Complexité : Compléments

Quentin Fortier

June 13, 2022

Complexité optimale

Objectif : trouver l'algorithme le plus efficace pour résoudre un problème donné.

Complexité optimale

Objectif : trouver l'algorithme le plus efficace pour résoudre un problème donné.

Cela revient à :

1. Trouver un algorithme pour le résoudre (souvent « facile »).
2. Montrer qu'il n'existe pas d'autre algorithme plus efficace (souvent difficile).

Complexité optimale

Objectif : trouver l'algorithme le plus efficace pour résoudre un problème donné.

Cela revient à :

1. Trouver un algorithme pour le résoudre (souvent « facile »).
2. Montrer qu'il n'existe pas d'autre algorithme plus efficace (souvent difficile).

« efficace » peut vouloir dire : en nombre d'opérations (exact ou O(...)), en espace mémoire, dans le pire des cas/en moyenne...

Recherche d'un minimum

Question

On souhaite trouver le minimum d'un tableau de n entiers distincts. Quel est le plus petit nombre de comparaisons nécessaire?

Recherche d'un minimum

Question

On souhaite trouver le minimum d'un tableau de n entiers distincts. Quel est le plus petit nombre de comparaisons nécessaire?

La solution naïve réalise $n-1$ comparaisons. Est-ce optimal?

Erreur classique

« Si un algorithme effectuait moins de $n - 1$ comparaisons pour trouver le minimum d'un tableau de taille n, un élément ne serait jamais comparé »

Erreur classique

« Si un algorithme effectuait moins de $n-1$ comparaisons pour trouver le minimum d'un tableau de taille n, un élément ne serait jamais comparé »

C'est faux : en comparant les éléments par paire, on effectue seulement $\approx \dfrac{n}{2}$ comparaisons et chaque élément est comparé.

Recherche d'un minimum

Si un algorithme calcule le minimum m d'un tableau, alors chaque élément $e \neq m$ doit avoir été comparé plus grand qu'un autre au moins une fois

Recherche d'un minimum

Si un algorithme calcule le minimum m d'un tableau, alors chaque élément $e \neq m$ doit avoir été comparé plus grand qu'un autre au moins une fois (sinon, l'algorithme se trompe si on remplace e par $m - 1$).

Recherche d'un minimum

Si un algorithme calcule le minimum m d'un tableau, alors chaque élément $e \neq m$ doit avoir été comparé plus grand qu'un autre au moins une fois (sinon, l'algorithme se trompe si on remplace e par $m - 1$).

Conclusion : le plus petit nombre de comparaisons pour trouver un minimum est bien $n - 1$.

Recherche d'un minimum

Si un algorithme calcule le minimum m d'un tableau, alors chaque élément $e \neq m$ doit avoir été comparé plus grand qu'un autre au moins une fois (sinon, l'algorithme se trompe si on remplace e par $m-1$).

Conclusion : le plus petit nombre de comparaisons pour trouver un minimum est bien $n-1$.

Pour prouver une borne inférieure sur la complexité d'un algorithme, il est souvent utile de considérer l'**arbre de décision** de cet algorithme.

Recherche d'un minimum

Considérons un algorithme quelconque calculant le minimum d'un tableau `t` de taille n.
Cet algorithme effectue une première comparaison, disons entre les éléments `t.(i)` et `t.(j)`.

Recherche d'un minimum

Considérons un algorithme quelconque calculant le minimum d'un tableau `t` de taille n.
Cet algorithme effectue une première comparaison, disons entre les éléments `t.(i)` et `t.(j)`.

Il va ensuite effectuer une 2ème comparaison entre des éléments `t.(k)` et `t.(l)`. Le choix de `k` et `l` dépend a priori du résultat de la comparaison de `t.(i)` et `t.(j)`.

Ainsi de suite... jusqu'au moment où l'algorithme s'arrête et renvoie le minimum.

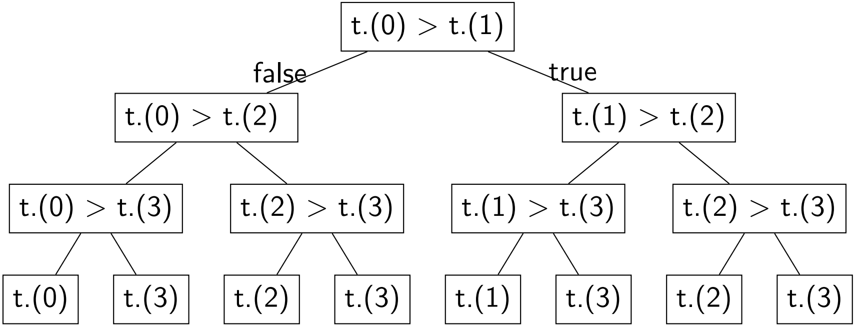
Arbre de décision

On peut dessiner l'arbre dont les noeuds sont les comparaisons, la racine est la comparaison initiale et les feuilles correspondent aux valeurs de retour de l'algorithme.

Le sous-arbre gauche (respectivement droit) correspond au cas où la comparaison est `false` (respectivement `true`)

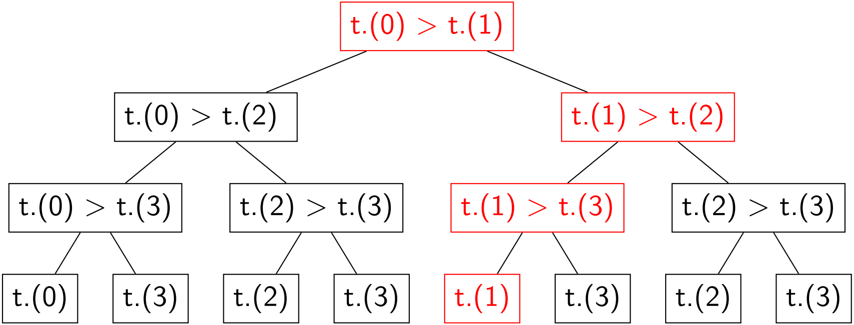
Arbre de décision

Arbre de décision de `mini` sur un tableau de taille 4 :



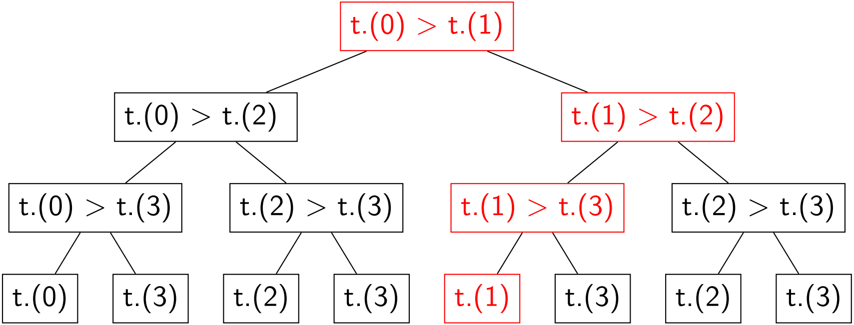
Arbre de décision

Exemple d'exécution sur `t = [|5; 1; 3; 2|]` :



Arbre de décision

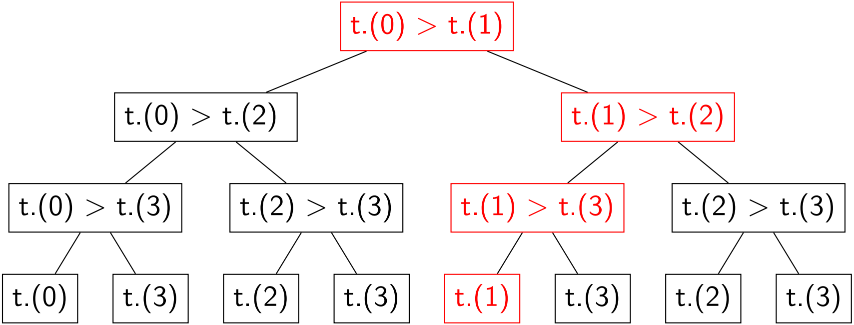
Exemple d'exécution sur `t = [|5; 1; 3; 2|]` :



<u>Complexité dans le pire cas</u> :

Arbre de décision

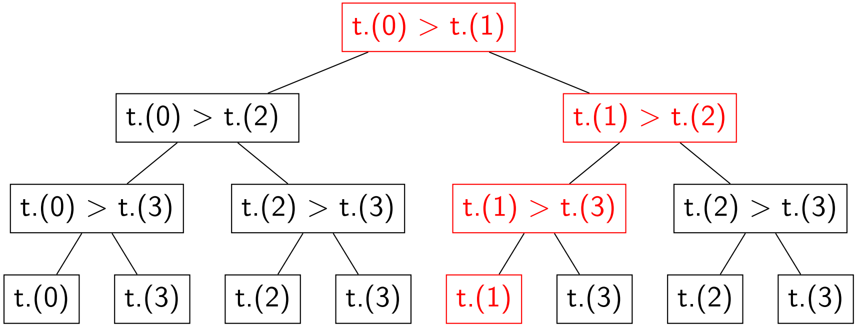
Exemple d'exécution sur `t = [|5; 1; 3; 2|]` :



<u>Complexité dans le pire cas</u> : hauteur de l'arbre de décision.

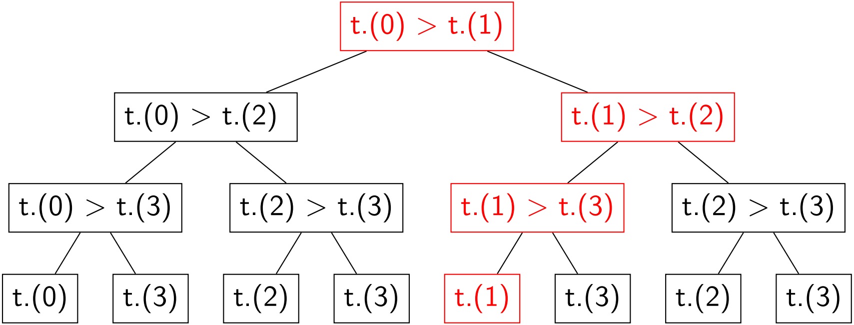
Arbre de décision

Exemple d'exécution sur `t = [|5; 1; 3; 2|]` :



Arbre de décision

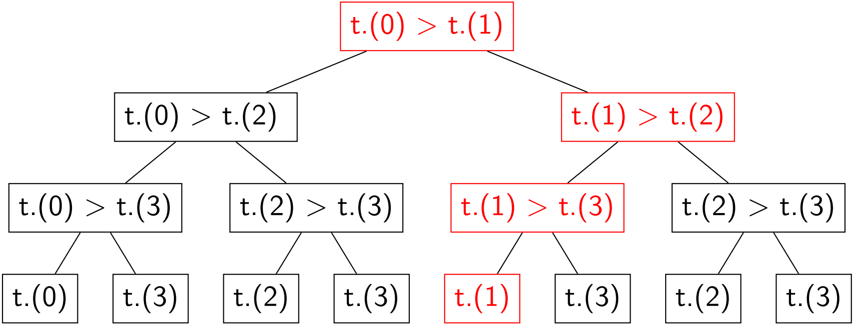
Exemple d'exécution sur `t = [|5; 1; 3; 2|]` :



<u>Complexité dans le meilleur cas</u> :

Arbre de décision

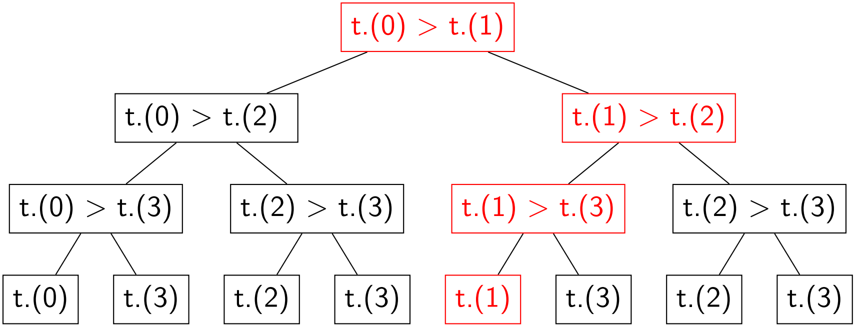
Exemple d'exécution sur `t = [|5; 1; 3; 2|]` :



<u>Complexité dans le meilleur cas</u> : profondeur minimum d'une feuille.

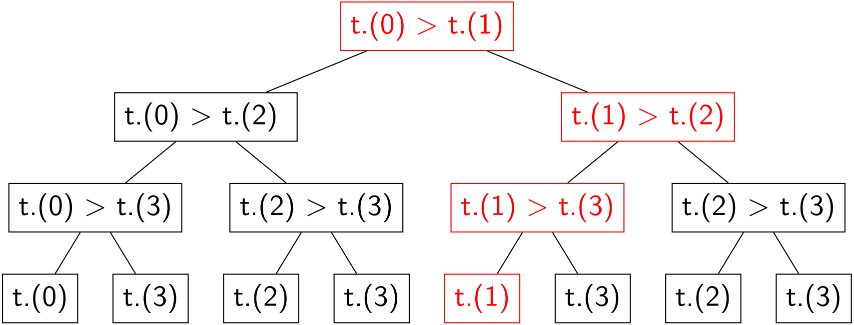
Arbre de décision

Exemple d'exécution sur `t = [|5; 1; 3; 2|]` :



Arbre de décision

Exemple d'exécution sur `t = [|5; 1; 3; 2|]` :



Complexité en moyenne : profondeur moyenne des feuilles.

Arbre de décision

Nous avons montré qu'un arbre de décision d'un algorithme qui calcule un minimum dans un tableau de taille n a une hauteur $\geq n-1$.

Et même mieux : que toutes les feuilles sont de profondeur $\geq n-1$.

Minimum ET maximum

Question

Quel est le nombre optimal de comparaisons effectuées par un algorithme calculant le minimum et maximum d'un tableau de taille n?

Minimum ET maximum

Question

Quel est le nombre optimal de comparaisons effectuées par un algorithme calculant le minimum et maximum d'un tableau de taille n?

(Naïf) On calcule minimum, puis maximum en $2(n-1)$ comparaisons.

Minimum ET maximum

Question

Quel est le nombre optimal de comparaisons effectuées par un algorithme calculant le minimum et maximum d'un tableau de taille n?

(Naïf) On calcule minimum, puis maximum en $2(n-1)$ comparaisons.

(Mieux) On conserve le minimum `mini` et maximum `maxi` vu jusqu'à présent. Soient a et b les prochains éléments du tableau.

Minimum ET maximum

Question

Quel est le nombre optimal de comparaisons effectuées par un algorithme calculant le minimum et maximum d'un tableau de taille n?

(Naïf) On calcule minimum, puis maximum en $2(n-1)$ comparaisons.

(Mieux) On conserve le minimum `mini` et maximum `maxi` vu jusqu'à présent. Soient a et b les prochains éléments du tableau.
Si $a < b$, on sait que b ne peut pas être le minimum et que a ne peut pas être le maximum.

Minimum ET maximum

Question

Quel est le nombre optimal de comparaisons effectuées par un algorithme calculant le minimum et maximum d'un tableau de taille n?

(Naïf) On calcule minimum, puis maximum en $2(n-1)$ comparaisons.

(Mieux) On conserve le minimum `mini` et maximum `maxi` vu jusqu'à présent. Soient a et b les prochains éléments du tableau.
Si $a < b$, on sait que b ne peut pas être le minimum et que a ne peut pas être le maximum. On compare donc seulement a avec `mini` et b avec `maxi`.

Minimum ET maximum

Question

Quel est le nombre optimal de comparaisons effectuées par un algorithme calculant le minimum et maximum d'un tableau de taille n?

(Naïf) On calcule minimum, puis maximum en $2(n-1)$ comparaisons.

(Mieux) On conserve le minimum `mini` et maximum `maxi` vu jusqu'à présent. Soient a et b les prochains éléments du tableau.
Si $a < b$, on sait que b ne peut pas être le minimum et que a ne peut pas être le maximum. On compare donc seulement a avec `mini` et b avec `maxi`.
De même symétriquement si $a > b$.

Minimum ET maximum

Question

Écrire une fonction `min_max : int list -> int * int` implémentant cet algorithme.

Minimum ET maximum

Question

Écrire une fonction `min_max : int list -> int * int` implémentant cet algorithme.

```
let rec min_max = function (* renvoie (minimum, maximum) *)
  | [] -> max_int, min_int
  | [a] -> a, a
  | a::b::q -> let mini, maxi = min_max q in
               if a < b then (min a mini), (max b maxi)
               else (min b mini), (max a maxi);;
```

Nombre de comparaisons :

Minimum ET maximum

Question

Écrire une fonction `min_max : int list -> int * int` implémentant cet algorithme.

```
let rec min_max = function (* renvoie (minimum, maximum) *)
  | [] -> max_int, min_int
  | [a] -> a, a
  | a::b::q -> let mini, maxi = min_max q in
               if a < b then (min a mini), (max b maxi)
               else (min b mini), (max a maxi);;
```

Nombre de comparaisons : $\left\lfloor \frac{3n}{2} \right\rfloor$.

On peut montrer qu'on ne peut pas faire mieux.

Trouver le 2ème minimum

Question

On veut maintenant trouver le 2ème plus petit élément d'un tableau (ou liste) de taille n. Combien faut-il de comparaisons?

Trouver le 2ème minimum

Question

On veut maintenant trouver le 2ème plus petit élément d'un tableau (ou liste) de taille n. Combien faut-il de comparaisons?

On peut utiliser une méthode similaire à la précédente, en traitant les éléments par paire :

```
let rec min2 = function (* renvoie (minimum, 2eme minimum) *)
  | [] -> max_int, max_int
  | [x] -> x, max_int
  | a::b::q when a < b -> let m1, m2 = min2 q in
                          if a < m1 then a, min m1 b
                          else if a < m2 then m1, a
                          else m1, m2
  | a::b::q -> ... (* similaire au cas précédent *)
```

Trouver le 2ème minimum

Question

On veut maintenant trouver le 2ème plus petit élément d'un tableau (ou liste) de taille n. Combien faut-il de comparaisons?

On peut utiliser une méthode similaire à la précédente, en traitant les éléments par paire :

```
let rec min2 = function (* renvoie (minimum, 2eme minimum) *)
  | [] -> max_int, max_int
  | [x] -> x, max_int
  | a::b::q when a < b -> let m1, m2 = min2 q in
                          if a < m1 then a, min m1 b
                          else if a < m2 then m1, a
                          else m1, m2
  | a::b::q -> ... (* similaire au cas précédent *)
```

Nombre de comparaisons :

Trouver le 2ème minimum

Question

On veut maintenant trouver le 2ème plus petit élément d'un tableau (ou liste) de taille n. Combien faut-il de comparaisons?

On peut utiliser une méthode similaire à la précédente, en traitant les éléments par paire :

```
let rec min2 = function (* renvoie (minimum, 2eme minimum) *)
  | [] -> max_int, max_int
  | [x] -> x, max_int
  | a::b::q when a < b -> let m1, m2 = min2 q in
                          if a < m1 then a, min m1 b
                          else if a < m2 then m1, a
                          else m1, m2
  | a::b::q -> ... (* similaire au cas précédent *)
```

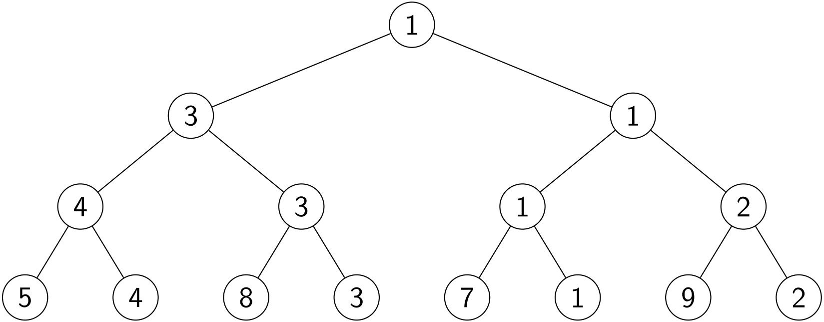
Nombre de comparaisons : $\left\lfloor \frac{3n}{2} \right\rfloor$. On peut faire mieux!

Trouver le 2ème minimum

Autre méthode pour calculer un minimum (**tournoi**) : comparer tous les éléments par paire, puis comparer les $\left\lceil \frac{n}{2} \right\rceil$ plus petits par paire, ..., jusqu'à obtenir le minimum.

Trouver le 2ème minimum

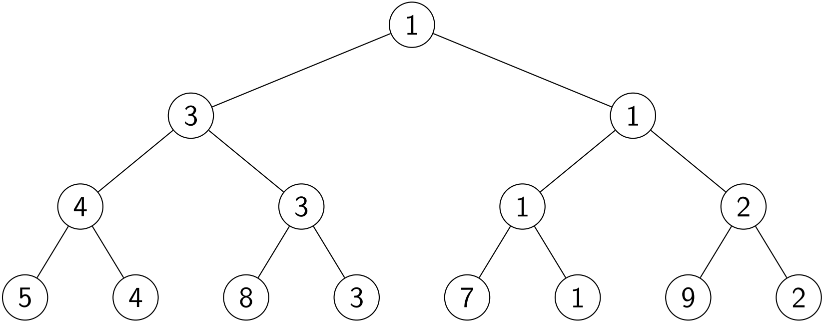
Autre méthode pour calculer un minimum (**tournoi**) : comparer tous les éléments par paire, puis comparer les $\left\lceil \frac{n}{2} \right\rceil$ plus petits par paire, ..., jusqu'à obtenir le minimum.



`t = [|5; 4; 8; 3; 7; 1; 9; 2|]`

Trouver le 2ème minimum

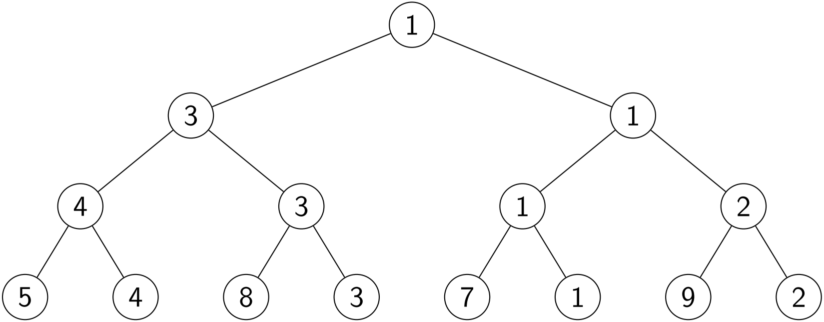
Autre méthode pour calculer un minimum (**tournoi**) : comparer tous les éléments par paire, puis comparer les $\left\lceil \frac{n}{2} \right\rceil$ plus petits par paire, ..., jusqu'à obtenir le minimum.




```
t = [|5; 4; 8; 3; 7; 1; 9; 2|]
```

C'est un arbre binaire (presque) complet.

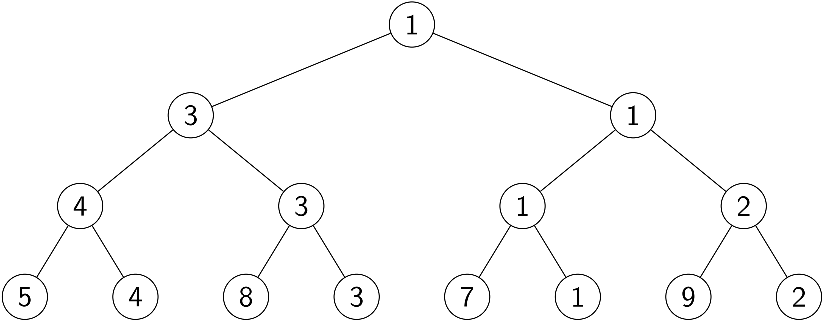
Trouver le 2ème minimum



`t = [|5; 4; 8; 3; 7; 1; 9; 2|]`

Le nombre de comparaisons effectuées est :

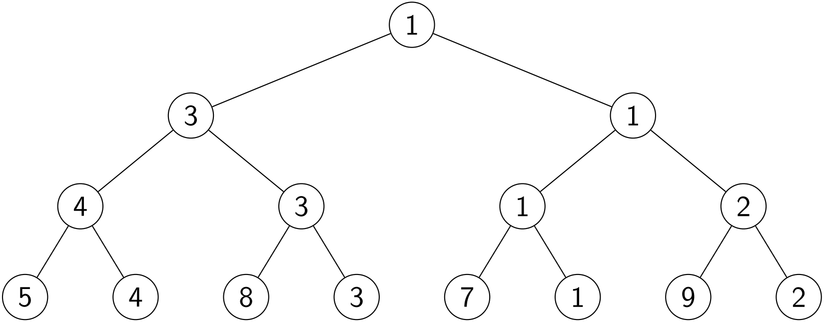
Trouver le 2ème minimum




```
t = [|5; 4; 8; 3; 7; 1; 9; 2|]
```

Le nombre de comparaisons effectuées est :
nb de noeuds internes =

Trouver le 2ème minimum




```
t = [|5; 4; 8; 3; 7; 1; 9; 2|]
```

Le nombre de comparaisons effectuées est :
nb de noeuds internes = nb feuilles − 1 = $n - 1$ (arbre binaire **strict**).

Trouver le 2ème minimum

Le 2ème minimum a forcément

Trouver le 2ème minimum

Le 2ème minimum a forcément été comparé au minimum m.

Il suffit dont de trouver le minimum des éléments comparés à m.

Trouver le 2ème minimum

Le 2ème minimum a forcément été comparé au minimum m.

Il suffit dont de trouver le minimum des éléments comparés à m.

Le nombre d'éléments comparés à m est égal à

Trouver le 2ème minimum

Le 2ème minimum a forcément été comparé au minimum m.

Il suffit dont de trouver le minimum des éléments comparés à m.

Le nombre d'éléments comparés à m est égal à la hauteur de m dans l'arbre précédent :

Trouver le 2ème minimum

Le 2ème minimum a forcément été comparé au minimum m.

Il suffit dont de trouver le minimum des éléments comparés à m.

Le nombre d'éléments comparés à m est égal à la hauteur de m dans l'arbre précédent : $\lceil \log_2(n) \rceil$ (arbre binaire complet)

Trouver le 2ème minimum

Le 2ème minimum a forcément été comparé au minimum m.

Il suffit dont de trouver le minimum des éléments comparés à m.

Le nombre d'éléments comparés à m est égal à la hauteur de m dans l'arbre précédent : $\lceil \log_2(n) \rceil$ (arbre binaire complet)

L'algorithme effectue donc

Trouver le 2ème minimum

Le 2ème minimum a forcément été comparé au minimum m.

Il suffit dont de trouver le minimum des éléments comparés à m.

Le nombre d'éléments comparés à m est égal à la hauteur de m dans l'arbre précédent : $\lceil \log_2(n) \rceil$ (arbre binaire complet)

L'algorithme effectue donc $\boxed{n - 1 + \lceil \log_2(n) \rceil - 1}$ comparaisons pour trouver le 2ème minimum. On peut montrer que c'est optimal.

Exercice

Programmer cet algorithme en OCaml.

Complexité des tris

1. Tri fusion :

Complexité des tris

1. Tri fusion : $\Theta(n \log(n))$
2. Tri à bulles :

Complexité des tris

1. Tri fusion : $\Theta(n \log(n))$
2. Tri à bulles : $\Theta(n^2)$
3. Tri rapide :

Complexité des tris

1. Tri fusion : $\Theta(n\log(n))$
2. Tri à bulles : $\Theta(n^2)$
3. Tri rapide : $\mathsf{O}(n^2)$ dans le pire des cas mais $\Theta(n\log(n))$ en moyenne.

Peut-on faire mieux que $\Theta(n\log(n))$, dans le pire des cas?

Complexité des tris

Considérons l'arbre de décision d'un algorithme triant un tableau de taille n.

Complexité des tris

Considérons l'arbre de décision d'un algorithme triant un tableau de taille n.

Une feuille correspond à

Complexité des tris

Considérons l'arbre de décision d'un algorithme triant un tableau de taille n.

Une feuille correspond à un résultat de l'algorithme, c'est à dire un réarrangement trié du tableau.
Combien y a t-il de feuilles?

Complexité des tris

Considérons l'arbre de décision d'un algorithme triant un tableau de taille n.

Une feuille correspond à un résultat de l'algorithme, c'est à dire un réarrangement trié du tableau.
Combien y a t-il de feuilles? au moins $n!$ (nombre de permutations)

Complexité des tris

Considérons l'arbre de décision d'un algorithme triant un tableau de taille n.

Une feuille correspond à un résultat de l'algorithme, c'est à dire un réarrangement trié du tableau.
Combien y a t-il de feuilles? au moins $n!$ (nombre de permutations)

La hauteur h est égale à la complexité dans le pire cas.

Complexité des tris

Considérons l'arbre de décision d'un algorithme triant un tableau de taille n.

Une feuille correspond à un résultat de l'algorithme, c'est à dire un réarrangement trié du tableau.
Combien y a t-il de feuilles? au moins $n!$ (nombre de permutations)

La hauteur h est égale à la complexité dans le pire cas.

$$h \geq \log_2(f) \geq \log_2(n!)$$

Complexité des tris

Considérons l'arbre de décision d'un algorithme triant un tableau de taille n.

Une feuille correspond à un résultat de l'algorithme, c'est à dire un réarrangement trié du tableau.
Combien y a t-il de feuilles? au moins $n!$ (nombre de permutations)

La hauteur h est égale à la complexité dans le pire cas.

$$h \geq \log_2(f) \geq \log_2(n!) \sim \boxed{n \log_2(n)}$$

Complexité des tris

Considérons l'arbre de décision d'un algorithme triant un tableau de taille n.

Une feuille correspond à un résultat de l'algorithme, c'est à dire un réarrangement trié du tableau.
Combien y a t-il de feuilles? au moins $n!$ (nombre de permutations)

La hauteur h est égale à la complexité dans le pire cas.

$$h \geq \log_2(f) \geq \log_2(n!) \sim \boxed{n\log_2(n)}$$

<u>Conclusion</u> : il est impossible de trier un tableau de taille n en $\mathsf{o}(n\log_2(n))$ comparaisons, dans le pire des cas.

Tri par dénombrement

Si on dispose d'hypothèses supplémentaires sur le tableau `t` en entrée, ou si l'algorithme effectue autre chose que des comparaisons pour réaliser le tri, le minorant $n \log_2(n)$ ne s'applique plus.

Tri par dénombrement

Si on dispose d'hypothèses supplémentaires sur le tableau `t` en entrée, ou si l'algorithme effectue autre chose que des comparaisons pour réaliser le tri, le minorant $n \log_2(n)$ ne s'applique plus.

Exemple : le **tri par dénombrement** trie un tableau dont les entrées sont des entiers entre 0 et k en complexité $\Theta(n + k)$.

Complexité en **moyenne** des tris

Le minorant $\Theta(n \log_2(n))$ a été établi pour la complexité dans *le pire des cas*.

Est-ce que l'on peut trier avec une meilleur complexité *en moyenne*?

Complexité en **moyenne** des tris

Le minorant $\Theta(n \log_2(n))$ a été établi pour la complexité dans *le pire des cas*.

Est-ce que l'on peut trier avec une meilleur complexité *en moyenne*?

Définition

Soit E l'ensemble des entrées possibles pour un algorithme $\mathcal{A}$.
Si $C(e)$ est le nombre d'opérations réalisées par $\mathcal{A}$ sur $e \in E$, alors la complexité en moyenne de $\mathcal{A}$ est :

$$\frac{\sum_{e \in E} C(e)}{|E|}$$

Complexité en **moyenne** des tris

Montrons par récurrence que dans un arbre binaire à n feuilles, la somme des profondeurs des feuilles est au moins $n\log_2(n)$.

1. C'est vrai pour $n = 1$.
2. Supposons la proposition vraie pour tout $n \leq N$. La somme des profondeurs des feuilles d'un arbre à $N+1$ feuilles est au moins :

$$n + 1 + \underbrace{k\log_2(k)}_{\text{fils gauche}} + \underbrace{(n+1-k)\log_2(n+1-k)}_{\text{fils droit}}(*)$$

Comme $f(x) = x\log_2(x)$ est convexe :

$$n + 1 + f(k) + f(n+1-k) \geq \underbrace{n + 1 + 2f(\frac{n+1}{2})}_{(n+1)\log_2(n+1)}$$

Complexité en **moyenne** des tris

La profondeur moyenne d'un arbre binaire à n feuilles est donc au moins $\log_2(n)$.

Complexité en **moyenne** des tris

La profondeur moyenne d'un arbre binaire à n feuilles est donc au moins $\log_2(n)$.

Un arbre de décision pour un tri a $n!$ feuilles, donc la profondeur moyenne de ses feuilles est au moins : $\log_2(n!) \sim n\log_2(n)$.

Complexité en **moyenne** des tris

La profondeur moyenne d'un arbre binaire à n feuilles est donc au moins $\log_2(n)$.

Un arbre de décision pour un tri a $n!$ feuilles, donc la profondeur moyenne de ses feuilles est au moins : $\log_2(n!) \sim n \log_2(n)$.

Conclusion : il n'est pas possible de trier un tableau de taille n avec moins de $n \log_2(n)$ comparaisons en moyenne.