

# Algorithmie

File de priorité

---

BOULCH Alexandre

File de priorité et HeapSort

Heap Sort

TP

On veut créer une file où les éléments sortent selon une priorité qui leur est donnée (un score).

Il faut donc maintenir la file triée au fur et à mesure qu'on insert des éléments.

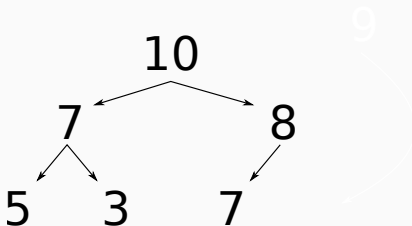
## Compromis

- Accéder en  $O(1)$ , implique une insertion en  $O(N)$   
*Exemple utiliser une liste chaînée (parcourir la file pour connaître l'emplacement)*
- Insérer en  $O(1)$ , implique un accès en  $O(N)$   
*Ajouter à la fin du tableau avec `push_back`, mais parcours en entier pour trouver le bon élément.*

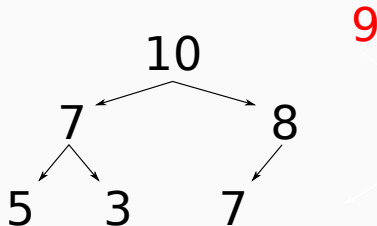
## Solution

On fait le choix d'une insertion et un accès en  $O(\log N)$ . Pour cela, on utilise une structure d'arbre binaire (chaque parents a au plus 2 fils). On construit l'arbre de sorte que chaque parent ait une priorité au moins aussi grande que ses fils.

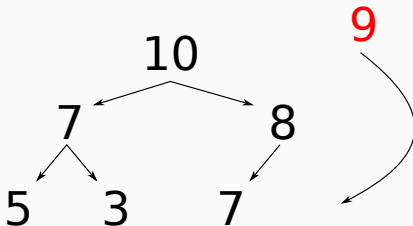
Le nouvel élément est inséré à hauteur minimale dans l'arbre (l'emplacement libre le plus proche de la racine).



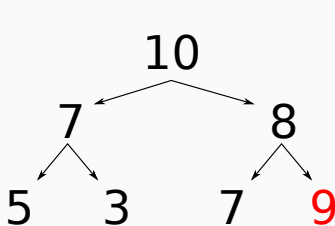
Le nouvel élément est inséré à hauteur minimale dans l'arbre  
(l'emplacement libre le plus proche de la racine).



Le nouvel élément est inséré à hauteur minimale dans l'arbre  
(l'emplacement libre le plus proche de la racine).

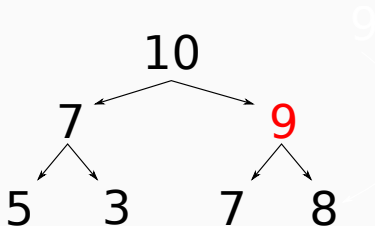


Le nouvel élément est inséré à hauteur minimale dans l'arbre (l'emplacement libre le plus proche de la racine).

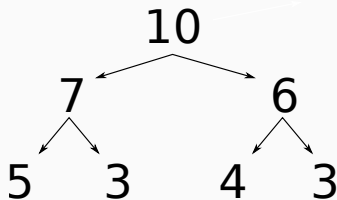




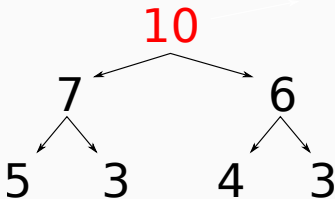
On échange l'élément avec son parent tant que son parent a une priorité plus faible que lui.



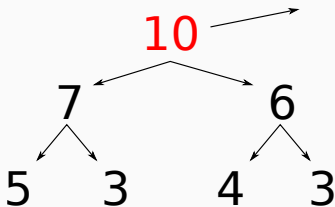
L'élément à retirer est à la racine.



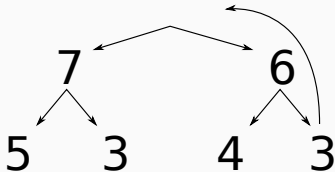
L'élément à retirer est à la racine.



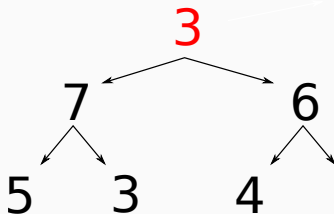
L'élément à retirer est à la racine.



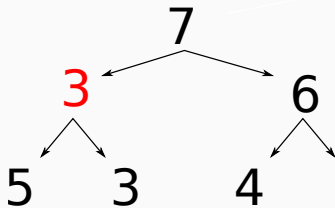
L'élément le plus profond est placé à la racine.



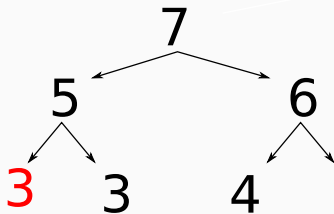
L'élément le plus profond est placé à la racine.



Il est ensuite échangé avec son fils de priorité maximale. On recommence tant qu'il y a un fils avec une priorité plus grande que l'élément considéré.

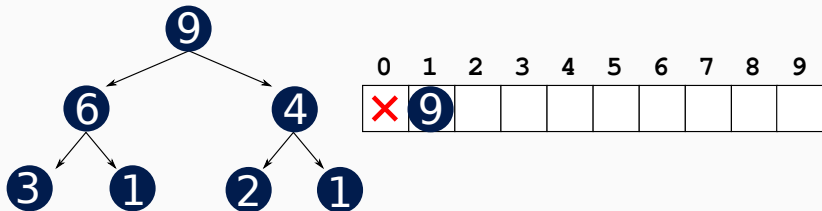


Il est ensuite échangé avec son fils de priorité maximale. On recommence tant qu'il y a un fils avec une priorité plus grande que l'élément considéré.

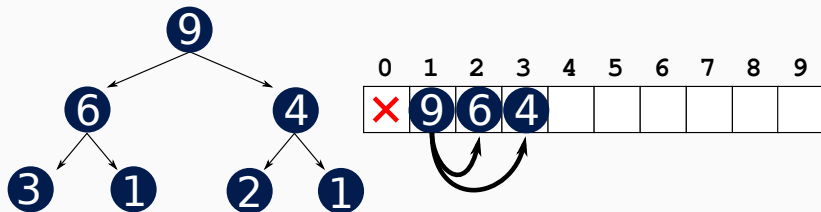




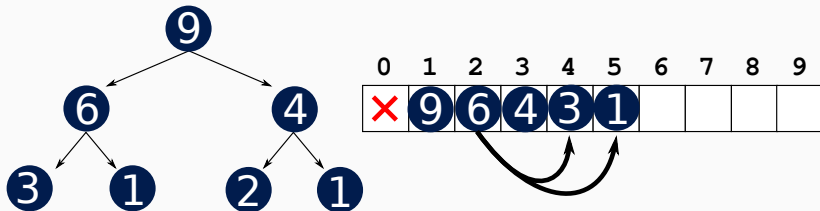
On stocke la file de priorité dans un tableau. On commence à l'indice 1, de sorte que les fils du nœud  $i$  soient placés à  $2i$  et  $2i + 1$ .



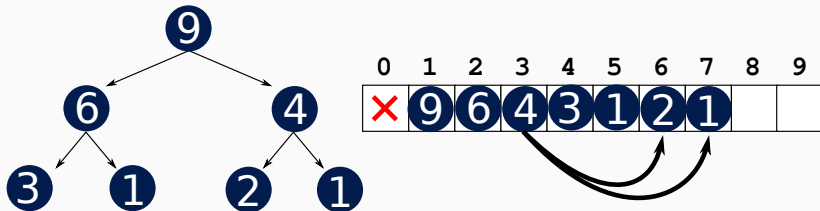
On stocke la file de priorité dans un tableau. On commence à l'indice 1, de sorte que les fils du nœud  $i$  soient placés à  $2i$  et  $2i + 1$ .



On stocke la file de priorité dans un tableau. On commence à l'indice 1, de sorte que les fils du nœud  $i$  soient placés à  $2i$  et  $2i + 1$ .



On stocke la file de priorité dans un tableau. On commence à l'indice 1, de sorte que les fils du nœud  $i$  soient placés à  $2i$  et  $2i + 1$ .



File de priorité et HeapSort

Heap Sort

TP

HeapSort remplit une file de priorité et puis retire les éléments un par un.

```
void HeapSort(std::vector<double> &v){  
    FilePriorite f;  
    for(int i=0; i<v.size(); i++){  
        f.push(v[i]);  
    }  
    for(int i=0; i<v.size(); i++){  
        v[i] = f.pop();  
    }  
}
```

HeapSort est un tri en  $O(N \log N)$  dans tous les cas. Cependant en comparaison à QuickSort, il utilise plus de mémoire et est plus long en moyenne.

En pratique c'est QuickSort le plus utilisé.

- Tri :  $O(N \log N)$
- Recherche dans un tableau trié :  $O(\log N)$
- Recherche dans un tableau non trié :  $O(N)$



File de priorité et HeapSort

Heap Sort

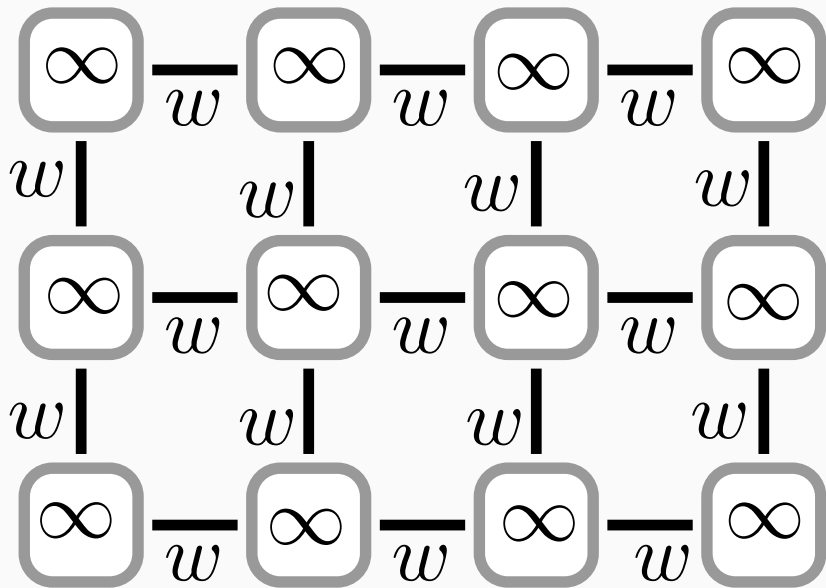
TP

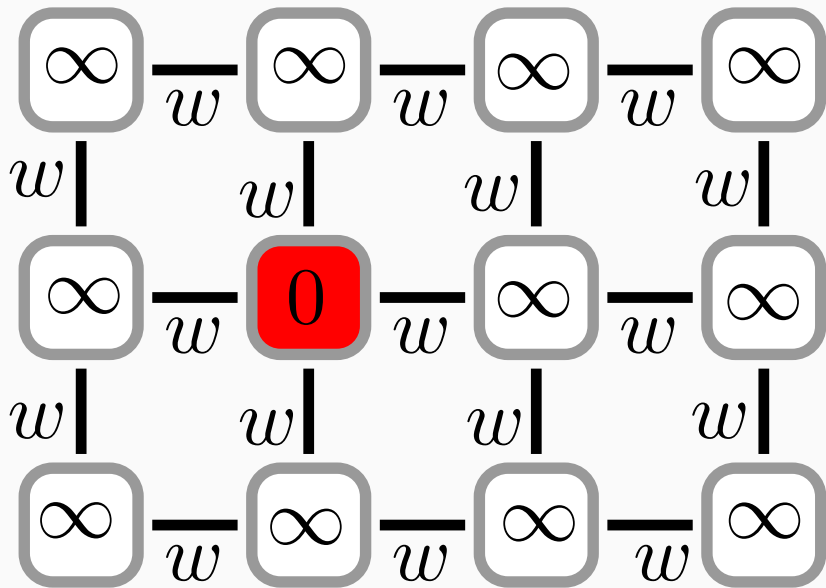
## TP

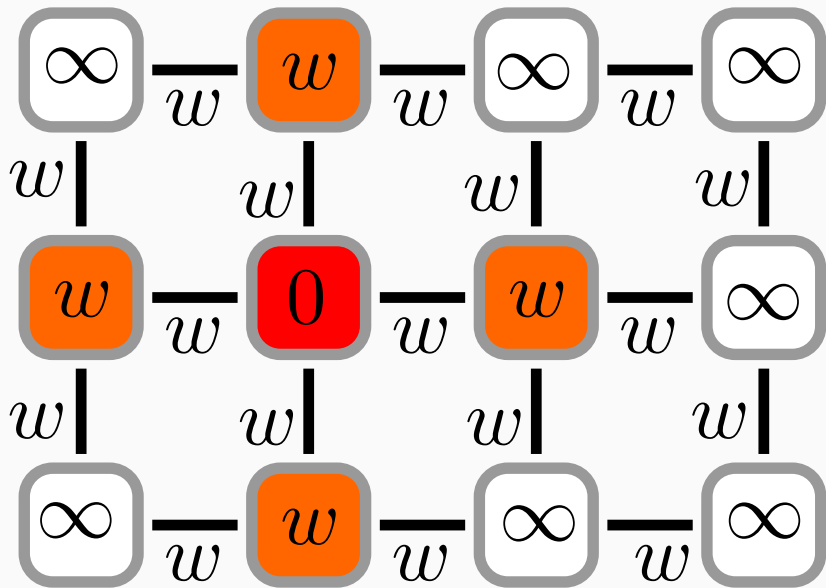
Deux parties :

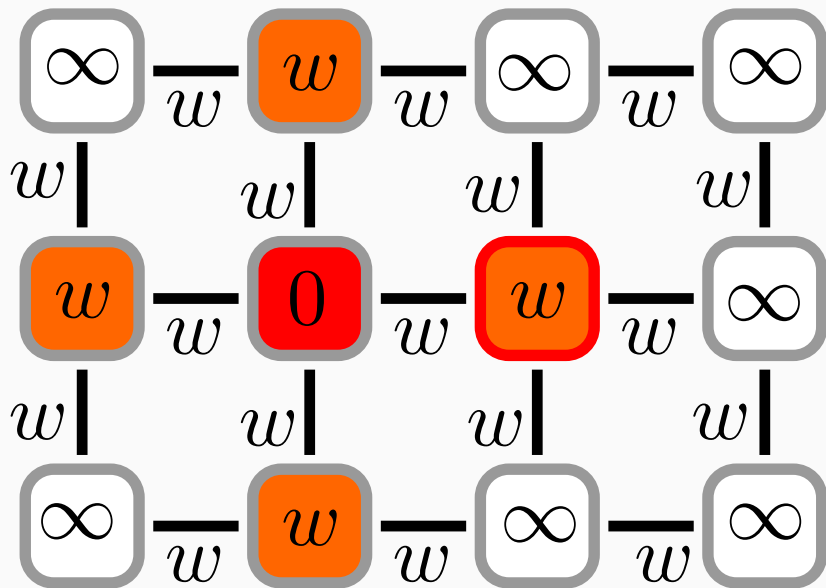
- Implémentation
- Application au fast marching

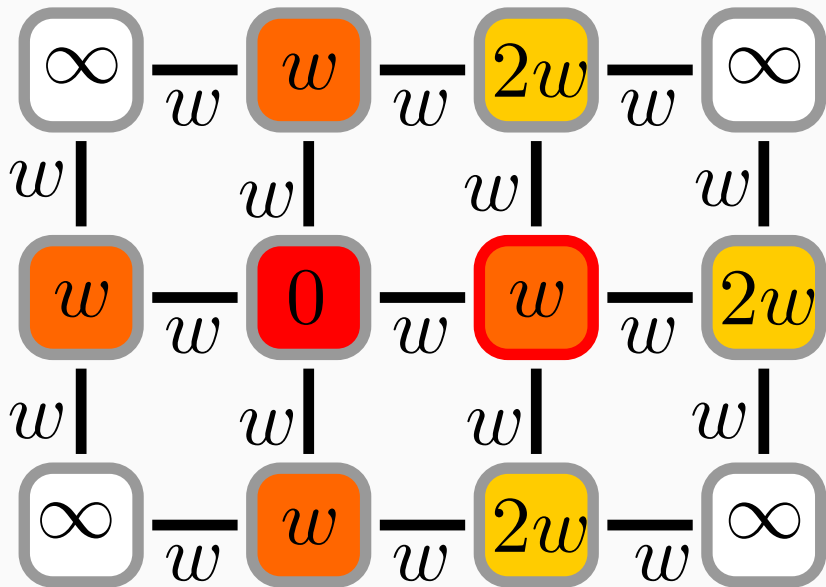
L'algorithme de Dijkstra calcul les plus courts chemins dans un graphe.



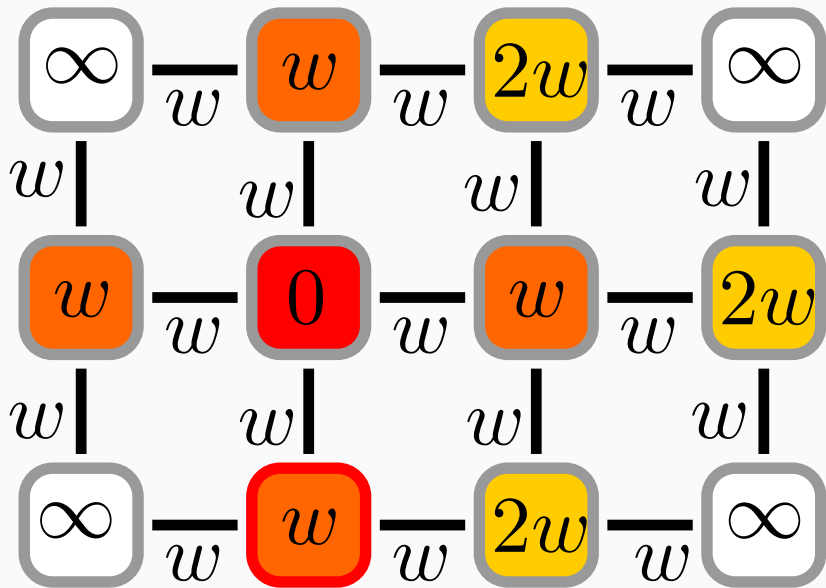


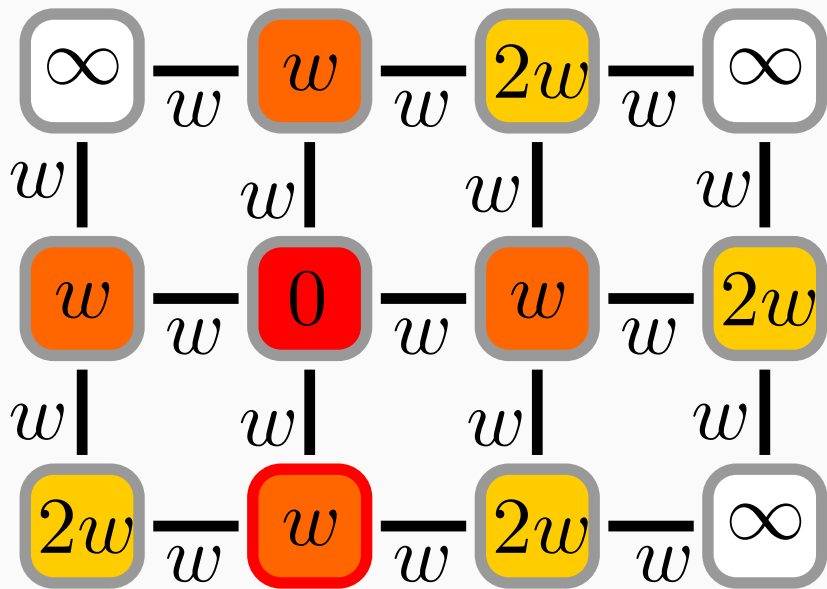


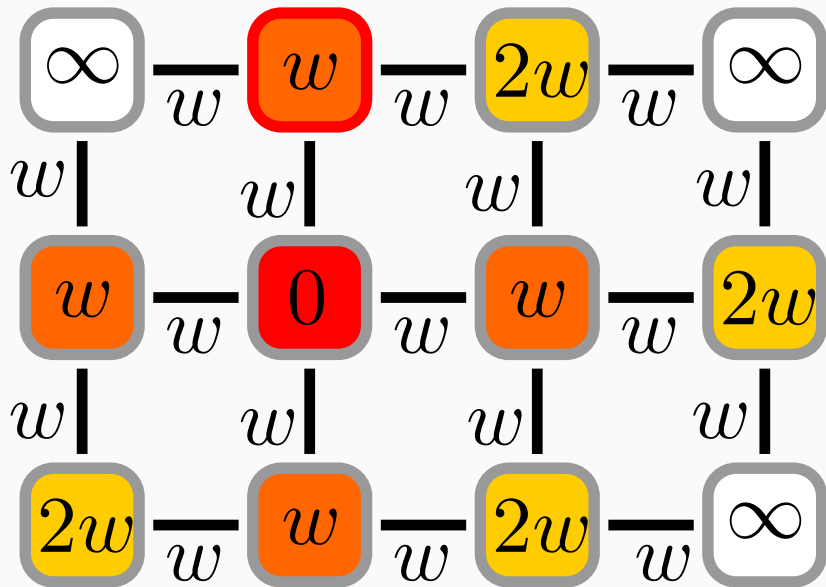


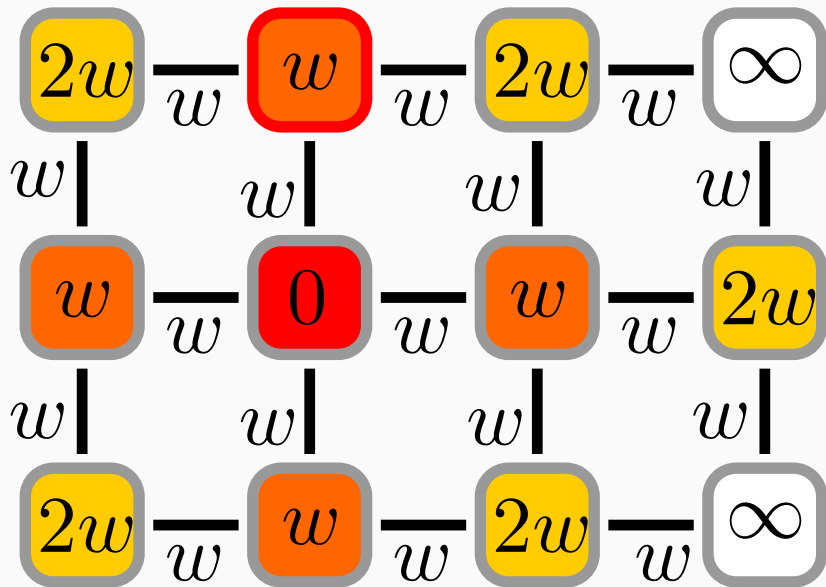


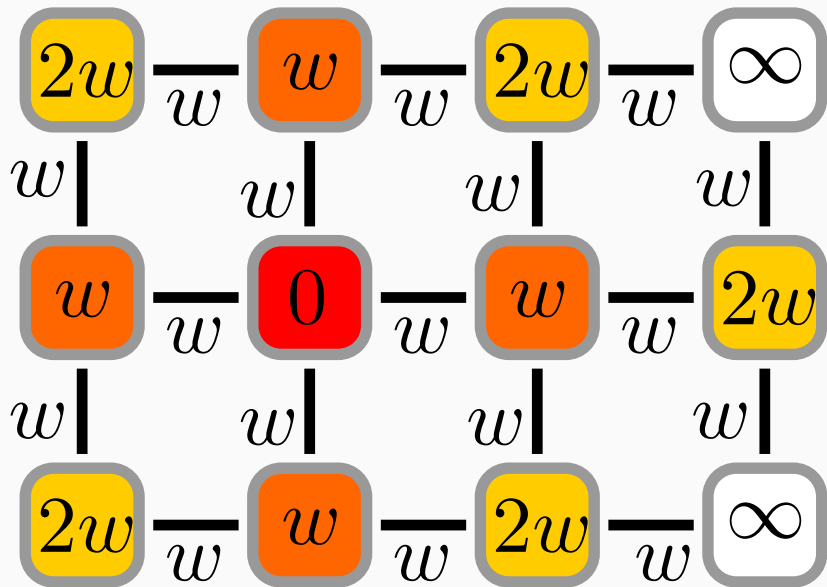


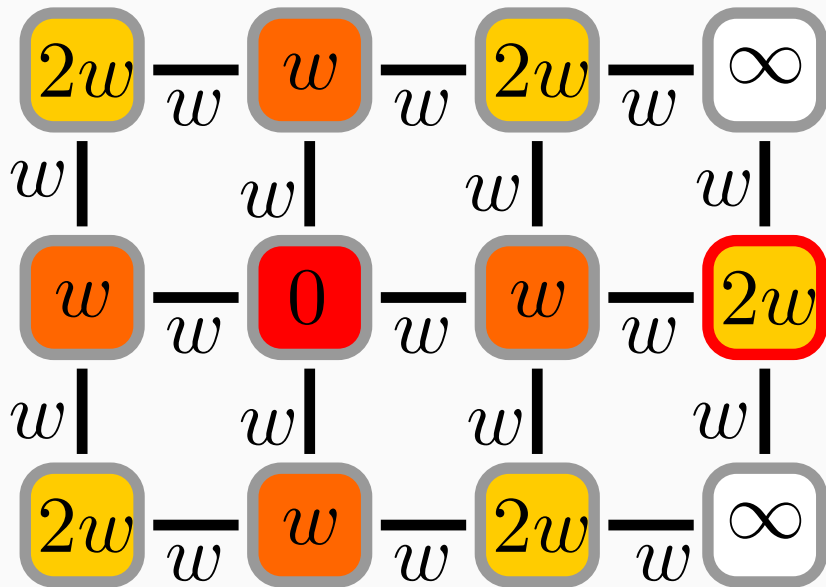


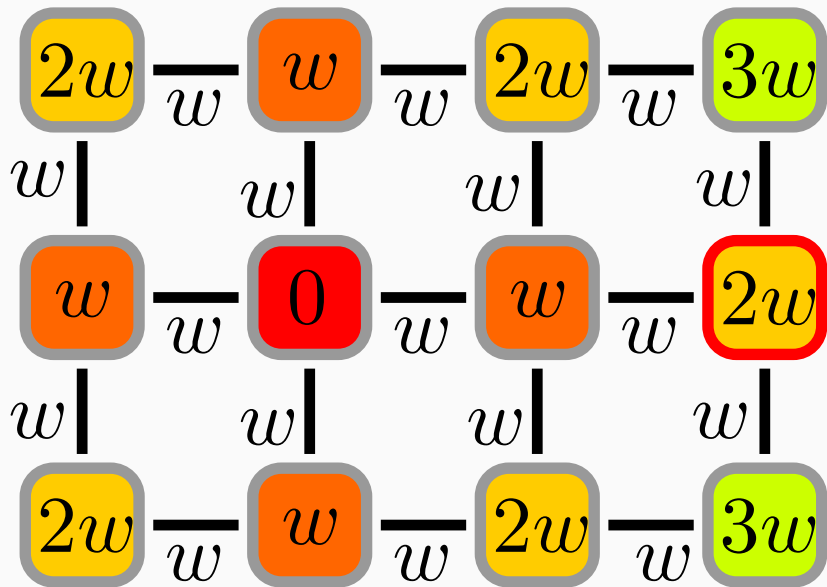








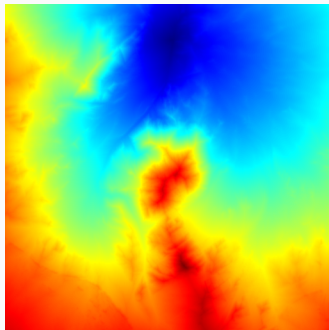
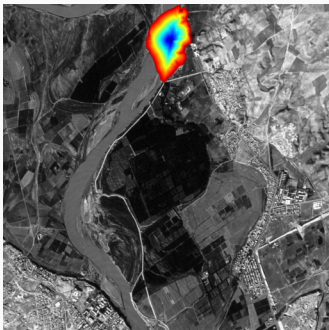




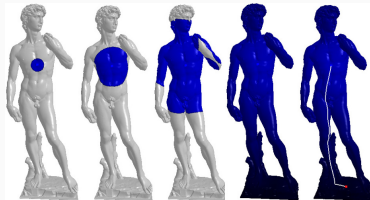
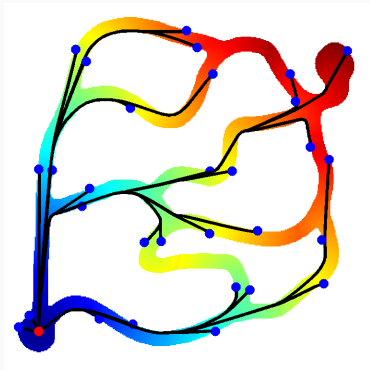
Le fast marching est une modification de l'algorithme précédent pour que les distances soient correctement calculées (exemple distance euclidienne).



## Calcul rapide de cartes de distances



## Calcul de plus court chemin



## **Exercice : swap**

Échanger deux nombres sans création de variable auxiliaire.

## **Exercice : Google Code Jam 2010**

Soit une liste  $L$  de nombres.  $L$  contient-elle une paire dont la somme est  $K$  ? si oui, quels sont-ils ?

## **Exercice : tours de Hanoï**

Proposer une méthode pour résoudre le problème pour  $K$  étages.

Input

```
3
100
3
5 75 25
200
7
150 24 79 50 88 345 3
8
8
2 1 9 4 4 56 90 3
```

Output

```
Case #1: 2 3
Case #2: 1 4
Case #3: 4 5
```

Tester toutes les paires.

## Solution naïve

Solution en  $O(N^2)$ . Risque de ne pas être efficace pour le gros dataset.

```
// charger les donnees du fichier
int S = ... // la somme
vector<int> articles = ...

// trier la liste des indices en fonction des prix
vector<int> id_tries = ...

// rechercher la paire dans un ensemble trie
int i=0, j=id_tries.size()-1;
while(i<j){
    if ( article[id_tries[i]]+article[id_tries[j]]== S )
        break;
    if ( article[id_tries[i]]+article[id_tries[j]] < S )
        i++;
    if ( article[id_tries[i]]+article[id_tries[j]] > S )
        j--;
}

// ecrire la solution
```

Solution en  $O(N \log N)$