ 1\ 1\ 0}_{b}\right] \to \left[\underbrace{0\ 0\ 0\ 0}_{a}\ \underbrace{0\ 1\ 1\ 0}_{b}\right]$$

- all the ones of the sequence were in a:
  - so after comparisons all the ones are in b and only zeros are in a.

$$\left[\underbrace{0\ 1\ 1\ 0}_{a}\ \underbrace{0\ 0\ 0\ 0}_{b}\right] \to \left[\underbrace{0\ 0\ 0\ 0}_{a}\ \underbrace{0\ 1\ 1\ 0}_{b}\right]$$

- there were in both a and b:
  - either there were not enough ones in b and all the ones are in b.

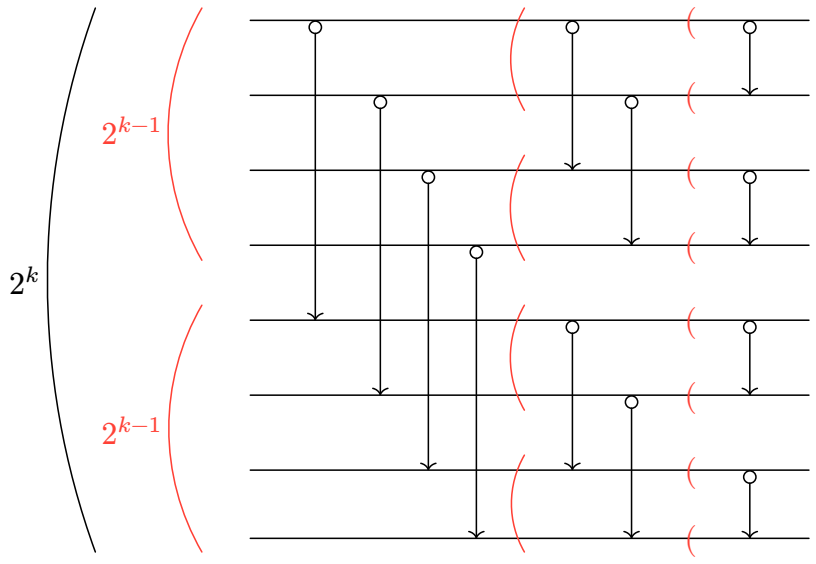
$$\left[\underbrace{0\ 0\ 1\ 1}_{a}\ \underbrace{1\ 0\ 0\ 0}_{b}\right] \to \left[\underbrace{0\ 0\ 0\ 0}_{a}\ \underbrace{1\ 0\ 1\ 1}_{b}\right]$$

  - there were to many ones so there is ones in a and in all b.

$$\left[\underbrace{0\ 1\ 1\ 1}_{a}\ \underbrace{1\ 1\ 0\ 0}_{b}\right] \to \left[\underbrace{0\ 1\ 0\ 0}_{a}\ \underbrace{1\ 1\ 1\ 1}_{b}\right]$$

As we can see in all case a and b are bitonic sequence.

So we can apply the same process as before but recursively:

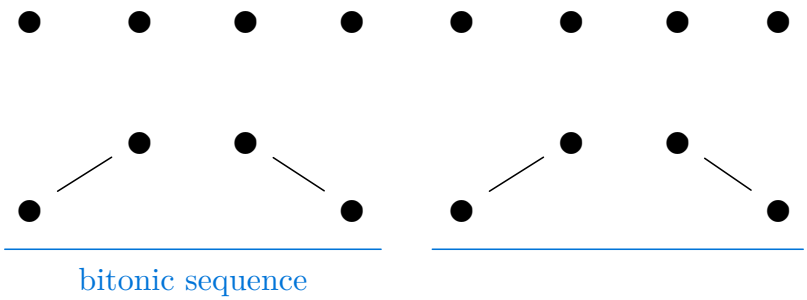


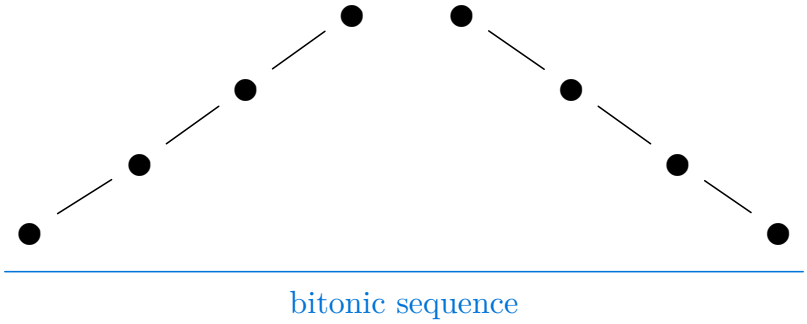
Here , we have:

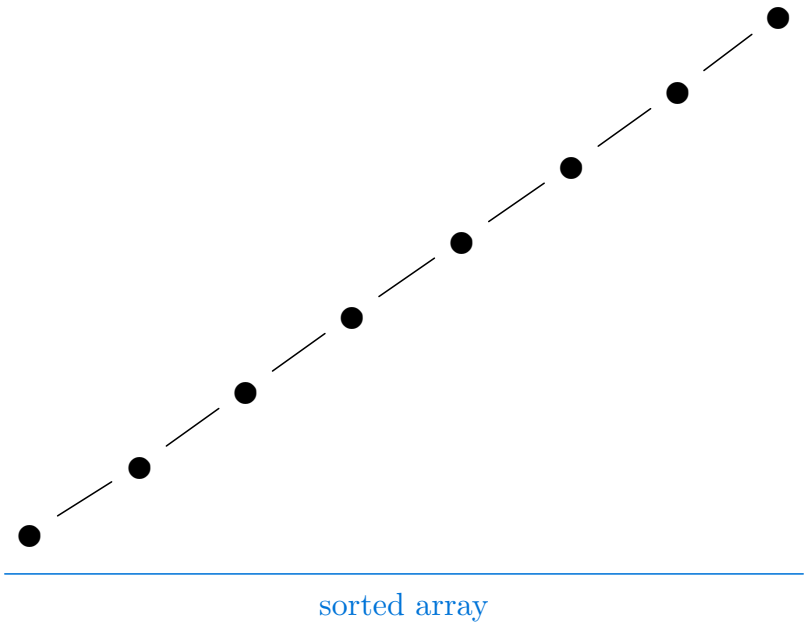
- $O(n \log n)$ comparators
- $O(\log n)$ depth

4.4.4.2 - Sort a non-bitonic sequence

Let's use bitonic sort to sort array of size 2 (because an array of size 2 in necessarily bitonic).







$n \log^2 n$ comparators

$\log^2 n$ depth

5 - Binary search

5.1 - Naive version

This is the basic technique to search an element in a list of elements.

5.1.1 - Common example

We are given a sorted array a and a target x.

We need to find some i in the array such as $x = a[i]$.

To achieve that, we are going to use 2 pointers:
- l = beginning at the leftmost index
- r = beginning at the rightmost index

We are ensuring the property: $x \in a[l..r]$



We pick m at the middle of this segment:

$$m = \left\lfloor \frac{l+r}{2} \right\rfloor$$

Actually, this technique may lead to an overflow in the computing of $l + r$ depending of our language and used type. A better way to pick m is:

$$m = l + \left\lfloor \frac{r-l}{2} \right\rfloor$$

- if a[m] < x
  - since the array is sorted, all the elements to the left of $a[m]$ are also less than x
  - we want to look at the right of m
  - l = m + 1
- else if a[m] > x
  - then it is the opposite case, all the elements to the right of $a[m]$ are also greater than x
  - we want to look at the left of m
  - r = m - 1
- else a[m] = x
  - we return m

We have our algorithm.

```
def bin_search(a, x)
    l = 0, r = n - 1
    while r - l + 1 > 1
        if a[m] = x
            return m
        else if a[m] < x
            l = m + 1
        else
            r = m - 1
```

If we have a more complex problem, this is not the more suitable version of binary search.

5.2 - Lower and upper bound

5.2.1 - Greater or equal

We are again given a sorted array a and a target x.

But this time we need to find $\min(i)$ such as $x \leq a[i]$

We are using the same two pointer as before.

But this time, we want to ensure those properties:
- $a[l] < x$
- $a[r] \geq x$

The objective here is to find the leftmost position that the pointer r can reach.

We need to be careful to where we put l and r:
- Indeed, if all the elements are $\geq x$, l cannot be placed at index 0.
- The same goes for r, if all elements are $< x$, r cannot be placed at index $n-1$.

to satisfy the properties.

To get around, we "add" 2 more elements in the array:
- $-\infty$ for the leftmost position and starting point of l.
- $+\infty$ for the rightmost position and starting point of r.



This is a common work around when we try to find element with properties.

Note that we do not actually put the elements in the array, but imagine l and r to have this value. (In some case we can put the elements, but not in here). It does not matter since we will never access these values.

```
def bin_search(a, x)
    l = -1, r = n
    while r > l + 1
        m = l + (r - l) / 2
        if a[m] >= x
            r = m
        else
            l = m
    return r // if r = n all elt >= x
```

5.2.2 - Less or equal

With the same idea, we can find $\max(i)$ such as $a[i] \leq x$

This time the properties we need to satisfy are:
- $a[l] \leq x$
- $a[r] > x$

```
def bin_search(a, x)
    l = -1, r = n
    while r > l + 1
        m = l + (r - l) / 2
        if a[m] > x
            r = m
        else
            l = m
    return m
```

5.3 - Integer problem

Common problems for binary search are:
- you take $\mathbb{Z}$
- In this set, some number are good the other are not

If x is good, $\forall y \in \mathbb{Z}, y > x \Rightarrow y$ is good. Find the minimum of the set of good numbers.

These problems can be tricky.

5.3.1 - Example

We have n rectangle of size $h \times w$.

x is a good number id we can put all the rectangles in a square of size $x \times x$. We want to find $\min(x)$.

Let's define a function good such that the function tells us if a number is good or not.

```
def good(x) -> bool
    return (x / w) * (x / h) >= n
```

This function is the trickiest part of those problems.

How we can use binary search here?
- l will be the pointer on the bad number.
- r will be the pointer on the good number.

We know then that our property to satisfy for l and r is:
- $\text{good}(l) = 0$
- $\text{good}(r) = 1$

We now need to find to what values initialize our two pointers.

Here we can chose $l = 0$ and $r = \max(h, w) * n$.

But if h and w are big, we may face integers overflows. There is no short answer to solve this problem.
- the most common work around is to check what could be the biggest possible number is for r.
- another one could be to use floating point number instead of integers to avoid overflow. But other problems would arise, for example accuracy of such number.
- The best idea is generally to find a good number small enough. To achieve that we can use another function.

In our case:

```
def find_small_good()
    r = 1
    while !good(r)
        r = r * 2
    return r
```

This is the safer way.

Now we can work our binary search algorithm.

```
def bin_search()
    r = find_small_good()
    l = r / 2 // work because of find_small_good definition
    while r > l + 1
        m = l + (r - l) / 2
        if good(m)
            r = m
        else
            l = m
    return r
```

5.4 - Double problem

Let's consider the following problem, we have a line and n people are living on this line at the position x_i.

For each person we know:
- his coordinate x_i
- his maximum moving speed v_i on the line

Each person can move left or right. Our task is to find the minimum time (t) we need to gather them at 1 point.

The number t is considered good when all the people can gather in t seconds.

It is our good property as in Section 5.3. We now need to define our good function.

How to check if we can get to the same point in t seconds.

At t seconds we need to check the interval where a person can be. This interval is:

$$[l_i, r_i] = [x_i - t \cdot v_i, x_i + t \cdot v_i]$$

Our goal is to find a point which is in the interval for every person:

$$\forall i \begin{vmatrix} x \geq l_i \\ x \leq r_i \end{vmatrix} \Rightarrow \begin{vmatrix} x \geq \max(l_i) \\ x \leq \min(r_i) \end{vmatrix}$$

```
def good(t) -> bool
    return max(x_i - t * v_i) <= min(x_i + t * v_i)
```

The next thing we need to consider is that in this example, t can not be an integer.

The basic idea stay the same:
- 1 pointer for the good values
- 1 pointer for the bad values

The first way to implement our binary search considering that our target is not an integer is to do the same as before but our loop will be:

`while r - l > EPS` , with EPS a small value.

```
def bin_search()
    EPS = 10**(-9)
    l = 0, r = 10**10
    while r - l > EPS
        m = l + (r - l) / 2
        if good(m)
            r = m
        else
            l = m
    return r
```

This way is not a good idea. Let's say that our answer $t \simeq 10^9$ we then have in our loop:

$$l = 3.1415...356$$
$$r = 3.1415...357$$

With approximately 16 digits for double. in this case $r - l \simeq 10^{-7}$.

We will never be able to reach our ε with double numbers, because in this example l and r are the closest they can be.

It is the same as having two following integer but here we have two following double.

How to fix this?

- first option: check if $m = l$ or r
  - if it is the case then we are in the previous situation so we can break
- second option: The safest way
  - instead of a while-loop we can use a for-loop and do a given amount of iterations.
    - i.e. `for i=0..100` instead of `while r - l > EPS`

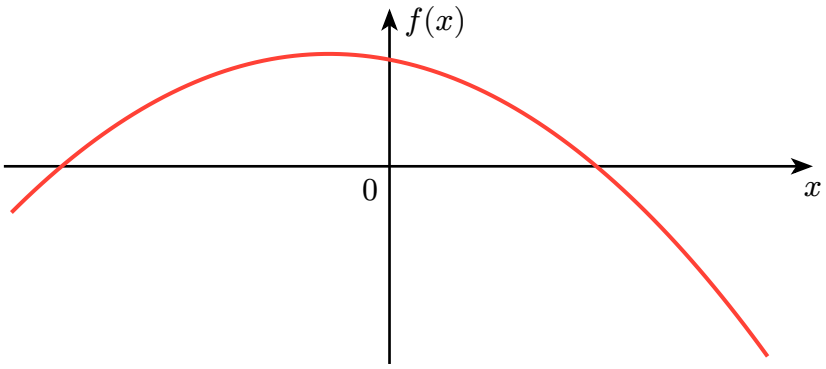
    since each time the difference between l and r is divided by 2, with this for-loop the distance will be divided by $2^{100} \simeq 10^{30}$. This is more than enough, 100 even may be to big and time out.

```
def bin_search()
    l = 0, r = 10**10
    for i=0..100
        m = l + (r - l) / 2
        if good(m)
            r = m
        else
            l = m
    return r
```

5.5 - Ternary search

5.5.1 - Example

Imagine you are given a function that you want to maximize. This function is of x.



On the interval where we want to maximize f, the function can be separated in 2 intervals, 1 where the function increase, the other one where the function decrease.

If at each point we can check if the function is increasing or not, then this check become our good function and we use the previous binary search example.

But if we can only compute the value of the given point:

- we set two pointers l and r
- we maintain following properties,
  - in l the function is increasing
  - in r the function is decreasing
- we pick 2 middle points m_1 and m_2
  - $m_1 = l + \frac{1}{3}(r - l)$
  - $m_1 = l + \frac{2}{3}(r - l)$

- now we compute $f(m_1)$ and $f(m_2)$
  - if $f(m_1) \geq f(m_2)$
    - it means that the function is decreasing in m_2
    - $r = m_2$
  - else
    - it means that the function is increasing in m_1
    - $l = m_1$

At each time, we decrease the length of the segment by $\frac{2}{3}$.

The tricky part here is to chose m_1 and m_2. Because in a case where the difference between m_1 and m_2 is little, we can face accuracy problems.

5.5.2 - Make the algorithm faster

The most of the computation time here is the evaluation of f in m_1 and m_2.

To save time, the idea is to chose carefully our m in order that only 1 of them change at each iterations i.e.:

- if $r = m_2$, m_1 become m_2 in the next iteration. Therefore the value of $f(m_2)$ is already known from the previous iteration s owe can save lots of time.
- if $l = m_1$, m_2 become m_1 in the next iteration. In this case we already know the value of $f(m_1)$.

In order to do that, we want to find α such as:

$$d_{l,m_1} = d_{m_2,r} = \alpha \cdot (r - l)$$
$$d_{m_1,m_2} = \alpha \cdot (r - m_1)$$

Let's find α:

$$d_{m_1,r} = (1 - \alpha) \cdot (r - l)$$
$$d_{m_2,r} = (1 - \alpha) \cdot (r - m_1)$$
$$(1 - \alpha) \cdot d_{m_1,r}$$
$$(1 - \alpha)^2 \cdot (r - l)$$

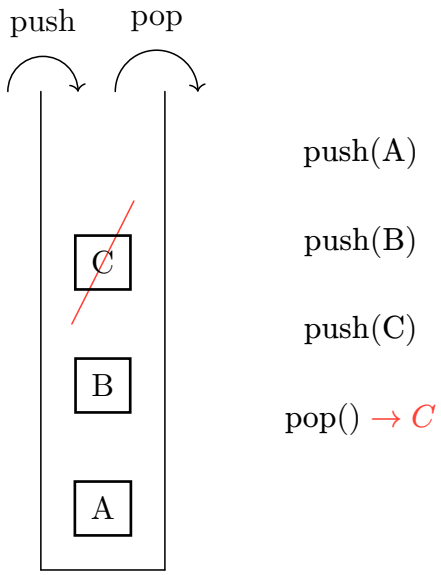
$$\alpha \cdot (r - l) = (1 - \alpha)^2 \cdot (r - l)$$
$$\alpha = 1 - 2\alpha + \alpha^2$$
$$\alpha^2 - 3\alpha + 1 = 0$$

This way to chose α will increase the speed by $\simeq 2$ because we will only have to compute one $f(m_i)$ each time and the computation of f are costing way more than the other operation.

6 - Stacks, Queues, Amortized costs

6.1 - Stack

6.1.1 - Definition



6.1.2 - Usages

Stack is a very simple data structure but it is widely used in computer science. Even if you never implemented it you used stack before.

Any recursion implementation is a stack.

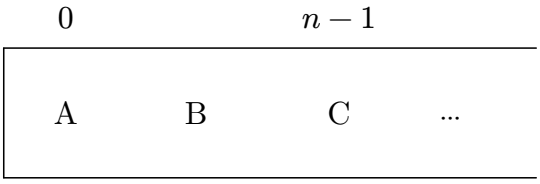
The computer create a stack frame in order to remember the values of var for each call:

- When we call a new f, it pushes a new block on the stack frame.
- When a f return, it comes back to the previous function call by popping the block.
- In each block we save a pointer which is used as a return address for the next function call.

Note: This is not only for recursive calls but for any call to a function from a function.

6.1.3 - Implementation

With an infinite sized array:



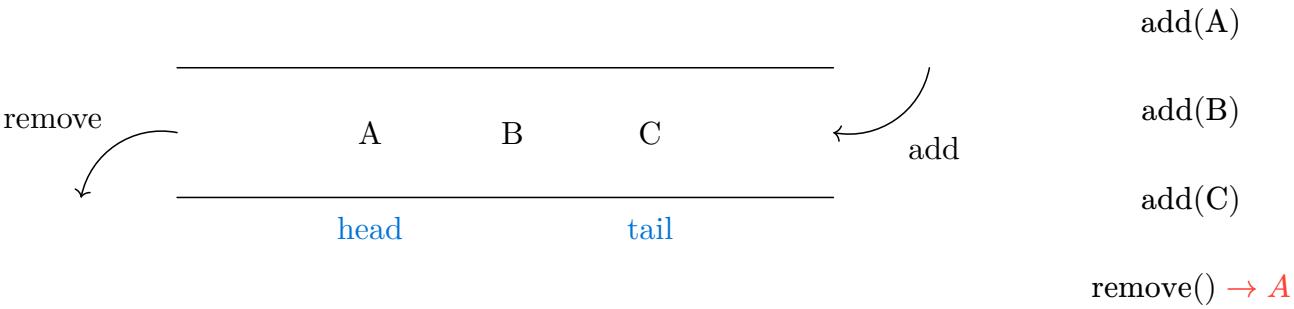

```
def push(x)
    stack[n++] = x
```

```
def pop()
    return stack[n--]
```

In C++, vectors are stacks.

6.2 - Queue

6.2.1 - Definition



6.2.2 - Usages

Queue is used when queue is needed.

Example:

- You have workers and tasks
- The workers are going to take the tasks from the queue and another part of the program will add tasks in the queue.

6.2.3 - Implementation

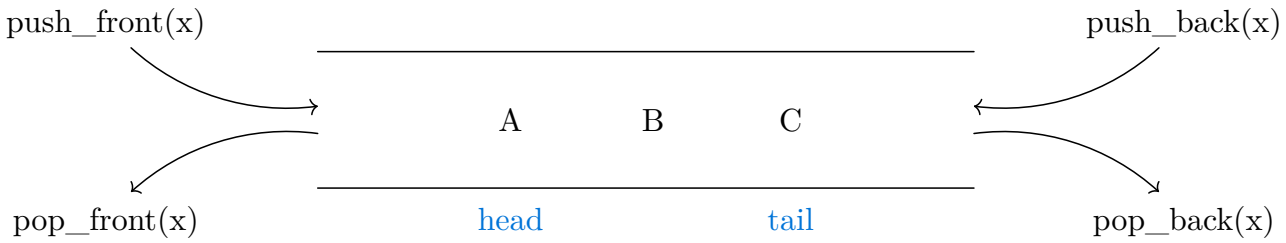
We take an array of infinite size. We also need two pointer, 1 at the head and 1 at the tail.

```
def add(x)
    queue[tail++] = x
```

```
def remove()
    return queue[head++]
```

6.3 - Dequeue

This data structure is just like a queue but we can add and remove from both end.



6.4 - Dynamic array

Ths problem with finite array:

- As we can see, there is a problem, in all the implementation using “infinite array”, which is not really possible.

Workaround, for stack and queue, we can work with finite size array.

6.4.1 - Stack:

If we know the max size of our stack we can just construct an array of size max and fill it bit by bit.

6.4.2 - Queue:

The idea is the same, if we know the maximum size of the queue, we create an of max size and fill it bit by bit.

But filling it is trickier since that contrary to the stack, the head can move from the 0^{th} index.

To got around this problem, we are moving head and tail with a modulo size max.

We see there that we can use fixed sized array to make stack and queue of known length.

But we can not work around if we do not know the max length, it is always better to allocate memory only when we use it and not have a big array almost empty much of the time.

6.4.3 - Dynamic structure

Lets do it on the stack to begin with.

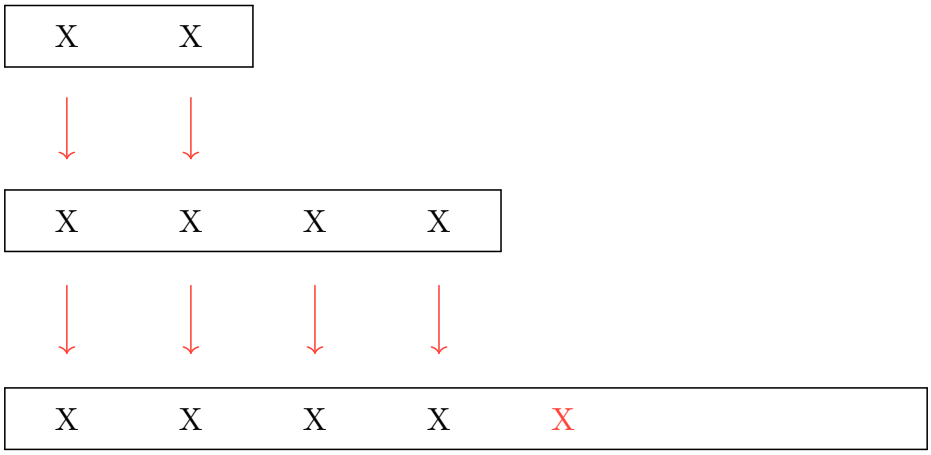
6.4.3.1 - Push function

We have a stack a which is full and we want to push something in it.

The simplest way to make it grow is to make it grow by copying this stack is a stack with more space allocated.

The tricky question is, how to make it grow?

- $a.\text{size} + 1$ is not a good option because it would make the push operation $\text{O}(n)$ each time.
- what we are going to do is to make it grow by 2.




```
def push(x)
    if n == stack.size
        stack' = new Array(2*n)
        stack'[0..n-1] = stack[0..n-1]
        stack = stack'
    stack[n++] = x
```

Our worst complexity is big ($\text{O}(n)$).

6.4.3.2 - Pop function

For the pop function, we could use the same pop function as before but we could encounter the same problem as before which is to make multiple pop and be left with an array mostly empty.

So we want to use a dynamic pop that shrink our stack.

```
def pop()
    ans = stack[--n]
    if n < stack.size / 4
        stack' = new Array(stack.size / 2)
        stack'[0..n-1] = stack[0..n-1]
        stack = stack'
    return ans
```

The question there is why we check for $\frac{\text{stack.size}}{4}$ and not $\frac{\text{stack.size}}{2}$.

Because if we make our stack size growth by 2 when it is full and shrink it by 2 when it is half empty, we can end up in a state where we chain $\text{push}(x)$ and $\text{pop}()$ calls, every call would then make us create a new array.

To prevent this state, we shrink the array by 2 when the three quarter of the stack is empty. This will allow us to make a $\text{push}(x)$ without remake the stack grow right away.

Again our worst complexity is big ($\text{O}(n)$).

For the push and the pop function we now have a worst complexity that is linear. But this complexity is not linear each time. We want to analyze what is average number of computation.

This is the amortized analysis.

6.5 - Amortized analysis

$$T(\texttt{op}) \text{ — real time}$$

$$\tilde{T}(\texttt{op}) \text{ — amortized time}$$

If we make m operations:

$$\sum_i T(o_i) \leq \sum_i \tilde{T}(o_i)$$

We want to prove $\tilde{T}(\text{push}) = \text{O}(1)$.

In order to do that, we have to show that:

$$\sum_{i=0}^{m-1} T(\text{push}) = c \cdot m$$

Then we will have:

$$\tilde{T}(\text{push}) = c = \text{O}(1)$$

There is multiple ways to analyze amortized costs:
- The naive idea is to use the definition
- The second method is the potential function method
- The last one is the accounting method

Lets see all these technique

6.5.1 - Definition method

Lets calculate $\sum T(o_i)$ for m push:

```
def push(x)
    if n == stack.size
        stack' = new Array(2*n)
        stack'[0..n-1] = stack[0..n-1]
        stack = stack'
    stack[n++] = x
```

Our affectation cost us 1 operation, repeated m times this cost us m operations.

Our if statement cost $\mathrm{O}(n)$ and is called for $n = 1, 2, ..., 2^k$. Its cost is:

$$1 + 2 + 4 + ... + 2^k = 2^{k+1} - 1$$

For m operations, we know that $m \geq 2^k$ because otherwise we would not have expanded the list.

$$m \geq 2^k \Rightarrow 2^{k+1} \leq 2 \cdot m$$

So the total time spend on copying element to the new array is no more than $2 \cdot m$.

We then have:

$$\begin{aligned} \sum T(o_i) &\leq m + 2 \cdot m \\ &\leq 3 \cdot m \end{aligned}$$

Per the definition of amortized cost, we proved that:

$$\tilde{T}(\text{push}) = \mathrm{O}(1)$$

6.5.2 - Potential function method

The first technique using the definition work here because the data structure is simple. But for much more complicated data structure it is unsuitable.

That why we have other method.

In this method, we introduce a special potential function to our data structure. This function will be a value assigned to the current state of the data structure.

6.5.2.1 - ϕ function:

Each time we make an operation, ϕ changes value.

$$\begin{array}{ccccccccccc} & & o_1 & & o_2 & & o_3 & & o_4 & \cdots & o_m \\ \phi_0 & & \phi_1 & & \phi_2 & & \phi_3 & & \cdots & \phi_{m-1} & \phi_m \end{array}$$

2 properties:

- $\phi_0 = 0$
- $\phi_i \geq 0$

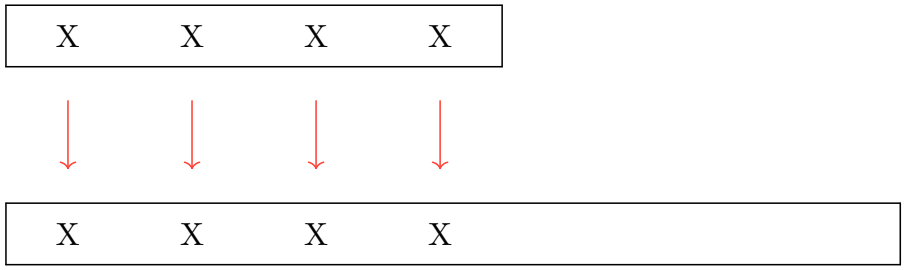
when ϕ is fixed:

$$\tilde{T} = T + \underbrace{\Delta\phi}_{\phi_{i+1}-\phi_i}$$

$$\begin{aligned}\Rightarrow \sum \tilde{T}(o_i) &= \sum (T(o_i) + \Delta\phi) \\ &= \sum T(o_i) + \underbrace{(\phi_m - \phi_0)}_{\geq 0} \\ &\geq \sum T(o_i)\end{aligned}$$

Now what we need to do is to find a potential function. To achieve that we need to have intuition about what to our data structure between two operations.

In slow operation:



$T = n$
$\Rightarrow \Delta\phi \simeq -n$

Because we want $\tilde{T}$ to be small

We observe that between our two state, the number of element in the second half of our array goes from being full (containing $\frac{n}{2}$ elements) to being empty (containing 0 element).

We want $\phi_{i+1} - \phi_i \simeq -n$:
- ϕ_{i+1} must be small
- ϕ_i must be big

When we look at our observation, we see that when ϕ is big, the right part of the array is full and when ϕ is small the right part of the array is empty.

Lets define:

$$\phi = 2 \cdot (\text{nb of elements in the second halt of the array})$$

With this definition, $T = n \Rightarrow \Delta\phi = -n$.

Then if we take our relation between amortized costs, real costs and the potential function we have:

$$\begin{aligned}\tilde{T} &= T + \Delta\phi \\ &= n - n \\ &= \mathrm{O}(1)\end{aligned}$$

Sometimes we can have a null or negative result, if this appends, it means that our amortized cost is constant.

This is a very powerful technique but it require some insight on how to define the potential function.

6.5.3 - Accounting method

This technique is basically the same as the previous one but it require less insight since it is a more visual one.

The idea is that we have an account where we can:
1. put_coin(t) // $\tilde{T} = t$
2. get_coin(t) // $\tilde{T} = -t$

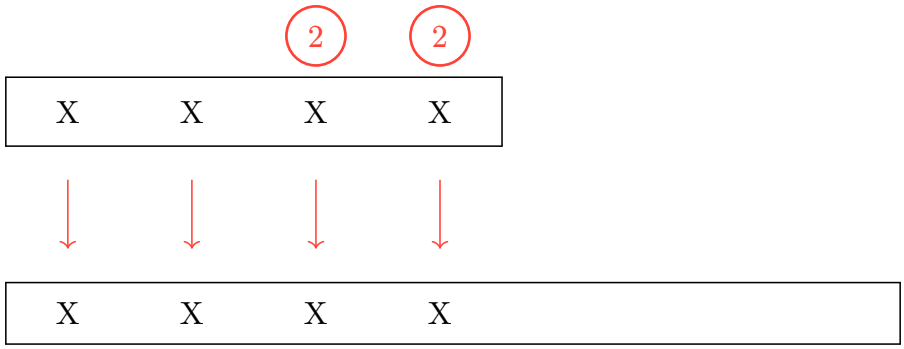
Our account store our time:
1. Reserve some time
2. Use some time

```
def push(x)
    if n == stack.size
        stack' = new Array(2*n)

        get_coin(n)

        stack'[0..n-1] = stack[0..n-1]
        stack = stack'

    put_coin(2)
    stack[n++] = x
```



When our array is full, we have n coins in our account. Coins that we use to copy our n elements to the new array.

We use $-n$ operations with get_coin(n) and n operations after to copy our array which cost 0 overall.

Then we use 2 operations to put_coin(2) and 1 to affect x.

We then have $\tilde{T}(\text{push}) = 3$.

We have proven multiple times that $\tilde{T}(\text{push}) = \text{O}(1)$, lets now prove that $\tilde{T}(\text{pop}) = \text{O}(1)$ with the accounting method again.

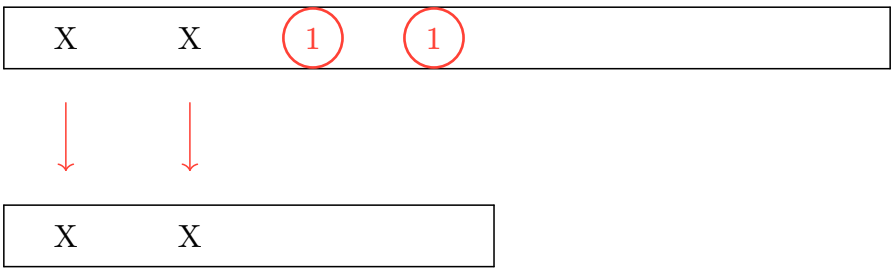
We need to save coins to create our new array while popping. We are going to save them when we do our fast pop.

```
def pop()
    ans = stack[--n]
    if n < stack.size / 4
        stack' = new Array(stack.size / 2)

        get_coins(n)

        stack'[0..n-1] = stack[0..n-1]
        stack = stack'

    put_coin(1)
    return ans
```



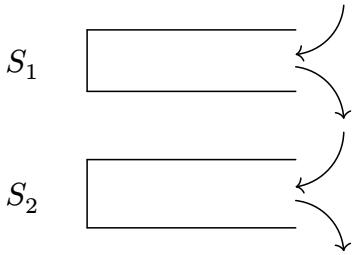
$\tilde{T}(\text{pop}) = 2 = \text{O}(1)$

We also have constant amortized cost.

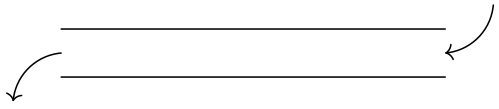
Sometime we have more coins than necessary and $\tilde{T}$ become negative, this is not a problem, just like for the potential function it means that $\tilde{T} = \text{O}(1)$.

6.6 - Queue from 2 stacks

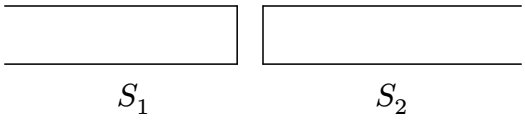
We are given two stacks:



We have to somehow make a queue from these 2 stacks.

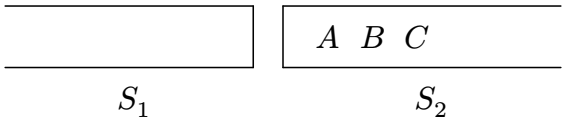


To make this append, we split our queue in 2. 1 half is S_1 the other one is S_2.



To add element, we push in S_2:

```
def add(x)
    s2.push(x)
```



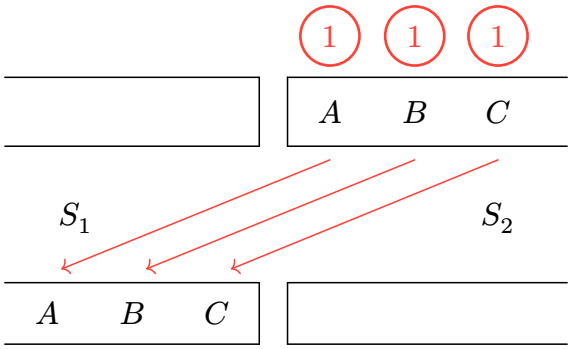
To remove them, we put elements of S_2 into S_1 ten pop from S_1:

```
def remove()
    if s1.empty()
        while !s2.empty()
            s1.push(s2.pop())
    if !s1.empty()
        return s1.pop()
```

That it, we have our stack, lets prove that $\tilde{T}(\text{remove}) = \text{O}(1)$, with the accounting method.

We need to save coins to do our `while` condition in remove.

We save 1 coin each time we add an element in S_2.



We then have:

$$\tilde{T}(\text{add}) = 2$$
$$\tilde{T}(\text{remove}) = 1$$

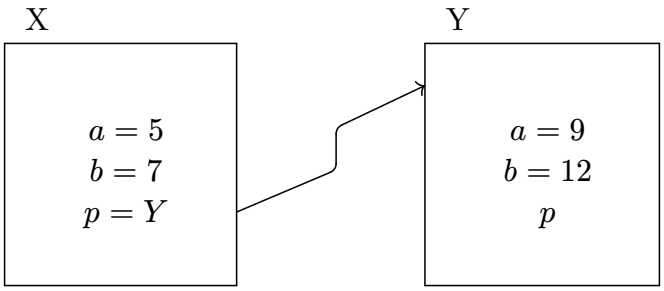
7 - Linked List, Pointer Machine

Until now, we have only used our RAM model.

In this lecture, we introduce a new computational model: Pointer Machine

7.1 - Pointer Machine

In this model, all our data is stored in some nodes.



The pointer p can point to nothing, itself...

Main differences between pointer machine and RAM model:
- In the PTR Machine model, we do not have arrays.
- We only have node and each node have a constant number of PTR.

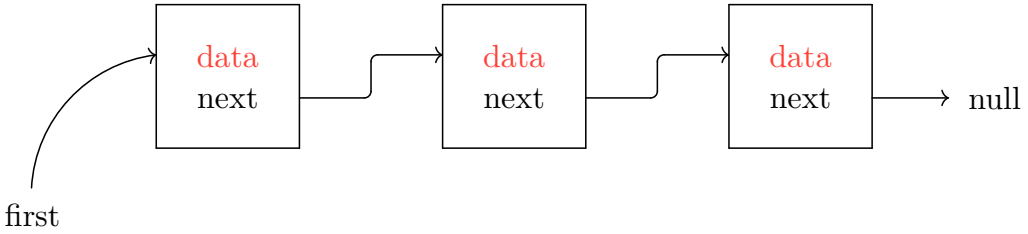
With those limitations, one may ask why would we use this model? This will be discussed later in detail but sometimes a simpler model is better, for now we want to make a structure that allow us to not use any array.

7.2 - Linked List

7.2.1 - Singly linked list

7.2.1.1 - Definition

We have a number of node, each node contain 1 element of the list and a pointer to the next element of the list.



7.2.1.2 - Iterate over the list

```
def print(first)
    x = first
    while x != null
        print(x)
        x = x.next
```

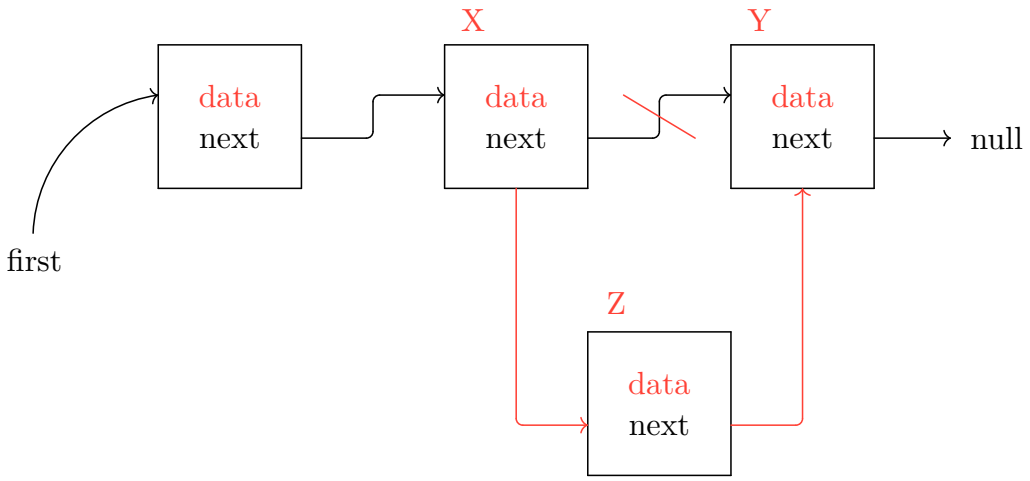
Complexity: $O(n)$ → same as array.

7.2.1.3 - Get the i^{th} element

```
def get(i)
    x = first
    j = 0
    while j < i and x != null
        x = x.next
    if j == i
        return x
```

Complexity: $O(n)$ → slower than array.

7.2.1.4 - Insert element

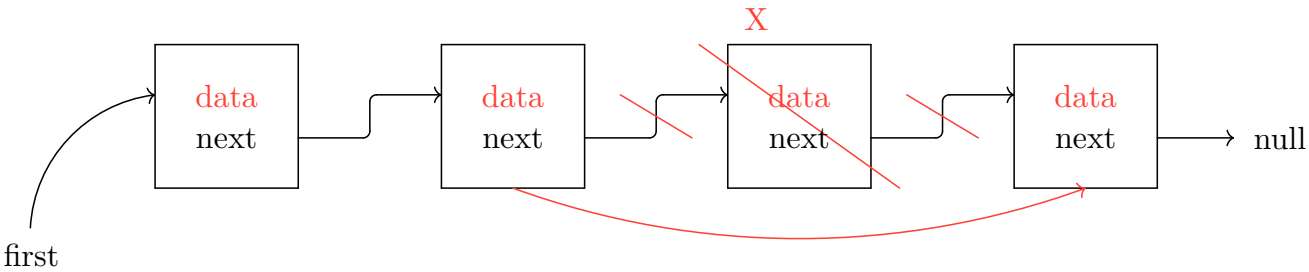



```
def insert_after(x,z)
    y = x.next
    z.next = y
    x.next = z
```

Complexity: $O(1)$ → faster than array.

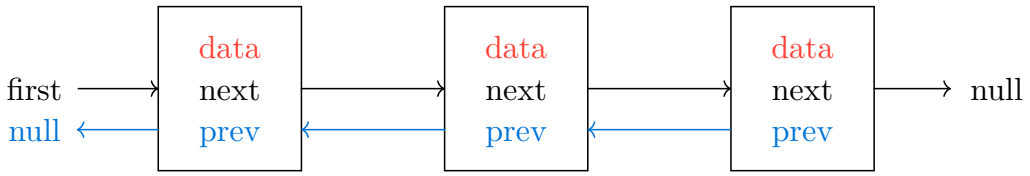
The only difficulty here is that we need a pointer to x.

7.2.1.5 - Remove an element



To remove X, we need to have a pointer to the previous element. To ensure that property, we use doubly linked list.

7.3 - Doubly Linked List



We add a second pointer in each node that point to the previous element.

7.3.0.1 - Insert after

```
def insert_after(x, z)
    y = x.next
    z.next = y
    z.prev = x
    x.next = z
    if y != null
        y.prev = z
```

Complexity: O(1) → faster than array.

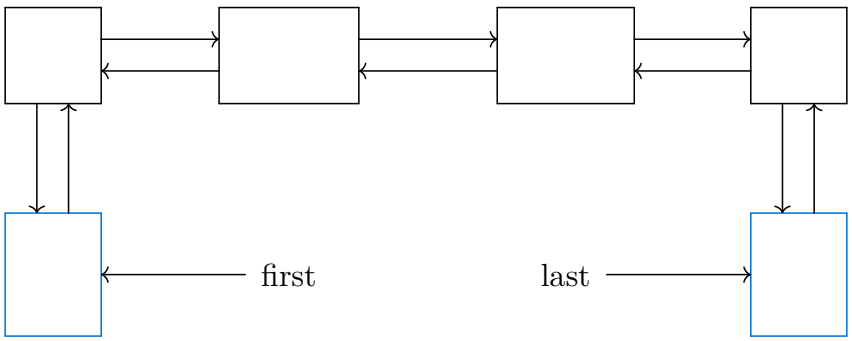
7.3.0.2 - Remove

```
def remove()
    y = x.prev
    z = x.next

    if z != null
        z.prev = y
    if y != null
        y.next = z
    if x = first
        first = z
```

We can see that even for a simple operation, we have many if-statements. Which is a problem because we need to test all of our if-statements.

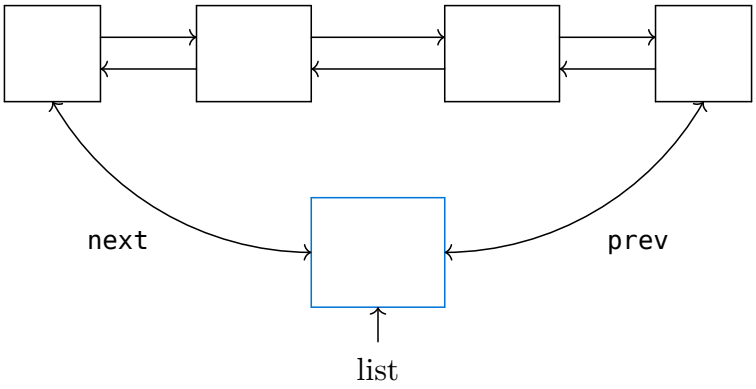
7.3.0.3 - Simplify the structure



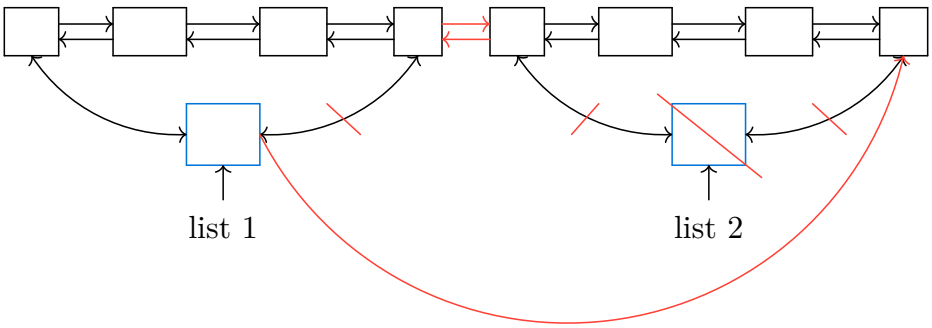
We add two elements to the linked list. Which allow us to get rid of our ifs because we do not change the added the element* only the ones in the middle*. And as we see for the middle elements, the previous and the next pointer are ≠ of null.

We can simplify the data structure a little more:

- Instead of adding two elements, we add only one and linked both end to it.



7.3.0.4 - Concatenate 2 linked list




```
def concat(l1, l2)
    l2.next.prev = l1.prev
    l1.prev.next = l2.next
    l1.prev = l2.prev
    l1.prev.next = l1
```

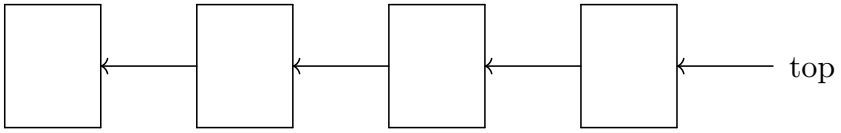
The same goes for the splitting…

```
def split(l1, x)
    l2 = new Node()
    l2.next = x.next
    l2.prev = l1.prev
    l1.prev = x

    x.next = l1
    l2.next.prev = l2
    l2.prev.next = l2

    reutrn l2
```

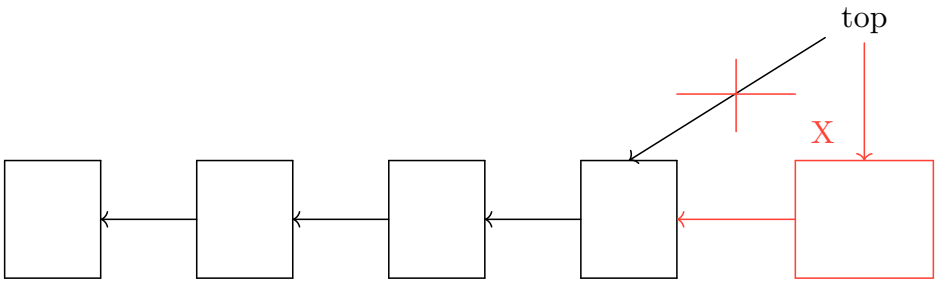
7.4 - Stack



A singly linked list is sufficient to implement a stack because the only thing we need is to go from top to the bottom.

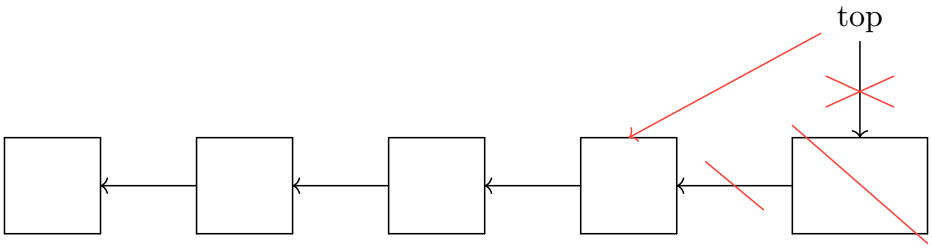
7.4.1 - Lets implement the stack operation.

7.4.1.1 - Push



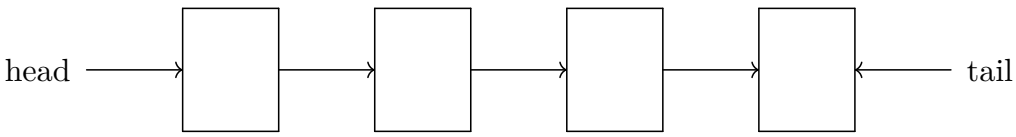

```
def push(x)
    x.next = top
    top = x
```

7.4.1.2 - Pop




```
def pop()
    ans = top
    top = top.next
    return ans
```

7.5 - Queue



Again, singly linked list are sufficient here. Since I just need to be able to go from head to tail. The tricky thing here is handling an empty queue.

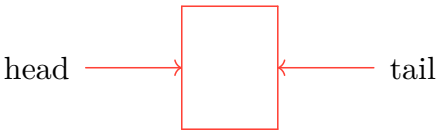
Handling empty queue, 2 way to do it:

- When the queue is empty, both of our pointer are null.
- we use our the trick from the linked list simplification and add an extra node that serve the purpose of handling the empty queue.

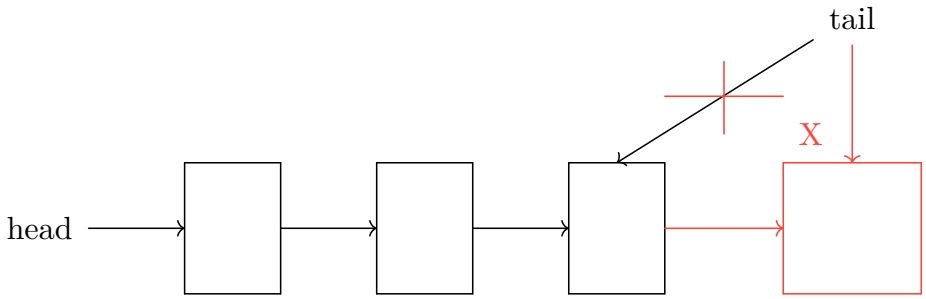
From we are going to go with the null pointer option.

7.5.1 - Add function

7.5.1.1 - Empty case

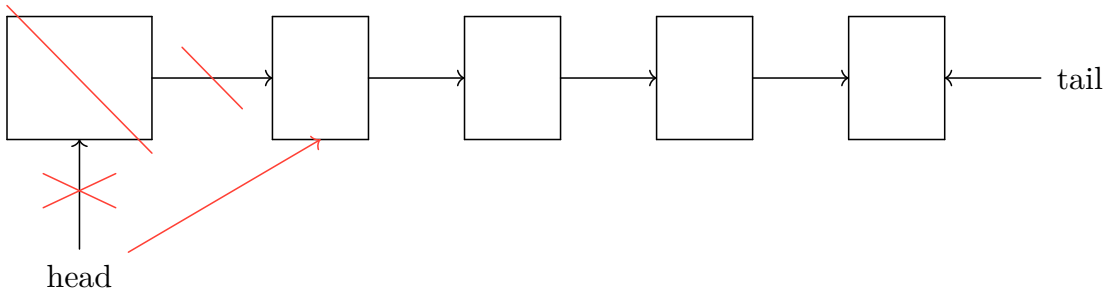


7.5.1.2 - Non-empty case




```
def add(x)
    if head == null
        head = tail = x
    else
        tail.next = x
        tail = x
```

7.5.2 - Remove




```
def remove()
    ans = tail
    if head == tail
        head = tail = null
    else
        head = head.next
    return ans
```

7.6 - Why pointer machine?

Sometime when we use simpler computational model, it is simpler to analyze it.

It is easier to check what part of the data structure is affected by the algorithm.

7.6.1 - Example

In RAM model, if you access our array, you can change it only by giving indexes. In pointer machine, you are obliged to have a pointer to the node to actually change it. Which make easier to check what element have access to our node.

For a node, you just need to have a list that contain all the pointer to this node and know at each instant which element has access to it.

This can reveal to be an important feature, in garbage collection for example. When you create an object with a program, it is linked with other objects. By keeping a list of who is linked to who, you can detect if an object is not accessible anymore (i.e. it is linked to no one) and delete it safely.

This can also be useful in a case of multiple threads, making concurrent call to the same object and want to be sure that only 1 thread at a time can modify your data.

7.7 - Persistent data structures

You have a data structure (any data structure). This data structure as a state, you make an operation on that data structure which changes its state.

Commonly, you forget the base state and keep track only of the new one.

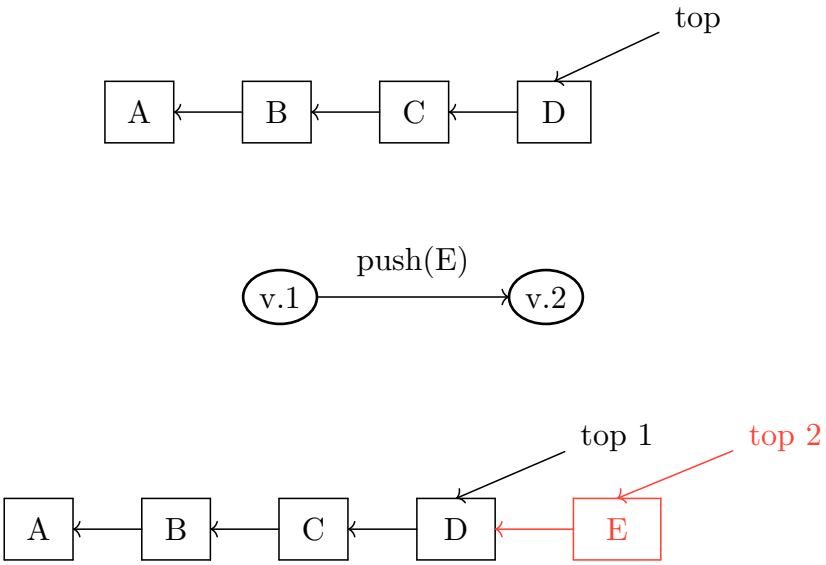
In persistent data structure, you want to maintain all the state an be able to access the previous state.

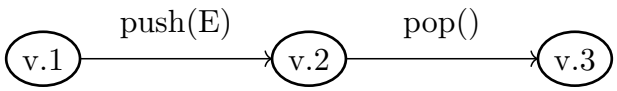
Why? In some algorithm it can be useful to keep the previous state in mind and have a persistent data structure (we will talk about persistent trees in semester 2).

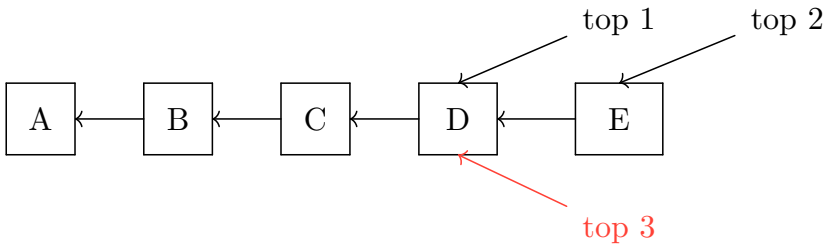
To simply fathom this idea, you can think of it as a version control system such as git.

Almost all data structures that can be done in the pointer machine model can be made persistent without any additional asymptotic costs.

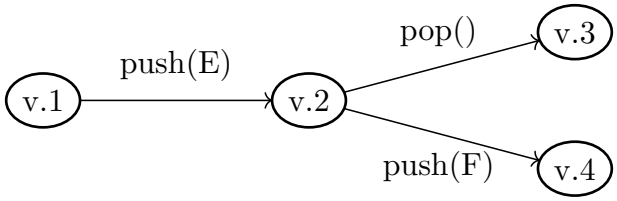
7.7.1 - Persistent stack

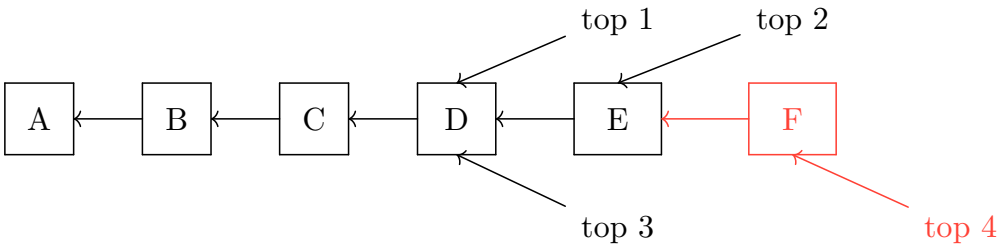




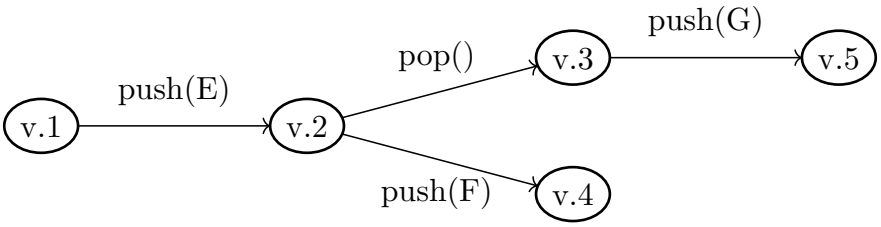


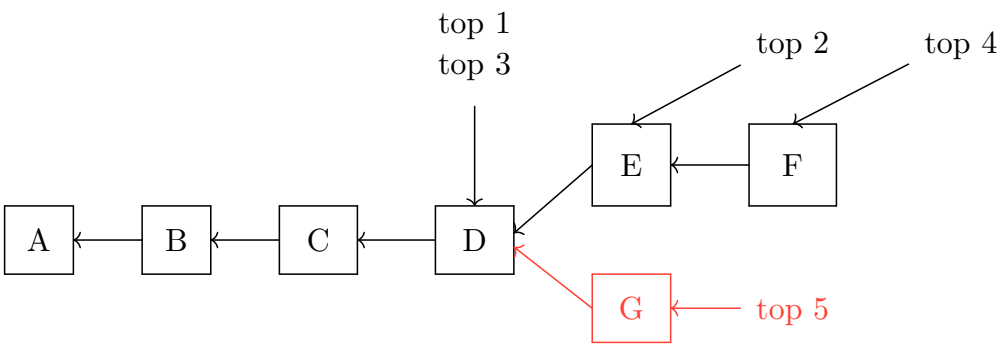
We can push F after $v.2$:





Now we push G after $v.3$:





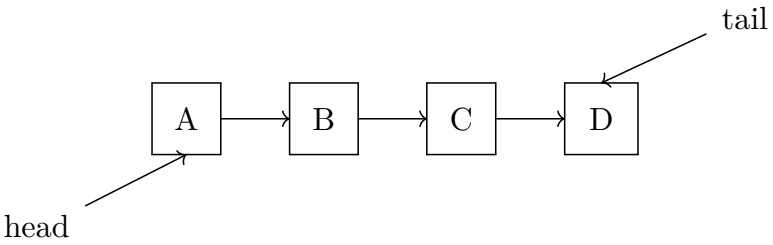
To use a version, you need to have the pointer to top $v.i$. You can for example save those pointer in a list.

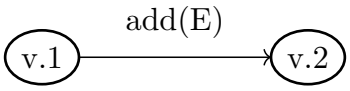
Note: You only need to keep track of the version you need not necessarily all of them.

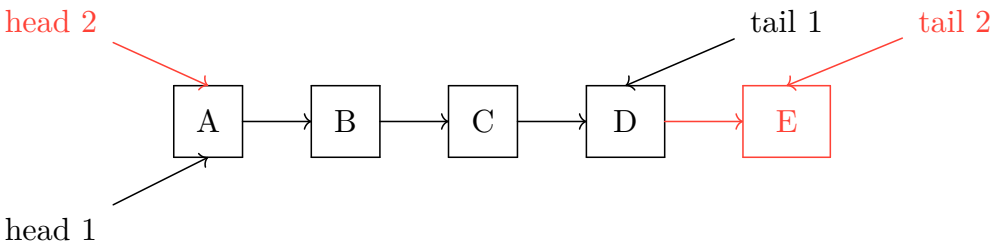
7.7.2 - Persistent queue

Let's try the idea of the stack for the queue, at first sight, there is not much of a difference.

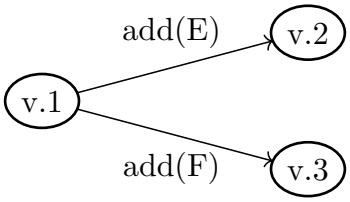
But actually, there is a difference. Let's try to use the same technique and see what goes wrong:

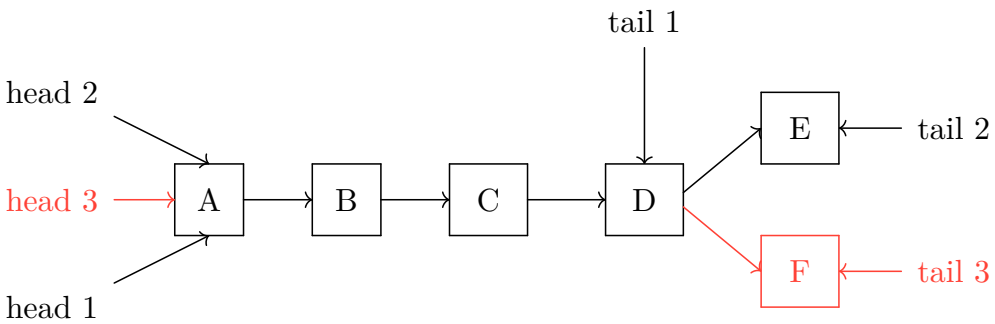






Until now, everything works the same, let's try to add another element after $v.1$:





And from mow we have a problem, which is that we cannot have multiple value for D.next. This is why we cannot use the same technique to create our persistent queue structure than for our persistent stack.

There are various way to solve this problem:

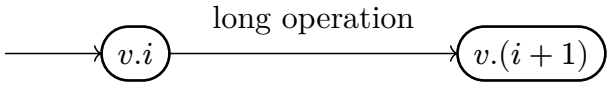
- One of the easiest is to build queue on top of stacks. In the previous lecture, we talk about making queue out of 2 stacks Section 6.6.

  Let's use that:

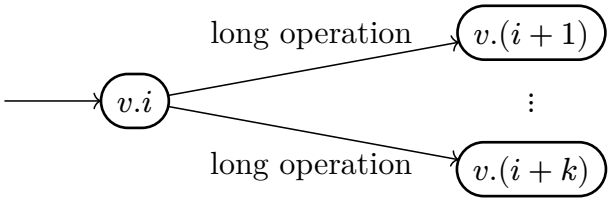
  ▸ You can take 2 persistent stack and it will solve the problem.

  But it will be slow because our queue made of 2 stacks as a constant amortized time complexity only and when you have a constant amortized time complexity, you cannot just make a persistent data structure of constant amortized time complexity.

  Indeed, for example if you are at a version $v.i$ and you want to make an operation:



  Normally our long operation is compensated by a decrease of ϕ. But in persistent data structure, you can go back to $v.i$ and do as many long operations as we want.



  Hence the amortized time cannot be compensated by ϕ.

  The way to solve this problem is to make queue out of stacks with a real constant complexity.

7.7.3 - Partially persistent data structure

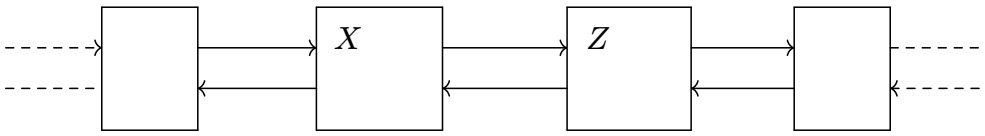
A data structure is said to be partially persistent when you can access to all the older states but make operations only on the last version. We have a linear ordering of our versions.

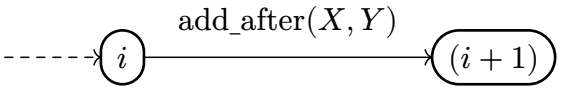
7.7.3.1 - Partially persistent linked lists

The authorized operations are:

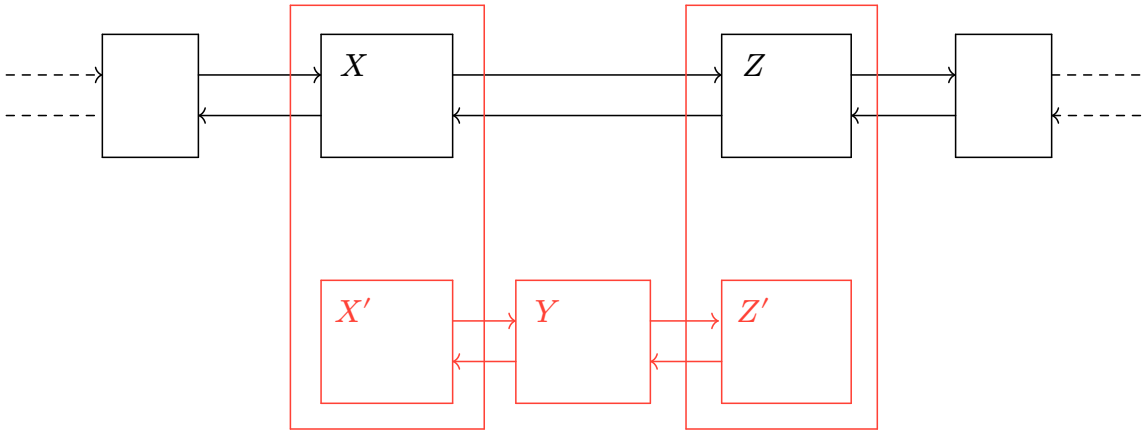
- add_after(x, y)
- remove(x)
- iterate()

7.7.3.1.1 - Add after





After the adding operation, we want to be able to access both version i and $i+1$. In order to achieve that we are going to maintain 2 version of our nodes X and Y.



The idea is to go through X, Z while iterating version i and X', Y, Z' while iterating version $i+1$.

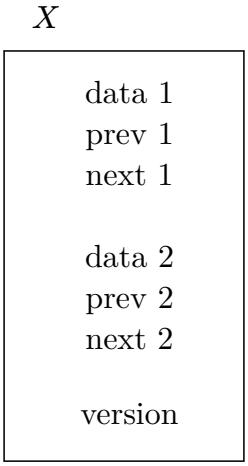
How to check what version of element X is to use when we go along the list?

- We maintain the limit number version (i.e. if version ≥ 6, we go through X' else we go though X.)

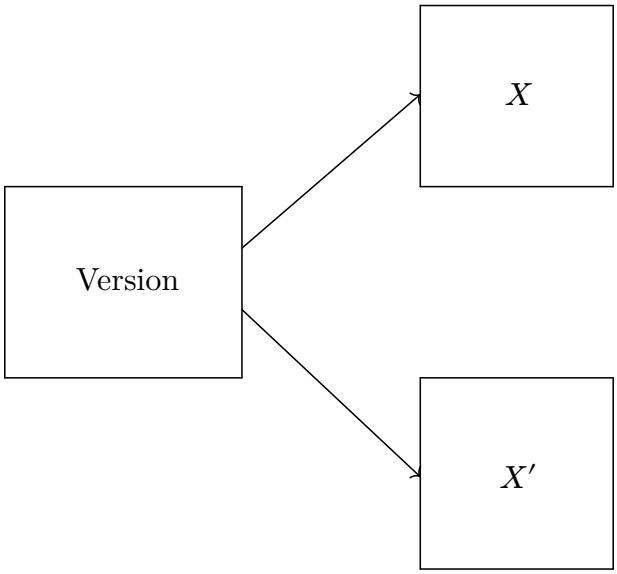
We do this for all the node that have multiple version.

How to apply this idea in a real pointer machine?

- With a big X node:



- With 2 X nodes and a version node:



We can actually do that because we will never have more than 2 versions of a node. If so we would have a list of node version, but it would be impossible to find the good version in $O(1)$ ($O(\log n)$ being the best with balanced binary tree).

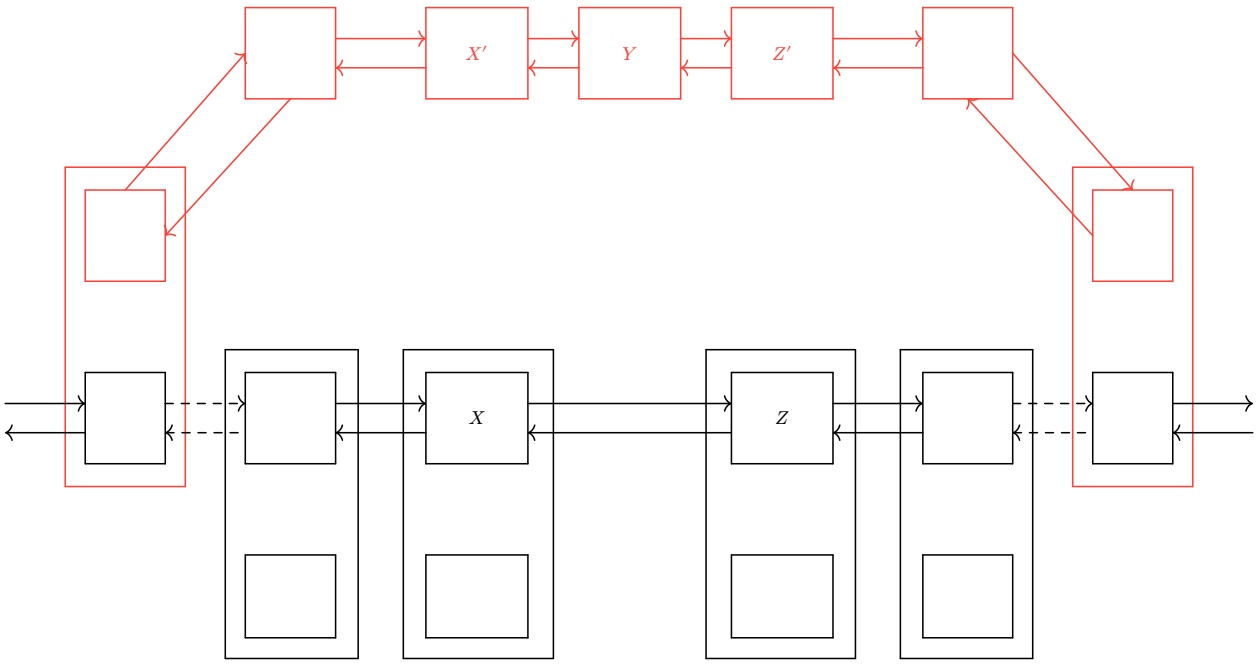
But if we add a new node between Y and Z in our example we will need a third version of Z. Therefore, to add a node Y between X and Z, we will do the following:

- if X have only 1 version, we create a second version for X
- else
  - we create a new node X'

We repeat this process until we reach a node that only have 1 version, and we do the same for node Z but for the element after.

If we reach an end, we add a version to each node.

This gives us the following result:



Time complexity:

Obviously, in a case where all our node have two version: $T(\text{add_after}) = \mathrm{O}(n)$.

But lets prove that since our linked list is partially persistent: $\tilde{T}(\text{add_after}) = \mathrm{O}(1)$.

ϕ definition:
- Our long operation is making a new node when one has already 2 versions.
- we need our ϕ to decrease when we are doing our long operation.

ϕ = number of fat nodes (nodes with 2 versions)

If you are making k copy:
- $T = k, \Delta\phi = -k + 2$ ($+2$ for the nodes on the left and the right).

We then have:

$$\tilde{T} = 2 = \mathrm{O}(1)$$

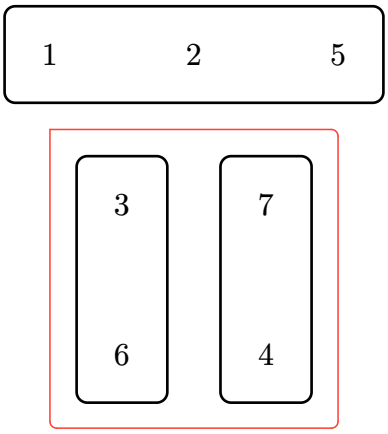
8 - Disjoint sets (Union-Find)

8.1 - Definition

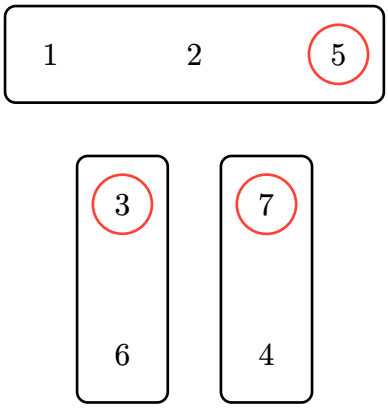
Disjoint sets is a pretty simple data structure. But this structure is important and will be used mostly in our semester 3 (about graph).

Imagine that you have objects and these objects are separated in disjoint sets, we want to perform 2 operations on those sets:

- *Union: it take two of our sets as input and output one set which is the union of those two.



- Find: it take an element x as input and return in which set the element is.
  - To do that, we use two type of object, the set and the element.
  - *To not be bothered by that, we are going to mark one element in each set. This element is unique and will represent the set so that our return type is also an element.



In this example:

- find(6) $\rightarrow$ 3
- find(5) $\rightarrow$ 5
- find(2) $\rightarrow$ 5

Element of the same set return the same result for find.

8.2 - Basic implementation

For this simple representation, we make an array of size n. For each object, we store in the array the representative element of his set:

$$p : \begin{bmatrix} 5 & 5 & 3 & 7 & 5 & 3 & 7 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{bmatrix}$$

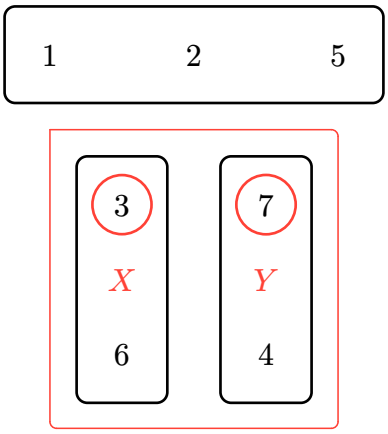
p is our disjoint set from the previous example.

8.2.1 - Find

```
def find(x)
    return p[x]
```

8.2.2 - Union

In order to make our union operation between X and Y, we want to change the representative element of the elements of X to the representative element of Y.



Here, to join our sets, we need to find in p all the elements that have 3 has a representative element and change it to 7.

```
def union(X,Y)
    x = find(X), y = find(Y)
    for i = 1..n
        if p[i] = x
            p[y] = y
```

The complexity of this union function is $\mathrm{O}(n)$.

The number of union operation that we can make in our structure is $\leq n - 1$.

$\rightarrow$ If at the beginning all the object are in a different set, we have n sets. At the end, all of our elements will be in the same set. Since each time we call the union we decrease the number of set by 1. We have at most $n - 1$ unions operations in our structure.

So we can compute the cost of all our union calls.

For this union function: $T(\text{total union}) = \mathrm{O}(n^2)$.

8.2.3 - Union optimization

To optimize the union function, we are going to maintain n list each list are going to store all the element that the object is the referent of.

In our previous example we would have:

$$\begin{aligned}\text{lists} : 1 &: \\ 2 &: \\ 3 &: 3, 6 \\ 4 &: \\ 5 &: 2, 5, 1 \\ 6 &: \\ 7 &: 7, 4\end{aligned}$$

We rewrite the union function using this new structure:

```
def union(X,Y)
    x = find(X), y = find(Y)
    for i: lists[x]
        p[i] = y
        lists[y].add(i)
    lists[x].clear()
```

Did we changed our total cost?

It depends, it is still possible to have: $T(\text{total union}) = \mathrm{O}(n^2)$.

$$\begin{aligned}\text{lists} : 1 &: 1 \\ 2 &: 2 \\ 3 &: 3 \\ 4 &: 4 \\ 5 &: 5 \\ 6 &: 6 \\ 7 &: 7\end{aligned} \qquad \begin{aligned}&\text{union} : 1, 2 \\ &T = 1\end{aligned}$$

$$\begin{aligned}\text{lists} : 1 &: 2 \\ 2 &: 2 \\ 3 &: 3 \\ 4 &: 4 \\ 5 &: 5 \\ 6 &: 6 \\ 7 &: 7\end{aligned} \qquad \begin{aligned}&\text{union} : 2, 3 \\ &T = 2\end{aligned}$$

$$\begin{aligned}\text{lists} : 1 &: 3 \\ 2 &: 3 \\ 3 &: 3 \\ 4 &: 4 \\ 5 &: 5 \\ 6 &: 6 \\ 7 &: 7 \\ &\dots\end{aligned} \qquad \begin{aligned}&\text{union} : 3, 4 \\ &T = 3\end{aligned}$$

In this case, we have $\sum T \simeq \Omega(n^2)$.

We did not reduced our asymptotic cost. But with this structure, we can actually reduce it with a simple trick: instead of always changing the the reference of the same set, we change the reference of the set that is the smaller.

Our union function is then:

```
def union(X,Y)
    x = find(X), y = find(Y)
    if lists(x).size > lists(y).size
        swap(x,y)
    for i: lists[x]
        p[i] = y
        lists[y].add(i)
    lists[x].clear()
```

With change, the previous example cannot occur again. Lets compute our new T(total union):

lists : 1 : 1
2 : 2
3 : 3
4 : 4
5 : 5
6 : 6
7 : 7

union : $1, 2$
$T = 1$

lists : 1 : 2
2 : 2
3 : 3
4 : 4
5 : 5
6 : 6
7 : 7

union : $2, 3$
$T = 1$

lists : 1 : 2
2 : 2
3 : 2
4 : 4
5 : 5
6 : 6
7 : 7
...

union : $3, 4$
$T = 1$

We have: $\sum T = \Omega(n)$, we have made our operation linear.

But now our worst case has changed, it is when we merge two sets of equal size.

Lets look at our element lifetime, how many times $p[x]$ changes? This will give us our T(total union) since our complexity come from there.

- We begin with x alone in a set, we combine it with another set of size ≥ 1.

- x is now in a set of size ≥ 2, we combine it with a set of size ≥ 2.
- x is now in a set of size ≥ 4, we combine it with a set of size ≥ 4.
- And again with sets of size ≥ 8…

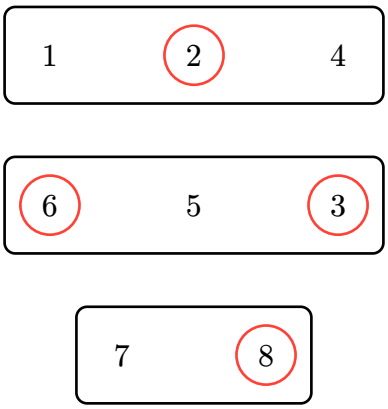
x then changes less than $\log n$ times.

In the worst case:

$$T(\text{total union}) = \sum T = \mathrm{O}(n \log n)$$

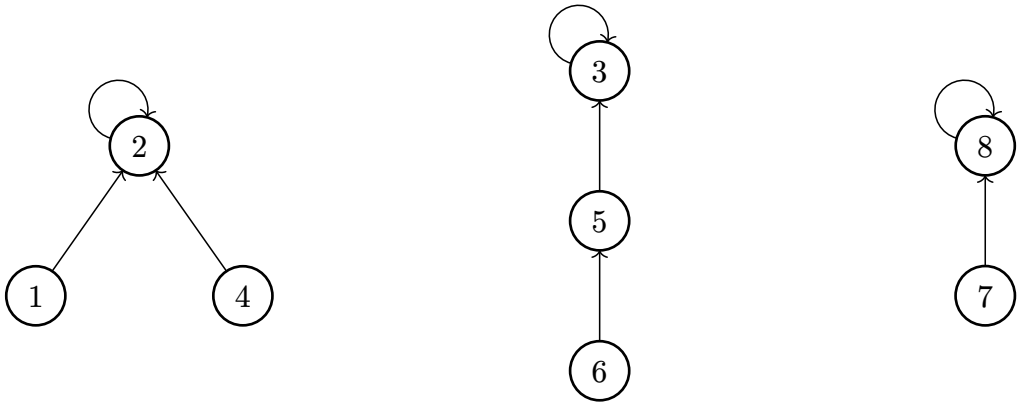
This is a simple yet important technique if you want to make a mergable data structure. You want to put the smallest of the two element by element in the second, which will give $\mathrm{O}(n \log n)$ total moves.

8.3 - Efficient implementation (rank & path compression)



We have our sets and there reference elements.

The idea is to store the sets as tree with the reference element as root.



$p : p[x]$— parent of x

$$p : [2 \quad 2 \quad 3 \quad 2 \quad 3 \quad 5 \quad 8 \quad 8]$$
$$1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8$$

8.3.1 - Find

```
def find(x)
    while p[x] != x
        x = p[x]
    return x
```

8.3.2 - Union

We need to do a find before just as in the previous implementation.

Then, we need to merge trees. The simplest way is to make x point to y.

```
def uinion(x,y)
    x = find(x), y = find(y)
    p[x] = y
```

The complexity of our union function is the same as the one of our find function.

So, what is the complexity of the find function?

Is it possible to have $T(\text{find}) = \mathrm{O}(n)$?

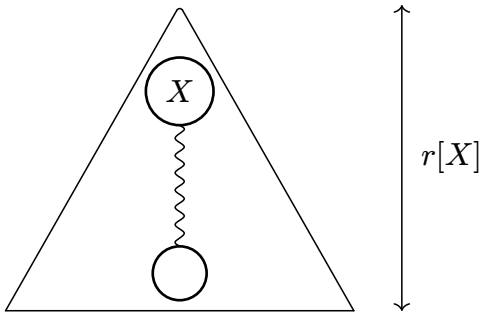
Yes, as before, if we make the merge in a bad order.

This is not a good news we need to work on our find function to improve its time complexity.

8.3.3 - Balance the tree

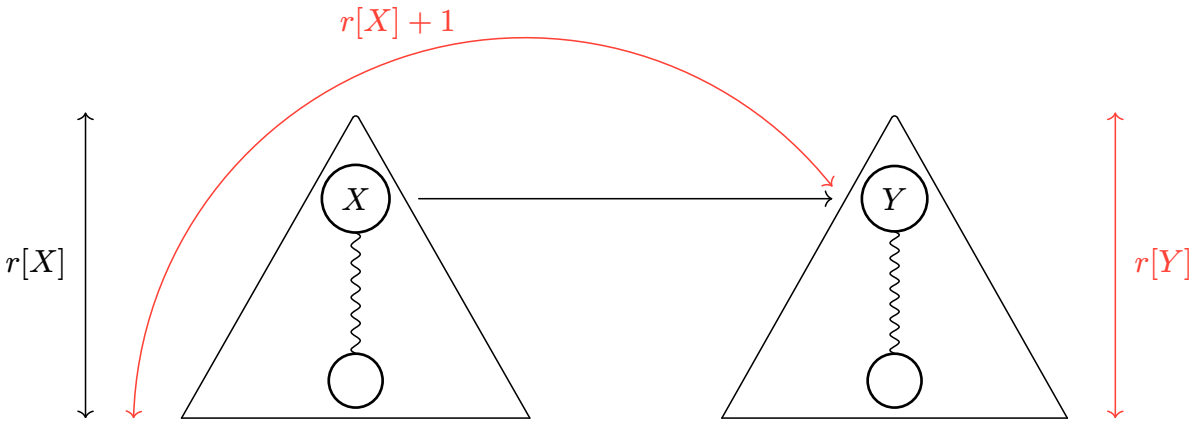
To avoid long path, we need to make the tree balanced. To do that, lets use the rank heuristic.

Rank heuristic: we maintain the rank of each node (the number of node between the node and a leaf):



What we want to do is to connect the tree with the smaller rank to the tree with the bigger rank.

Rank value after union:



$$\text{if} : r[X] < r[Y] : r[X] + 1 \leq r[Y]$$
$$\text{else} : r[X] = r[Y] : r[Y] + 1$$

```
def union(X,Y)
    x = find(X), y = find(Y)
    if r[x] > r[y]
        swap(x,y)
    p[x] = y
    if r[x] = r[y]
        r[y]++
```

Lets prove by induction that if we use this union function, find function has an $O(n \log n)$ complexity.

Our introduction property is that a set with a reference element X of rank $r[X]$ has at least $2^{r[X]}$ elements.

Initialization:
- $r[X] = 0 \rightarrow$ then our set a only 1 element. $1 > 2^0$ so property is true for $r[X] = 0$.

After union operation:
- we join 2 trees:
  - the first as a rank of $r[X]$ and is number of elements is $\geq 2^{r[X]}$
  - the second as a rank of $r[Y]$ and is number of elements is $\geq 2^{r[Y]}$
- if $r[X] < r[Y]$:
  - we saw that $r[Y]$ stay the same.
  - number of element is at least $2^{r[Y]}$ since number of element os first tree is ≥ 1.
- if $r[X] = r[Y]$:
  - $r[Y] = r[Y] + 1$
  - we have at leat $2^{r[Y]+1}$ element in our final tree since both of our trees had at least $2^{r[Y]}$ element at the beginning.

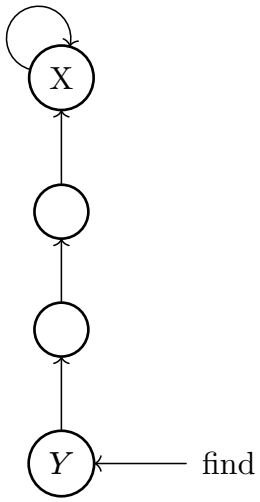
The property stay true after the union operation and is true at the beginning, The property is always true.

Since a tree as at least $2^{r[X]}$ element, we have $r[X] \leq \log n$.

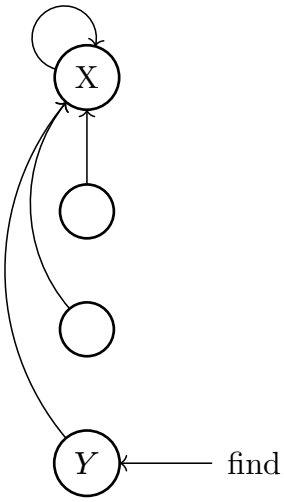
8.3.4 - Path compressing

We want to optimize our find even more.

The first time we call find, we follow this path:



but, we change our pointer to:



so that the next time we try to do a find for an element of this set it will be done in constant time.

```
def find(x)
    y = x
    while p[y] != y
        y = p[y]
    while p[x] != x
        z = p[x]
        p[x] = y
        x = z
    return y
```

Path compression costs: $\tilde{T}(\text{find}) = \mathrm{O}(\log n)$

Path compression + rank heuristics costs:

$$\tilde{T}(\text{find}) = \mathrm{O}(\alpha(m, n))$$

where:

- α is the reverse Ackerman function

- n is the size of the tree (i.e. the number of element in the set).
- m is the number of finds call.

When:

$$n \nearrow \Rightarrow \alpha(m, n) \nearrow$$
$$m \nearrow \Rightarrow \alpha(m, n) \searrow$$

See the first bonus chapter at the end of this semester for a proof.

In the mean time, we are going to prove the case

$$m = n \Rightarrow \alpha(m, n) = \log^* n$$

Where $\log^*$ is the iterated logarithm function:

$$\log^* n \stackrel{\text{def}}{=} \begin{cases} 0 \text{ if } n \leq 1 \\ 1 + \log^*(\log n) \end{cases}$$

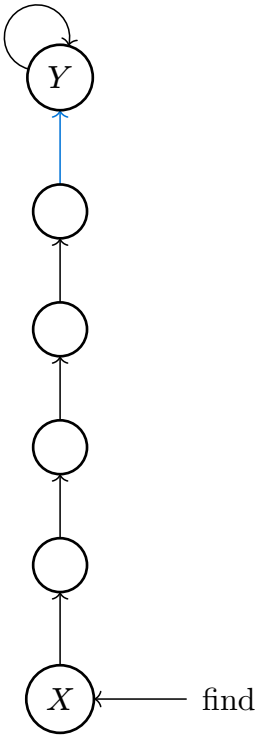
Example for $\log^*$:

$$2^{65536} \underset{\log}{\to} 2^{16} \underset{\log}{\to} 2^4 \underset{\log}{\to} 2^2 \underset{\log}{\to} 2^1 \underset{\log}{\to} 2^0 \Rightarrow \log^*(2^{65536}) = 5$$

Lets prove that $\tilde{T}(\text{find}) = \mathrm{O}(\log^* n)$:

$$T(\text{find}) = \text{length of path between } X \text{ and the root } Y$$

We want $\tilde{T}(\text{find})$ so we want the sum of all path in my find call.



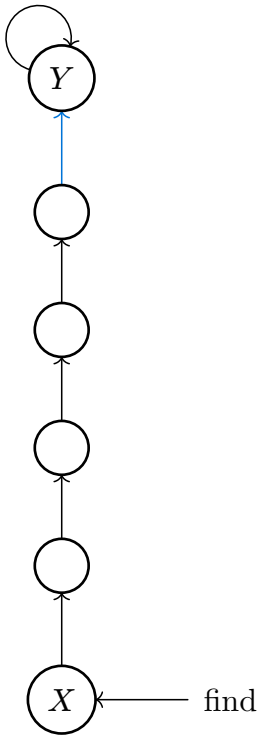
In each operation we have a last edge*.

To calculate the other edge, we split them in two groups:

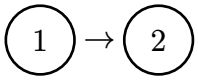
- *small jump: $r[p[X]] < 1.9^{r[X]}$
- *big jump: $r[p[X]] \geq 1.9^{r[X]}$

The number of big jump is $\leq \log^* n$ because $r[X] \leq \log n$.

Number of small jump:



When we call find, $r[p[X]]$ is increasing because the link from:



and $r[Y] > r[p[X]]$

How many jump will it take for Y to have a jump that is bug and not small?

- for X: $\leq 1.9^{r[X]}$ operation until $X \to p[X]$ become a big jump.

$$\sum_x 1.9^{r[X]} = \underbrace{\sum_r \text{cnt}[r] \cdot 1.9^r}_{\text{we group the } X \text{ by rank}}$$

The number of node of rank r is $\leq \frac{n}{2^r}$ since each tree as at least 2^r elements.

$$\sum_r \text{cnt}[r] \cdot 1.9^r \leq n \cdot \underbrace{\sum \left(\frac{1.9}{2}\right)^r}_{=\mathrm{O}(n)}$$

So:

$$\begin{aligned}\sum T(\text{find}) &\leq m + m \cdot \log^* n + n \\ &\leq \mathrm{O}(n \log^* n)\end{aligned}$$

since $m = n$ by hypothesis.

We have proved that $\tilde{T}(\text{find}) = \mathrm{O}(n \log^* n)$

This structure can be modified to compute function on the tree, this is wht it is such an important data structure.

9 - Fibonacci Heap

This is a heap that support more operations than a classical one.

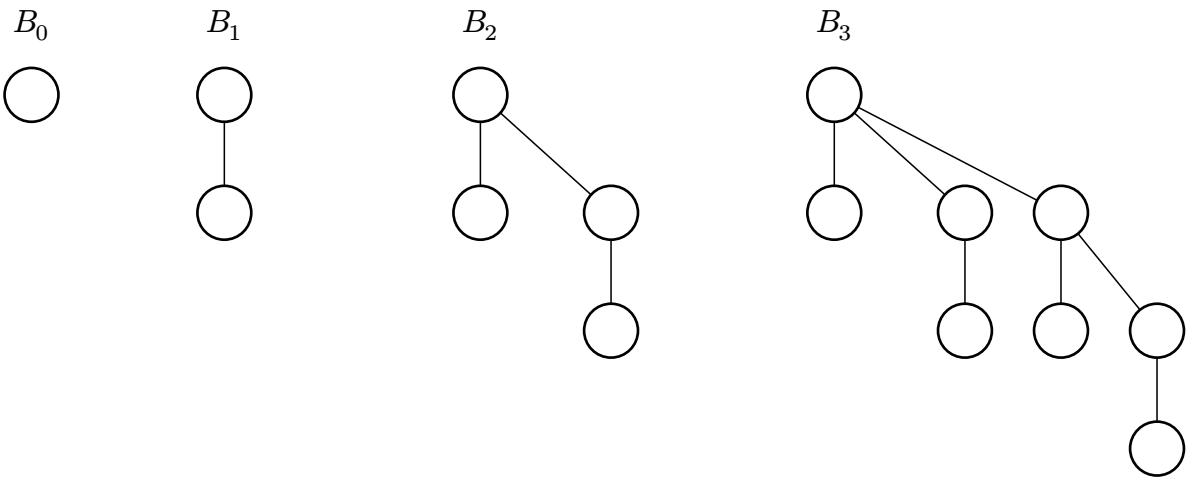
Classical operations are:
- add(x)
- remove_min()

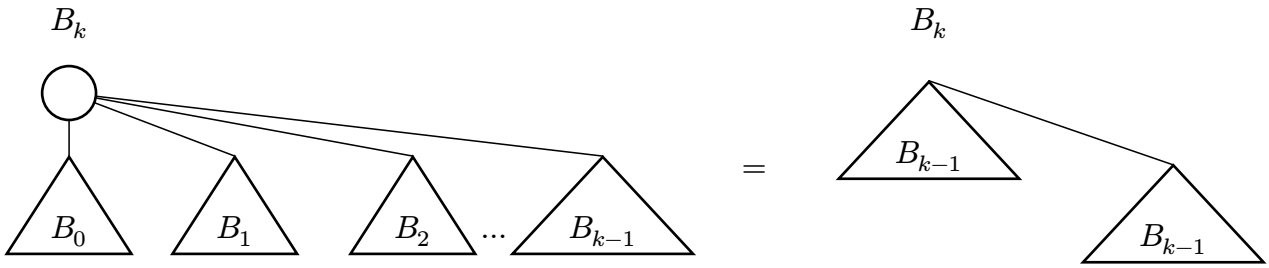
New one are:
- merge(H_1, H_2) → merge two heap into 1 big one
- decrease_key(x, y) → change the value of node x by y, can only be done when $y < x$

For binary heap, we have:
- add(x) → $\log n$
- remove_min() → $\log n$
- decrease_key(x, y) → $\log n$

9.1 - Binomial trees

Before going onto the Fibonacci heap, we are going to talk about binomial trees and heap. This will ease our comprehension when we slide to the Fibonacci heap.





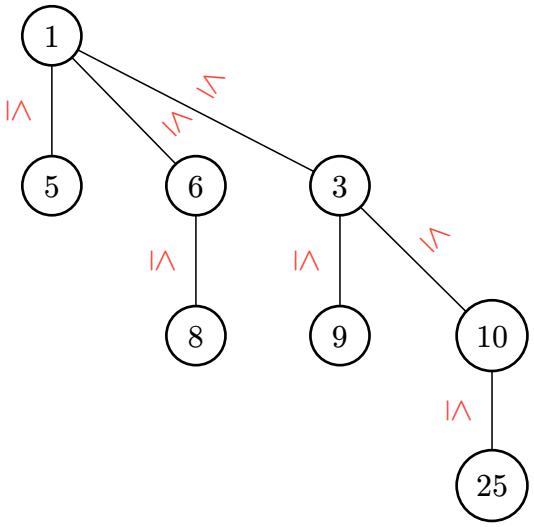
Number of nodes in $B_k = 2^k$.

9.2 - Binomial heap

9.2.1 - Definition

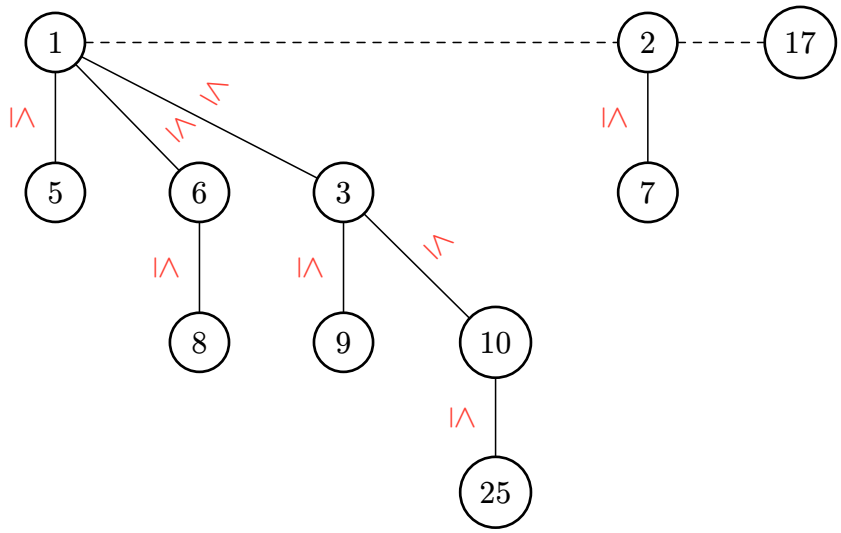
We build a heap on top of binomial tree like we did with binary tree.

Example with $n = 8$:



But if $n \neq 2^k$ for example $n = 11$.

Then, we take multiple tree:

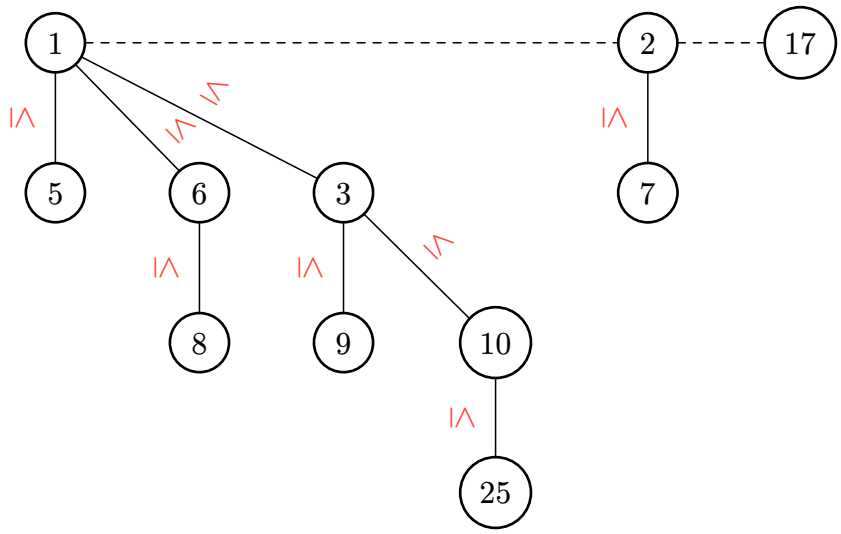


For a given number of element, the tree structure will be the same. And it will be composed of trees from decomposition in power of 2 (the minimum number of tree).

Two consequences:

1. All the ranks are different because if 2 trees have the same rank, we can use the superior power of binomial tree.
2. number of trees $\leq \log n$.

9.2.2 - Add



We want to add the node 4 to this binomial heap. The node 4 is a binomial tree of rank 0. What we want to do is to add the node as a tree in our heap.

- if B_0 is not already present in our heap, we add it to the heap as B_0.
- otherwise, we merge the B_0 of our heap and the node we want to add.
  - to do that we take the minimal root and set it as the root of our B_1 tree.

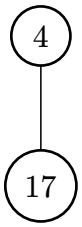
  Now, we only have a B_1 tree to add to the binomial heap.
- the same process is repeated, if B_1 is not already present in our heap, we add it to the heap as B_1.
- otherwise, we merge the B_1 of our heap and the node we want to add.
  - to do that we take the minimal root and set it as the root of our B_2 tree.

  Now, we only have a B_2 tree to add to the binomial heap.

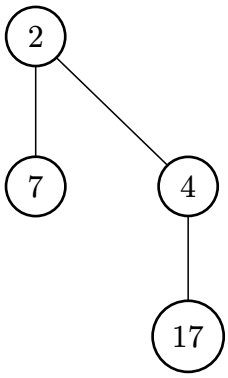
We do the same until we merge a B_k that is not in our heap.

Lets see what our example gives us:

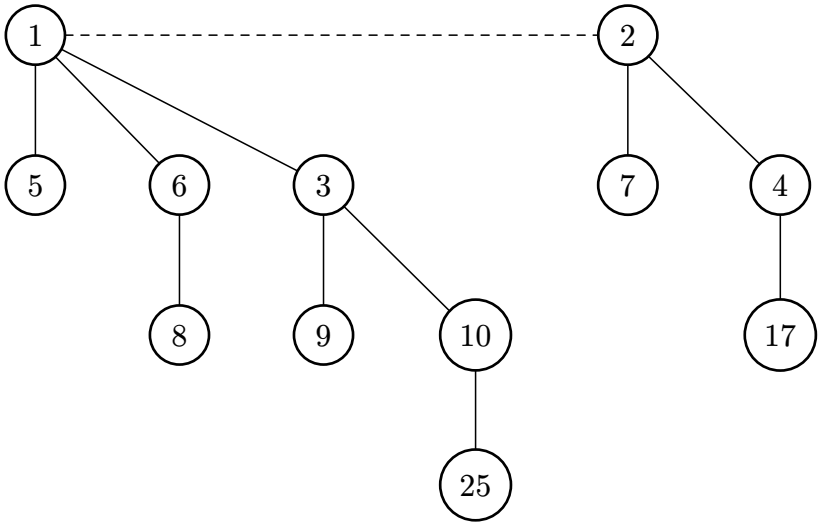
- B_0 is in our tree, so we merge our node 4 with the B_0 of the heap, we obtain:



- B_1 is in our tree, so we merge our B_1 with the B_1 of the heap:



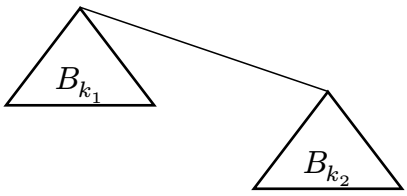
- B_2 is not in our tree so we can add it to the binomial heap and get our resulting heap after adding:



Since our addition is based on merged, to have a fast addition, we need to have a fast merge operation.

9.2.3 - Merge

$B_k + 1$

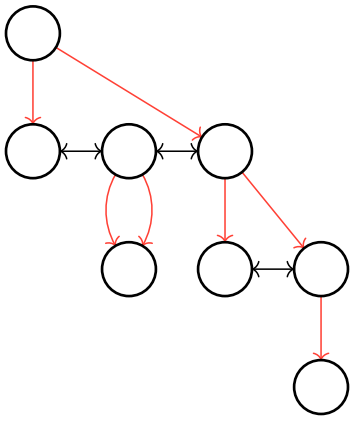


With B_{k_1} root less than B_{k_2} root.

This can be done in constant time if we use a good data structure to make our trees.

We represent the child of a node as a linked list. The parent has access to the first and last element of the linked list (i.e. the first and the last child).

Example fort B_3:



With this representation, we can just merge our elements only by putting the root element of our second heap at the end of the child list of the root of the first heap.

With this technique that allow us to merge two trees of same rank in constant time, our merge function will work in $O(\log n)$.

Indeed our technique to merge is the same idea as the merge sort:
- We put a pointer on the smallest tree of each heap
- If the size of the tree for both pointer is the same we merge them,
  - We try to put our result in the resulting heap
- Otherwise we take the smallest,
  - We try to put the smallest tree in the result heap
- To add it in the result heap,
  - We check if a tree of the same size is in already in the heap,
  - If not we add it at the end of our heap
  - Otherwise, we merge the two and try add the result in the result heap
  - We repeat that until we are able to place the tree in the heap

With this technique the time complexity of our merge function is $O(m)$ where m is the number of tree in the heap. And since there is at most $\log n$ tree in a heap, the merging complexity is $O(\log n)$.

9.2.4 - Remove minimum

The minimum is at the root of the frist tree, we remove the root and take it child.

All of those children are the root of a binomial heap of different length, so we make a heap of those child. Now, we have two binomial heap, the one we just made with the child of the first tree. And the part that was left in our beginning tree.

We just merge the two of them in order to have one only tree.

This put the complexity of our remove minimum function to $O(\log n)$.

9.2.5 - Decrease key

The idea is to do the same as in the binary heap:
- Change the value
- Sieve up to recover heap properties.

9.3 - Fibonacci heap

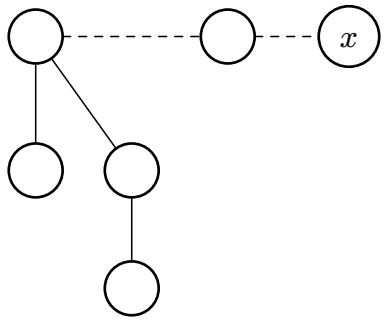
The idea of Fibonacci heap is to have the same function as the binomial heap but with a complexity of $O(1)$ for some of them.

Our goal is:
- $T(\text{add}) = O(1)$
- $T(\text{merge}) = O(1)$
- $\tilde{T}(\text{decrease key}) = O(1)$
- $T(\text{remove min}) = O(\log n)$

9.3.1 - Add

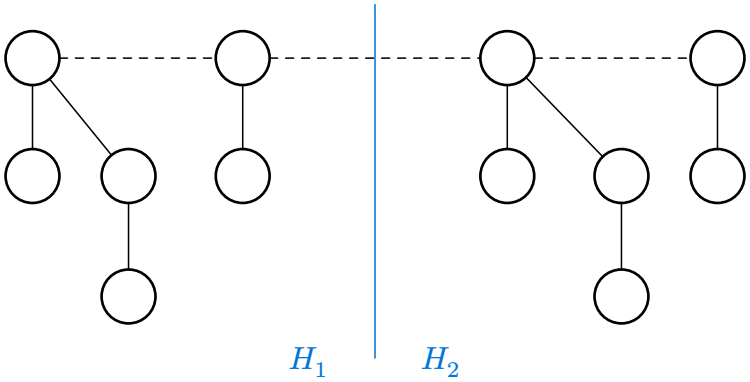
By removing the property that all tree must have different rank, adding in constant time is a trivial problem.

We just take the element we want to add and add it at the end of the heap as a B_0.



9.3.2 - Merge

With our rank property gone, merging has become as hard as adding.



We then have our merge of complexity O(1).

9.3.3 - Remove minimum

We need to maintain a pointer to the minimum element:

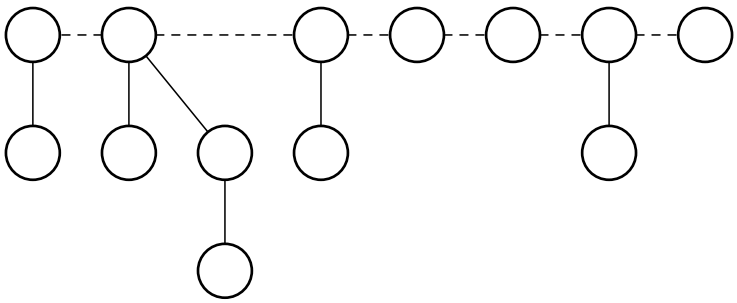
- When we add, we check if our new element is less than our current minimum
- Same for merging, both of the tree have a minimum, we keep the smaller.

With this, we can just remove the minimum element. After that, we take each child as a tree and add them in the heap.

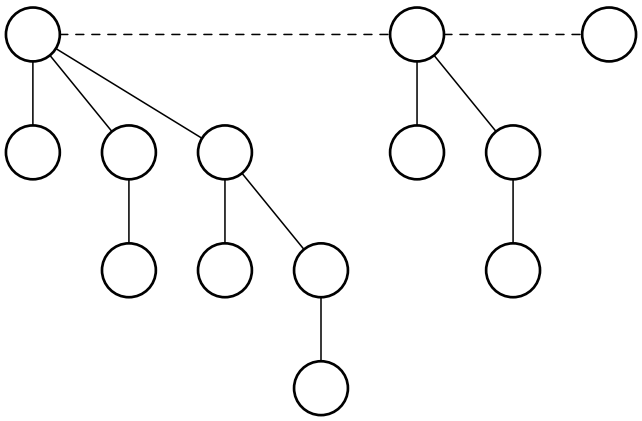
Now we need to find the new minimum we need to check for each tree and we have at most n of them. Since this operation is going to cost us some time, we are going to use it to balance our trees.

Lets take an array listing the rank of trees and if 2 trees have the same rank, we merge them.

From this:



We end up with this:



At the end of this clean up, all the tree have different rank. There is at most $\log n$ tree, we can iterate overthem to find the new minimum.

The cost of this clean up will define the cost of the remove minimum function.

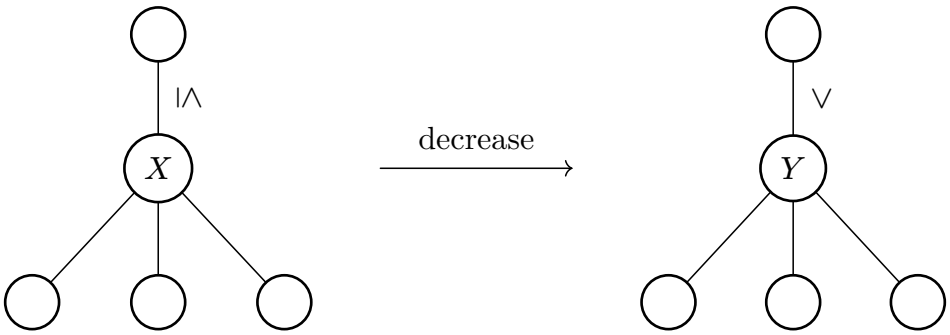
We go from m to $\log n$ tree.

$$\phi \overset{\text{def}}{=} \text{number of trees} = \log n - m$$

$$\begin{aligned}\tilde{T}(\text{remove min}) &= T + \Delta\phi \\ &= m + \log n - m \\ &= \mathrm{O}(\log n)\end{aligned}$$

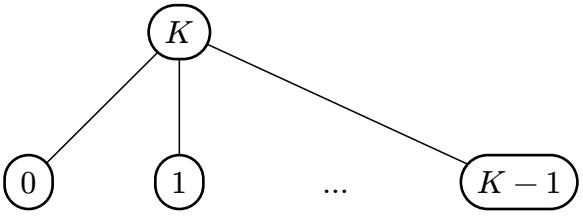
9.3.4 - Decrease key

When the value of our node break the heap property:

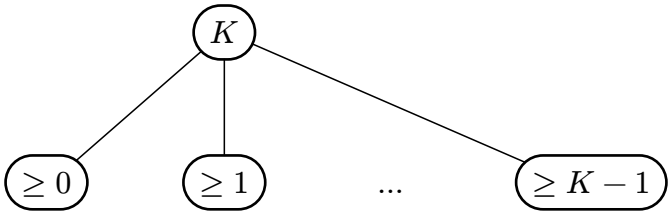


We have a problem, because the structure of the binomial tree is to rigid to achieve constant amortized time for the decrease key function.

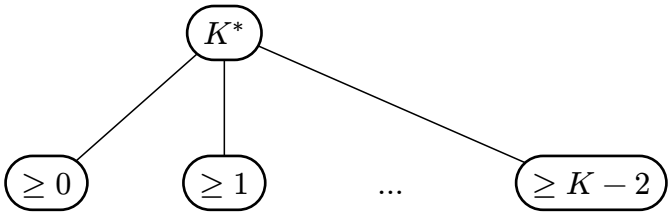
For now, our tree look like:



Lets change the tree structure to:



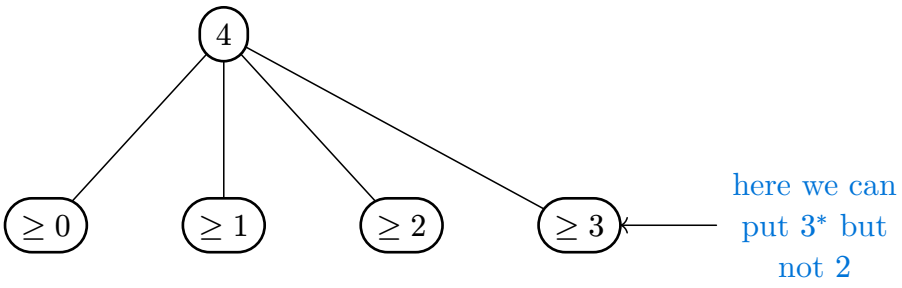
and



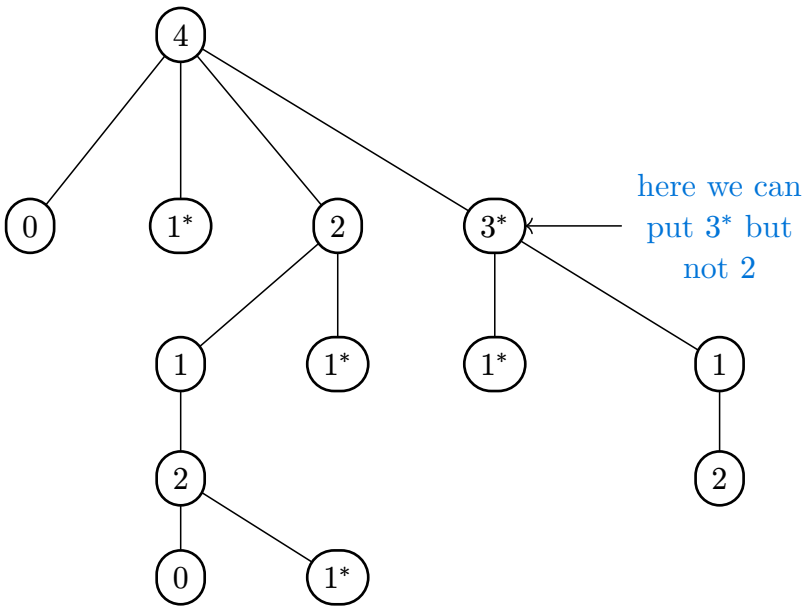
K^* is a tree of rank K that have a child less.



The difference between those two trees is that 3^* is of rank 3 and not 2.



This is a valid Fibonacci tree:



The rank of this tree is less or equal than $c \cdot \log n$.

How do we decrease the key:

- If our new element X is less than is parent A, we put the tree of root X at the root. We put a mark on A
- If A was already marked, we cannot remove 1 more of its child. So we put the tree of root X and the tree of root A at the root. And we put the mark on the parent of A.

We do that until the parent is not marked then we add a mark to it.

For the marked tree we moved to the root, their rank goes from K^* to $K-2$.

$$\tilde{T} = \underbrace{T}_{\substack{= \text{number of moved marked tree} + 1 (\text{for } X) \\ = H+1}} + \Delta\phi$$

Lets redefine ϕ for it to be enough for all our operations.

$$\phi \overset{\text{def}}{=} \underbrace{\text{number of tree}}_{\text{from remove min}} + 2 \cdot \text{number of marked tree}$$

$$\begin{aligned} \tilde{T}(\text{decrease key}) &= (H+1) + (H+1-2H+2) \\ &= (H+1) + (3-H) \\ &= 4 \\ &= \mathrm{O}(1) \end{aligned}$$

We decrease key in constant time.

Thats it we have our Fibonacci heap with the desired complexity for each function.

10 - Dynamic Programming - PART I

Dynamic programming is one of the basic technique of algorithm theory. It is used in various algorithms to solve various type of problems.

In this first part we are going to see common simple problem that will make us understand the basics of dynamic programming.

10.1 - Fibonacci sequence

10.1.1 - Definition

$$F_0 = 0; F_1 = F_2 = 1;$$
$$\forall n > 1, F_n = F_{n-1} + F_{n-2}$$

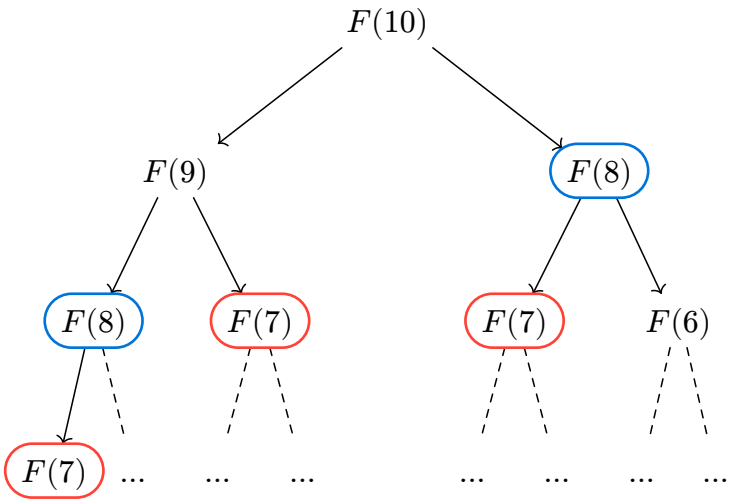
10.1.2 - Naive implementation

```
def F(n)
    if n == 0
        return 0
    if n <= 2
        return 1
    else
        return f(n-1) + f(n-2)
```

This solution is totally correct, but its time complexity is far too big.

$$\begin{aligned} T(n) &= 1 + T(n-1) + T(n-2) \\ &\geq F_n \\ &\geq \varphi^n \\ &= \mathrm{O}(2^n) \end{aligned}$$

An exponential complexity is not usable in real life. Lets try to understand why our complexity is that big. Then we will try to decrease it.



As we can see, we are computing the same value multiple time (red and blue nodes for example). Our objective is that our next implementation compute each value only one time.

In order to do that, we will use memoization. We save the Fibonacci number for each n. When we will have to use again that number, we will get it from our saved array instead of computing it again.

10.1.3 - Memoization

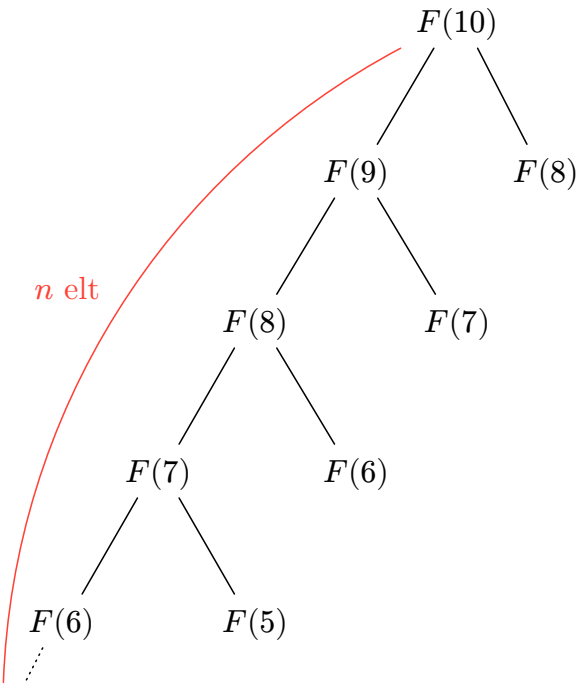
The objective here is to save the Fibonacci number for each n. When we will have to use again that number, we will get it from our saved array instead of computing it again.

```
def F(n)
    res[0] = 0, res[1] = res[2] = 1
    if res[n] != null
        return res[n]
    else
        res[n] = F(n - 1) + F(n - 2)
    return res[n]
```

Since we do the recursive call 1 time for each n and that the call of get in our array is $\mathrm{O}(1)$, we have:

$$T(n) = \mathrm{O}(n)$$

Our new call tree is now:



10.1.4 - Bottom up approach

Lets simplify the procedure using for-loop instead of recursion.

```
def F(n)
    res[0] = 0, res[1] = res[2] = 1
    for i = 3..n
        res[i] = res[i-1] + res[i-2]
    return res[n]
```

Note 1: if you want you can make this function keeping only the 2 previous result to achieve the computation of F_n in order to optimize the memory usage. But it is not the object of this session.

Note 2: Fibonacci sequence is well know and we can achieve even better complexity to compute it. If you want to see cool technique to do that such as fast Fourier transform and other things, see the video → One second to compute the largest Fibonacci number I can.

10.2 - Grasshopper problem

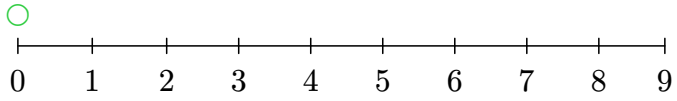
We are given an array, a grasshopper is on the leftmost cell of this array. This grasshopper want to go to the rightmost cell of the array.

It can make a small jump:
- from index i to $i+1$

and big jump:
- from i to $i+2$

We want to calculate the number of different path that exist between leftmost and rightmost position.

Example:



If we have a path that make us arrive on the cell 9:
- It means that we either came from cell 8 with a small jump
- Or cell 7 from a big jump

The number of path to the cell 9 is then the number of path from cell 8 + the number of path from the cell 7.

Lets define:

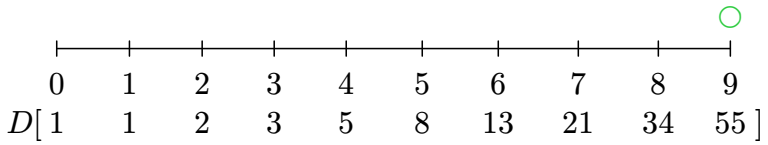
$$D[n] \overset{\text{def}}{=} \text{number of path to cell } n$$
$$D[n] = D[n-1] + D[n-2]$$

We have the same recursion relation than for the Fibonacci sequence. But here our initial values are:

$$D[0] = D[1] = 1$$

```
def grasshopper(n)
    D[0] = D[1] = 1
    for i = 2..n
        D[i] = D[i-1] + D[i-2]
    return D[n]
```

Example answer:



10.3 - Number of binary vector not containing `11`

We have a binary vector of length n. We want a vector not to contain the sequence `11`.

How many different vectors to not contain `11` are they for a given n?

Lets define:

$$D[n] \overset{\text{def}}{=} \text{number of different vectors without `11`}$$

$$D[n] = \underbrace{D[n-1]}_{\text{our vector end by `0`}} + \underbrace{D[n-2]}_{\text{our vector end by `01`}}$$

We have $D[0] = 0$, $D[1] = 2$.

This kind of problems can occurs in some encoding theory algorithm. It is more practical than the other problems but not so much.

10.4 - Grasshopper with k jumps

10.4.1 - 3 jumps

We simply add one more layer to our first grasshopper algorithm.

```
def grasshopper_3(n)
    D[0] = D[1] = 1, D[2] = 2
    for i = 3..n
        D[i] = D[i-1] + D[i-2] + D[i-3]
    return D[n]
```

The algorithm begin to be bad. We have many special case, which is never a good thing.

This will become a real problem if we add more jumps.

10.4.2 - k jumps

Now our grasshopper may do k types of jumps:

- i^{th} cell $\to$ $(i+1)^{\text{th}}$ cell
- i^{th} cell $\to$ $(i+2)^{\text{th}}$ cell
- ...
- i^{th} cell $\to$ $(i+k)^{\text{th}}$ cell

In this case we cannot use the same technique of just adding more layer as the one for 3 jumps.

We can only compute the first value by hand:

$$D[0] = 1$$

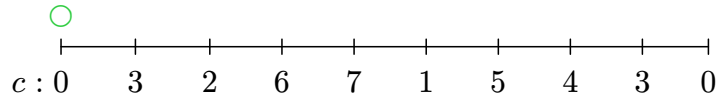
The idea is to make a cycle for all the possible jump in our loop. We check if the jump is possible, if so, we add this value to our result.

```
def grasshopper_k(n,k)
    D[0] = 1
    for i = 0..n
        for j = 1..k
            if i - j >= 0
                D[i] += D[i-j]
    return D[n]
```

The complexity here is $\mathrm{O}(n \cdot k)$. This can be easily done in $\mathrm{O}(n)$ using prefix sum instead of the second loop.

10.5 - Grasshopper with cell cost

Now, each cell the grasshopper can hop on as a cost.



You want to go from the leftmost to the rightmost element spending the less amount of coins.

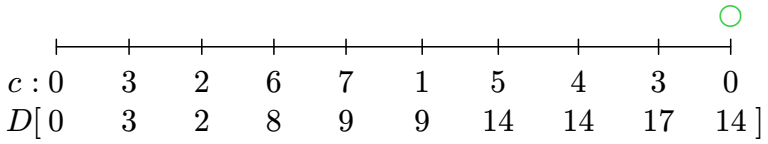
We do the same as before, split the different path in some sets.

If you want to reach the n^{th} cell then the last jump was:
1. from $(n-1)^{\text{th}}$ cell
2. from $(n-2)^{\text{th}}$ cell

$$D[n] \overset{\text{def}}{=} \text{miminum cost to reach the cell } n$$

$$D[n] = \underbrace{\min}_{\text{optimal cost}} \left(\underbrace{D[n-1]}_{1}, \underbrace{D[n-2]}_{2} \right) + \underbrace{c[n]}_{\text{cost of last jump}}$$

Lets see the result for our example:




```
def grasshopper_cost(n,c)
    D[0] = 0, D[1] = c[1]
    for i = 1..n
        D[i] = min(D[i-1],D[i-2]) + c[i]
    return D[n]
```

10.6 - Grasshopper with cell cost + k jumps

The idea is the same but instead of having the minimum of the k previous element:

$$D[n] \overset{\text{def}}{=} \text{miminum cost to reach the cell } n$$

$$D[n] = \min_{i \in [\![1,k]\!]} (D[n-i]) + c[n]$$

```
def grasshopper_k_jump(n,k,c)
    D[0] = 1
    for i = 1..n
        D[i] = + inf
        for j = i..k
            if i-j >= 0
                D[i] = min(D[i],D[i-j]+c[i])
    return D[n]
```

The complexity here is also $\mathrm{O}(n \cdot k)$. Again, this can be improved to $\mathrm{O}(n)$ using sliding windows and maintaining is minimum.

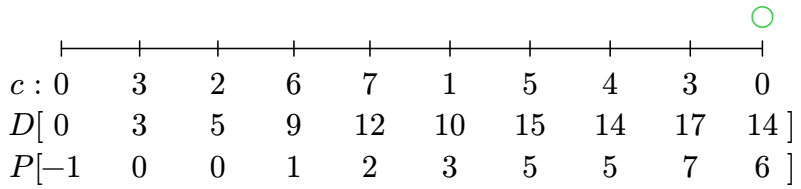
Now that we now the cost of the optimal path, we want to get the path itself.

The idea is to start from the n^{th} cell and check where are we coming from. Then we repeat that for our result until we arrive on the first cell.

Analyzing that, we see that we are just running the algorithm backward. So what we want to do is to maintain an other array. Lets call it P. Where:

$$P[n] \overset{\text{def}}{=} \text{the last cell on the path}$$

For our previous example we got:



To get our path when we have P, we start from $P[n]$ and go on the next index until reaching -1.

```
def grasshopper_path(n,k,c)
    D[0] = 0
    for i = 1..n
        D[i] = + inf
        for j = i..k
            if i-j >= o
                 if D[i-j] + c[i] < D[i]
                    D[i] = D[i-j] + c[i]
                    P[i] = i-j
    x = n
    while x != -1
        path.add(x)
        x = P[x]
    return D[n], reverse(path)
```

10.7 - Turtle on a $2D$ grid

○	3	2	3
1	6	5	7
2	3	1	2
8	2	1	6
3	2	1	×

We have a turtle on the upper-left corner of a $2D$ grid. This turtle want to go at the bottom right corner. She can only go down or right. On each cell there is an amount of salad.

What is the maximum amount of salad the turtle can eat on her path.

The basic idea is the same as the previous problems. We want to maintain an array that contain the maximum for each cell.

Here we are in $2D$. Our array will then also be in $2D$:

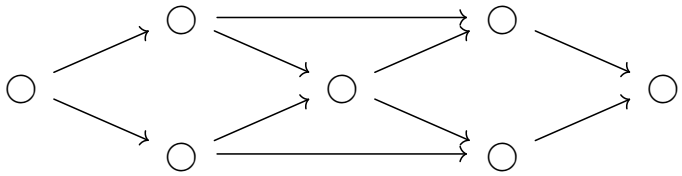
$$D[i,j] \stackrel{\text{def}}{=} \text{max sum to reach cell } (i,j)$$

If we are on the cell (i,j) we either come from:
- $(i-1,j)$
- $(i,j-1)$

We then need to get the maximum between optimal path for $(i,j-1)$ and $(i-1,j)$ in order to get the optimal one for (i,j).

The code is the same but with 2 loops.

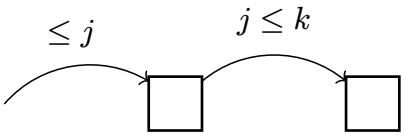
You can actually do the same as this grid problem in any acyclic graph.



We will talk about them during the semester 3.

10.8 - Grasshopper with jump restrictions

Here we stay with the same base rule than before but we add constrain:
- if the j^{th} jump was of length k then $(j+1)^{\text{th}}$ jump must be of length at most k.



There is 2 way to think about this:
- if we arrive on the n^{th} call with a jump of length k:
  - it mean we came from cell $(n-k)$ and the jump to $(n-k)$ was of length $\leq k$

  We have 2 parameters for our problem:
  - landing cell
  - length of the last jump

- lets view our start and end cell as a state. We want to describe our states with at least variable as possible.

  Here we have 2 of them:
  - where do we stand
  - what was our last jump

$$D[n,k] \stackrel{\text{def}}{=} \text{number of path to cell } n \text{ with last jump} \leq k$$

$$D[n,k] = \sum_{j=1}^{k} D[n-j,j]$$

```
def grasshopper(n,k)
    D[0] = 1
    for i = 1..n
        for h = 1..k
            for j = 1..h
                D[i,h] += D[i-j,j]
```

Here we have $\mathrm{O}(n^3)$ but we can improve to $\mathrm{O}(n^2)$.

The same algorithm construction can be made for any optimization problem of minimal cost.

11 - Dynamic Programming - PART II

11.1 - Levenshtein distance

We have two text:

```python
s = 0
for i in range(n):
    s+= a[i]
print(s)
```

```python
s = float('-inf')
for i in range(n):
    if a[i] > s:
        s = a[i]
print(s)
```

The changes between those texts are:
- line 1: $s = 0 \rightarrow s = \text{float}('-\text{inf}')$
- line 3: add the line `if a[i] > s:`
- line 4: change `+=` $\rightarrow$ `=`

We want to be able to look at two files and create the list of operation that we should do to the file A to obtain the file B as an output.

This is not trivial since we do not have any information other than the two raw files.

There are multiple list of changes that produce B from A. The simplest being erase all the file A and put B at its place.

The difficulty here is to get the minimal possible list of changes that produces B from A.

11.1.1 - Levenshtein distance between 2 words

You want to fo from one word to another using the minimal number of operation.

Our authorized operations are:
- change letter
- add letter
- remove letter

Lets look at an example:

$$A = \text{APPLE}$$
$$B = \text{ALPINE}$$

$$\underset{\text{L}}{A\not{P}PLE} \qquad \underset{\text{I}}{ALP\not{L}E} \qquad ALPINE$$

We need 3 operations to get from A to B. This is what we call the Levenshtein distance (or the edit distance).

The set of operation is not necessarily immutable from one problem to an other. We just need to set it when we define it.

We now want to make an algorithm that compute that edit distance.

We have two words:

$$A : \underbrace{\qquad\qquad}_{x} \qquad B : \underbrace{\qquad\qquad}_{y}$$

1. If the last character between A and B is the same ($x = y$), we can let it here and repeat the same problem on A' and B' where $A' = A[:-1]$ and $B' = B[:-1]$.

$$A : \underbrace{\qquad\qquad}$$
$$B : \underbrace{\qquad\qquad}$$

   We do the same as in the first dynamic programming lecture: repeat the same problem on a reduced input.

2. If ($x \neq y$):
   - change (x, y) so that both letter are the same and then we are in 1. situation.

$$A : \underbrace{\qquad\qquad}y$$
$$B : \underbrace{\qquad\qquad}y$$

   - remove x, then we repeat our problem between A' and B.

$$A : \underbrace{\qquad\qquad}$$
$$B : \underbrace{\qquad\qquad}y$$

   - add y at the end of A then repeat our problem between A and B'.

$$A : \underbrace{\qquad\qquad}xy$$
$$B : \underbrace{\qquad\qquad}y$$

   Those are the option we have if the end letter are not the same. Each time, we reduce our problem to a smaller one of the same type, and each time the smaller problems input are prefix of A and B.

Lets define our dynamic programming state:

$$D[i, j] \stackrel{\text{def}}{=} \text{min munber of operation to produce } B[0..j-1] \text{ from } A[0..i-1]$$

Lets define our transition value for each transition define before:

- same ending

$$D[i, j] = D[i-1, j-1]$$

- change x to y

$$D[i, j] = 1 + D[i-1, j-1]$$

- remove x

$$D[i, j] = 1 + D[i-1, j]$$

- add y

$$D[i, j] = 1 + D[i, j-1]$$

To get $D[i, j]$, if $x = y$ then we take this solution, else we take the minimum between every other possibilities.

```
def levenshtein_dist(A,B)
    for i = 0..|A|
        for j = 0..|B|
            if i = 0 or j = 0
                D[i,j] = i+j
                continue
            if A[i-1] = B[j-1]
                D[i,j] = D[i-1,j-1]
            else
                D[i,j] = 1 + min(D[i-1,j-1], D[i-1,j], D[i,j-1])
    return D[|A|,|B|]
```

Lets try to run this algorithm on our previous example:

$$A = \text{APPLE}$$
$$B = \text{ALPINE}$$

D	0	A:1	L:2	P:3	I:4	N:5	E:6
0	0	1	2	3	4	5	6
A:1	1	0	1	2	3	4	5
P:2	2	1	1	1	2	3	4
P:3	3	2	2	1	2	3	4
L:4	4	3	2	2	2	3	4
E:5	5	4	3	3	3	3	3

Now, we want to not just get the minimum number of change but also get the change-log of what operation need to be done.

The change-log can be interpreted as the path from the beginning to the end. We need to recover it.

To do that when we make our if, we save what transition we used.

Note: The levensthein distance is symmetrical, the distance to get A from B or B from A is the same.

11.1.2 - Levenshtein distance between 2 files

Now we can go back to our base problem, 2 files instead of 2 words.

This is the same thing: a letter become a line of our file.

The algorithm we made as a complexity of $O(|A| \cdot |B|)$. It is sufficient for practical use:
- if both of our file are not that long

If each file is really big: $|A| \simeq |B| \simeq 10^6$, our algorithm will not be good enough.

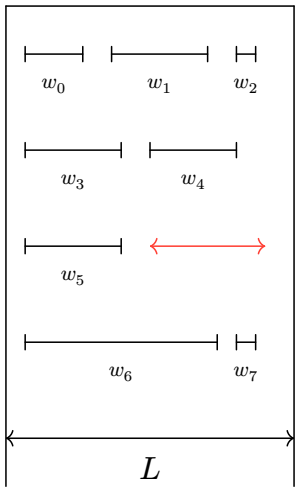
In a real world calculation, in git diff for example, we use additional heuristics to decrease complexity:
- If we have 2 big files with not to many difference, we arrange just to build the table around the optimal path and not the whole things.

- Sometimes it will not give you real optimal solution, we made a trade of between optimality and complexity.
- The algorithm work "less" correctly if the files are really different. But usually, we use diff on file that have similarity.

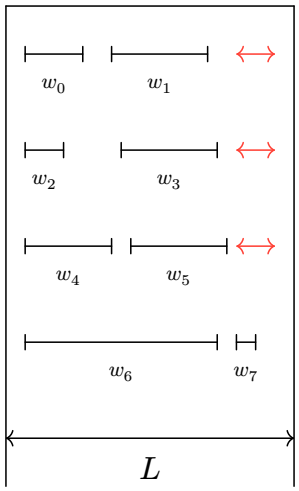
11.2 - Word alignment and text justification

$$\text{Text} : \underbrace{l_0 \quad l_{m-1}}_{w_0} \quad \underbrace{l_0 \quad l_{m'-1}}_{w_1} \quad \underbrace{l_0 \quad l_{m''-1}}_{w_2} \quad \dots \quad \underbrace{l_0 \quad l_{m-1}}_{w_{n-1}}$$

We know the length of each words. We want to print them on a sheet of width L.



We have a big gap after word 5. What we want is to justify the text to not have those kind of big gaps.



First we need to define a measurement to know if the text is good. For example, if we call our end gap x, we have: $\text{badness} = x^3$.

Our badness function does not change the code this function need to be found by experimenting. We want to minimize the total badness ($\min(\sum \text{badness})$) of our text.

What we want is to take our word array and split it in segment representing our lines.

$$\underbrace{\underbrace{l_0 \quad l_{m-1}}_{w_0} \quad \underbrace{l_0 \quad l_{m'-1}}_{w_1}}_{\text{line 1}} \quad \underbrace{\underbrace{l_0 \quad l_{m''-1}}_{w_2} \quad \dots}_{\text{line 2}} \quad \underbrace{\dots \quad \underbrace{l_0 \quad l_{m-1}}_{w_{n-1}}}_{\text{line 3}}$$

Lets define the function computing the badness of a segment.

If we put words from index l to $r-1$ in a segment then:

$$\text{bad}(l,r) = \left(L - \sum_{i=l}^{r-1} w_i\right)^3$$

If we want to put the gap not in the end of the line but to have all the gap the same between each words.

We have:

$$\text{bad}(l,r) = \left(\frac{L - \sum_{i=l}^{r-1} w_i}{r-l-1}\right)^3 \cdot (r-l-1)$$

Lets solve our problem:

$$\text{word array} : \left[w_0, w_1, w_2, w_3, ..., \underbrace{w_{n-2}, w_{n-1}}_{\text{last line}}\right]$$

We put those words in the last line and then, we have the same problem without the last segment. Which is a prefix of our words array.

$$D[i] \overset{\text{def}}{=} \min \sum \text{bad for } w[0..i-1]$$

```
def word_justification(w)
    D[0] = 0
    for i = 1..n
        D[i] = +infinity
        s = 0
        for k = 1..i
            s += w[i-k]
            if s >= L
                break
            D[i] = min(D[i], D[i-k]+(L-s)**3)
    return D[n]
```

We have a complexity of $O(n^2)$.

Here we only compute the total badness. In an actual problem, we want to have the actual answer. The path. We do the same as every-time, go from the end and retrace our step back.

11.3 - Run length encoding (RLE)

Basic RLE: AAAB $\rightarrow$ 3AB

RLE we are going to solve:

$$\text{AAABAAAB} \rightarrow 2(\text{3AB})$$

Given a string S, what is the minimum size of encoded S.

11.3.1 - Encoding the string S

Lets take a look at the first character of the string S:
1. either c_1 is a letter alone in our encoded string:

$$S \to c_1(S[1..n-1])$$

1. it is part of a repeating group:

$$S : \underbrace{\underline{l_0 ... l_{x-1}} \quad ... \quad \underline{l_{(K-1)x} ... l_{Kx-1}}}_{K \text{ times the same group of length } x} \quad ... \quad l_{n-1} \to K(S[0..x-1])(S[Kx..n-1])$$

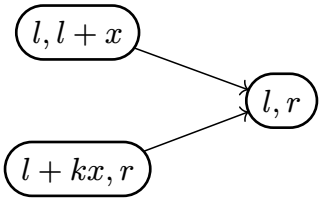
In each case, we reduce our problem to a subset of the same problem:
- In the first one we reduce to $S[1..n-1]$
- In the second one, we reduce to $S[0..x-1]$ and $S[Kx..n-1]$

Our state is:

$$D[l, r] \stackrel{\text{def}}{=} \text{minimum emcodding for string } S[l..r-1]$$

Our transition value are:
1. $1 + D[l+1, r]$
2. $\text{len}(K) + 2 + D[l, l+x] + D[l+Kx, r]$

We use the 2 previously calculated state to get the new one:



We see there that on the contrary of our previous algorithms, we need to compute our array from the bottom to up.

And we need to assess that the K repetition of the group string are equal (use a string algorithm that we will see in the Semester 3).

Again here we compute the minimum value, we need to take the path from the last element to get our encoded string.

12 - Knapsack

Knapsack problem is a fundamental problem in computer science. You can encounter it in many situation and real life problem.

It is then an important thing to know how to recognize those problems and know what are the options to solve it.

12.1 - Problem statement

You have: - n items

Each items has a weight and a cost: $\forall i \in [\![0, n-1]\!] \mid I_i \to (w_i, c_i)$

You also have a knapsack of capacity S.

Your objective is to put items in the knapsack in order to maximize its cost while not exciding its capacity:

$$\sum_i w_i \leq S$$

$$\sum_i c_i \text{ is maximized}$$

This problem is NP-complete (we will discuss and defined NP-complete more in the last lecture of semester 2) for now, lets just say that we do not know if a solution exist for this problem.

If there is no additional constrain, there is a good chance that this problem is not solvable.

But, with additional constrain it is possible.

12.2 - No cost

$$\forall i, c_i = w_i$$

Then we want:

$$\sum_i w_i \leq S$$

$$\sum_i w_i \text{ is maximized}$$

This is still NP-complete.

But what we can do is:

$$D[i, j] \stackrel{\text{def}}{=} \text{it is possible to have subset of } I[0..i-1] \text{ with} \sum w = j$$

1. set contain $i-1$: $D[i, j] = D[i-1, j-w_{i-1}]$
2. set does not contain $i-1$: $D[i, j] = D[i-1, j]$

At the end:

$$D[i, j] = D[i-1, j-w_{i-1}] \vee D[i-1, j]$$

Example:

$$w : 5, 3, 2, 3$$
$$S = 9$$

D	0	1	2	3	4	5	6	7	8	9
0	+	-	-	-	-	-	-	-	-	-
1	+	-	-	-	-	+	-	-	-	-
2	+	-	-	+	-	+	-	-	+	-
3	+	-	+	+	-	+	-	+	+	-
4	+	-	+	+	-	+	+	+	+	-

$$+ = \text{true} \ / \ - = \text{false}$$

To maximize the weight, we take the rightmost element in the last row.

To get what we put in our maximized set, we retrace our steps.

12.3 - Cost, all integers

$$\forall i, w_i \in \mathbb{N}, S \in \mathbb{N}, S \lesssim 10^6$$

To solve this problem, we use the same technique as for the previous element. But, instead of true or false, we put our cost in the cell.

$$D[i, j] \stackrel{\text{def}}{=} \max \sum_i c_i \text{ for subset of } [0..i-1] \text{ with } \sum_i w_i \leq j$$

We use the same approach:
1. set contain $i - 1$

$$D[i - 1, j - w_{i-1}] + c_{i-1}$$

2. set does not contain $i - 1$

$$D[i - 1, j]$$

$$D[i, j] = \max(D[i - 1, j - w_{i-1}] + c_{i-1}, D[i - 1, j])$$

Example:

$$w : 5, 3, 2, 3$$
$$c : 3, 2, 5, 3$$
$$S = 9$$

D	0	1	2	3	4	5	6	7	8	9
0	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
1	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	3	$-\infty$	$-\infty$	$-\infty$	$-\infty$
2	0	$-\infty$	$-\infty$	2	$-\infty$	3	$-\infty$	$-\infty$	5	$-\infty$
3	0	$-\infty$	5	2	$-\infty$	7	$-\infty$	8	5	$-\infty$
4	0	$-\infty$	5	3	$-\infty$	8	5	8	10	$-\infty$

Our answer is $\max_j D[n, j]$

To get all our chosen elements, we retrace our steps.

Our algorithm complexity is $O(n \cdot S)$.

In those case, we can use the fact that since $\sum w \leq S$ and S is small, $\sum w$ is also small. And since we only have integers there are not to many case to handle. This allow us to have array of the size. The fact that S is small is really important have since S is part of our complexity.

12.4 - Small integers costs

$$\sum_i c_i \to \text{small}$$

$$c_i - \text{integers}$$

This is part of the question of the home tasks. The idea is basically the same but instead of having the weight as the state of our dynamic programming we have the costs.

12.5 - n is very small

$$2^n - \text{small enough}$$

$$n \simeq 25$$

The idea here is since n is small, the solution will be the naive one. We iterate over all the possible subsets.

To achieve that, we want to enumerate all the subsets.

We want to make a bijection between:

$$\text{subset of } [0..n-1] \leftrightarrow [\![0..2^n - 1]\!]$$

We are going to use the bit in our number to enumerate our subsets.

$$x = \{1, 2, 4\} \leftrightarrow x = 0010110_2 = 22_{10}$$

Lets try to make all the common subsets operations with this integers.

12.5.1 - Singleton

$$x = \{i\} \leftrightarrow x = 1 \ll i = 2^i$$

12.5.2 - Union

The i^{th} bit of our number should be equal to one i at least one of the i^{th} bit of our two sets we want to make the vision of is set at once.

This is the bitwise or:

$$x \cup y \leftrightarrow x \mid y$$

12.5.3 - Intersection

$$x \cap y \leftrightarrow x \& y$$

Elements that in x and y. The i^{th} bit of our resulting set is set to 1 if both i^{th} bit are set to 1 for x and y. This is the bitwise and.

12.5.4 - Difference

We want all the elements of x that are not in y.

In the general case: $x \setminus y \leftrightarrow x\&(\sim y)$

If $\forall y \subset x$: $x \setminus y \leftrightarrow x - y$ or $x \hat{} y$

12.5.5 - Appartenance of an element to the subset

$$i \in x \leftrightarrow x\&(1 \ll i) > 0 \text{ or } (x \gg i)\&1 > 0$$

From now, we iterate over all subsets is just iterate from 0 to $n-1$.

```
def knapsack(w,c,S)
    ans = 0
    for x = 0..2**n - 1
        sw = 0
        sc = 0
        for i = 0..n-1
            if x & (1<<i) > 0
                sc += c[i]
                sw += w[i]
            if sw <= S
                ans = max(sc,ans)
    return ans
```

Our complexity here is $O(2^n \cdot n)$. We want to optimize that complexity (the 2^n part especially).

12.5.6 - Meet in the middle optimization

$$\underbrace{\underbrace{0\ 0\ 0\ 0}_{\frac{n}{2}}\ \ldots\ \underbrace{0\ 0\ 0\ 0}_{\frac{n}{2}}}_{n} : X; Y$$

We want to iterate over x and y separately.

$$(1)\begin{cases} \sum_{i\in x} w_i + \sum_{i\in y} w_i \leq S \\ \sum_{i\in x} c_i + \sum_{i\in y} c_i \rightarrow \text{maximized} \end{cases}$$

We do not want to iterate over all x and y or our optimization will be worth nothing.

We want the optimal y for each x.

Lets see, if we fix the value of x, what constrain do we have on y from (1).

$$(2)\begin{cases} \sum_{i\in y} w_i \leq S - \sum_{i\in x} w_i \\ \sum_{i\in y} c_i \rightarrow \text{maximized} \end{cases}$$

Our new objective is to solve (2) for y without going through every possible y. To achieve that, we make a sorted array of y, sorted by $\sum_{i\in y} w_i$. We can then separate this array into two part:

$$[Y_0, ..., Y_i, Y_{i+1}, ..., Y_{n-1}] \rightarrow \forall j \in [\![0, i]\!], \sum_{k\in Y_j} w_k \leq S - \sum_{k\in X} w_k$$

We can find this prefix using binary search.

Then we take the max value of $\sum_{i \in y} c_i$ from this prefix.

With this optimization, our complexity is now $\mathrm{O}\left(2^{\frac{n}{2}} \cdot n\right)$.

12.6 - Multi-knapsack

$$n - \text{items}$$
$$\text{weights} : w_i$$
$$S$$

We want to minimize the number of knapsacks to all items.

Example:

$$w = 2, 5, 3, 1, 4, 8$$
$$S = 9$$

$$K_1 = \{5, 2\}$$
$$K_2 = \{8, 1\}$$
$$K_3 = \{3, 4\}$$

$$\forall i, \sum_{w \in K_i} w \leq S$$

For those kind of problem, even if integers are small, it is hard to solve because we want to remember all knapsack value not only 1.

We define our algorithm state by:

$$D[X] \stackrel{\text{def}}{=} \text{min number ok knapsacks to put all elements in } X$$

$$D[X] = \min_{\substack{Y \subset X \\ \sum_{w \in Y} w \leq S}} (1 + D[X \setminus Y])$$

```
def multi_knapsack(w,S)
    D[0] = 0
    for X = 1..(2**n - 1)
        for Y = 1..(2**n - 1)
            if Y subset of X and sum(w) of Y <= S // can precompute the sum
                D[X] = min(D[X], D[X-Y]+1)
```

This algorithm runs in $\mathrm{O}(n^4)$. Which is bad.

Lets optimize our algorithm by iterating only over the Y that are a subset of X.

```
def multi_knapsack(w,S)
    D[0] = 0
    for X = 1..(2**n - 1)
        for Y subset of X
            if  sum(w) of Y <= S
                D[X] = min(D[X], D[X-Y]+1)
```

Here we go through all couple (X, Y) such as $Y \subset X$

We only have 3 possible cases:
- the element is in neither X or Y
- the element is in X but not Y
- the element is in X and Y

We then have only $O(n^3)$ possibilities to go through.

With this optimization, our complexity goes from $O(n^4)$ to $O(n^3)$.

The idea to iterate over all the subset of X is to begin with $X = Y$ and then by decreasing Y from X to 0, we get all the its subset.

$$\begin{aligned} X &= 10010110 \\ Y \rightarrow\ & 10010110 \\ & 10010100 \\ & 10010010 \\ & 10010000 \\ & 10000110 \\ & \cdots \\ & 00000000 \end{aligned}$$

To go from one iteration of Y to another:

$$\begin{aligned} X &: \underbrace{\text{———}1011010}_{} \\ Y &: \underbrace{\text{———}1011010}_{} \\ Y' &: \underbrace{\text{———}0011010}_{} \end{aligned}$$

$$Y' = (Y - 1)X$$

Iterating with this formula for Y will give all the subset of X.

```
def multi_knapsack(w,S)
    D[0] = 0
    for X = 1..(2**n - 1)
        for (Y = X; Y > 0; Y = (Y - 1) & X)
            if sum(w) of Y <= S
                D[X] =  min(D[X], D[X-Y]+1)
```

This is quite a universal technique when we want to split a set into subsets maintaining a property.

But in this specific problem, we can do even better. Our number of state is $O(n^2)$ but our overall complexity is $O(n^3)$. This append because we have too many transition between our states.

Instead of adding elements in group, we want to add them 1 by 1.

We have A knapsacks and only 1 of them is not full. We add element one by one in it, we keep track of its weight B. When this knapsack is full, we create a new empty one.

Our state is:

$$D[X] \stackrel{\text{def}}{=} (A, B)$$

- Here, we want to optimize both A and B. Usually this is not possible. It is here because we want to minimize the pair. We optimize A and then B.

  Compare (A_1, B_1) and (A_2, B_2):
  - $A_1 \leq A_2, B_1 \leq B_2 \Rightarrow (A_1, B_1) \leq (A_2, B_2)$
  - $A_1 < A_2, B_1 > B_2 \Rightarrow (A_1, B_1) \leq (A_2, B_2)$
    - Because $(A_1, B_1) \leq (A_1 + 1, 0) \leq (A_2, B_2)$

  In short, we have two parameters to optimize but they do not have the same priority.

In our transition we go from set $X \setminus \{i\} \rightarrow X$:

- If $(B + w_i) \leq S$

  $D[X] = (A, B + w_i)$
- Else $(B + w_i) > S$

  $D[X] = (A + 1, w_i)$

Our number of transition from one state to another is now 2 instead of 2^n.

With this technique, we came to a complexity of $\mathrm{O}(2^n \cdot n)$.

13 - Dynamic programming on subsets, on profiles

13.1 - Dynamic programming on subset

This is when you do dynamic programming and the parameter of your program is a subset.

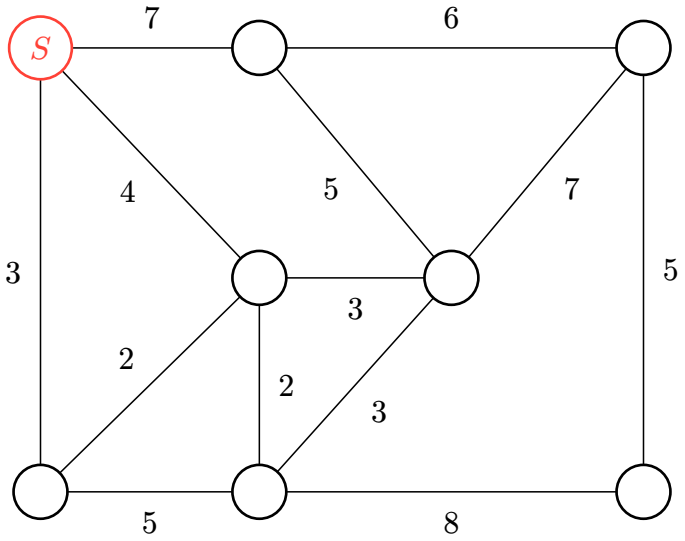
$$D[X] - X \text{ is a subset}$$

We already talk a little about that on lecture 12.

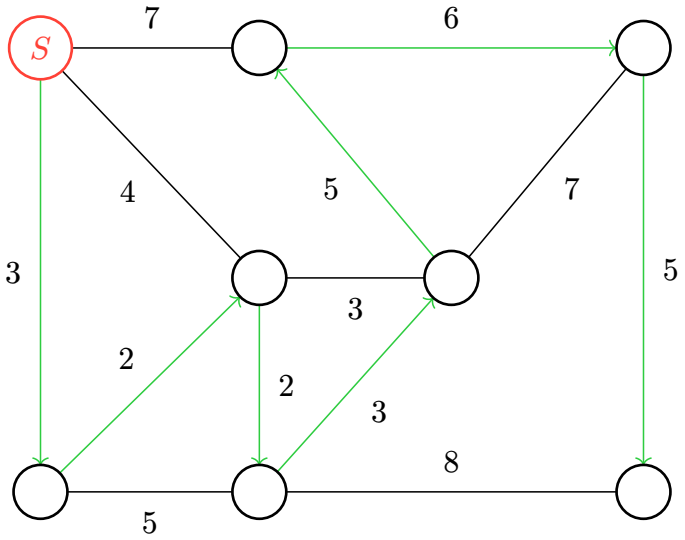
13.1.1 - Traveling salesman problem

You have a graph, each vertices represent a town, each edge represent a road between two towns.

Example:



Goal: you start from S and you need to visit all the cities while minimizing your traveling distance.



Here our traveling distance is 26.

This is a common graph problem. And it is know to be NP-Complete as the knapsack problem (again, we will talk about complexity classes in S02L15). But for now, we just keep in mind that it mean we do nt have an efficient solution.

But, we can make a working algorithm with complexity of $O(2^n)$ where n is the number of vertices (which is much better than the naive algorithm working in $O(n!)$) to find a solution. And we remember from the precedent lecture that algorithm with complexity of $O(2^n)$ are usable with small n.

When we are running our program, we construct our solution going from 1 vertices to an other. When we arrive on a vertices, the state of our algorithm is the following:
- we have visited a subset X of G vertices (where G is the subset of all the graph vertices).
- the path we took to do that is of length m.
- our path end on vertices V

From that we have:

$$D[X, V] \stackrel{\text{def}}{=} \text{the minimal cost (shortest path) to visit subset } X \text{ while ending on } V$$

Now, we need to define our transition between 2 states. To do that, you can either find all the previous state or all the next states. Lets see forward in this case and find all our next states.

When we are on V, we can use an edge to go on vertices U.

$$(X, V) \to (X \cup \{U\}, U)$$

We can do this for each U that we can access from V that are not already in X.

```
def traveling_salesman(S,graph)
    D[1<<S,S] = 0
    for X = 0..(2**n - 1)
        for V = 0..n-1
            for V -> U: out(V)
                if X & (1<<U) = 0 // U not in X
                    Y = X + (1<<U)
                    D[Y,U] = min(D[Y,U],D[X,V] + len(V,U))
    ans = 0
    for V = 0..n-1
        ans = min(ans, D[(1<<n) - 1, V])
```

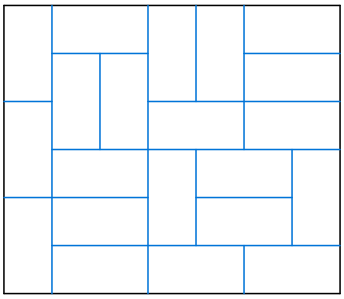
This is an example of problem that you can solve using dynamic programming on a subset. These kind of problems are represented by state of dynamic programming that are dependent of a subset of objects (here our visited vertices).

13.2 - Dynamic programming on profiles (Domino tilling)

We have a rectangle of size $n \times m$. We fill this rectangle with small rectangles of size 1×2.

The objective is to count the number of ways to fill this rectangle with 1×2 rectangle.

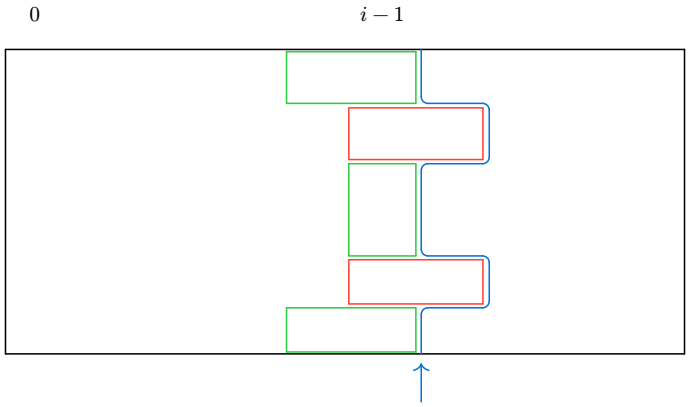
Example with $n = 6, m = 7$:



Note: We can find this solution in polynomial time using algebra and graphs but this is not the objective here and much more complicated. See further reading for article on Domino Tilling.

Here, what we want to do here is to fill the rectangle column by column from left to right.

13.2.1 - Full profile approach



At this state, we have filled column 0 to $i-1$. To fill the column i, I want to know what tile is straddling column $i-1$ and i*.

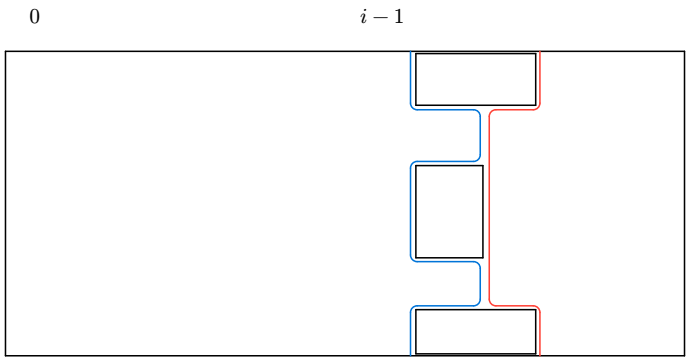
We want to remember the profile of the blue line. To do that, we use a bit sequence*:

- 0 mean that the tile is not straddling
- 1 mean that the tile is straddling

We encode those bits as the number p and we can define our state as:

$$D[i, p] \stackrel{\text{def}}{=} \text{number of ways to fill the first } i \text{ columns with the profile } p$$

Now that we have our dynamic programming state, lets find our transitions.



$p = 010010, q = 100001$

$$(i, p) \to (i + 1, q)$$

To check if we can go from profile p to q:

- When we have 2 profiles, there is a space between the 2 profiles. If this space can be filled by our tiles, then we can go from profile p to q.

Our first state is 0 column filled and then an empty profile $\to D[0, 0] = 1$.

```
def solution(n,m)
    D[0,0] = 1
    for i = 0..m-1
        for p = 0..(2**n - 1)
            for q = 0..(2**n - 1)
                if comp(p,q)
                    D[i+1,q] += D[i,p]
    return D[m,0]
```

When $i = m - 1$ we do not need to iterate over all q just 0.

Check that we can go from p to q:

- if $p = 0, q = 1$ you have to put an horizontal tile
- if $p = 0, q = 0$ you have to put a vertical tile

If there is a solution, it is unique. But in some case we could not be able to put a tile, for example when we want to put a vertical tile, we need two consecutive bits to have the same configuration. If we do not have that we can go from p to q.

To achieve the `comp` function, we could iterate over the bits. But lets use bitwise operation to achieve that in constant time. Lets enumerate our possible state for the bits of p and q ($b_{i,p} \stackrel{\text{def}}{=}$ the i^{th} bit of profile p):

- $b_{i,p} = b_{i,q} = 1$, we cannot go from p to q.
  - to ensure that we do not have this state, we want $p \& q \neq 0$
- $b_{i,p} = 1, b_{i,q} = 0$, we can go from p to q and we do not need to put a tile.
- $b_{i,p} = 0, b_{i,q} = 1$, we can go from p to q and we need to put an horizontal tile.
- $b_{i,p} = 0, b_{i,q} = 0$, we can go from p to q and we need to put a vertical tile.

  But to do that, we need two consecutive row with this state. Which mean that we need:

- $\big(b_{i,p}=0, b_{i,q}=0\big) \wedge \big(b_{i+1,p}=0, b_{i+1,q}=0\big)$
- or $\big(b_{i,p}=0, b_{i,q}=0\big) \wedge \big(b_{i-1,p}=0, b_{i-1,q}=0\big)$

to be able to put a vertical tile.

We compute: $x = (\sim p \& \sim q)$

$$x = 00111101100110$$

Where a 1 in x mean that we have $b_{i,p}=0, b_{i,q}=0$. Now, we need to check if all our one can be splited by pair. The easiest way to do this is to divide x by 3. Because $11_2 = 3_{10}$.

$$x = 00111101100110$$
$$\frac{x}{3} = 00010100100010$$

So, if $x \bmod 3 \neq 0$, it mean that we cannot divide all our 1 into pair.

Then, we want to check that $y = \frac{x}{3}$ do not have two consecutive 1. To do that we check that $y \& (y \ll 1) \neq 0$.

We have our `comp` function working in $\mathrm{O}(1)$:

```
def comp(p,q)
    if (p & q != 0) return false
    x = (~p & ~q)
    if (x%3 != 0) return false
    y = x/3
    if (y & (y << 1) != 0) return false
    return true
```

Now lets check our overall complexity for our solution. It is the product of our for-loop since $T(\text{comp}) = \mathrm{O}(1)$:

$$T(\text{solution}) = \mathrm{O}(m \cdot 4^n)$$

Using a technique we previously used we can decrease our complexity, instead of iterating over all the possible profile and check if they are compatible, we can iterate only over the compatible ones. This would decrease our complexity to less than $\mathrm{O}(m \cdot 3^n)$ (we get rid of profile containing $b_{i,p} = b_{i,q} = 1$ and containing $b_{i,p} = 0, b_{i,q} = 0$ not in pair).

13.2.2 - Broken profile approach

The objective here is to reduce our time and code complexity.

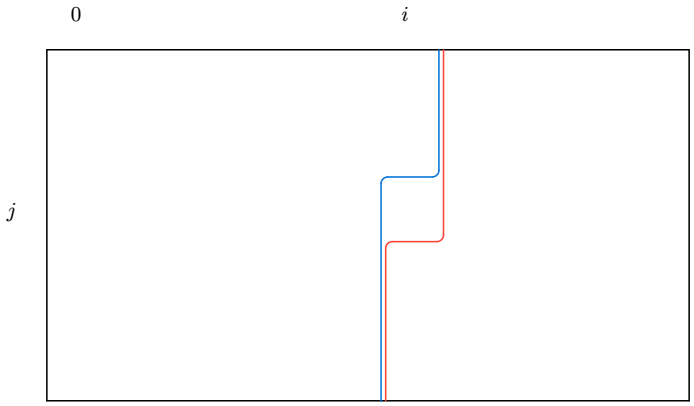
We face a problem that we already encountered once, we have to many transition between 2 states and this makes our complexity explode. We need to redefine our states and there transitions. Until now, when we were going from $i-1$ to i we were adding n square to our column. From now we want to add only 1 square to our column from 1 state to another.

We have filled our first $i-1$ columns and the first j rows of the i^{th} column.

Again, we keep track of the straddling tiles:

$$D[i,j,p] \stackrel{\text{def}}{=} \text{number of way to fill the first } i \text{ columns}$$
$$+j \text{ squares with profile } p$$

Our state looks a bit complicated but we are going to have much simpler transition.



$$(i,j) \to (i,j+1)$$

There are 2 possibilities for (i,j):

1. The square we want to fill is already filled (by a straddling tile).

   It mean that $b_{j,p} = 1$, it become 0 in the profile q ($b_{j,q} = 0$). The next state is:

   $$b_{j,p} = 1 \Rightarrow (i,j,p) \to (i,j+1,q) \text{ where } b_{j,q} = 1$$

2. It is empty, then 2 choices:
   - put an horizontal tile:

     $$b_{j,p} = 0 \Rightarrow (i,j,p) \to (i,j+1,q) \text{ where } b_{j,q} = 1$$

   - put a vertical tile. To do that, we need $b_{j+1,p} = 0$

     $$b_{j,p} = 0 \wedge b_{j+1,p} = 0 \Rightarrow (i,j,p) \to (i,j+1,q) \text{ where } b_{j,q} = 0 \wedge b_{j+1,q} = 1$$

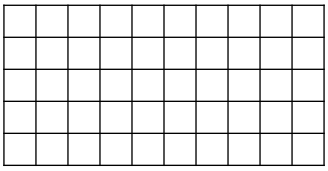
```
def solution(n,m)
    D[0,0,0] = 1
    for i = 0..m-1
        for j = 0..n-1
            for p = 0..(2**n - 1)
                if p & (1 << j) != 0
                    q = p - (1 << j)
                    D[i,j+1,q] += D[i,j,p]
                else
                    q = p + (1 << j)
                    D[i,j+1,q] += D[i,j,p]
                    if (j < n-1) && (p & (1 << (j+1)) == 0)
                        q = p + (1 << (j+1))
                        D[i,j+1,q] += D[i,j,p]
        for p = 0..(2**n - 1)
            D[i,n,p] = D[i+1,n,p] // Transition from i -> i+1
    return D[m,0,0]
```

Our overall complexity is $O(m \cdot 2^n)$

These techniques are useful when you build a solution layer by layer only using the previous layer. Mostly in $2D$ grid problems that we fill row by row or column by column.

Here we counted the number of way to fill the grid but we can imagine optimization problems too, where for example each square as a cost and the manner of filling it cost less or more.

13.3 - A second problem on profile (Grid coloring)



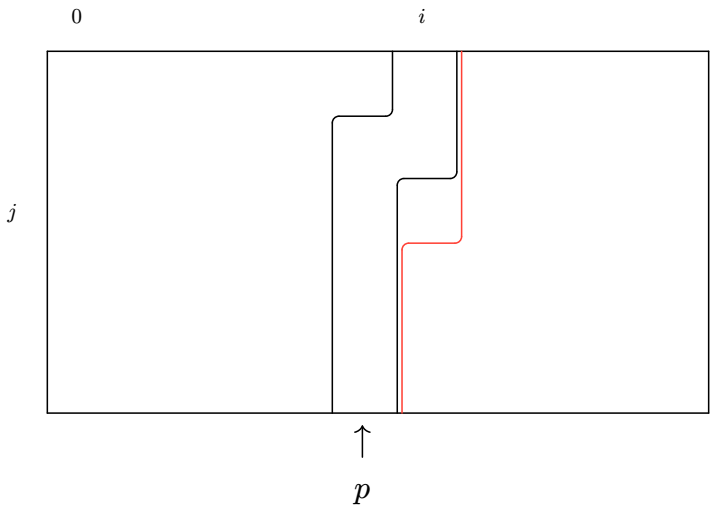
We have an $n \times m$ rectangle. We color each 1×1 square in black or white in such way that neither of those pattern exist:





Here, we can see that is we are coloring column by column, we only meed to know the state of the last colored column to fill the new one.

Using the broken profile approach, we do:



- If each adjacent square are the same color, then we have no other choice than coloring our current square with the other color.
- Else, we can do both.

The algorithm is basically the same but p is of length $n + 1$ because we need to keep track of the adjacent square of position $(i - 1, j - 1)$.

14 - Hash Tables

Hash tables are really important data structures. They are randomized and can cause problem in some situation. We will see when to use them and when not to.

There are 2 major data structures that are based on hash tables:

- `set`: as the name suggest, this is a set of objects. Its base operations are:
  - `add(x)`
  - `remove(x)`
  - `contain(x)`
- `map`: these are object mapping a key to a value, we access value with the key. Its base operations are:
  - `put(k,v)`
  - `get(k)`

Those are widely used primitive data structures.

14.1 - Simple implementations

14.1.1 - Small range integers keys

$$\text{key} \in [\![0, m-1]\!]$$
$$m - \text{small}$$

We use the key as an index of an array:

```
def put(k,v)
    a[k] = v
```

```
def get(k)
    return a[k]
```

This is the most simple case to implement `maps`.

14.1.2 - Big range integers key

$$\text{key} \in [\![0, n-1]\!]$$
$$n - \text{big}$$

What wee can do is to make a function that take a big integer and return a small one. Then we use this result as a key in our array.

$$h \left| \begin{array}{l} [\![0, n-1]\!] \to [\![0, m-1]\!] \\ k \mapsto k\%m \end{array} \right.$$

What we can do now is:

```
def put(k,v)
    a[h(k)] = v
```

```
def get(k)
    return a[h(k)]
```

Example:

$$a : [\ |\ \ |\ \ |\ \ |\ \ ]$$
$$0\ \ 1\ \ 2\ \ 3\ \ 4$$

`put(37,3)` $\rightarrow$ `h(37) = 2`

$$a : [\ |\ \ |\ 3\ |\ \ |\ \ ]$$
$$0\ \ 1\ \ 2\ \ 3\ \ 4$$

`get(37)` $\rightarrow$ `h(37) = 2` $\rightarrow$ `3`

Since the function h is the same for one hash table for both function put and get, we end on the same index. This is the basic idea of hashing.

But our current implementation suffer from one big problem:
- Collision: when we have: $x, y \in [\![0, n-1]\!], x \neq y \wedge h(x) = h(y)$

For example with our previous example we have: $37 \neq 52 \wedge h(37) = h(52) = 2$

So we can get wrong result from collision.

To patch that problem, instead of putting only the value in our array, we put the couple $(\text{key}, \text{value})$. So that when we will have collision we can check for key equality.

There is still one thing to do change, now we can ensure that we will not return a bad answer but we can store only 1 couple for a given index. To patch that, we store a list of couple instead so we can save multiple value a the same index.

Example:

$$a : [\ |\ \ |\ (37,3);(52,5)\ |\ \ |\ \ ]$$
$$0\ \ 1\ \ \ \ \ \ 2\ \ \ \ \ \ 3\ \ 4$$

```
def put(k,v)
    a[h(k)].add((k,v))
```

This function is fine if our k is not already in the hash table. But it does not really work if it it the case, because we want to replace the value not ad it again.

```
def get(k)
    for (x,y) : a[h(k)]
        if x = k
            return y
    return null
```

Both of our function have a worst case linear complexity.

Since our function h is really simple, the data is not really randomized and we can end in a situation where most of our value have the same index.

If we want to avoid collision, we need to increase our array size. Indeed, we have n elements so the number of pair is of the order of n^2. Each pair have the same probability to be putted in our hash table. Since our hash table array as a length of m, if we want the length of our

collision list to be of $O(1)$ length, we want the probability of choosing a cell to be the same as the one of a pair. The probability to chose a cell at random is $\frac{1}{m}$. Which mean that we need

$$\frac{1}{m} = \frac{1}{n^2}$$

So m need to be at least of the order of n^2 to avoid collision.

This is a problem because it forces us to make really big array to avoid collision for big value of n. So we cannot use this function is n is not small.

14.2 - Random hash function

What we then want is to use a random function from all $h \mid [\![0, n-1]\!] \to [\![0, m-1]\!]$. Which gives us m^n possibilities.

If we take a random function, what is the expected complexity for get?

$$\begin{aligned}\mathbb{E}(T(\text{get})) &= \mathbb{E}(\forall k, \text{number of } x \neq k : h(k) = h(x)) \\ &= \frac{n}{m} \because k \neq x \Rightarrow \mathbb{P}(h(k) = h(x)) = \frac{1}{m}\end{aligned}$$

So if $m \simeq n \to T(\text{get}) = O(1)$

But we need to chose one of m^n function, this is too big of a number to be exploitable. So, in practice, you do not have a completely random hash function. What we do is take a set of good functions from our possible ones. And we pick one of this set.

Our function set is the following, we call it H.

$$H \stackrel{\text{def}}{=} \left\{ h_{p,A} \middle| \begin{matrix} \mathbb{N} \to [\![0 \\ m-1]\!] \\ k \mapsto ((k \cdot A)\%p)\%m \end{matrix} : p \text{ big random prime number}, A \in [\![0, p-1]\!] \right\}$$

To get a function, we just generate p and A. Lets find if our time complexity is still $O(1)$ with H as our function set.

In the previous proof, the property that was ensuring that $T(\text{get}) = O(1)$ is:

$$\mathbb{P}(h(x) \underset{x \neq k}{=} h(k)) = \frac{1}{m}$$

If we prove that getting a function from H maintain this property we will have proved that $T(\text{get}) = O(1)$.

$$\begin{aligned} & h(x) = h(y) \\ \Rightarrow & ((x \cdot A)\%p)\%m = ((y \cdot A)\%p)\%m \\ \Rightarrow & (((x-y) \cdot A)\%p)\%m = 0 \\ \Rightarrow & (((x-y) \cdot A)\%p)\%m = k \cdot m \end{aligned}$$

Where $k \in [\![0, p-1]\!]$, we then have $\simeq \frac{p}{m}$ possibilities for $k \cdot m$.

Since p is prime and $x \neq y$:

$$\forall k \in [\![0, p-1]\!], \exists! A \in [\![0, p-1]\!], (x-y) \cdot A \equiv k \cdot m [p]$$

The probability that we have picked such A is the same as $\mathbb{P}(h(x) = h(k)) = \frac{1}{m}$.
$x \neq k$

And the probability that we have picked such A is also:

$$\underbrace{\frac{p}{m}}_{\substack{\text{number} \\ \text{of } k \cdot m}} \cdot \underbrace{\frac{1}{p}}_{\substack{\text{number} \\ \text{of } \neq A}} = \frac{1}{m}$$

So, we have proved that $\mathbb{P}(h(x) = h(k)) = \frac{1}{m}$.
$x \neq k$

Then we still have $T(\text{get}) = \mathrm{O}(1)$ with our new function set.

Attention, working with hash function is a dangerous thing, indeed in our case we only have one property to satisfy to have $T(\text{get}) = \mathrm{O}(1)$. But be careful when working with other data structures, because the property to satisfy could be different, more complex and multiple.

14.3 - Open addressing

Open addressing is a different solution to the collision problem.

The idea is the following, instead of treating the collision by making a list of couple at the same index:

$$\begin{array}{c} a : [\ \ |\ \ |\ (37,3); (52,5)\ |\ \ |\ \ ] \\ \quad\ 0\ \ 1 \qquad\quad 2 \qquad\quad 3\ \ 4 \end{array}$$

We put the collision element into the first empty cell to the right of the collided one:

$$\begin{array}{c} a : [\ \ |\ \ |\ (37,3)\ |\ (52,5)\ |\ \ ] \\ \quad\ 0\ \ 1 \qquad 2 \qquad\quad 3 \qquad 4 \end{array}$$

```
def put(k,v)
    i = h(k)
    while a[i] != null
        i = (i+1) % m
    a[i] = (k,v)
```

To get a key, we have two possibilities:
- The key is in the table
- We find an empty cell before our searched key which that our key is not in the table

```
def get(k)
    i = h(k)
    while a[i] != null
        if a[i].first = k
            return a[i].second
    return null
```

Array size and complexity:
- n elements – array of size $m = n$

This does not work because at the end we will not have any empty space so the get function will not work.

- lets try with n elements $-$ array of size $m = 2n$

  Approximately half of our array is empty so $\mathbb{P}(a[i] = \text{null}) = \frac{1}{2}$.

  So, the probability of making i step before finding (or not) our key is:

$$\mathbb{P}(\text{number of steps} = i) = \frac{1}{2^i}$$

  So:

$$\mathbb{E}(\text{number of steps}) = \sum_i \frac{1}{2^i} = 1$$

  This is actually not totally true because our probability are not independent. This is because of clustering:

$$\begin{array}{c} a : [\ |\ \ |\ \ |\times|\times|\times|\times|\times|\ \ |\ \ |\ \ ] \\ \quad\ 0\ 1\ 2\ 3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8\ 9\ 10 \end{array}$$

  In this case, the probability of putting our next value in the cluster increase because of how the previous value are disposed in the array. Therefore probability are dependent of each other. But we will talk after of avoiding cluster. So we can still say that:

$$T(\text{get}) = \mathbb{E}(\text{number of steps}) = \mathrm{O}(1)$$

How to avoid clustering?

- When we iterate over the right, we change the size of the steps. The change need to follow a function, what is important is to use the same jump function for put and get function.

  The function can be:
  - Incremental: (1^{st} jump -> 1 cell,2^{nd} jump -> 2 cells,3^{rd} jump -> 3 cells,…)
  - Polynomial: (1^{st} jump -> 1 cell,2^{nd} jump -> 4 cells,3^{rd} jump -> 9 cells,…)
  - Or, what we can do is to select a second hash function h_2 created at the same time as the first one. In this case length of jump $= h_2(k)$

  But if this function make us make big jump, our function will be less efficient.

  It is a tradeoff.

Why use open addressing when the previous technique seems to work just fine?

- Memory efficiency:
  - No other lists
  - No other pointers
- Useful for caching because we iterate on a continuous block of memory

In practice this technique can be more efficient than the first one.

14.4 - Hash tables poisoning

The only random point that ensure the working of our algorithms is the moment we chose a hash function randomly from our set.

If someone was able to find what function we use, he would be able to create key that are all colliding in order to make our hash table algorithms inefficient.

To detect such thing, we check the size of our list in each of the hash map cell. If one of the list is too big, there is a problem because we proved that those size are expected to be constant.

If we detect such big list, what we can do is to pick another hash function and recreate the hash table with a new function.

14.5 - Dynamic hash table

In practice, we rarely know how many element we are going to put in our hash table.

To make it grow, we use the same idea we use to deal against poisoning. when we detect that our table is to little (one list of cell has become too big), we pick a function & recreate the hash map bigger.

Just like the array, our worst case complexity will become $O(n)$ but the amortized one will stay $O(1)$.

15 - Perfect & Cuckoo hashing, Bloom filters

Last lecture, we saw 2 data structures:

`set`:	`map`:
• `insert(x)`	• `put(k,v)`
• `contain(x)`	• `get(k)`

All of these function were working in O(1) average when $\to \mathbb{P}(h(x) = h(y)) \simeq \frac{1}{m}$.
$x \neq y$

The goal of this lecture is to make a hash table that allow us to have `get` and `contain` function working in O(1) worst case instead of average.

Why? Because it happens a lot that we make a hash table at the beginning at the program and then make many `get` operations.

There is 2 techniques to do that, perfect hashing and cuckoo hashing.

15.1 - Perfect hashing

Last lecture we had:

```
def get(k)
    i = h(k)
    find k in a[i]
```

Since $a[i]$ was a list, our find operations had a worst case complexity of $O(n)$. What we want to do is use another data structure that will allow us to have a find operation working in constant time.

And what data structure allows us to find in constant time? Yes hash table. Instead of lists, we will have hash tables to store our collision.

We will use small hash tables inside a big hash table.

We need a different hash function between the big and small hash tables. If not, our optimization would be useless. And will use a different hash function for each small hash table.

$$a[i] - \text{hash table with hash function } \ g_i$$

At this moment you might ask yourself:
- But what is we have collision in $a[i]$, we also put smaller hash tables in it again?
- At one point, we need to stop the hash tables Matryoshka.

What are we going to do is to keep $a[i]$ collision free. We talked about the possibility of doing that in the previous lecture, we can have collision free table if the array is big enough compared to number of element inserted.

We could not use that for our main table because the number of element is to big but there since our tables are small we can afford to do it.

$$n_i = \text{number of elements in } a[i]$$
$$\text{size}(a[i]) = n_i^2$$

If we do that, we have a high probability that we will not have any collision. If one occur anyway, we get a new hash function until we do not have any collision.

Lets calculate the average number of collision. Since we have n_i element:

$$\mathbb{E}(\text{collision}) = \left(\frac{n(n-1)}{2}\right) \cdot \frac{1}{n^2} \simeq \frac{1}{2}$$

$$\mathbb{P}(\text{collision} \geq 1) \leq \frac{1}{2}$$

So we will not have to try many times to find a good function g_i.

Since our small hash table are collision free, our `find k in a[i]` work in $\mathrm{O}(1)$ worst case.

Now that we have a working hash table, lets see what is its size:

$$\text{size} = \sum_i n_i^2$$

$$\sum_i n_i^2 = \text{number of pair } (x,y), h(x) = i, h(y) = i$$

$$\begin{aligned}
\mathbb{E}\left(\sum_i n_i^2\right) &= \mathbb{E}(\text{number of pairs } (x,y) : h(x) = h(y)) \\
&= \underbrace{n}_{\substack{n \text{ pairs} \\ \text{where } x=y}} + \mathbb{E}(\text{number of pairs } (x,y) : x \neq y, h(x) = h(y)) \\
&= n + \left(\frac{n(n-1)}{2}\right) \cdot \frac{1}{n} \\
&= \mathrm{O}(n)
\end{aligned}$$

The size of our hash table stays linear.

In theory, this is a good technique but in practice, the constant factor is much bigger than the classical way we saw in the previous lecture + it uses more memory. So, in practice, with the exception of special weird case, you do not want to use it (remember we made the same commentary on Fibonacci heap, great theory but mostly unusable).

15.2 - Cuckoo hashing

We have 2 hash function (h_1 and h_2) and 2 array (a_1 and a_2) with $\text{size}(a_1) + \text{size}(a_2) >$ number of elements.

Each element have two possible placement, one in a_1 and one in a_2.

```
def get(k)
    look in a1[h1(k)]
    look in a2[h2(k)]
```

But how we put elements x in our tables?

- If one of the two positions: $a_1[h_1(x)]$ or $a_2[h_2(x)]$, we put x in this empty position.
- When we have an element in $a_1[h_1(x)]$ and in $a_2[h_2(x)]$ (respectively A and B):
  - Both of our possibility are taken
  - We put x at the place of A
  - We try to put A in a_2 at $a_2[h_2(A)]$:
    - If an element C is in there then we put A at the place of C and try to put C in a_1

And so on until we reach an empty position.

A problem here is that we can en in a cycle:

$$\begin{gathered}
X \text{ goes in } a_1[h_1(X)] = a_1[h_1(A)] \\
A \text{ goes from } a_1[h_1(A)] \to a_2[h_2(A)] = a_2[h_2(B)] \\
B \text{ goes from } a_2[h_2(B)] \to a_1[h_1(B)] = a_1[h_1(C)] \\
C \text{ goes from } a_1[h_1(C)] \to a_2[h_2(C)] = a_2[h_2(D)] \\
D \text{ goes from } a_2[h_2(D)] \to a_1[h_1(D)] = a_1[h_1(E)] \\
E \text{ goes from } a_1[h_1(E)] \to a_2[h_2(E)] = a_2[h_2(A)] \\
A \text{ goes from } a_2[h_2(A)] \to a_1[h_1(A)] = a_1[h_1(X)] \\
X \text{ goes from } a_1[h_1(X)] \to a_2[h_2(X)] = a_2[h_2(D)]
\end{gathered}$$

We can see that here we have a full cycle from we cannot escape. And we cannot fully prevent those kind of cycle.

So how to do deal with that?
- Use a cycle detection algorithm
- Keep the value of the number of changes. If we go this number become bigger than $\simeq 100$ it mean we probably have a cycle.

When we detect a cycle, we recalculate a hash function and make our structures from the base.

Now, to keep our complexity small, we want to ensure that the probability of cycle is small.

The proof will be in the further reading because it is kinda black magic and use the property of k-universality of our function set. which roughly mean:

$$\forall (x_1, ...x_k), (y_1, ...y_k) \in \mathbb{N}^k, \mathbb{P}(h(x_1) = h(y_1), ...h(x_k) = h(y_k)) = \frac{1}{m^k}$$

In our case, if our set H is $\log(n)$-universal our probability will stay low enough.

This is complex but what we need to grasp is that with a good hash function this works good. Just be careful in production and try to ensure that your possible function are sufficiently good.

We can add that at the beginning we said that $\text{size}(a_1) + \text{size}(a_2) > \text{number of elements}$. We can say that we want to have $\text{size}(a_1) + \text{size}(a_2) > c \cdot \text{number of elements}$ with $c \geq 1$

The probability of our cycle will decrease with the growing of c (our memory consumption grow with c so it is a tradeoff).

But anyway these cycle only affect our put speed. Our get will always works in $O(1)$ worst case.

15.3 - Bloom filters

We want to create a set data structure with classical operations (insert and contain).

The objective is to use the less memory we can. Basically, the smallest amount is the memory used to remember all the elements of the set. We want to go lower than that.

One might think it is impossible, and one would be right in general. But, we are going to introduce a property that make it possible.

We are going to allow our structure to be wrong. Sometimes our structure will say x is in the set but x will not be in it.

The error threshold for contains operation should not be more than ε. The size of our structure will be a tradeoff of our probability.

$$\varepsilon \nearrow \text{size} \searrow$$
$$\varepsilon \searrow \text{size} \nearrow$$

15.3.1 - Set of 1 element

You have one element x and a function that tells you if a given y is equal to x.

```
def equal_x(y)
    return x == y
```

To optimize the memory, we do not store x but h_x which is the hash of x.

```
def equal_x(y)
    hy = hash(y)
    return hx == hy
```

We can see that if y and x collide, we have a wrong answer.

But, we saw that with a good function,

$$\text{for memory} = k \text{ bits}$$
$$\mathbb{P}(\underset{x \neq y}{h(x) = h(y)}) = \frac{1}{2^k}$$

We can now see the tradeoff between memory usage and the probability of error.

15.3.2 - Actual set

We have an array of bits of length m and k hash functions $(h_1, ..., h_k)$.

Instead of computing and storing the hash of x, we do the following:

- for insert function:
  - for all i in $[\![1, k]\!]$ compute $h_{i(x)}$.
  - for all i in $[\![1, k]\!], a[h_i(x)] = 1$
- for contain function:
  - for all i in $[\![1, k]\!]$ we check that $a[h_i(x)] = 1$

```
def insert(x)
    for i = 1..k
        a[h[i](x)] = 1
```

```
def contain(x)
    for i = 1..k
        if a[h[i](x)] == 0
            return false
    rewturn true
```

Again, we see from where our errors can come. We can insert element that collide with x for a specific h_i. So if $\forall i \in [\![1, k]\!], \exists y, h_i(x) = h_i(y)$ we have contain equal true for x when x is not actually here.

Lets now fix our k and m:

- We want m such as $\mathbb{P}(a[i] = 1) = 2$ (i.e. when we pick an element at random from a, we have the same chance to get 0 or 1).

  Since for each element we add k ones, we add $k \cdot n$ ones total.

  If we want halfs the number of 1 and 0 in a, we need $m \simeq 2 \cdot k \cdot n$ (actually it is smaller ($m = 1.44 \cdot k \cdot n$) because some ones are colliding).

Since $\mathbb{P}(a[i] = 1) = \frac{1}{2}$, the probability of a false positive is $\frac{1}{2}$ for each iteration.

We then have: $\mathbb{P}(\text{false positive}) \simeq \frac{1}{2^k}$

And from definition we have: $\mathbb{P}(\text{false positive}) = \varepsilon$

- From that, we fix $k = \log \frac{1}{\varepsilon}$

Again, we clearly see the dependance between memory usage and number of error.

Example:

$$n = 10^6 (\text{big objects})$$
$$\varepsilon \simeq 10^{-6}$$
$$k \simeq 20, m \simeq 2.10^7 \simeq 2.5 \text{ MB}$$

Which is very compact.

15.4 - Cuckoo filter

The idea is to combine both cuckoo hashing and bloom filter idea.

The idea is the following:

- for insert function:
  - We compute x', a k bits fingerprint (hash) of x.
  - we insert the element in the cuckoo table:
    - we compute $h_1(x)$ and $h_2(x)$
    - but instead of putting x as in the cuckoo hashing, we put x' in one of the array
- for the check function: we check if $a_1[h_1(x)]$ or $a_2[h_2(x)]$ contain x'

We have a problem here:

- During the insertion, if both places are occupied, we move one of the present element (lets called it A)
- For cuckoo hashing, we were just putting A in $a_2[h_2(A)]$
- Here, we cannot do that because we cannot compute $a_2[h_2(A)]$ since it is A' in the array.

To go around this problem we define:

$$h_1(x) = \text{hash}(x)$$
$$h_2(x) = h_1(x) \oplus \text{hash}(x')$$

With this:

- If we go from $a_1 \to a_2$:
  - we know the value of $h_1(x)$ because it is the value of the index of A' in a_1.
  - we can compute $\text{hash}(x')$ because we have x'

  So we can compute $h_2(x)$
- If we go from $a_2 \to a_1$:

  We know $h_2(x)$ and can compute $\text{hash}(x')$, so we have: $h_1(x) = h_2(x) \oplus \text{hash}(x')$ from xor property.

Here, we define:

$$k = \log \frac{1}{\varepsilon} + 1$$

$$\text{size} \simeq \frac{k \cdot n}{\text{load factor}}$$

Note 1: It is really difficult to make a bloom or a cuckoo filter that allow to remove elements.

Note 2: We have skipped some math in this lecture (especially in the cuckoo sections), this is because the mathematics behind this are pretty hardcore and require deep knowledge.