Stratégies de résolution de problèmes

Brute-Force

Brute-Force

- Consiste à appliquer la solution la plus directe à un problème

- Généralement obtenue en appliquant à la lettre la définition du problème

Brute-Force

- Exemple:
  - Rechercher un élément dans un tableau (trié ou non) en le parcourant linéairement
  - Calculer a^n en multipliant a n fois avec lui-même
- Souvent pas très efficace en terme de temps de calcul mais facile à implémenter et fonctionnel

Recherche exhaustive

- Une solution par force brute au problème de la recherche d'un élément possédant une propriété particulière

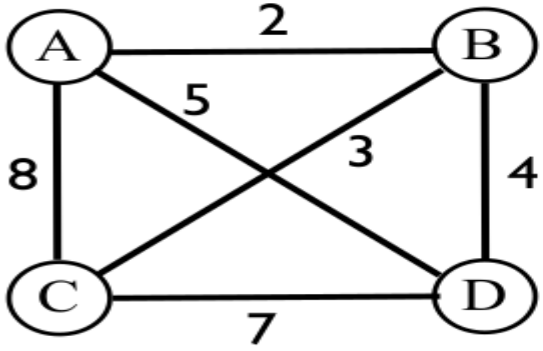
- Générer toutes les solutions possibles jusqu'à en obtenir une qui possède la propriété recherchée

Recherche exhaustive

- Généralement utilisable seulement pour des problèmes de petite taille
- Dans la plupart des cas, il existe une meilleure solution
- Dans certain cas, c’est la seule solution possible

Problème du voyageur de commerce -TSP-

- Etant donné n villes et les distances entre ces villes
- Trouver le plus court chemin qui passe par toutes les villes exactement une fois avant de revenir à la ville de départ



Tour	Coût
A-B-C-D-A	17
A-B-D-C-A	21
A-C-B-D-A	20
A-C-D-B-A	21
A-D-B-C-A	20
A-D-C-B-A	17

- Recherche exhaustive : O(n!)
- On n’a pas encore pu trouver un algorithme de complexité polynomiale

Diviser pour régner

Diviser pour régner

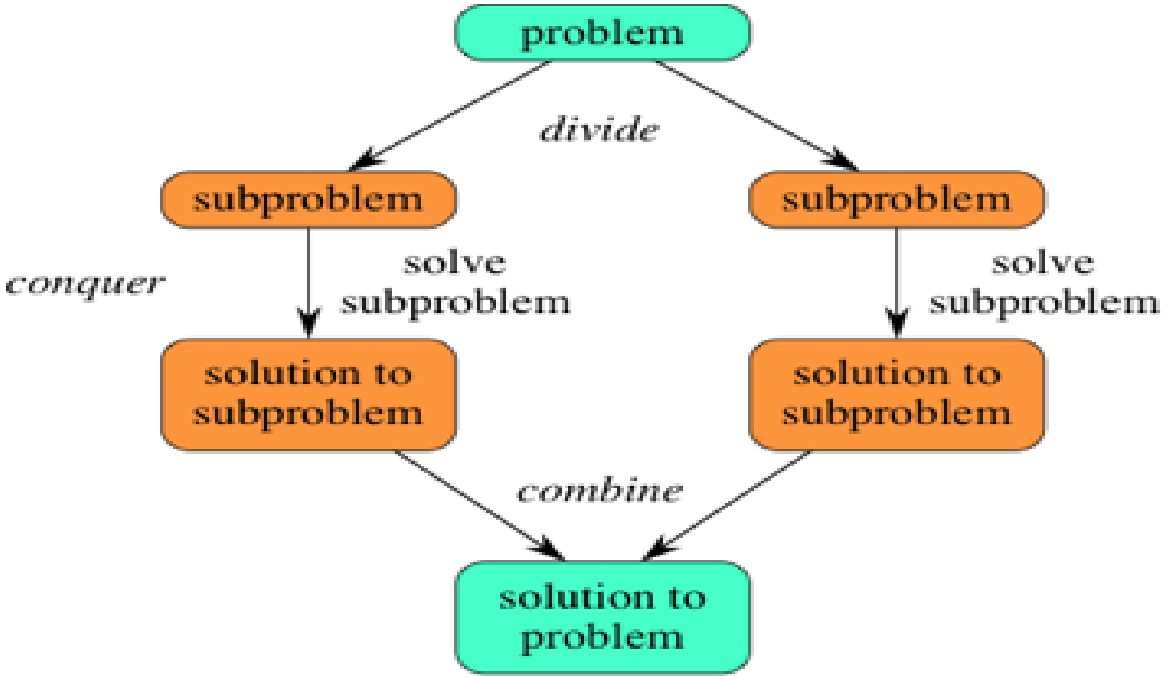
- Divide and conquer est un paradigme pour concevoir certains algorithmes
- La méthode de diviser pour régner est une méthode qui permet, parfois de trouver des solutions efficaces à des problèmes algorithmiques
- L'idée est de découper le problème initial, de taille n, en plusieurs sous-problèmes de taille sensiblement inférieure, puis de recombiner les solutions partielles.

Forme générale

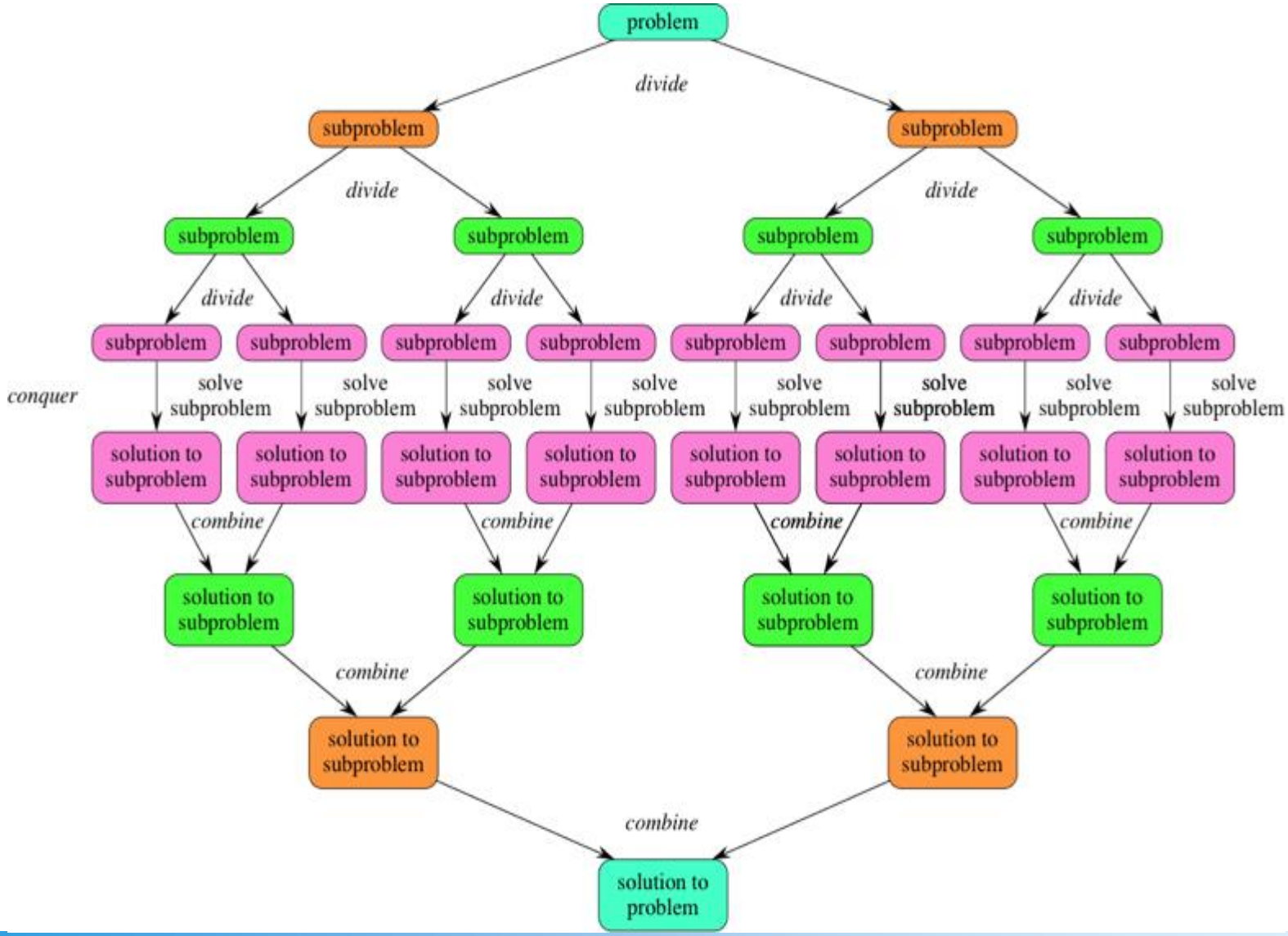
La forme générale considérée dans ce cours va être :

- **Diviser** :
- On décompose le problème à résoudre en sous-problèmes plus simples.
- **Régner** :
- On trouve la solution des sous-problèmes.
- **Recombiner** :
- On combine les solutions des sous-problèmes pour obtenir la solution du problème initial.

Diviser pour régner



Diviser pour régner



Forme générale 2

- La forme générale considérée dans ce cours va être :
- – **Diviser** :
- on découpe le problème en a sous-problèmes de tailles n/b , qui sont de même nature, avec a ≥ 1 et b > 1.
- – **Régner** :
- les sous-problèmes sont résolus récursivement.
- – **Recombiner** :
- on utilise les solutions aux sous-problèmes pour reconstruire la solution au problème initial en temps $O(n^d)$, avec d ≥ 0.
- L'équation qu'on aura à résoudre quand on traduit

$$\begin{cases} T(1) = \text{ constante} \\ T(n) \approx aT\left(\frac{n}{b}\right) + O\left(n^d\right) \end{cases}$$

Diviser pour régner

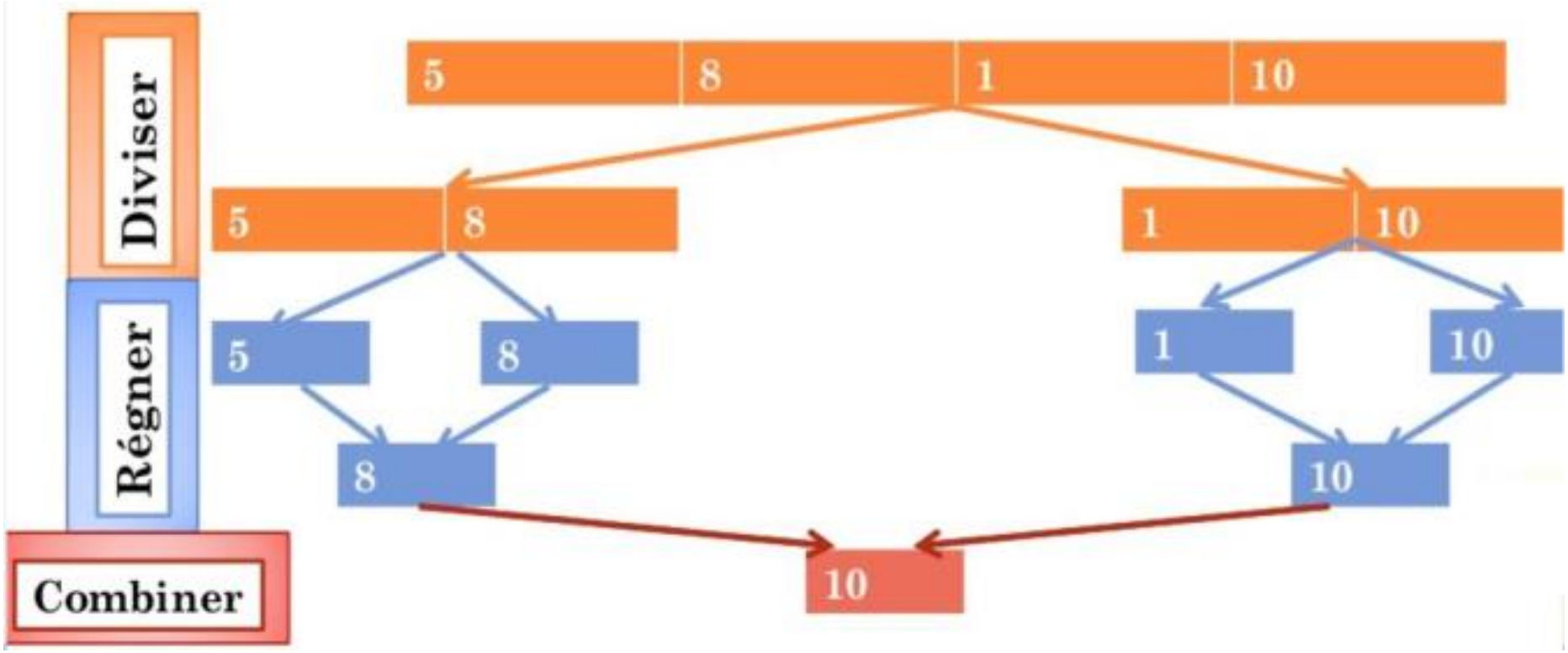
- Approche descendante à la résolution d'un problème
- Cette approche conduit, de façon naturelle (mais pas obligatoirement), à un algorithme récursif :
  - Q: comment obtenir la solution des sous-problèmes?
  - A: en appliquant la même approche diviser-pour-régner descendante aux sous-problèmes, et ce jusqu'à ce que le problème à résoudre soit trivial.

Diviser pour régner

- Pour que l’approche diviser-pour-régner puisse conduire à un algorithme récursif, il faut que les sous-problèmes résultants soient similaires au problème initial.

- Si ce n’est pas le cas, on peut quand même considérer qu’on utilise une approche diviser-pour- régner, mais sans récursivité.

Example :Recherche du maximum



Example

- **Recherche de l'indice du maximum dans tableau d'entiers**
- Fonction **maximum**(Tab, i_deb, i_fin){
- si(i_deb=i_fin) return i_deb
- sinon {
- **//division de problème en 2 sous problèmes**
- m=(i_deb+i_fin)/2
- **//régner sur le 1er sous-problème**
- k1=maximum(tab, i_deb, m)
- **//régner sur le 2er sous-problème**
- k2=maximum(tab, m+1, i_fin)
- **//Combiner les solutions**
- si(Tab[k1]> Tab[k2]) return k1 sinon return k2 }}

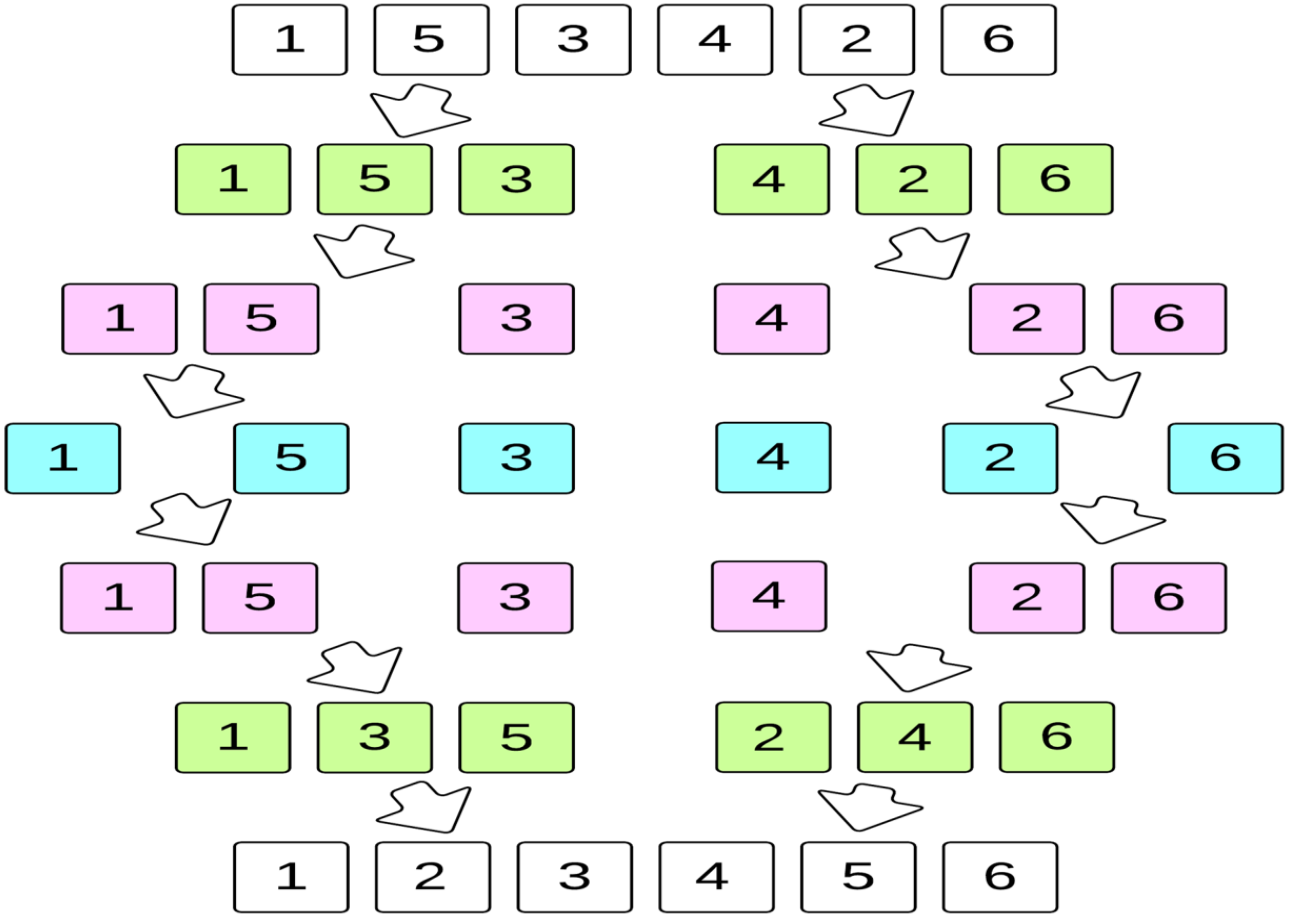
$$T(n) = 2T(n/2) + c$$

Exemple: recherche dichotomique

```
dicho _recursif (T:tab, i_deb, i_fin,  x: entier):booleen
si (i_deb<=  i_fin)
        alors m ← (i_deb +  i_fin) div 2
si (T[m] = x)
         alors retourn (vrai)
sinon si (T[m] > x)
       alors retourn dicho_recursif(T, i_deb, m-1, x)
sinon
       retourn dicho_recursif(T, m+1,  i_fin, x)
sinon    retourn (faux)
 Fin
```

$$T(n) = T(n/2) + c$$

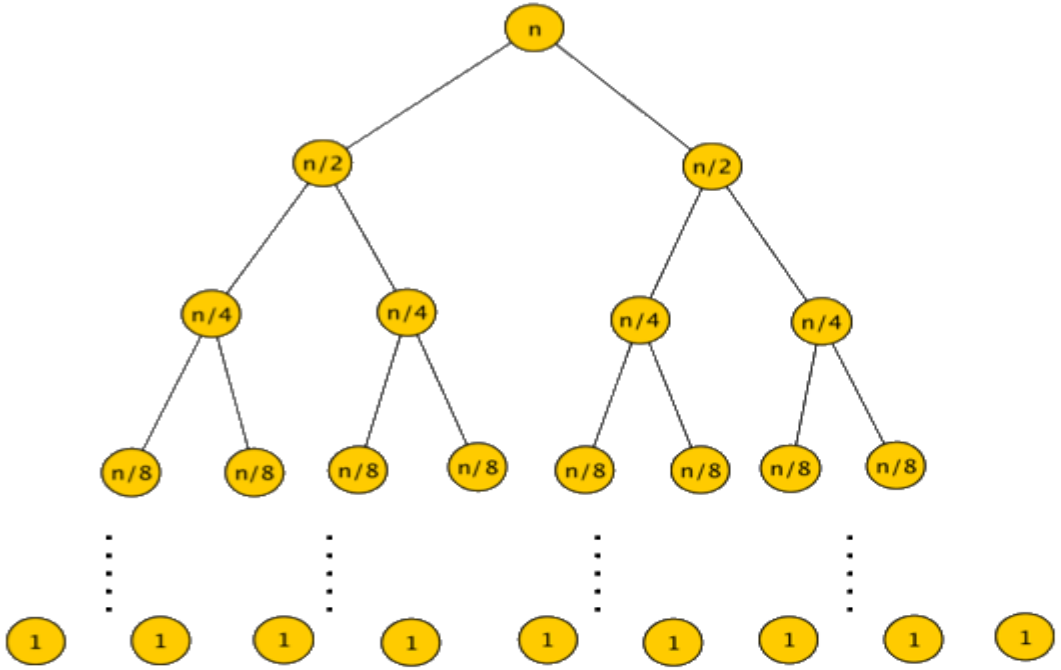
Tri par Fusion



Tri par fusion-l'arbre des appels récursifs-

- -Le cout du fusion de deux tableaux de taille n/2 est O(n)

$$T(n) = 2T(n/2) + n$$



Coût de chaque niveau

$$n$$

$$2 \times \left(\frac{n}{2}\right) = n$$

$$2^2 \times \left(\frac{n}{2^2}\right) = n$$

$$2^3 \times \left(\frac{n}{2^3}\right) = n$$

$$2^{\log_2(n)} \times \left(\frac{n}{2^{\log_2(n)}}\right) = n$$

- -Donc O(n log(n))

Théorème Maître-Master Theorem-

- Soient **a ≥ 1, b > 1**,et T une fonction vérifiant

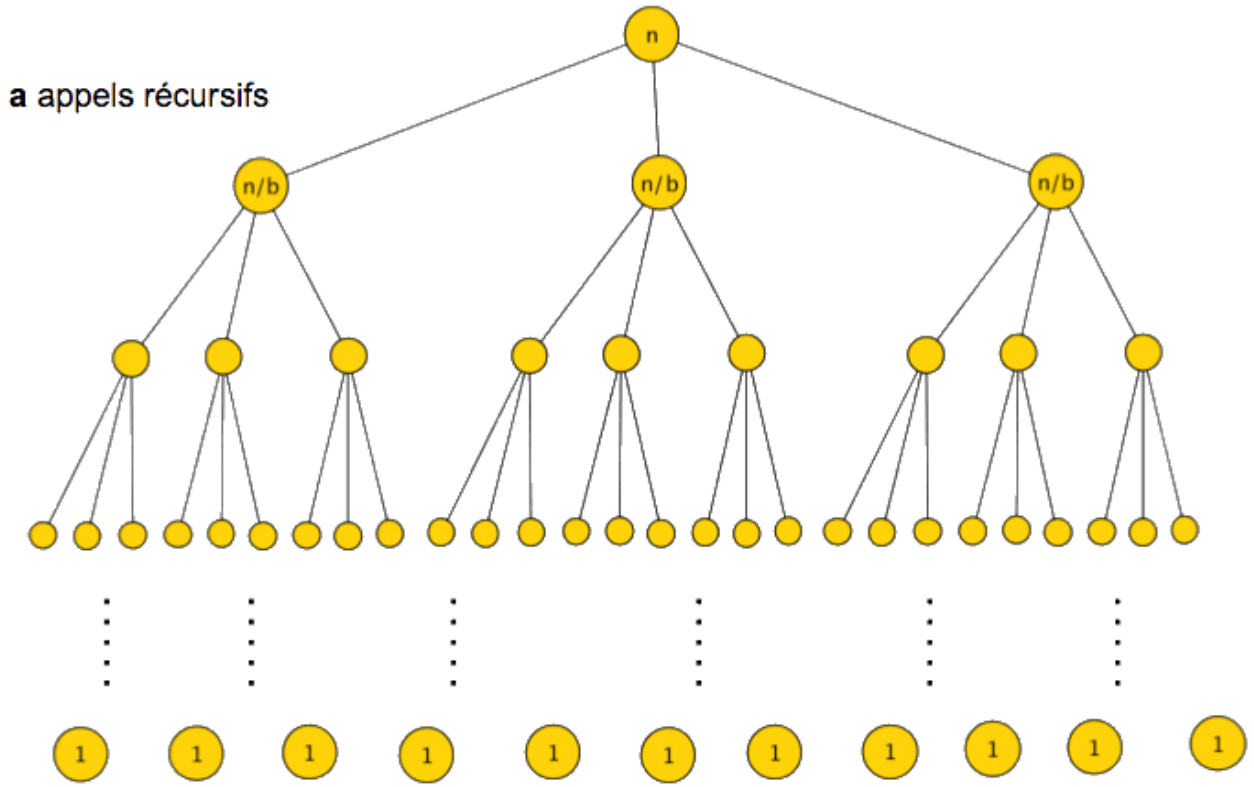
$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- Les différents paramètres s'interprètent comme suit :
  - ***n*** est la taille du problème.
  - ***a*** est le nombre de sous-problèmes.
  - ***n / b*** est le taille de chaque sous-problèmes.
  - ***f(n)*** est le coût de la subdivision et de la combinaison des solutions des sous-problèmes. $f(n) = \Theta\left(n^d\right)$

- Oı

$$T(n) = \begin{cases} \Theta\left(n^d\right) & \text{si } \log_b(a) < d \quad (a < b^d) \\ \Theta\left(n^d \log(n)\right) & \text{si } \log_b(a) = d \quad (a = b^d) \\ \Theta\left(n^{\log_b(a)}\right) & \text{si } \log_b(a) > d \quad (a > b^d) \end{cases}$$

Master Theorem



Coût de chaque niveau

$$n^d$$

$$a \times \left(\frac{n}{b}\right)^d$$

$$a^2 \times \left(\frac{n}{b^2}\right)^d$$

$$a^3 \times \left(\frac{n}{b^3}\right)^d$$

$$a^{\log_b(n)} \times \left(\frac{n}{b^{\log_b(n)}}\right)^d$$

$$T(n) = \sum_{j=0}^{\log_b(n)} a^j \times \left(\frac{n}{b^j}\right)^d = n^d \sum_{j=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^j$$

Example

- $T(n)=2\ T(n/2)+1$
- $T(n)=T(n/2)+1$
- $T(n)=2T(n/2)+n$

$a=?\ b=?\ d=?$

Programmation dynamique

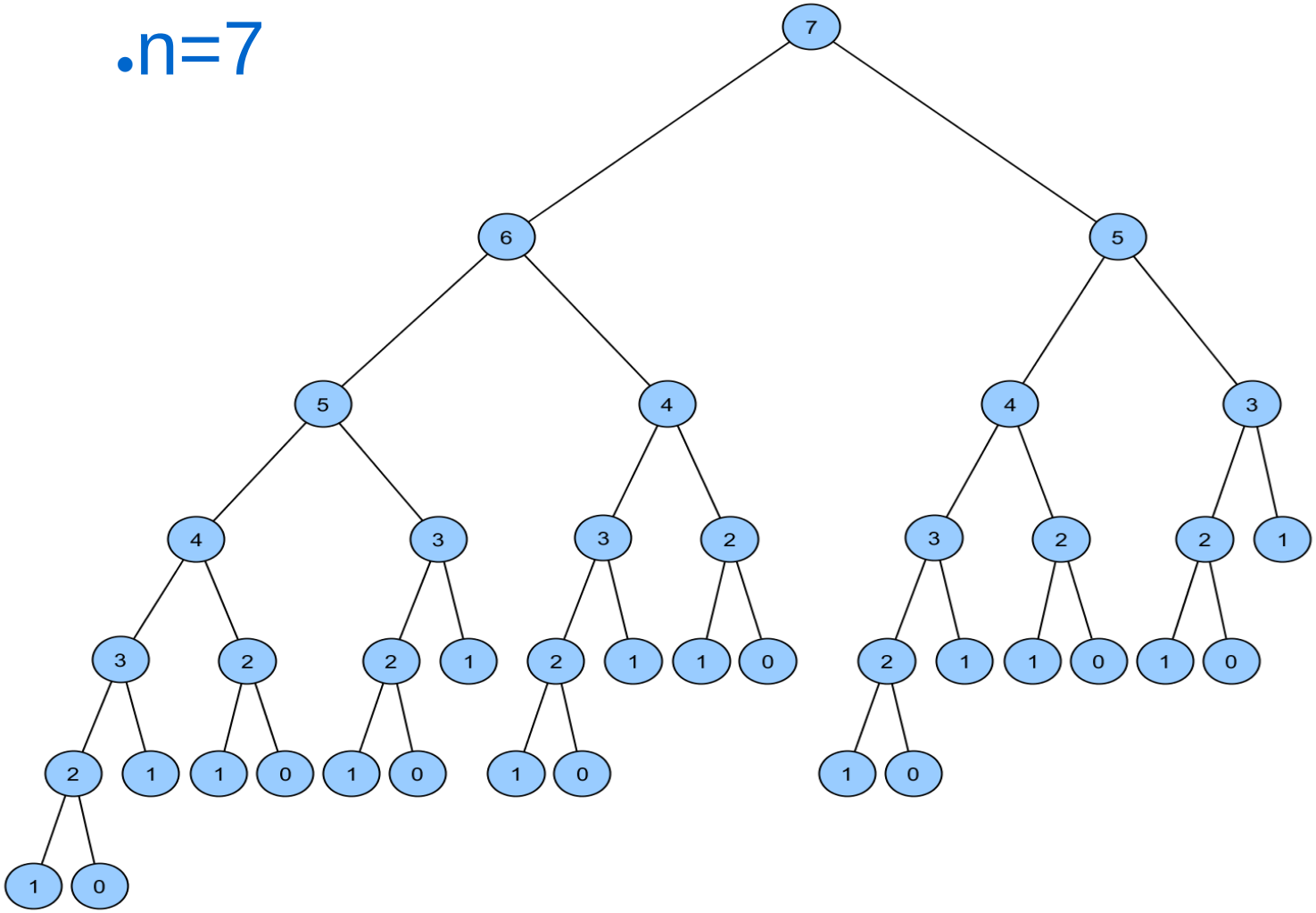
la suite de Fibonacci

$$u_n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ u_{n-1} + u_{n-2} & \text{si } n \geq 2 \end{cases}$$

```
Fibo(n){
 if( n==0 || n==1 )        return n
 else         return Fibo(n-1) + Fibo(n-2)}
```

la suite de Fibonacci

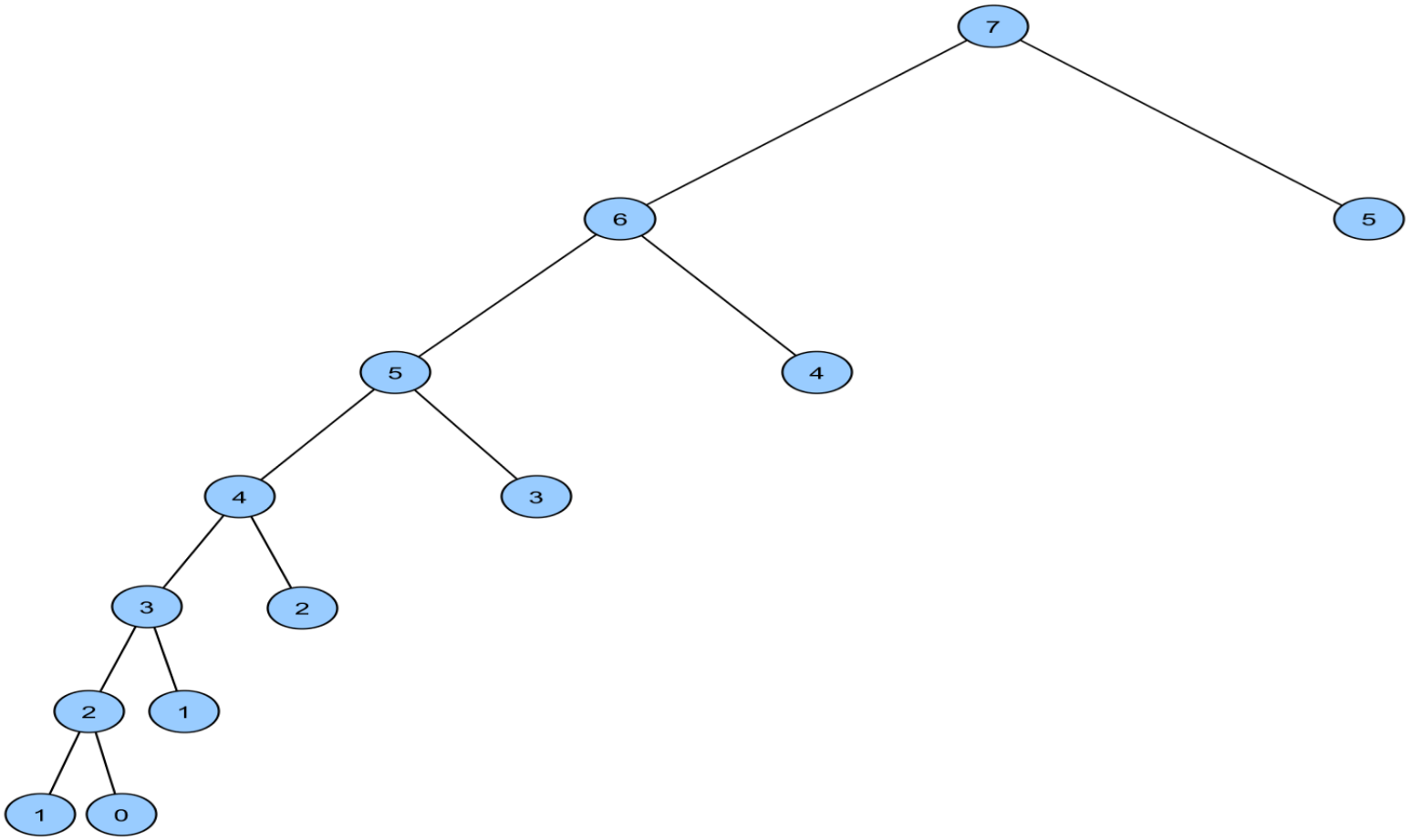
- n=7



la suite de Fibonacci

- int function Fib(int n, table [ ]){
- if (Fib(n) solution est dans la table)
- return table[n] ;
- if (n<= 2)
- return 1;
- else {
- sol = Fib(n-1) + Fib(n-2)
- //sauvegarder sol dans table comme solution à Fib(n) ;
- return (sol) ;}}

la suite de Fibonacci



la suite de Fibonacci

```
int function Fib(int n){
Allouer F avec n + 1 cases
F[1] =1 ; F[2] =1 ;
For (i=2 ; i<=n ; i++)
F[i] = F[i-1] + F[i-2];
return (F[i]);
}
```

Programmation dynamique

La programmation dynamique est en général appliquée aux problèmes d'optimisation :

- problèmes qui peuvent admettre plusieurs solutions, parmi lesquelles on désir choisir
- une solution optimale (maximale ou minimale).

Programmation dynamique

Un algorithme de programmation dynamique résout chaque problème une unique fois et mémorise sa solution dans un tableau s’épargnant ainsi le calcul de la solution à chaque fois que le sous-problème est rencontré.

Exemple : 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 pour nombre entier de centimètres
- Le prix de vente d'une tige dépend (non linéairement) de sa longueur

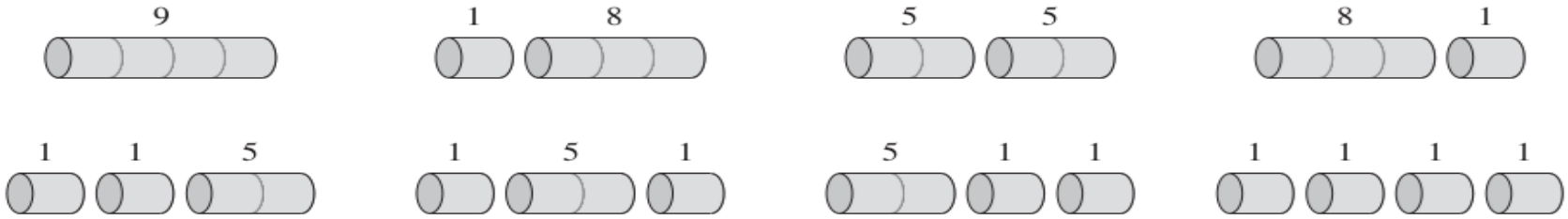
- On veut déterminer le revenu maximum qu'on peut attendre de la vente d'une tige de n centimètres

Découpage de tiges d'acier

- 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



- Meilleur revenu = 10 (2 tiges de 2cm)

Approche par force brute:

- Enumérer toutes les découpes, calculer leur revenu, déterminer le revenu maximum

- Il y a 2^{n-1} manières de découper une tige de longueur n (on peut couper ou non après chacun des n-1 premiers centimètres)

- Plusieurs découpes sont équivalentes (1+1+2 et 1+2+1 par exemple) mais même en prenant cela en compte, le nombre de découpes reste exponentiel

découpage de tiges d'acier

- Soit r_i le revenu maximum pour une tige de longueur i
- Peut-on formuler r_n de manière récursive ?
- Déterminons r_i pour notre exemple :

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 . . .

Formulation récursive de r_n : version naïve

- r_n peut être calculé 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
  - ...
  - $r_{n-1} + r_1$.
- Donc $r_n = \max(p_n , r_1 + r_{n-1} , r_2 + r_{n-2} , \ldots , r_{n-1} + r_1)$

Formulation récursive de r_n : version simplifiée

- Toute solution optimale a un découpe la plus à gauche
- On peut calculer r_n en considérant toutes les tailles pour la première
- découpe et en combinant avec le découpage optimal pour la partie à droite
- Pour chaque cas, on n'a donc qu'à résoudre un seul sous-problème (au lieu de deux), celui du découpage de la partie droite
- En supposant $r_0 = 0$, on obtient ainsi :
- $r_n = \max (p_i + r_{n-i})$ $1 <= i <= n$

Implémentation récursive directe

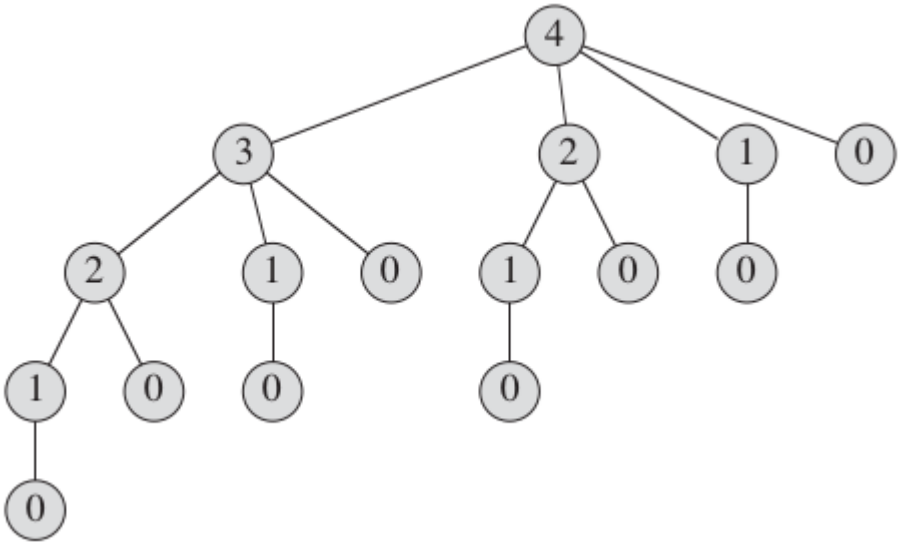
- La formule récursive peut être implémentée directement

```
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
```

- (p est un tableau de taille n contenant les prix des tiges de tailles 1 à n)

Implémentation récursive directe :

- L’algorithme est extrêmement inefficace à cause des appels récursifs redondants



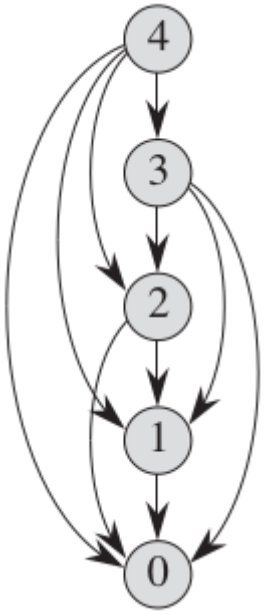
Approche descendante avec mémoization

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

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
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
```

Approche ascendante

```
BOTTOM-UP-CUT-ROD(p, n)
1   Let r[0..n] be a new array
2   r[0] = 0
3   for j = 1 to n
4       q = -∞
5       for i = 1 to j
6           q = max(q, p[i] + r[j - i])
7       r[j] = q
8   return r[n]
```



$\Theta(n^2)$

Reconstruction de la solution

- Fonction Bottom-up-Cut-rod calcule le revenu maximum mais ne donne pas directement la découpe correspondant à ce revenu
- On peut étendre l'approche ascendante pour enregistrer également la solution dans une autre table
- s[j] contient la coupure la plus à gauche d'une solution optimale au problème de taille j

```
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
```

Reconstruction de la solution

- Pour afficher la solution, on doit “remonter” dans s

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      PRINT s[n]
4      n = n − s[n]
```

- Exemple :

i	0	1	2	3	4	5	6	7	8
$p[i]$	0	1	5	8	9	10	17	17	20
$r[i]$	0	1	5	8	10	13	17	18	22
$s[i]$	0	1	2	3	2	2	6	1	2

- print-cut-rod-solution(p, 8)) → ”2 6”

Programmation dynamique : généralités

- 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
  - Chevauchement des sous-problèmes : Certains sous-problèmes distincts partagent une partie de leurs sous-problèmes
- Implémentation directe récursive donne une solution de complexité exponentielle
- Sauvegarde des solutions aux sous-problèmes donne une complexité linéaire dans le nombre d'arcs et de sommets du graphe des sous-problèmes

Programmation dynamique vs diviser-pour-régner

- L’approche Diviser-pour-régner décompose aussi le problème en sous-problèmes
- Mais ces sous-problèmes sont significativement plus petits que le problème de départ (n --> n/2)
  - Alors que la programmation dynamique réduit généralement un problème de taille n en sous-problèmes de taille n-1
- Et ces sous-problèmes sont indépendants
  - Alors qu’en programmation dynamique, ils se recouvrent

Programmation dynamique:Grandes étapes

.

- Caractériser la structure du problème
- Définir de manière récursive la valeur de la solution optimale
- Calculer les valeurs de la solution optimale (c'est-à-dire remplir un tableau)
- Reconstruire la (une) solution optimale à partir de l'information calculée ("bottom-up")

Applications

- Command unix diff(comparaison de fichiers)
- Algorithme de Viterbi (reconnaissance vocale)
- Alignement de séquences d’ADN (Smith-Waterman)
- Plus court chemin dans un graphe (Bellman-Ford)
- Compilateurs (analyse syntaxique et optimisation du code)

Algorithme glouton
(GREEDY ALGORITHM)

Algorithme glouton

Algorithme glouton

- Utilisé pour résoudre des problèmes d’optimisation (comme la programmation dynamique)
- Idée principale :
- 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)

Algorithme glouton (greedy)

- 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 : rendre la monnaie

- Objectif : Etant 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.

- Exemple : 34 cents :
- {1, 1, 2, 5, 5, 20} ! 6 pièces
- {2, 2, 10, 20} ! 4 pièces
- 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)

Exemple : rendre la monnaie

- Algorithme de la caissière : A chaque itération, ajouter une pièce de la plus grande valeur qui ne dépasse pas la somme restant à rembourser
- Complexité : O(n) (sans compter le tri des pièces)
- Cet algorithme permet-il de trouver une solution optimale ?

```
CoinChangingGreedy(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 ith coin in solution
4  CoinCount = 0
5  for i = 1 to n
6        s[i] = ⌊x/c[i]⌋
7        x = x − s[i] ∗ c[i]
8        CoinCount = CoinCount + s[i]
9  return (s, CoinCount)
```

Exemple : rendre la monnaie

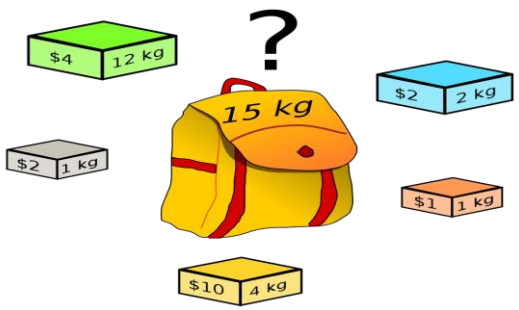
- L’approche greedy n’est correcte que pour certains choix particuliers de valeurs de pièces
- OK pour la plupart des monnaies courantes, €, $. . .
- Contre-exemple : C = [1, 10, 21, 34, 70, 100] (valeurs de timbres aux USA) et x = 140
- Algorithme glouton : 100, 34, 1, 1, 1, 1, 1, 1
- Solution optimale : 70, 70
- Solution pour résoudre le cas général : programmation dynamique

le problème du sac à dos (01 knapsack)

Un voleur se rend dans un musée pour commettre un méfait avec un sac à dos pouvant contenir W kg.

- Le musée comprend n œuvres d'art, chacune de poids p_i et de prix v_i (i = 1, . . . , n)

- Le problème pour le voleur est de déterminer une sélection d'objets de valeur totale maximale et n'excédant pas le poids total admissible dans le sac à dos.



knapsack problem

- Example :W=11

i	v_i	p_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

- Formellement :
- Soit un ensemble S de n objets de poids $p_i > 0$ et de valeurs $v_i > 0$
- Trouver $x_1, x_2, \ldots, x_n \in \{0, 1\}$ tels que : $\sum_{i=1}^{n} x_i \cdot p_i \leq W$ et $\sum_{i=1}^{n} x_i \cdot v_i$ est maximal
- Recherche exhaustive:$O(n\ 2^n)$

Greedy knapsack algorithm

- **Idée d'algorithme :**
- Ajouter à chaque itération l'objet de rapport p_i / v_i maximal qui rentre dans le sac
- Implémentation très proche du problème de change :

- W=11
- Solution greedy : {5, 2, 1} valeur=35
- Solution DP : {4, 3} valeur=40

i	v_i	p_i	v_i / p_i
1	1	1	1
2	6	2	3
3	18	5	3,6
4	22	6	3,7
5	28	7	4

Algorithme glouton

- Très efficaces quand ils fonctionnent. Simples et faciles à implémenter
- Ne fonctionnent pas toujours. Leur correction peut être assez difficile à prouver
- Applications :
  - Arbre de couverture minimal
  - Plus court chemin dans un graphe (algorithme de Dijkstra)
  - Allocation de resources
  - Codage de Huffman

Conclusion

- **Approche par force brute** : résoudre directement le problème, à partir de sa définition ou par une recherche exhaustive
- **Diviser pour régner** : diviser le problème en sous-problèmes, les résoudre, fusionner les solutions pour obtenir une solution au problème original
- **Programmation dynamique** : obtenir la solution optimale à un
- problème en combinant des solutions optimales à des sous-problèmes similaires plus petits et se chevauchant
- **Approche gloutonne** : construire la solution incrémentalement, en optimisant de manière aveugle un critère local