Université de Liège

Structures de données et algorithmes

Théorie 2016 - 2017

2ème bachelier en ingénieur civil

Auteurs :
Antoine Wehenkel
Antoine Louis
Tom Crasset
Benjamin Delvoye

Professeur :
P. Geurts

Année académique 2016-2017

Table des matières

1 Introduction et récursivité

1.1 Qu'est-ce qu'un algorithme ? Comment peut-on le décrire ?

Un algorithme est une suite finie et non-ambiguë d'opérations ou d'instructions permettant de résoudre un problème.

Un problème algorithmique est souvent formulé comme la transformation d'un ensemble de valeurs (entrées), en un nouvel ensemble de valeurs (sorties).

1.2 Qu'est-ce qu'un algorithme correct, partiellement correct et approximatif ?

- Un algorithme est totalement correct lorsque pour chaque instance, il se termine en produisant la bonne sortie.
- Un algorithme est partiellement correct lorsque sa terminaison n'est pas assurée mais qu'il fournit la bonne sortie lorsqu'il se termine.
- Un algorithme est approximatif lorsqu'il fournit une sortie inexacte mais néanmoins proche de l'optimum.

1.3 Qu'est-ce qu'une structure de données ? Un type de données abstrait ? Donnez un ou plusieurs exemples.

Une structure de données est une méthode pour stocker et organiser les données pour en faciliter l'accès et la modification.

Un type de données abstrait (TDA) représente l'interface d'une structure de données.

Exemple : une file à priorités

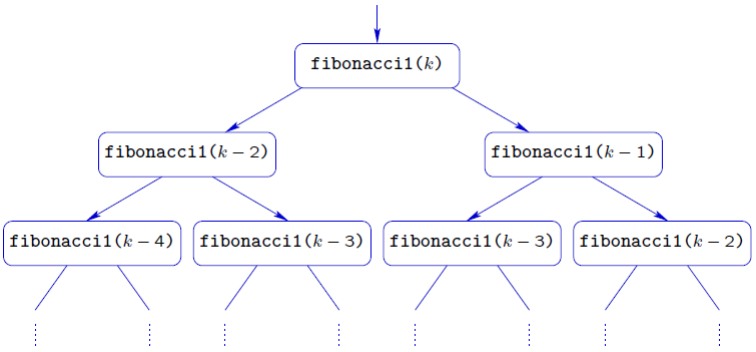
- Données gérées : des objets possédant chacun une clé et une valeur quelconque.
- Opérations : création de la file, INSERT(S,x),...
- Nombreuses façons d'implémenter ce TDA : liste triée, tableau non trié,...

1.4 Qu'est-ce qu'un algorithme récursif ? La récursivité simple ? La récursivité terminale ? Donnez à chaque fois un exemple.

- Un algorithme est récursif s'il s'invoque lui-même directement ou indirectement.
  Exemple : FACTORIAL(n)
- On parle de récursivité multiple lorsque le corps d'une fonction contient plusieurs appels récursifs.
  Exemple : FIBONACCI(n)
- Une procédure est récursive terminale si elle n'effectue plus aucune opération après s'être invoquée récursivement.
  Exemple : FACTORIAL2-REC(n,i,f)

1.5 Représentez l'arbre des appels récursifs d'un algorithme donné.

Représentons l'arbre de la fonction récursive de Fibonacci :



1.6 Etant donné une fonction récursive terminale, écrivez un algorithme itératif équivalent.

Prenons l'algorithme récursif terminal du calcul de la factorielle :

```
FACTORIAL2(n)
1  return FACTORIAL2-REC(n, 2, 1)
```

```
FACTORIAL2-REC(n, i, f)
1  if i > n
2        return f
3  return FACTORIAL2-REC(n, i + 1, f.i)
```

On peut l'écrire en version itérative :

```
FACTORIAL2(n)
1  while i <= n
2        f = f.i
3        f++
4  return f
```

2 Outils d'analyse

2.1 Comment vérifie-t-on qu'un algorithme itératif/récursif est correct ?

En pratique, nous combinerons deux solutions :

1. En testant concrètement l'algorithme : cela suppose d'implémenter l'algorithme dans un langage et de le faire tourner. Cela suppose également qu'on peut déterminer les instances du problème à vérifier. Notons qu'il est très difficile de prouver empiriquement qu'on n'a pas de bug.
2. En dérivant une preuve mathématique formelle : pas besoin d'implémenter et de tester toutes les instances du problème.

2.2 Montrez qu'un algorithme donné itératif/récursif est correct.

- Pour les algorithmes itératifs, on utilise les triplets de Hoare ou les invariants de boucle :
  - Un code est correct si le triplet "$\{P\}code\{Q\}$" est vrai où la pré-condition P représente les conditions que doivent remplir les entrées valides de l'algorithme et où la post-condition Q représente les conditions qui expriment que le résultat de l'algorithme est correct.
  - La preuve par invariant est basée sur le principe général de la preuve par induction.
- Pour les algorithmes récursifs, on utilise des preuves par induction :
  - Cas de base : cas de base de la récursion.
  - Cas inductif : on suppose que les appels récursifs sont corrects et on en déduit que l'appel courant est correct.
  - Terminaison : on montre que les appels récursifs se font sur des sous-problèmes (souvent trivial).

2.3 Expliquez le principe de preuve par induction. Montrez par induction qu'une propriété simple donnée est correcte.

On veut montrer qu'une propriété est vraie pour une série d'instances. On suppose l'existence d'un ordonnancement des instances.

- On montre explicitement que la propriété est vraie pour la ou les premières instances.
- On suppose que la propriété est vraie pour les k premières instances et on montre qu'elle l'est alors aussi pour la $(k+1)$-ième instance quel que soit k.

Par le principe d'induction, la propriété sera vraie pour toutes les instances.

2.4 Qu'entend-on par complexité en temps et complexité en espace ?

- La complexité en temps rend compte du temps de calcul de l'algorithme.
- La complexité en espace rend compte de l'espace mémoire consommé par l'algorithme.

2.5 Que signifient les notations $O(.), \Theta(.), \Omega(.)$?

- $f(n) \in O(g(n)) \simeq f(n) \leq g(n)$: $g(n)$ est borne supérieure asymptotique pour $f(n)$.
- $f(n) \in \Omega(g(n)) \simeq f(n) \geq g(n)$: $g(n)$ est borne inférieure asymptotique pour $f(n)$.
- $f(n) \in \Theta(g(n)) \simeq f(n) = g(n)$: $g(n)$ est borne serrée asymptotique pour $f(n)$.

2.6 Qu'est-ce que la complexité au pire cas, meilleur cas, en moyenne ? Quel cas est le plus pertinent en pratique ?

Soit D_n l'ensemble des instances de taille n d'un problème et $T(i_n)$ le temps de calcul pour une instance $i_n \in D_n$:

- Complexité au pire cas : $T(n) = max\{T(i_n)|i_n \in D_n\}$
- Complexité au meilleur cas : $T(n) = min\{T(i_n)|i_n \in D_n\}$
- Complexité moyenne : $T(n) = \sum_{i_n \in D_n} Pr(i_n).T(i_n)$, où $Pr(i_n)$ est la probabilité de rencontrer i_n.

On se focalise généralement sur le cas le plus défavorable car il donne une borne supérieure sur le temps d'exécution. De plus, le meilleur cas n'est pas représentatif et le cas moyen est difficile à calculer.

2.7 Calculez et justifiez la complexité d'un algorithme donné (itératif).

Règles pour les algorithmes itératifs :

- Affectation, accès à un tableau, opérations arithmétiques, appel de la fonction $\rightarrow O(1)$
- Instructions "if - then - else" $\rightarrow O$(complexité max des deux branches).
- Séquence d'opérations $\rightarrow$ l'opération la plus coûteuse domine (règle de la somme).
- Boucle simple $\rightarrow O(n.f(n))$ si le corps de la boucle est $O(f(n))$.
- Boucle incrémentales doubles $\rightarrow O(n^2)$.
- Boucle avec un incrément exponentiel (ex : "$i = 2i$") $\rightarrow O(log\ n)$.

Notons que : $O(1) \subset O(log\ n) \subset O(n) \subset O(n\ log\ n) \subset O(n^{\alpha>1}) \subset O(2^n)$

2.8 Un algorithme de complexité X est-il systématiquement meilleur/moins bon qu'un algorithme de complexité Y ? Justifiez.

2.9 Montrez que le problème de tri est $\Omega(n)$. Montrez qu'il est aussi $O(n^2)$.

Analysons la complexité en temps du tri par insertion :

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j - 1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

où t_j est le nombre de fois que la condition du while est testée. Le temps d'exécution $T(n)$ pour un tableau de taille n vaut alors :

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Le pire cas correspond à un tableau trié par ordre décroissant : $t_j = j$. Le temps de calcul devient :

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\tfrac{n(n+1)}{2} - 1\right) + c_6\left(\tfrac{n(n-1)}{2}\right) + c_7\left(\tfrac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\tfrac{c_5}{2} + \tfrac{c_6}{2} + \tfrac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \tfrac{c_5}{2} - \tfrac{c_6}{2} - \tfrac{c_7}{2} + c_8\right) n - (c_2 + c_4 + c_5 + c_8) \\ &= an^2 + bn + c \Rightarrow T(n) \in O(n^2) \end{aligned}$$

De plus, prouvons par l'absurde que le problème de tri est $\Omega(n)$:

- Supposons qu'il existe un algorithme moins que $O(n)$ pour résoudre le problème du tri. Cet algorithme ne peut pas parcourir tous les éléments du tableau, sinon il serait au moins $O(n)$.
- Il y a donc au moins un élément du tableau qui n'est pas vu par cet algorithme.
- Il existe donc des instances de tableau qui ne seront pas triées correctement par cet algorithme.
- Il n'y a donc pas d'algorithme plus rapide que $O(n)$ pour le tri.

3 Tri

3.1 Qu'est-ce que le problème de tri ? Qu'est-ce qu'un algorithme de tri en place/itératif/récursif/stable/comparatif ? Donnez à chaque fois un exemple.

- Le problème de tri :
  - Entrée : une séquence de n nombres $\langle a_1, a_2, \ldots, a_n \rangle$
  - Sortie : une permutation de la séquence de départ $\langle a'_1, a'_2, \ldots, a'_n \rangle$ telle que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$
- *Tri en place* : modifie directement la structure qu'il est en train de trier. Ne nécessite qu'une quantité de mémoire supplémentaire.
  Exemple : Insertion-Sort
  Contre-exemple : Merge-Sort
- *Tri itératif* : basé sur un ou plusieurs parcours itératifs du tableau.
  Exemple : Insertion-Sort
- *Tri récursif* : basé sur une procédure récursive.
  Exemple : Merge-Sort
- *Tri stable* : conserve l'ordre relatif des éléments égaux(au sens de la méthode de comparaison).
  Exemple : Insertion-Sort
  Contre-exemple : Quicksort
- *Tri comparatif* : basé sur la comparaison entre les éléments (clés).
  Exemple : Insertion-Sort

3.2 Qu'est-ce qu'un arbre ? Qu'est-ce qu'un arbre binaire/ordonné/binaire entier/binaire parfait/binaire complet ? Qu'est-ce que la hauteur d'un arbre ?

- Définition : Un arbre (*tree*) T est un graphe dirigé (N, E), où :
  - N est un ensemble de nœuds
  - $E \subset (N \times N)$ est un ensemble d'arcs,

  possédant les propriétés suivantes :
  - T est connexe et acyclique
  - Si T n'est pas vide, alors il possède un nœud distingué appelé racine (*root node*). Cette racine est unique.
  - Pour tout arc $(n_1, n_2) \in E$, le nœud n_1 est le parent de n_2.
    - La racine de T ne possède pas de parent.
    - Les autres nœuds de T possèdent un et un seul parent.

- Un arbre *binaire* est un arbre ordonné possédant les propriétés suivantes :
  - Chacun de ses nœuds possède au plus deux fils.
  - Chaque nœud fils est soit un fils gauche, soit un fils droit.

  - Le fils gauche précède le fils droit dans l'ordre des fils d'un nœud.

- Un arbre *ordonné* est un arbre dans lequel les ensembles de fils de chacun de ses nœuds sont ordonnés.
- Un arbre *binaire entier* ou propre (*full* or *proper*) est un arbre binaire dans lequel tous les nœuds internes possèdent exactement deux fils.
- Un arbre *binaire parfait* est un arbre binaire entier dans lequel toutes les feuilles sont à la même profondeur.
- Un arbre *binaire complet* est un arbre binaire tel que :
  - Si h dénote la hauteur de l'arbre :
    * Pour tout $i \in [0, h-1]$, il y a exactement 2^i nœuds à la profondeur i.
    * Une feuille a une profondeur h ou $h-1$.
    * Les feuilles de profondeur maximale (h) sont "tassées" sur la gauche.
- La *hauteur* (*height*) d'un nœud n est le nombre d'arcs d'un plus long chemin de ce nœud vers une feuille. La *hauteur de l'arbre* est la hauteur de sa racine.

3.3 Établissez en utilisant les notations asymptotiques le lien qui existe entre la hauteur d'un arbre binaire (entier ou non) et le nombre de nœuds. Justifiez.

Dans n'importe quel arbre binaire, on a que la hauteur de l'arbre est $\leq$ au nombre de nœuds internes [1]. En effet si h est la hauteur de l'arbre, cela veut dire qu'il existe un chemin qui contient h branches dans l'arbre, ce chemin est constitué de h + 1 nœuds, seul le dernier nœud est externe, le chemin contient donc h nœuds internes. L'arbre contient donc au moins h nœuds internes. Sachant cela, on en déduit que $n \in \Omega(h)$. En effet $n > n_{interne}$ et $n_{interne} \geqslant h$ donc $n > h$.

Le nombre de nœuds d'un arbre binaire est maximum quand celui-ci est parfait, dans ce cas $n = 2^{h+1} - 1$, n'importe quel arbre binaire a donc un nombre de nœuds $n < 2.2^h$ et donc $n \in O(2^h)$.

3.4 Qu'est-ce qu'un tas binaire ? Comment implémente-t-on un tas dans un vecteur ?

- Un *tas binaire* (binary heap) est un arbre binaire complet tel que :
  - Chacun de ses nœuds est associé à une clé.
  - La clé de chaque nœud est supérieure ou égale à celle de ses fils (*propriété d'ordre du tas*).
- Un tas peut être représenté de manière compacte à l'aide d'un tableau A.
  - La racine de l'arbre est le premier élément du tableau.
  - $\textsc{Parent}(i) = \lfloor i/2 \rfloor$
  - $\textsc{Left}(i) = 2i$
  - $\textsc{Right}(i) = 2i + 1$

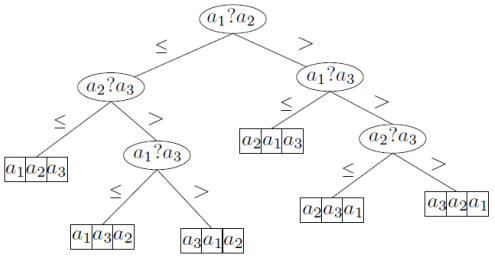
Propriété d'ordre du tas : $\forall i, A[\textsc{Parent}(i)] \geq A[i]$

1. Un nœud qui possède au moins un fils est un nœud interne.

3.5 Construisez l'arbre de décision correspondant à un algorithme de tri X sur un tableau de taille Y .

Un algorithme de tri = un arbre binaire de décision (entier) :

- Feuille de l'arbre : permutation des éléments du tableau initial.
- Tri : chemin de la racine à la feuille correspondant au tableau trié.
- Hauteur de l'arbre : pire cas pour le tri.
- Branche la plus courte : meilleur cas pour le tri.
- Hauteur moyenne de l'arbre : complexité en moyenne du tri.



(arbre de décision pour le tri par insertion du tableau $[e_0, e_1, e_2]$)

3.6 Montrez que le problème de tri (comparatif) est $\Omega(n\ log\ n)$.

Un arbre binaire de hauteur h a au plus 2^h feuilles. Le nombre de feuilles de l'arbre de décision est exactement $n!$ où n est la taille du tableau à trier (par l'absurde : si moins que $n!$, certains tableaux ne seraient pas triés). On a donc :

$$n! \leq 2^h \Rightarrow log(n!) \leq h$$

Par la formule de Stirling, on écrit :

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta(\frac{1}{n}\right) \Rightarrow n! \geq \left(\frac{n}{e}\right)^n$$

Ainsi,

$$h \geq log(n!) > log\left((\frac{n}{e})^n\right) = n\ log\ n - n\ log\ e \Rightarrow h = \Omega(n\ log\ n)$$

4 Structure de données élémentaires

4.1 Pile

4.1.1 Description

- Ensemble dynamique d'objets accessibles selon une discipline LIFO ("Last-in first-out").
- Interface
  - STACK-EMPTY(S) renvoie vrai si et seulement si la pile est vide
  - PUSH(S, x) pousse la valeur x sur la pile S
  - POP(S) extrait et renvoie la valeur sur le sommet de la pile S
- Implémentations :
  - avec un tableau (taille fixée a priori)
  - avec une liste liée (allouée de manière dynamique)
  - ...
- Applications : vérification de l'appariement de parenthèses dans une chaine de caractères

4.1.2 Implémentation par un tableau

- S est un tableau qui contient les éléments de la pile
- $S.top$ est la position courante de l'élément au sommet de S

```
PUSH(S, x)
1  if S.top == S.length
2        error "overflow"
3  S.top = S.top + 1
4  S[S.top] = x
```

```
STACK-EMPTY(S)
1  return S.top == 0
```

```
POP(S)
1  if STACK-EMPTY(S)
2        error "underflow"
3  else S.top = S.top − 1
4        return S[S.top + 1]
```

- Complexité en temps et *en espace* : $O(1)$
  (Inconvénient : L'espace occupé ne dépend pas du nombre d'objets)

4.2 File simple

4.2.1 Description

- Ensemble dynamique d'objets accessibles selon une discipline FIFO ("First-in first-out").
- Interface
  - ENQUEUE(Q, s) ajoute l'élément x à la fin de la file Q
  - DEQUEUE(Q) retire l'élément à la tête de la file Q
- Implémentation
  - A l'aide d'un tableau circulaire
  - A l'aide d'une liste liée

4.2.2 Implémentation à l'aide d'une liste liée

- Q est une liste simplement liée
- $Q.head$ (resp. $Q.tail$) pointe vers la tête (resp. la queue) de la liste

```
ENQUEUE(Q,x)
1  x.next = NIL
2  if Q.head == NIL
3        Q.head = x
4  else Q.tail.next = x
5  Q.tail = x
```

```
DEQUEUE(Q)
1  if Q.head == NIL
2        error "underflow"
3  x = Q.head
4  Q.head = Q.head.next
5  if Q.head == NIL
6        Q.tail = NIL
7  return x
```

- Complexité en temps $O(1)$, complexité en espace $O(n)$ pour n opérations

4.3 File double

4.3.1 Description

Combine accès LIFO et FIFO.Double ended-queue (dequeue).

- Généralisation de la pile et de la file
- Collection ordonnée d'objets offrant la possibilité
  - d'insérer un nouvel objet avant le premier ou après le dernier
  - d'extraire le premier ou le dernier objet
- Interface :
  - INSERT-FIRST(Q, x) : ajoute x au début de la file double
  - INSERT-LAST(Q, x) : ajoute x à la fin de la file double
  - REMOVE-FIRST(Q) : extrait l'objet situé en première position
  - REMOVE-LAST(Q) : extrait l'objet situé en dernière position
  - ...
- Application : équilibrage de la charge d'un serveur

4.3.2 Implémentation à l'aide d'une liste doublement liée

Soit la file double Q :

- Q.head pointe vers un élément sentinelle en début de liste
- Q.tail pointe vers un élément sentinelle en fin de liste
- Q.size est la taille courante de la liste

INSERT-FIRST(Q, x)

1 $x.prev = Q.head$
2 $x.next = Q.head.next$
3 $Q.head.next.prev = x$
4 $Q.head.next = x$
5 $Q.size = Q.size + 1$

INSERT-LAST(Q, x)

1 $x.prev = Q.tail.prev$
2 $x.next = Q.tail$
3 $Q.tail.prev.next = x$
4 $Q.head.prev = x$
5 $Q.size = Q.size + 1$

```
REMOVE-FIRST(Q)
1  if (Q.size == 0)
2        error
3  x = Q.head.next
4  Q.head.next = Q.head.next.next
5  Q.head.next.prev = Q.head
6  Q.size = Q.size − 1
7  return x
```

```
REMOVE-LAST(Q)
1  if (Q.size == 0)
2        error
3  x = Q.tail.prev
4  Q.tail.prev = Q.tail.prev.prev
5  Q.tail.prev.next = Q.head
6  Q.size = Q.size − 1
7  return x
```

Complexité $O(1)$ en temps et $O(n)$ en espace pour n opérations.

4.4 Liste

4.4.1 Description

- Ensemble dynamique d'objets ordonnés accessibles relativement les uns aux autres, sur base de leur position
- Généralise toutes les structures vues précédemment
- Interface :
  - Les fonctions d'une liste double (insertion et retrait en début et fin de liste)
  - INSERT-BEFORE(L, p, x) : insére x avant p dans la liste
  - INSERT-AFTER(L, p, x) : insère x après p dans la liste
  - REMOVE(L, p) : retire l'élement à la position p
  - REPLACE(L, p, x) : remplace par l'objet x l'objet situé à la position p
  - FIRST(L), LAST(L) : renvoie la première, resp. dernière position dans la liste
  - PREV(L, p), NEXT(L, p) : renvoie la position précédant (resp. suivant) p dans la liste
- Implémentation similaire à la file double, à l'aide d'une liste doublement liée (avec sentinelles)

4.4.2 Implémentation à l'aide d'une liste doublement liée

INSERT-BEFORE(L, p, x)

```
1  x.prev = p.prev
2  x.next = p
3  p.prev.next = x
4  p.prev = x
5  L.size = L.size + 1
```

REMOVE(L, p)

```
1  p.prev.next = p.next
2  p.next.prev = p.prev
3  L.size = L.size − 1
4  return p
```

INSERT-AFTER(L, p, x)

```
1  x.prev = p
2  x.next = p.next
3  p.next.prev = x
4  p.next = x
5  L.size = L.size + 1
```

Complexité $O(1)$ en temps et $O(n)$ en espace pour n opérations.

4.5 Vecteur

4.5.1 Description

- Ensemble dynamique d'objets occupant des rangs entiers successifs, permettant la consultation, le remplacement, l'insertion et la suppression d'éléments à des rangs arbitraires
- Interface
  - ELEM-AT-RANK(V, r) retourne l'élément au rang r dans V.
  - REPLACE-AT-RANK(V, r, x) remplace l'élément situé au rang r par x et retourne cet objet.
  - INSERT-AT-RANK(V, r, x) insère l'élément x au rang r, en augmentant le rang des objets suivants.
  - REMOVE-AT-RANK(V, r) extrait l'élément situé au rang r et le retire de r, en diminuant le rang des objets suivants.
  - VECTOR-SIZE(V) renvoie la taille du vecteur.
- Applications : tableau dynamique, gestion des éléments d'un menu,...
- Implémentation : liste liée, tableau extensible...

4.5.2 Implémentation par un tableau extensible

- Les éléments sont stockés dans un tableau extensible $V.A$ de taille initiale $V.c$.
- En cas de dépassement, la capacité du tableau est doublée.
- $V.n$ retient le nombre de composantes.
- Insertion et suppression :

INSERT-AT-RANK(V, r, x)

```
1   if V.n == V.c
2       V.c = 2 · V.c
3       W = "Tableau de taille V.c"
4       for i = 1 to n
5           W[i] = V.A[i]
6       V.A = W
7   for i = V.n downto r
8       V.A[i + 1] = V.A[i]
9   V.A[r] = x
10  V.n = V.n + 1
```

REMOVE-AT-RANK(V, r)

```
1  tmp = V.A[r]
2  for i = r to V.n − 1
3      V.A[i] = V.A[i + 1]
4  V.n = V.n − 1
5  return tmp
```

Complexité en temps

- Insert-At-Rank :
  - $O(n)$ pour une opération individuelle, où n est le nombre de composantes du vecteur
  - $\Theta(n^2)$ pour n opérations d'insertion en début de vecteur -> coût amorti par opération est $\Theta(n)$
  - $\Theta(n)$ pour n opérations d'insertion en fin de vecteur -> coût amorti par opération est $\Theta(1)$
- Remove-At-Rank :
  - $O(n)$ pour une opération individuelle, où n est le nombre de composantes du vecteur
  - $\Theta(n^2)$ pour n opérations de retrait en début de vecteur -> coût amorti par opération est $\Theta(n)$
  - $\Theta(n)$ pour n opérations de retrait en fin de vecteur -> coût amorti par opération est $\Theta(1)$

Complexité en espace $Theta(n)$

4.6 File à priorité

4.6.1 Description

- Ensemble dynamique d'objets classés par ordre de priorité
  - Permet d'extraire un objet possédant la plus grande priorité
  - En pratique, on représente les priorités par les clés
  - Suppose un ordre total défini sur les clés
- Interface :
  - INSERT(S, x) : insère l'élément x dans S.
  - MAXIMUM(S) : renvoie l'élément de S avec la plus grande clé.
  - EXTRACT-MAX(S) : supprime et renvoie l'élément de S avec la plus grande clé.
- Remarques :
  - Extraire l'élément de clé maximale ou minimale sont des problèmes équivalents
  - La file FIFO est une file à priorité où la clé correspond à l'ordre d'arrivée des élements.
- Application : gestion des jobs sur un ordinateur partagé

4.6.2 Implémentation à l'aide d'un tas

- Accès et extraction du maximum :

```
HEAP-MAXIMUM(A)
1  return A[1]

HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2        error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1) // reconstruit le tas
7  return max
```

- Complexité : $\Theta(1)$ et $O(\log n)$ respectivement (voir chapitre 3)
- INCREASE-KEY(S, x, k) augmente la valeur de la clé de x à k (on suppose que $k \geq$ à la valeur courante de la clé de x).

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2        error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5        swap(A[i], A[PARENT(i)])
6        i = PARENT(i)
```

- Complexité : $O(\log n)$ (la longueur de la branche de la racine à i étant $O(\log n)$ pour un tas de taille n).
- Pour insérer un élément avec une clé key :
  - l'insérer à la dernière position sur le tas avec une clé $-\infty$,
  - augmenter sa clé de $-\infty$ à key en utilisant la procédure précédente

HEAP-INSERT(A, key)
1 $A.heap\text{-}size = A.heap\text{-}size + 1$
2 $A[A.heap\text{-}size] = -\infty$
3 HEAP-INCREASE-KEY$(A, A.heap\text{-}size, key)$

- Complexité : $O(\log n)$.

$\Rightarrow$ Implémentation d'une file à priorité par un tas : $O(\log n)$ pour l'extraction et l'insertion.

4.7 Montrez que les parcours infixe, préfixe et postfixe sont $\Theta(n)$.

Tous les parcours en profondeur sont $\Theta(n)$ en temps. En effet :

- Soit $T(n)$ le nombre d'opérations pour un arbre avec n nœuds
- On a $T(n) = \Omega(n)$ (on doit au moins parcourir chaque nœud).
- Etant donné la récurrence, on a :

$$T(n) \leq T(n_L) + T(n - n_L - 1) + d$$

où n_L est le nombre de nœuds du sous-arbre à gauche et d une constante

- On peut prouver par induction que : $T(n) \leq (c + d)n + c$, où $c = T(0)$ [2].
- $T(n) = \Omega(n)$ et $T(n) = O(n) \Rightarrow T(n) = \Theta(n)$

4.8 Montrez que le coût amorti d'une opération d'insertion à la fin d'un vecteur extensible dont on double la taille est $O(1)$ et qu'elle serait $\Theta(n)$ si on augmentait cette taille d'une constante c.

- Si la capacité du tableau passe de c_0 à $2^k c_0$ au cours des n opérations d'insertion en fin de vecteur, alors le coût des transferts entre tableaux s'élève à :

$$c_0 + 2c_0 + ... + 2^{k-1}c_0 = (2^k - 1)c_0$$

Puisque $2^{k-1}c_0 < n \leq 2^k c_0$, ce coût est $\Theta(n)$.

- On dit que le coût amorti par opération est $O(1)$.

4.9 Étant donné un problème algorithmique, pouvoir déterminer quelle structure est la plus appropriée.

2. Cas de base : $n = 0$ ok car $T(0) = c \leq c$
Cas inductif : Pour tout $m \leq n : T(m) \leq (c + d)m + c \Rightarrow T(n) \leq (c + d)n + c$
Demo : $T(n) = T(n_L) + T(n - n_L - 1) \leq (c + d)n_L + c + (c + d)(n - n_L - 1) + c + d \leq (c + d)n + c$

5 Dictionnaires

5.1 Qu'est-ce qu'un dictionnaire (principe, interface, exemples d'application) ? Énumérez au moins 4 manières de l'implémenter en précisant la complexité des opérations principales.

- Définition : un dictionnaire est un ensemble dynamique d'objets avec des clés comparables qui supportent les opérations suivantes :
  - SEARCH(S, k) retourne un pointeur x vers un élément dans S tel que $x.key = k$, ou NIL si un tel élément n'appartient pas à S.
  - INSERT(S, x) insère l'élément x dans le dictionnaire S. Si un élément de même clé se trouve déjà dans le dictionnaire, on met à jour sa valeur
  - DELETE(S, x) retire l'élément x de S. Ne fait rien si l'élément n'est pas dans le dictionnaire.
- Deux objectifs en général :
  - minimiser le coût pour l'insertion et l'accès au données
  - minimiser l'espace mémoire pour le stockage des données
- Exemples d'applications :
  - Table de symboles dans un compilateur
  - Table de routage d'un DNS
  - ...
- Précisons 4 manières de l'implémenter :

	Pire cas			*En moyenne*		
Implémentation	SEARCH	INSERT	DELETE	SEARCH	INSERT	DELETE
Liste	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Vecteur trié	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
ABR	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
AVL	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

5.2 Comment peut-on implémenter un dictionnaire à l'aide d'un vecteur ?

- On suppose qu'il existe un ordre total sur les clés
- On stocke les éléments dans un vecteur qu'on maintient trié
- Recherche dichotomique (approche "diviser-pour-régner")

```
BINARY-SEARCH(V, k, low, high)
1  if low > high
2       return NIL
3  mid = ⌊(low + high)/2⌋
4  x = ELEM-AT-RANK(V, mid)
5  if k == x.key
6       return x
7  elseif k > x.key
8       return BINARY-SEARCH(V, k, mid + 1, high)
9  else return BINARY-SEARCH(V, k, low, mid − 1)
```

Complexité au pire cas : $\Theta(\log n)$

- Insertion : recherche de la position par BINARY-SEARCH puis insertion dans le vecteur par INSERT-AT-RANK (=décalage des éléments vers la droite). Complexité au pire cas de $\Theta(N)$.
- Suppression : recherche puis suppression par REMOVE-AT-RANK (=décalage des éléments vers la gauche). Complexité au pire cas de $\Theta(N)$.

5.3 Qu'est-ce qu'un arbre binaire de recherche ?

- Une structure d'arbre binaire implémentant un dictionnaire, avec des opérations en $O(h)$ où h est la hauteur de l'arbre
- Chaque nœud de l'arbre binaire est associé à une clé
- L'arbre satisfait à la propriété d'arbre binaire de recherche
  - Soient deux nœuds x et y.
  - Si y est dans le sous-arbre de gauche de x, alors $y.key < x.key$
  - Si y est dans le sous-arbre de droite de x, alors $y.key \geq x.key$

5.4 Comment peut-on trier avec un arbre binaire de recherche ? Comparez cet algorithme aux autres algorithmes de tri vu au cours.

```
BINARY-SEARCH-TREE-SORT(A)
1  T = "Empty binary search tree"
2  for i = 1 to n
3       TREE-INSERT(T, A[i])
4  INORDER-TREE-WALK(T.root)
```

- Complexité en temps identique au Quicksort
  - Insertion : en moyenne, $n \cdot O(\log n) = O(n \log n)$, pire cas : $\Theta(n^2)$
  - Parcours de l'arbre en ordre : $\Theta(n)$
  - Total : $\Theta(n \log n)$ en moyenne, $\Theta(n^2)$ pour le pire cas
- Complexité en espace cependant plus importante, pour le stockage de la structure d'arbres.

5.5 Qu'est-ce qu'une table de hachage ? Décrivez les opérations d'insertion et de suppression et donnez leur complexité.

- Idée :
  - Utiliser une table T de taille $m \ll |U|$ où $U = \{0, 1, ..., m-1\}$ correspond à l'univers associé aux clés de chaque élément.

– stocker x à la position $h(x.key)$, où h est une fonction de hachage :

$$h : U \to \{0, \ldots, m-1\}$$

CHAINED-HASH-INSERT(T, x)
1 LIST-INSERT$(T[h(x.key)], x)$

CHAINED-HASH-DELETE(T, x)
1 LIST-DELETE$(T[h(x.key)], x)$

- Complexité :
  - Insertion : $O(1)$
  - Suppression : $O(1)$ si liste doublement liée, $O(n)$ pour une liste de taille n si liste simplement liée.

Dans le cas de résolution des collisions par chainage.

5.6 Qu'est-ce qu'une collision ? Qu'est-ce que le facteur de charge ? Expliquez le principe des deux méthodes principales de gestion des collisions, donnez leurs complexités (sans preuve) et comparez les.

- Collision : Lorsque deux clés distinctes k_1 et k_2 sont telles que $h(k_1) = h(k_2)$
- Le facteur de charge d'une table de hachage est donné par $\alpha = \frac{n}{m}$ où :
  - n est le nombre d'éléments deans la table
  - m est la taille de la table(c'est-à-dire, le nombre de listes liées)
- Il existe deux grandes façons de gèrer les collisions :
  1. Le chaînage : mettre les éléments qui sont hachés vers la même position dans une liste liée(simple ou double)
     - Insertion : $\Theta(1)$ dans tous les cas
     - Suppression : $\Theta(1)$ en moyenne, $O(n)$ dans le pire cas.
     - Recherche : $\Theta(1)$ en moyenne, $O(n)$ dans le pire cas.
  2. Adressage ouvert : Stocke tous les éléments dans le tableau, lorsqu'il y a collision on sonde le tableau pour trouver une case vide. On fera en sorte de toujours avoir α inférieur à 1.
     - Insertion : $\Theta(1)$ si α est constant.
     - Suppression : $\Theta(1)$ si α est constant.
     - Recherche : $\Theta(1)$ si α est constant.
- Comparaison entre l'adressage ouvert et le chainage
  - Chaînage :
    - Peut gérer un nombre illimité d'éléments et de collisions
    - Performances plus stables
    - Surcoût lié à la gestion et le stockage en mémoire des listes liées
  - Adressage ouvert :
    - Rapide et peu gourmand en mémoire
    - Choix de la fonction de hachage plus difficile (pour éviter les grappes)
    - On ne peut pas avoir $n > m$
    - Suppression problématique

5.7 Qu'est-ce que l'adressage ouvert ? Décrivez les différentes fonctions de sondages.

L'adressage ouvert est une alternative au chaînage pour gérer les collisions. Tous les éléments sont stockés dans le tableau (pas de listes chainées).

Pour insérer une clé k, on sonde les cases systématiquement à partir de $h(k)$ jusqu'à en trouver une vide.

Notons que l'adressage ouvert ne fonctionne que si $\alpha \leq 1$.

Pour une séquence de sondage $h_k =< h(k,0), h(k,1), ..., h(k,m-1) >$ correspondant à la clé k, il existe différentes méthodes en fonction de la stratégie de sondage :

1. Sondage linéaire
$$h(k,i) = (h'(k) + i) \bmod m$$
où $h'(k)$ est une fonction de hachage ordinaire à valeurs dans $\{0, 1, \ldots, m-1\}$.

   Propriétés :

   - très facile à implémenter.
   - effet de grappe fort : création de longues suites de cellules occupées
     - La probabilité de remplir une cellule vide est $\frac{i+1}{m}$ où i est le nombre de cellules pleines précédant la cellule vide.
   - pas très uniforme.

2. Sondage quadratique
$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$
où h' est une fonction de hachage ordinaire à valeurs dans $\{0, 1, \ldots, m-1\}$, c_1 et c_2 sont deux constantes non nulles.

   Propriétés :

   - nécessité de bien choisir les constantes c_1 et c_2 (pour avoir une permutation de $\langle 0, 1, \ldots, m-1 \rangle$).
   - effet de grappe plus faible mais tout de même existant :
     - Deux clés de même valeur de hachage suivront le même chemin
$$h(k,0) = h(k',0) \Rightarrow h(k,i) = h(k',i)$$
   - meilleur que le sondage linéaire.

3. Double hachage
$$h(k,i) = (h_1(k) + i.h_2(k)) \bmod m$$
où h_1 et h_2 sont des fonctions de hachage ordinaires à valeurs dans $\{0, 1, \ldots, m-1\}$.

   Propriétés :

   - difficile à implémenter à cause du choix de h_1 et h_2 ($h_2(k)$ doit être premier avec m pour avoir une permutation de $< 0, 1, ..., m-1 >$).
   - très proche du hachage uniforme.
   - bien meilleur que les sondages linéaires et quadratiques.

5.8 Qu'est-ce qu'une fonction de hachage ? Quelles sont les caractéristiques d'une bonne fonction de hachage ? Comment traiter des clés non numériques ?

- Fonction de hachage : fonction d'une clé qui renvoit une position dans le tableau pour cette clé.
- Fonction de hachage idéale :
  - Doit être facile à calculer : $O(1)$.
  - Doit satisfaire l'hypothèse de hachage uniforme simple.
- Pour traiter des clés non numériques, on utilise une fonction de codage.

5.9 Décrivez deux exemples de familles de fonctions de hachage

1. Méthode de division
   - La fonction de hachage calcule le reste de la division entière de la clé par la taille de la table
     $$h(k) = k \bmod m$$
     Exemple : $m = 20$ et $k = 91 \Rightarrow h(k) = 11$.

     Avantages : simple et rapide (juste une opération de division)

     Inconvénients : Le choix de m est très sensible et certaines valeurs doivent être évitées
   - Si la fonction de hachage produit des séquences périodiques, il vaut mieux choisir m premier
   - En effet, si m est premier avec b, on a :
     $$\{(a + b \cdot i) \bmod m | i = 0, 1, 2, \ldots\} = \{0, 1, 2, \ldots, m - 1\}$$

   $\Rightarrow$ Bonne valeur de m : un nombre premier pas trop près d'une puissance exacte de 2

2. Méthode de multiplication
   - Fonction de hachage :
     $$h(k) = \lfloor m \cdot (kA \bmod 1) \rfloor$$
     où
     - A est une constante telle que $0 < A < 1$.
     - $kA \bmod 1 = kA - \lfloor kA \rfloor$ est la partie fractionnaire de kA.

     Avantages : la valeur de m n'est plus critique

     Inconvénients : plus lente que la méthode de division
   - La méthode marche mieux pour certaines valeurs de A. Par exemple :
     $$A = \frac{\sqrt{5} - 1}{2}$$

5.10 Comparez les arbres binaires de recherche et les tables de hachage en énumérant leurs avantages et défauts respectifs.

Tables de hachage :

- Faciles à implémenter
- Seule solution pour des clés non ordonnées
- Accès et insertion très rapides en moyenne (pour des clés simples)
- Espace gaspillé lorsque α est petit
- Pas de garantie au pire cas (performances “instables”)

Arbres binaire de recherche (équilibrés) :

- Performance garantie dans tous les cas (stabilité)
- Taille de structure s’adapte à la taille des données
- Supportent des opérations supplémentaires lorsque les clés sont ordonnées (parcours en ordre, successeur, prédécesseur, etc.)
- Accès et insertion plus lente en moyenne

6 Résolution de problèmes

6.1 Pour chaque technique de programmation (force brute, diviser-pour-régner, programmation dynamique et algorithme glouton) : expliquez le principe de la technique et donnez un exemple de problème et sa solution.

6.1.1 Force brute

Principe

Résoudre directement le problème à partir de sa définition ou par une recherche exhaustive. On va donc appliquer la solution la plus directe à un problème, généralement en appliquant à la lettre la définition du problème.

Exemples : calculer a^n en multipliant a n fois avec lui-même, parcourir un tableau linéairement à la recherche d'un élément.

Exemple pour le tri :

- Générer toutes les permutations du tableau de départ (une et une seule fois)
- Vérifier si chaque tableau permuté est trié. S'arrêter si c'est le cas.
- Complexité : $O(n! \cdot n)$

Avantages :

- Simple et d'application très large
- Un bon point de départ pour trouver de meilleurs algorithmes
- Parfois, faire mieux n'en vaut pas la peine

Inconvénients :

- Produit rarement des solutions efficaces
- Moins élégant et créatif que les autres techniques

6.1.2 Diviser pour régner

Principe

Si le problème est trivial, on le résout directement. Sinon :

- Diviser le problème en sous-problèmes de taille inférieure (Diviser)
- Résoudre récursivement ces sous-problèmes (Régner)
- Fusionner les solutions aux sous-problèmes pour produire une solution au problème original

Exemple : MergeSort

- Diviser : Couper le tableau en deux sous-tableaux de même taille
- Régner : Trier récursivement les deux sous-tableaux
- Fusionner : fusionner les deux sous-tableaux
- Complexité : $\Rightarrow O(n \log n) \qquad (force\ brute \Rightarrow (n^2))$

Exemple plus concret : recherche de pics dans un tableau

Soit un tableau A ; définition d'un pic : $A[i]$ est un pic s'il n'est pas plus petit que ses voisins :

$$A[i-1] \leq A[i] \geq A[i+1]$$

Approche par force brute : tester toutes les positions séquentiellement.
Complexité dans le pire cas : $O(n)$

```
PEAK1D(A)
1  for i = 1 to A.length
2       if A[i − 1] ≤ A[i] ≥ A[i + 1]
3            return i
```

Meilleure idée : diviser pour régner.
On va sonder un élément $A[i]$ et ses voisins $A[i-1]$ et $A[i+1]$, si c'est un pic on renvoie i, sinon :

- les valeurs doivent croitre au moins d'un côté $A[i-1] > A[i]$ ou $A[i] < A[i+1]$
- Si $A[i-1] > A[i]$, on cherche le pic dans $A[1...i-1]$
- Si $A[i+1] > A[i]$, on cherche le pic dans $A[i+1...A.length]$

```
PEAK1D(A, p, r)
1  q = (p + r)/2
2  if A[q − 1] ≤ A[q] ≥ A[q + 1]
3       return q
4  elseif A[q − 1] > A[q]
5       return PEAK1D(A, p, q − 1)
6  elseif A[q] < A[q + 1]
7       return PEAK1D(A, q + 1, r)
```

Complexité : T(n) = $O(\log n)$.

Résumé

Mène à des algorithmes très efficaces mais qui ne sont pas toujours applicable, mais quand même très utile.

Applications :

- Tris optimaux
- Recherche binaire
- Problème de sélection
- Trouver la paire de points les plus proches
- Recherche de l'enveloppe convexe (conver-hull)
- Multiplication de matrices (Méthode de Strassens)

Notons qu'il y a beaucoup d'autres exemples importants dans le cours, notamment : recherche d'un max dans un tableau 2D, achat/vente d'actions.

6.1.3 Programmation dynamique

Principe

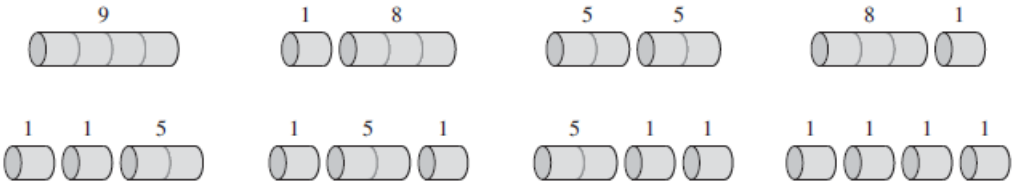
Obtenir la solution optimale à un problème en combinant des solutions optimales a des sous-problèmes similaires plus petits et se chevauchant.

Exemple 1 : découpage de tiges d'acier

Soit une tige d'acier qu'on découpe pour la vendre morceau par morceau. La découpe ne peut se faire que par nombre entier de centimètre et le prix de vente d'une tige dépend (mais non linéairement) de sa longueur. On veut donc déterminer le revenu maximum qu'il peut atteindre avec la tige à n cm. Soit la table de prix :

Longueur i	1	2	3	4	5	6	7	8	9	10
Prix p_i	1	5	8	9	10	17	17	20	24	30

Découpes possibles d'une tige de longueur $n = 4$:



On peut voir que le meilleur revenu est obtenu lorsqu'on découpe la tige en 2 tiges de 2 centimètres, ce qui nous donne un revenu de 10.

L'**approche par force brute** est d'énumérer toutes les découpes, de calculer leur revenu et de déterminer le revenu maximal. Or, la complexité est exponentielle ce qui rend cette approche infaisable pour un n un peu grand. En effet, il y a 2^{n-1} manières de découper une tige de longueur n.

Approche récursive naïve

Soit r_i le revenu maximal pour une tige de longueur i, formulons ceci de manière récursive :

i	r_i	solution optimale
1	1	1 (pas de découpe)
2	5	2 (pas de découpe)
3	8	3 (pas de découpe)
4	10	2+2
5	13	2+3
6	17	6 (pas de découpe)
7	18	1+6 ou 2+2+3
8	22	2+6 . . .

Ainsi, r_n peut être calculé naïvement comme le maximum de :

- p_n : le prix sans découpe

- $r_1 + r_{n-1}$: le revenu max pour une tige de 1 et une tige de $n-1$
- $r_2 + r_{n-2}$: le revenu max pour une tige de 2 et une tige de $n-2$
- ...
- $r_{n-1} + r_1$

$$\Rightarrow r_n = max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, ..., r_{n-1} + r_1)$$

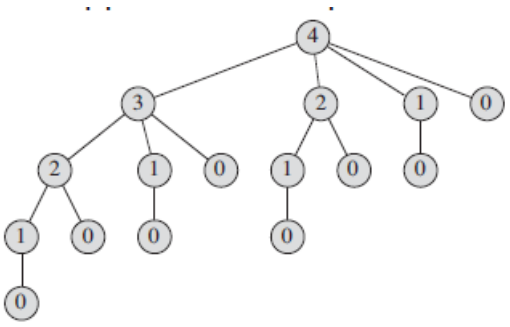
Une version plus directe est la suivante : on peut calculer r_n en considérant toutes les tailles pour la première découpe et en combinant avec le découpage optimal pour la partie de droite. Ainsi, pour chaque cas on ne doit résoudre qu'un seul sous problème : le découpage de la partie de droite . On a ainsi :

$$r_n = max(p_i + r_{n-1}) \qquad avec\ 1 \leq i \leq n$$

On peut l'implémenter comme ceci :

```
CUT-ROD(p, n)
1  if n == 0
2        return 0
3  q = -∞
4  for i = 1 to n
5        q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

Bien qu'on ai formulé l'approche récursivement, la complexité est toujours exponentielle en n (en général, nb. de nœuds d'un arbre est T(n) = 2^n) car on fait beaucoup d'appels récursif redondants. Ci-dessous on trouve l'arbre des appels récursifs pour le calcul de r_n



La solution nous vient alors grâce à la programmation dynamique : plutôt que de résoudre les mêmes sous problèmes plusieurs fois, pourquoi ne pas les résoudre qu'une seule fois et sauvegarder les solutions dans une table et ensuite se référer à la table à chaque demande de résolution d'un sous problème déjà rencontré. Cela permet de transformer une solution d'une complexité exponentielle en une solution à complexité polynomiale. On a donc 2 implémentations possible : la méthode descendante (top-down) avec mémoization et ascendante (bottom-up).

Approche ascendante

Principe : résoudre les sous-problèmes par taille en commençant d'abord par les plus petits.
On peut étendre l'approche ascendante pour enregistrer également la solution dans une autre table.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  Let r[0..n] and s[1..n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n
 4        q = −∞
 5        for i = 1 to j
 6              if q < p[i] + r[j − i]
 7                    q = p[i] + r[j − i]
 8                    s[j] = i
 9        r[j] = q
10  return r and s
```

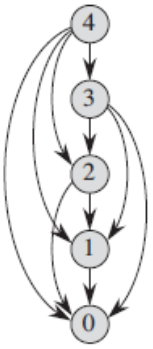
où $s[j]$ contient la coupure la plus la gauche d'une solution optimale au problème de taille j.

Approche descendante

```
MEMOIZED-CUT-ROD(p, n)
1  Let r[0..n] be a new array
2  for j = 1 to n
3        q = −∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n)
1  if r[n] ≥ 0
2        return r[n]
3  if n == 0
4        q = 0
5  else q = −∞
6        for i = 1 to n
7              q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

La solution ascendante est clairement de complexité $\theta(n^2)$ car on a deux boucles imbriquées. La solution descendante l'est également, chaque sous problème est résolu une et une seule fois et la résolution d'un sous-problème passe par une boucle à n itérations. Ci dessous est illustré le graphe des sous problèmes (une flèche de x a y indique que la résolution de x dépend de la résolution de y) :



Une multitude d'exemples supplémentaire sont présent dans le cours du Pr.Geurts, notamment la suite de fibonacci, la recherche d'une sous séquence maximale, la recherche d'une sous-séquence commune, le problème du sac à dos, etc...

6.1.4 Algorithme glouton

Principe

Construire la solution incrémentalement en optimisant de manière aveugle un critère local. Ces algorithmes sont utilisés pour résoudre des problèmes d'optimisation (comme la programmation dynamique).

L'idée principale est que quand on a un choix local à faire, faire le choix (glouton) qui semble le meilleur tout de suite (et ne jamais le remettre en question). Pour que l'approche fonctionne, le problème doit satisfaire deux propriétés :

- Propriété des choix gloutons optimaux : On peut toujours arriver à une solution optimale en faisant un choix localement optimal
- Propriété de sous-structure optimale : Une solution optimale du problème est composée de solutions optimales à des sous-problèmes

Exemple 1 : rendre la monnaie

Objectif : Étant donné des pièces de 1, 2, 5, 10, et 20 cents, trouver une méthode pour rembourser une somme de x cents en utilisant le moins de pièces possible.

Algorithme de la caissière : A chaque itération, ajouter une pièce de la plus grande valeur qui ne dépasse pas la somme restant à rembourser.

Exemple : 49 cents {20 ; 20 ; 5 ; 2 ; 2} (5 pièces)

Cela permet-t-il de trouver une solution optimale ? L'approche greedy n'est correcte que pour certains choix particuliers de valeurs de pièces.

```
CoinchainingGreedy(x, c, n)
1  //c[1..n] contains the n coin values in decreasing order
2  Let s[1..n] be a new table
3  // s[i] is the number of i-th coin in solution
4  CoinCount = 0
5  for i = 1 to n
6        s[i] = floor(x/c[i])
7        x = x - s[i]*c[i]
8        CoinCount = CoinCount + s[i]
9  return (s,CoinCount)
```

Contre-exemple : C = [1 ; 10 ; 21 ; 34 ; 70 ; 100] (valeurs de timbres aux USA) et x = 140

- Algorithme glouton : {100 ; 34 ; 1 ; 1 ; 1 ; 1 ; 1 ; 1}
- Solution optimale : {70 ; 70}

Pour résoudre le cas général, il faut recourir à la programmation dynamique.

Résumé

Les algorithmes gloutons sont très efficaces quand ils fonctionnent et sont simples et faciles à implémenter. Par contre, ils ne fonctionnent pas toujours et leur corrections peut être difficile à prouver.

6.2 Quelles sont les deux conditions nécessaires pour pouvoir appliquer la programmation dynamique ? Illustrez avec un exemple.

La programmation dynamique s'applique aux problèmes d'optimisation qui peuvent se décomposer en sous-problèmes de même nature, et qui possèdent les deux propriétés suivantes :

- Sous-structure optimale : on peut calculer la solution d'un problème de taille n à partir de la solution de sous-problèmes de taille inférieure.
- Certains sous-problèmes distincts partagent une partie de leurs sous-problèmes.

L'implémentation directe récursive donne une solution de complexité exponentielle. C'est pour ça qu'on sauvegarde des solutions des sous-problèmes, ce qui donne une complexité linéaire dans le nombre d'arc et de sommets du graphes des sous-problèmes.

6.3 Comment prouver qu'une approche gloutonne est correcte ? Illustrez en montrant que l'algorithme CoinChangingGreedy est optimal pour les pièces 1, 2, 5, 10 et 20.

Analyse de CoinChangingGreedy

Théorème : l'algorithme CoinChangingGreedy est optimal pour $c = [20, 10, 5, 2, 1]$

Preuve :

- Soit $S^*(x)$ l'ensemble optimal de pièces pour un montant x et soit c^* le plus grand $c[i] \leq x$. On doit montrer que :
  1. $S^*(x)$ contient c^* *(propriété des choix gloutons optimaux)*
  2. $S^*(x) = \{c^*\} \cup S^*(x - c^*)$ *(propriété de sous-structure optimale)*

- Propriété (2) découle directement de (1)
  - $S^*(x)$ contient c^* par (1)
  - Donc $S^*(x) \setminus \{c^*\}$ représente le change pour un montant de $x - c^*$
  - Ce change doit être optimal sinon $S' = \{c^*\} \cup S^*(x - c^*)$ serait une meilleure solution que $S^*(x)$ pour un montant de x
  - On a donc $S^*(x) = \{c^*\} \cup S^*(x - c^*)$

- Propriété (1) : $S^*(x)$ contient c^*
  - Avec $c = [20, 10, 5, 2, 1]$, une solution optimale ne contient jamais :
    - plus d'une pièce de 1, 5, ou de 10 (car $2 \times 1 = 2$, $2 \times 5 = 10$, $2 \times 10 = 20$)
    - plus de deux pièces de 2 (car $3 \times 2 = 5 + 1$)
  - Analysons les différents cas pour x :

$x = 1$: $c^* = 1$, meilleure solution $S^*(x) = \{1\}$ contient c^*
$2 \leq x < 5$: $c^* = 2$, avec un seul 1, on ne peut pas obtenir $x \Rightarrow c^* \in S^*(x)$
$x = 5$: $c^* = 5$, meilleure solution $S^*(x) = \{5\}$ contient c^*
$5 < x < 10$: $c^* = 5$, avec un seul 1, et deux 2, on ne peut pas obtenir $x \Rightarrow c^* \in S^*(x)$
$x = 10$: $c^* = 10$, meilleure solution $S^*(x) = \{10\}$ contient c^*
$10 < x < 20$: $c^* = 10$, avec un seul 1, deux 2, et 1 seul 5, on ne peut pas obtenir $x \Rightarrow c^* \in S^*(x)$
$x = 20$: $c^* = 20$, meilleure solution $S^*(x) = \{20\}$ contient c^*
$x > 20$: $c^* = 20$, avec un seul 1, deux 2, 1 seul 5 et 1 seul 10, on ne peut pas obtenir $x \Rightarrow c^* \in S^*(x)$

  - $S^*(x)$ contient donc toujours bien c^*

□

6.4 Comparez la programmation dynamique au diviser-pour-régner et à l'approche gloutonne.

Programmation dynamique vs diviser-pour-régner

L'approche Diviser-pour-régner décompose aussi le poblème en sous-problèmes, mais ces sous-problèmes sont significativement plus petits que le problème de départ ($n \to n/2$). Alors que la programmation dynamique réduit généralement un problème de taille n en sous problèmes de taille $n-1$.

De plus, en DPR , les sous-problèmes sont indépendants alors qu'il se recouvrent en programmation dynamique.

Pour ces deux raison, la récursivité ne marche pas pour la programmation dynamique.

Programmation dynamique vs approche gloutonne

Tous deux nécessitent la propriété de sous-structure optimale Les algorithmes gloutons ne nécessitent que la propriété de choix gloutons optimaux soit satisfaite

- On n'a pas besoin de solutionner plus d'un sous-problème
- Le choix glouton est fait avant de résoudre le sous-problème
- Il n'y a pas besoin de stocker les résultats intermédiaires

La programmation dynamique marche sans la propriété des choix gloutons optimaux

- On doit solutionner plusieurs sous-problèmes et choisir dynamiquement l'un deux pour obtenir la solution globale
- La solution doit être assemblée "Bottom-up"
- Les sous-solutions aux sous-problèmes sont réutilisées et doivent donc être stockées

7 Graphes

7.1 Qu'est-ce qu'un graphe ? Un graphe dirigé/non dirigé ? Acyclique ? Connexe ? Une composante connexe d'un graphe ? Le degré d'un graphe ? Un graphe pondéré ?

- Un graphe est un couple (V, E) où :
  - V est l'ensemble des noeuds (sommets) du graphe.
  - E est l'ensemble des arêtes (arcs) du graphique. Chaque élément de E est défini par 2 noeuds ($E \subseteq V \times V$).
- Graphe non dirigé :
  - les éléments de E définis par les sommets i et j de V, (i, j) et (j, i), sont égaux.
  - Deux noeuds sont adjacents si ils sont liés par une même arrête.
  - Une arrête $(v1, v2)$ est dites incidentes aux noeuds $v1$ et $v2$
  - le degré d'un noeud est égal au nombre de ses arrêtes incidentes.
  - le degré d'un graphe non dirigé est le degré du noeud de plus haut degré de ce graphe.

  Dans le cadre des graphes non dirigés, un graphe est connexe s'il existe un chemin de tout sommet à tout autre.
  Une composante connexe d'un graphe non orienté est définie comme un sous-graphe connexe maximal de ce graphe.
- Graphe dirigé :
  - Une arrête d'un graphe dirigé $(v1, v2)$ est formée de son origine $v1$ et de sa destination $v2$, $\forall v1, v2 \in V$. On dit également de l'arrête qu'elle est sortante pour $v1$ et rentrante pour $v2$.
  - Le degré entrant (in-degree) et le degré sortant (out-degree) sont respectivement égaux au nombre d'arêtes entrantes et sortantes.

  Dans le cadre des graphes dirigés, un graphe est acyclique si il n'est pas possible de suivre les arêtes du graphe à partir du sommet v et de revenir à ce même sommet v, $\forall v$.
- Un graphe est pondéré si les arêtes sont annotées par des poids. (Exemple : réseau entre villes avec la distance entre deux villes comme poids à chaque arête.

7.2 Quelles sont les deux manières principales de représenter un graphe ? Donnez les complexités en fonction de |V| et |E| des principales opérations dans les deux cas. Comparez les deux représentations. Donnez pour chaque représentation un algorithme pour laquelle elle est appropriée.

Représentation par listes adjacentes

Une structure G de type graphe est composé de :

- une liste G.V (de longueur |V|) des noeuds du graphe.
- un tableau G.Adj de |V| listes tel que :
  - Chaque sommet $u \in G.V$ est représenté par un élément du tableau G.Adj.
  - G.Adj[u] est la liste d'adjacence du sommet u, c'est à dire la liste des sommets v tel que $(u, v) \in E$

Cette méthode permet de représenter des graphes dirigés ou non : Si le graphe est dirigé (resp. non dirigé), la somme des longueurs des listes de G.Adj est |E| (resp 2|E|). Permet également de représenter un graphe pondéré en associant un poids à chaque élément des listes d'adjacence.

Opération	Complexité	Efficacité
Espace mémoire	$\Theta(\|V\| + \|E\|)$	optimal
Accéder à un sommet	$\Theta(1)$	optimal
Parcourir tout les sommets	$\Theta(\|V\|)$	optimal
Parcourir toutes les arêtes	$\Theta(\|V\| + \|E\|)$	ok
Vérifier l'existence d'une arête (u, v)	$O(\|V\|)$	mauvais

Représentation par matrice d'adjacence

Une autre représentation d'un graphe peut-être donnée par une structure contenant :

- un vecteur G.V des noeuds du graphe contenant |V| éléments.
- une matrice G.A de dimension $|V| \times |V|$ où G.A $= (a_{ij})$ tel que :

$$a_{ij} = \begin{cases} 1 & \text{si (i,j)} \in \text{E} \\ 0 & \text{sinon} \end{cases} \tag{1}$$

Permet de représenter des graphes orientés ou non (G.A est symétrique si non orienté). Pour un graphique pondérée, remplacer 1 par le poids lorsque l'arête existe.

Opération	Complexité	Efficacité
Espace mémoire	$\Theta(\|V\|^2)$	potentiellement très mauvais
Accéder à un sommet	$\Theta(1)$	optimal
Parcourir tout les sommets	$\Theta(\|V\|)$	optimal
Parcourir toutes les arêtes	$\Theta(\|V\|^2)$	potentiellement très mauvais
Vérifier l'existence d'une arête (u, v)	$\Theta(1)$	optimal

Comparaisons des deux méthodes

$$RAPPEL : \begin{cases} \text{Graphe creux :} & |E| \approx |V|^2 \\ \text{Graphe dense :} & |E| << |V^2| \end{cases} \tag{2}$$

- Listes d'adjacence :
  - Complexité en espace optimal.
  - Préférable pour graphe creux ou de faible degré.
  - A éviter si l'algorithme doit accéder régulièrement aux arêtes.
- Matrices d'adjacence :
  - Complexité très mauvaise en espace. En particulier si le graphe est creux, énormément de ressources mobilisées inutilement.
  - Préférable pour graphes denses.
  - Approprié pour des algorithmes désirant accéder aléatoirement aux arêtes.

7.3 Comment modifier les algorithmes de parcours pour parcourir tous les sommets d'un graphe ? Appliquez cette idée au parcours en largeur/profondeur.

Dans un parcours classique d'un arbre que ce soit en largeur (BFS) ou en profondeur (DFS), un noeud de référence est passé en argument. Ce noeud sert de point de départ au parcours de l'arbre. Il est alors possible que certains noeuds du graphe ne soit pas visité (dans le cas de graphes dirigés par exemple).
Pour palier à cela, il suffit d'enclencher le parcours à partir de chaque noeud de G.V qui n'a pas encore été visité par un parcours précédent. On peut alors construire un sous-graphe de liaison (une forêt : ensemble d'arbre) à partir d'un parcours.

- les sommets sont les sommets du graphe.
- un sommet u est le fils d'un sommet v dans la forêt si `DFS-rec(G, u)` est appelé depuis `DFS-rec(G, v)`

7.4 Qu'est-ce qu'un chemin ? Le poids d'un chemin ? Le plus court chemin entre deux sommets ?

- Soit un graphe dirigé $G = (V, E)$ et une fonction de poids $w : E \to \mathbb{R}$,
  Un chemin (du sommet v_1 au sommet v_k) est une séquence de noeuds $v_1, v_2, ..v_k$ telle que :
  $\forall i = 1, ..k - 1 : (v_i, v_{i+1}) \in E$
- Le poids (le coût) d'un chemin p est la somme du poids des arêtes qui le compose :

$$w(p) = w(v_1, v_2) + w(v_2, v_3)... + w(v_{k-1}, v_k) \tag{3}$$

- Un plus court chemin entre deux sommets u et v est un chemin p de u à v de poids w(p) le plus faible possible. On le note :

$$\delta(u, v) = min\{w(p) | \text{p est un chemin de u à v}\} \tag{4}$$

  S'il n'y a pas de chemin entre u et v, $\delta = \infty$ par définition.

  Propriété de sous-structure optimale :
  Tout sous-chemin d'un chemin le plus court est un chemin le plus court entre ses extrémités.

7.5 Comment gérer les cycles dans le cadre de la recherche d'un plus court chemin ?

Un chemin le plus court ne peut pas contenir de cycle

- De poids négatifs :
  Si un cycle négatif (somme aller-retour négative) est présent dans le graphe, il est possible de diminuer arbitrairement le coût de tout chemin passant par ce cycle.
  Par définition, on fixera $\delta(u, v) = -\infty$ s'il y a un chemin de u à v qui passe par un cycle négatif. Cependant il nous faudra exclure ceux-ci pour ne pas rentrer dans une boucle infinie.
- De poids positifs :
  Ceux-ci seront systématiquement éliminés car il ne contribue qu'à augmenter le poids de $\delta(u, v)$.
- De poids nuls :
  Aucune raison de passer par un cycle de poids nul, nous les éliminons également.

7.6 Expliquez le problème de recherche du plus court chemin à origine unique, présentez le schéma général d'un algorithme et expliquez le principe de relâchement. Donnez l'invariant maintenu par l'algorithme et montrez qu'il est bien vérifié.

Origine unique : trouver tous les plus courts chemins d'un sommet à tous les autres.

- Entrées : un graphe dirigé pondéré et un sommet s (origine).
- Sorties : deux attributs pour chaque sommet ($\forall v \in V$) :
  - $v.d = \delta(s, v)$ la plus courte distance de s vers chaque noeud.
  - $v.\pi$ = le prédécesseur de chaque sommet v dans un plus court chemin de s à v.

Schéma général de l'algorithme

- Objectif : calculer $v.d = \delta(s, v)$, $\forall v \in V$
- Idée d'algorithme :
  - Initialisation : $v.d = +\infty$ ($\forall v \in V$)
  - Invariant : $v.d \geq \delta(s, v)$
  - $v.d$ à une itération donnée contient une estimation du poids d'un plus court chemin de s à v.
  - A chaque itération, on tente de diminuer $v.d$ en maintenant l'invariant. L'amélioration est basé sur l'inégalité triangulaire. A la convergence, $v.d = \delta(s, v)$

Inégalité triangulaire et relâchement

Inégalité triangulaire

$\forall u, v, x \in V$, on a

$$\begin{aligned} &\delta(u, v) \leq \delta(u, x) + \delta(x, v) \\ &\text{D'où, } \forall (u, v) \in E : \delta(s, v) \leq \delta(s, u) + w(u, v) \end{aligned} \tag{5}$$

Relâchement

RELAX(u, v, w)
1 **if** $v.d > u.d + w(u, v)$
2 $\quad v.d = u.d + w(u, v)$

Preuve de l'invariant

- Cas de base : l'invariant est vérifier après l'initialisation
- Cas inductif :
  - Soit un appel à $relax(u, v, w)$
  - Avant l'appel, on suppose que l'invariant est vérifié et donc $u.d \geq \delta(s, u)$
  - Par l'inégalité triangulaire, on a

$$\begin{aligned} \delta(s, v) &\leq \delta(s, u) + \delta(u, v) \\ &\leq \delta(s, u) + w(u, v) \end{aligned} \tag{6}$$

  - Suite à l'assignation $v.d = u.d + w(u, v)$, on a bien : $v.d \geq \delta(s, v)$

7.7 Qu'est-ce qu'un arbre couvrant ? Un arbre couvrant de poids minimal ? Donnez un exemple d'application.

Définitions

- un arbre couvrant (spanning tree) pour un graphe connexe (V, E) non dirigé est un arbre (i.e. un graphe acyclique) T tel que :
  - l'ensemble des noeuds de T est égal à |V|
  - l'ensemble des arcs de T est un sous ensemble de E

  Il peut être construit par un BFS/DFS (graphe de liaison)
- un arbre couvrant de poids minimum (minimum spanning tree) pour un graphe pondéré connexe (V, E) est un arbre (V, E') tel que :
  - (V, E') est un arbre couvrant de (V, E)
  - la valeur de $\sum_{e \in E'} w(e)$ est minimal parmis tous les arbres couvrants de (V, E), où $w(e)$ dénote le poids de l'arc e.

Application

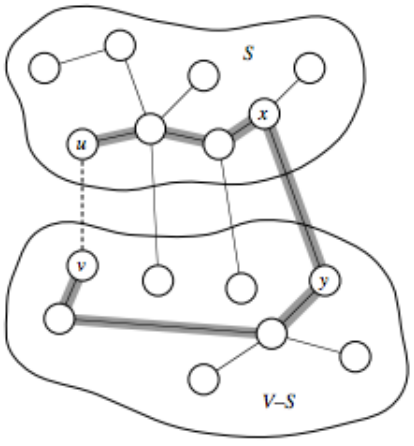
Conception de réseaux :

Connecter des entités en minimisant le coût de la connection.

- Raccorder des maisons à un central téléphonique en minimisant les longueurs de cables.
- Élaborer un système routier pour connecter des maisons.
- ...

7.8 Montrez que le choix d'une arête de poids minimal traversant une coupure est sûre et expliquez pourquoi cela prouve que les algorithmes de Kruskal et de Prim sont corrects.

- `Coupure (cut)`: (S, V \ S) est une partition des sommets en deux ensembles disjoints S et V \ S
- Une arête `traverse` (crosses) une coupure (S, V \ S) si une extrémité est dans S et l'autre dans V \ S
- Une `coupure` respecte A ssi il n'y a pas d'arête dans A qui traverse la coupure



Preuve :

Soit T un ACM qui inclut A.
Supposons que T ne contienne pas (u, v) et montrons qu'il est possible de construire un arbre T' qui inclut A $\cup$ (u, v). Puisque T est un arbre, il n'y a qu'un unique chemin p entre u et v et ce chemin traverse la coupure (S, V $\setminus$ S).
Soit (x, y) une arête de p qui traverse la coupure (S, V $\setminus$ S).
Puisque (u, v) est l'arête de poids minimum qui traverse la coupure, on a :

$$w(u, v) \leq w(x, y)$$

Puisque la coupure respecte A, l'arête (x, y) n'est pas dans A.
Soit T' = (T (x, y)) $\cup$ (u, v) :

- T' est un spanning tree

T' est donc bien un ACM tel que A $\cup$ (u, v) $\subseteq$ T' $\Rightarrow$ (u, v) est sûre pour A.

`Kruskal` est correct car :

- On connecte à chaque fois deux composantes connexes disjointes, le graphe reste acyclique et à la terminaison, on obtient un arbre couvrant
- On sélectionne une arête de poids minimal à chaque étape, le théorème précédent garantit qu'on arrivera à un ACM

Principe de `Prim` :

- A est toujours un arbre (plus une forêt)
- Initialisé comme une seule racine r choisie de manière arbitraire.
- A chaque étape, choisir une arête de poids minimal traversant la coupure (V_A, V $\setminus$ V_A), où V_A est l'ensemble des sommets connectés par des arêtes de A, et l'ajouter à A.

$\Rightarrow$ Toujours correct vu la preuve.