Université Clermont Auvergne
ISIMA 1A par apprentissage
Algorithmique

Florent Foucaud
2023-2024

TD3 - Complexité en temps, partie II

1 Diviser pour régner

La formule vient de la politique ou de la guerre : *Divide ut regnes*, Philippe II de Macédoine, père d'Alexandre Legrand ; *Divide et impera*, Machiavel.

En algorithmique, il s'agit de couper un problème algorithmique en plusieurs sous-problèmes, généralement en 3 étapes :

- DIVISER : le problème est divisé en sous-problèmes
- RÉGNER : les sous-problèmes sont plus faciles à résoudre
- COMBINER : les solutions des sous-problèmes sont combinées

Exemples classiques : PGCD, Exponentiation rapide, Tri par fusion, Karatsuba (multiplication rapide), Strassen (multiplication de matrices), Tours de Hanoï...

Exercice 1 (Recherche dichotomique récursive)**.**

Quelle est la complexité en temps $T(N)$ de l'algorithme de recherche dichotomique ci-dessous, appelé avec $m = 1$ et $n = N$? Pour simplifier, on supposera que $N = 2^k$ est une puissance de deux.

rechercheDicho(TAB, m, n, x): calcul d'une position de x dans TAB

Entrée : Un tableau d'entiers $TAB[1...N]$ triés par ordre croissant, un entier x, deux positions m et n

Sortie : 0 si $x \notin TAB$, et une position de x dans TAB (entre m et n) sinon

- Si $m == n$:
  - ⋆ Si $TAB[m] == x$:
    Retourner m
  - ⋆ Sinon :
    Retourner 0
- Sinon :
  - ⋆ $k = \lfloor \frac{m+n}{2} \rfloor$
  - ⋆ Si $TAB[k] < x$:
    Retourner rechercheDicho(TAB,$k+1$,n,x)
  - ⋆ Sinon :
    Retourner rechercheDicho(TAB,m,k,x)

Solution.

De façon informelle, on peut dessiner l'arbre des appels récursifs, et on voit que cet arbre est en fait une chaîne, de hauteur $\log_2(N)$. À chaque étape, on a un nombre constant d'opérations, on va donc avoir une complexité de l'ordre de $\Theta(\log(N))$.

Plus formellement, on peut faire l'analyse suivante.

Soit $N' = m - n + 1$ (la taille du sous-tableau $TAB[m...n]$). Au premier appel, on a $N' = N$. À chaque appel récursif, au plus 10 opérations de base sont réalisées. À chaque étape, k divise le tableau exactement en deux parties égales et on fait un appel récursif sur un tableau deux fois moins grand, sauf si $N' = 1$.

Dans ce (pire des) cas, on peut done écrire que $T(N') \leq 10+T(N'/2)$. Au deuxième appel récursif, on a $T(N'/2) \leq 10 + T(N'/4)$, ce qui donne $T(N') \leq 10 + 10 + T(N'/4) \leq 20 + T(N'/4)$. Avec la même logique, au bout de i appels récursifs, on a $T(N') \leq 10 \times i + C\left(\frac{N'}{2^i}\right)$.

Combien y a-t-il d'appels récursifs ? On s'arrête quand $m = n$, soit $N' = 1$ (cas de base de l'algorithme). On veut donc résoudre l'équation $\frac{N'}{2^i} = 1$, cela donne $N' = 2^i$ soit $i = \log_2(N')$. Pour nous, $N = N' = 2^k$, on obtient donc $i = \log_2(N) = k$.

On a donc $T(N) \leq 10 \times \log_2(N) + T(1)$, et $T(1) = 4$, donc $T(N) \leq 10 \times \log_2(N) + 4$ ce qui appartient à la classe de complexité $\Theta(\log(N))$.

Exercice 2 (Diviser pour régner : le tri par fusion)**.**

Pour simplifier, on supposera que $N = 2^k$ est une puissance de deux.

1. Estimer la complexité de la sous-fonction `interClassement(`TAB_1`,`TAB_2`)` en fonction de N_1 et N_2.
2. Dessiner l'arbre des appels récursifs de `triFusion` pour un tableau de taille $N = 2^4$. Quelle est la complexité approximative de l'algorithme sur *chaque niveau* de l'arbre ?
3. En déduire une estimation informelle de la complexité globale de l'algorithme.
4. On va maintenant calculer plus précisément cette complexité. Exprimer la complexité $T(N)$ de `triFusion` en fonction de $T(N/2)$.
5. Appliquer la formule précédente à $T(N/2)$ pour substituer $T(N/2)$ dans la formule. Ainsi on exprime $T(N)$ en fonction de $T(N/4)$.
6. Faire encore une étape en exprimant $T(N)$ en fonction de $T(N/8)$.
7. En déduire $T(N)$ en fonction de $T(N/2^i)$ ($i$ème étape récursive).
8. On rappelle que $N = 2^k$, donc si $i = k$ on a $T(N/2^i) = T(1)$. En déduire l'expression de $T(N)$ uniquement en fonction de N.

`triFusion`(TAB, m, n)`: tri du sous-tableau` $TAB[m...n]$ `d'entiers par ordre croissant`

`Entrée : Un tableau d'entiers` $TAB[1...N]$ `et deux positions` m, n `avec` $m \leq n$

`Sortie : Le sous-tableau` $TAB[m...n]$ `trié`

- `Si` $m < n$ `:`
  - ⋆ $k = \lfloor \frac{m+n}{2} \rfloor$
  - ⋆ $TAB_1 =$ `triFusion`(TAB, m, k)
  - ⋆ $TAB_2 =$ `triFusion`($TAB, k+1, n$)
  - ⋆ $TAB[m...n] =$ `interClassement`(TAB_1, TAB_2)
- `Retourner` $TAB[m...n]$

`interClassement`(TAB_1, TAB_2)`:`

`Entrée : Deux tableaux d'entiers` $TAB_1[1...N_1]$, $TAB_2[1...N_2]$ `triés par ordre croissant`

`Sortie : L'union` TAB `de` TAB_1 `et` TAB_2 `triée par ordre croissant`

- $i = 1$, $j = 1$, $k = 1$
- TAB `est un tableau vide à` $N_1 + N_2$ `éléments`
- `TABant que` $i \leq N_1$ `et` $j \leq N_2$ `:`
  - `Si` $TAB_1[i] < TAB_2[j]$ `:`
    $TAB[k] = TAB_1[i]$
    $i = i+1$
  - `Sinon :`
    $TAB[k] = TAB_2[j]$
    $j = j+1$
  - $k = k+1$
- `Tant que` $i \leq N_1$ `:`
  $TAB[k] = TAB_1[i]$
  $i = i+1$
  $k = k+1$
- `Tant que` $j \leq N_2$ `:`
  $TAB[k] = TAB_2[j]$
  $j = j+1$
  $k = k+1$
- `Retourner` TAB

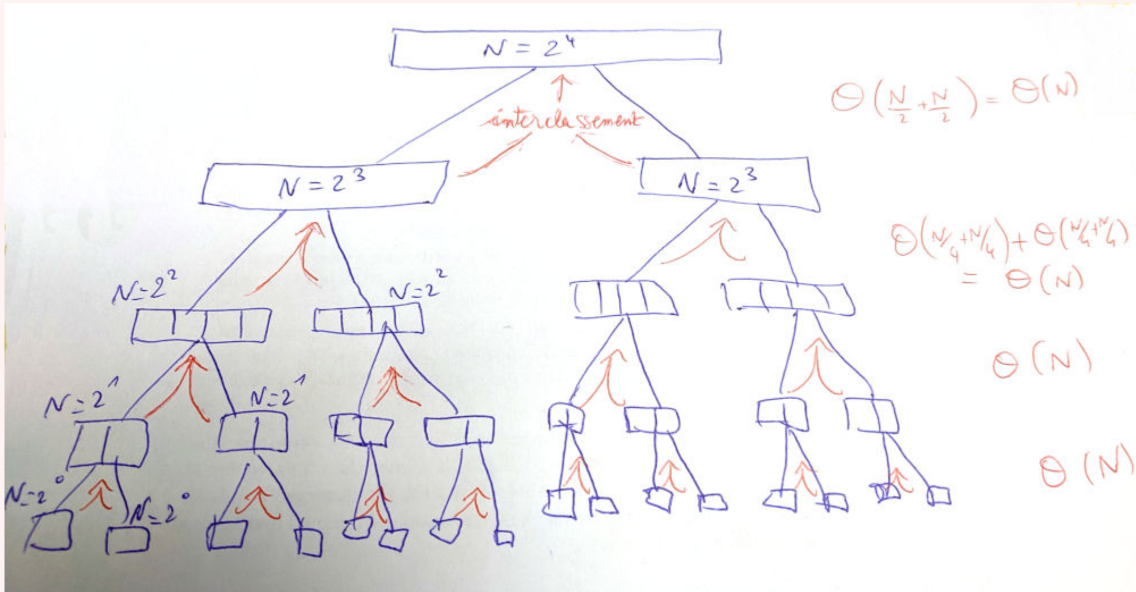
Solution.

1. Estimons la complexité de `interClassement`(TAB_1, TAB_2). Noter que seule la première et l'une des deux dernières boucles seront parcourues. Il y a au plus 13 opérations de base par tour de boucle sur la première boucle $(3+3+3+2+2)$, et 8 pour les autres boucles $(1+3+2+2)$. Donc, dans le pire des cas, on épuise au maximum les deux tableaux, et on fait $N_1 + N_2 - 1$

tours de la première boucle, et 1 tour de l'une des deux suivantes. Il y a 3 affectations de i, j, k, un retour de TAB (4 instructions de base). Il y a un "tant que" de chaque boucle exécuté en plus avant la sortie, donc $3 + 1 + 1$ opérations en plus.

On a donc (dans le pire de cas) $13(N_1 + N_2 - 1) + 4 + 5 + 8 = 13(N_1 + N_2) + 4$ opérations.

2. À chaque niveau, le total est une somme d'interclassements, au total c'est linéaire.



3. Pour l'algorithme principal, de nouveau, de façon informelle, on note qu'on appelle récursivement l'algorithme sur deux fois la moitié du tableau, et pour chacun de ces deux appels, on fait appel à `interClassement(`TAB_1, TAB_2`)`. A chaque niveau, on fait un traitement linéaire puisqu'on fait des interclassements sur des sous-tableaux dont la somme des tailles fait N. On peut représenter la structure de ces appels par un arbre binaire dont les feuilles sont les sous-tableaux de taille 1. Cet arbre binaire a une hauteur de $\log_2(N)$, et pour chaque niveau de l'arbre, on a une complexité de $\Theta(N)$ (la somme des appels à `interClassement(`TAB_1, TAB_2`)` pour ce niveau). Au total, on a donc une complexité de $\Theta(N \log(N))$.

4. On a $N = m - n$. On fait une comparaison, 4 opérations mathématiques, 2 appels récursifs et 2 affectations du résultat, un appel à `interClassement(`TAB_1, TAB_2`)` avec une affectation, et un retour. Cela donne donc

$$
\begin{aligned}
T(N) &\leq 1 + 4 + 2 + 2T(N/2) + (13N + 4) + N + 1 \\
&= 2T(N/2) + 14N + 12
\end{aligned}
$$

5. $T(N/2) \leq 2T(N/4) + 14N/2 + 12$. On substitue dans la formule précédente :

$$
\begin{aligned}
T(N) &\leq 2T(N/2) + 14N + 12 \\
&= 2(2T(N/4) + 14N/2 + 12) + 14N + 12 \\
&= 4T(N/4) + 2 \times 14N + 3 \times 12
\end{aligned}
$$

6. $T(N/4) \leq 2T(N/8) + 14N/4 + 12$. On substitue dans la formule précédente :

$$\begin{array}{rcl} T(N) & \leq & 4T(N/4) + 2 \times 14N + 3 \times 12 \\ & = & 4(2T(N/8) + 14N/4 + 12) + 2 \times 14N + 3 \times 12 \\ & = & 8T(N/8) + 3 \times 14N + 7 \times 12 \end{array}$$

7. Avec la même logique, au bout de i appels récursifs, on a $T(N) \leq (14N+14) \times (2^i - 1) + 2^i T\left(\frac{N}{2^i}\right)$.

$$\begin{array}{rcl} T(N) & \leq & 2^i T\left(\frac{N}{2^i}\right) + i \times 14N + \sum_{j=0}^{i-1} 2^j \times 12 \\ & = & 2^i T\left(\frac{N}{2^i}\right) + i \times 14N + (2^i - 1) \times 12 \end{array}$$

8. On s'arrête quand $\frac{N}{2^i} = 1$ (cas de base). On veut donc résoudre l'équation $\frac{N}{2^i} = 1$, cela donne $N = 2^i$ soit $i = \log_2(N)$. Pour nous, $N = 2^k$, on obtient donc $i = \log_2(N) = k$.
   On a donc :

$$\begin{array}{rcl} T(N) & \leq & 2^k T\left(\frac{N}{2^k}\right) + k \times 14N + (2^k - 1) \times 12 \\ & = & N \times T(1) + 14N \log_2(N) + 12(N-1) \\ & = & 14N \log_2(N) + 14N - 12 \end{array}$$

   ce qui appartient à la classe de complexité $\Theta(N \log(N))$.

2 Programmation dynamique

La *programmation dynamique* est une technique algorithmique qui consiste à calculer la solution pour des sous-problèmes, et où la solution du sous-problème suivant dépend directement de un ou plusieurs sous-problèmes précédemment calculés.

Souvent, c'est le moyen de rendre itératif un algorithme récursif, en passant d'une complexité exponentielle à une complexité polynomiale.

Les sous-problèmes sont souvent stockés dans une tableau, ou bien, on peut aussi faire de la programmation dynamique sur des structures arborescentes ou des graphes orientés acycliques.

Exercice 3 (Suite de Fibonacci)**.**

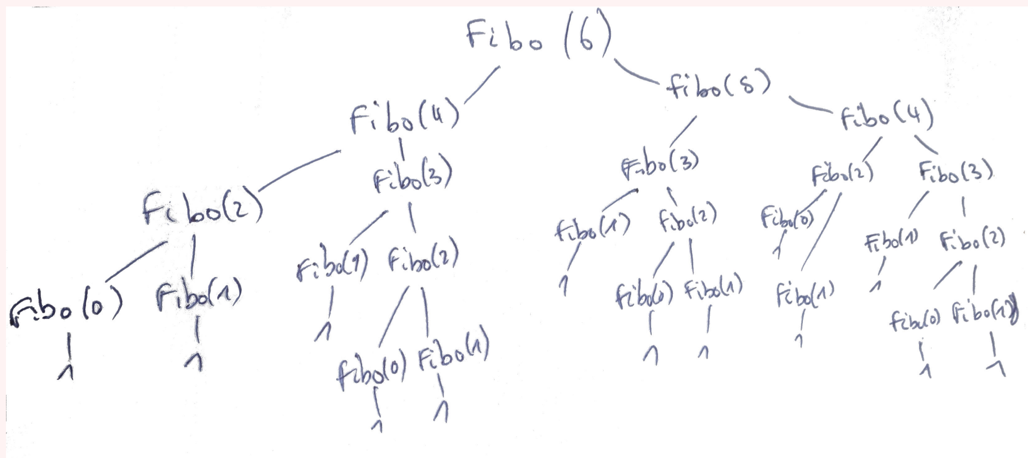
On considère le programme récursif suivant, permettant d'approximer la taille de la $n$ème génération d'une population de lapins.

```
Fibo(n):

Entrée : un entier n

Sortie : la nème valeur de la suite de Fibonacci

  • Si n == 0 ou n == 1:
    Retourner 1

  • Sinon:
    Retourner Fibo(n − 1) + Fibo(n − 2)
```

1. Dessiner l'arbre des appels récursifs pour $Fibo(6)$. Intuitivement, quelle est la complexité du programme ?
2. L'algorithme récursif effectue le même calcul un grand nombre de fois. Pour calculer $Fibo(n)$ de façon plus efficace, on pourrait simplement stocker les valeurs calculées au fur et à mesure dans un tableau. Écrire un tel algorithme.

Solution.

1. On se rend compte que cela "a l'air" exponentiel :



2.

```
Fibo(n):

Entrée : un entier n

Sortie : la nème valeur de la suite de Fibonacci

  - Initialiser un tableau Fibo
  - Fibo[0] = 1
  - Fibo[1] = 1
  - i = 2
  - Tant que i ≤ n faire :
      ⋆ Fibo[i] = Fibo[i-2] + Fibo[i-1]
      ⋆ i = i + 1
  - Retourner Fibo[n]
```

Exercice 4 (Rendu de monnaie).

On a une liste de dénominations de pièces et billets, par exemple `L=[1,2,5,10,20,50,100,200,500]`. Étant donné un entier `n` de monnaie à rendre en euros, on veut savoir quelle est la meilleure combinaison de pièces/billets (on souhaite minimiser le nombre total de pièces/billets à rendre).

On pourrait proposer un algorithme glouton pour ce problème : on choisit la plus grande dénomination `d` qui ne dépasse pas `n`, et on continue avec `n-d` jusqu'à arriver à 0. Cependant, cet algorithme n'est pas optimal : par exemple pour `L=[5,10,20,25]` et `n=40`, cet algorithme va renvoyer 3 $(25+10+5)$ alors que l'optimum est 2 $(20+20)$. Il faut donc trouver une méthode plus exhaustive...

On note $s[i]$ la solution pour la somme i (plus petit nombre de pièces/billets dont la somme est i).

1. Exprimer $s[i]$ selon deux cas : (1) i est dans la liste L des dénominations, ou sinon (2), en fonction de $s[i-d]$ pour tous les d dans L et $d<i$.
2. En déduire un algorithme récursif `MonnaieRec(n,L)` pour calculer la valeur optimale.
3. Estimer la complexité de l'algorithme récursif en fonction de n et de la taille $|L|$ de L.
4. Le transformer en algorithme itératif `MonnaieProgDyn(n,L)` utlisant le principe de programmation dynamique.
5. Quelle est sa complexité en temps en fonction de n et de la taille $|L|$ de L ? Comme $N=n+|L|$ est la taille de l'entrée, quelle est la complexité en fonction de N ?
6. Quelle modification peut-on faire pour aussi calculer la combinaison optimale (pas juste la valeur) ?

Solution.

1. Si i est dans la liste L des dénominations, on a $s[i]=1$.
   Sinon, $s[i]=\min\{s[i-d]+1$ sur tous les d dans L avec $d<i\}$
2.

```
MonnaieRec(n,L):

Entrée : un entier n, une liste L d'entiers

Sortie : le nombre minimum de pièces/billets de L dont la somme est n

   - Si n est dans L:
     Retourner 1

   - Sinon:

       ⋆ min = +∞

       ⋆ Pour chaque d dans L tel que d<n:

           - val = MonnaieRec(n-d,L)

           - Si val < min:
             min = val

       ⋆ Retourner min + 1
```

3. Complexité exponentielle, car on a un arbre des appels récursifs où chaque noeud a potentiellement $|L|$ enfants et la hauteur est d'ordre n, donc $|L|^n$ noeuds au total. (Chaque noeud représente un appel récursif.)

4.

```
MonnaieProgDyn(n,L):

Entrée : un entier n, une liste L d'entiers
Sortie : le nombre minimum de pièces/billets de L dont la somme est n

  • Initialiser un tableau des valeurs optimales s[1..n]
  • Pour i allant de 1 à n:
      - Si i est dans L:
        s[i] = 1
      - Sinon:
          ⋆ min = +∞
          ⋆ Pour chaque d dans L tel que d < i:
              - val = s[i-d]
              - Si val < min + 1:
                min = val
          ⋆ s[i] = min
  • Retourner s[n]
```

5. $\Theta(n \times |L|)$ car pour chaque valeur de i on fait une boucle sur L. Si $N = n + |L|$ est la taille de l'entrée, cela donne $\Theta(N^2)$.
6. On peut faire un backtrack, en se souvenant pour chaque $s[i]$ quelle était la dernière valeur de d pour laquelle $s[i] = s[i-d] + 1$. De fil en aiguille on peut ainsi reconstruire la solution optimale.

Voici le code Python de l'implémentation des deux algos :

```python
import sys

L=[1,2,5,10,20,50,100,200,500]

def MonnaieRec(n,L):

    #case de base
    if n in L:
        return 1

    dict_valeurs = {}

    #calculs ré cursifs
    for d in L:
        if n-d > 0:
            dict_valeurs[d] = MonnaieRec(n-d,L)

    valeur_min = sys.maxsize
    elt_min = 0

    #trouver le minimum parmi les valeurs calcul ées
    for d in dict_valeurs:
        if dict_valeurs[d] < valeur_min:
            valeur_min = dict_valeurs[d]
```

```
25            elt_min = d
26
27    return valeur_min + 1
28
29 print(MonnaieRec(36,L))#ça commence àprendre du temps!
30
31
32 def MonnaieProgDyn(n,L):
33     #tableau de solutions
34     valeurs = (n+1)*[sys.maxsize]
35     valeurs[0] = 1
36
37     for i in range (1,n+1):
38         #si on peut rendre un seul billet/pièce
39         if i in L:
40             valeurs[i] = 1
41         else:
42             #on cherche la meilleure possibilité
43             #pour la dernière pièce/billet
44             valeur_min = sys.maxsize
45             for d in L:
46                 if i-d > 0 and valeur_min > valeurs[i-d]:
47                     valeur_min = valeurs[i-d]
48             valeurs[i] = valeur_min + 1
49
50     return valeurs[i]
51
52 print(MonnaieProgDyn(176854,L))
53
54 def MonnaieProgDynAvecCombinaison(n,L):
55     valeurs = (n+1)*[sys.maxsize]
56     valeurs[0] = 1
57
58     for i in range (1,n+1):
59         if i in L:
60             valeurs[i] = 1
61         else:
62             valeur_min = sys.maxsize
63             for d in L:
64                 if i-d > 0 and valeur_min > valeurs[i-d]:
65                     valeur_min = valeurs[i-d]
66             valeurs[i] = valeur_min + 1
67
68     # -- PARTIE NOUVELLE POUR LE BACKTRACK --
69     # initialiser dico de solutions à 0
70     #pour chaque dénomination, on va compter
71     #le nombre qu'il faut
72     dict_solution = {}
73     for d in L:
74         dict_solution[d] = 0
75
76     #backtracking:
77     j = n
78     while j > 0:
79         if j in L:
80             dict_solution[j] += 1
```

```
81            j = 0
82        else:
83            valeur_j = valeurs[j]
84            #on cherche la piece/billet qui permet d'obtenir la valeur précédente
85            for d in L:
86                if j-d >= 0 and valeurs[j-d] == valeur_j - 1:
87                    j = j-d
88                    dict_solution[d] += 1
89                    break
90
91    print(dict_solution)
92    return valeurs[n]
93
94 print(MonnaieProgDynAvecCombinaison(655338,L))
```

Listing 1 – Rendu de monnaie

Exercice 5 (Répartition de médicaments)**.**

Une ONG dispose d'un stock de denrées (médicaments) en quantité n. Quand l'ONG distribue une quantité i de denrées à un endroit, l'expérience montre que cela sauve $S[i]$ personnes. Le tableau S a été établi par des experts et mis à disposition à l'ONG. On cherche à calculer le nombre maximal de personnes que l'on pourra sauver avec le stock en le découpant en sous-parties. Par exemple, avec le tableau $S = [1, 3, 2]$ (c'est-à-dire $S[1] = 1$, $S[2] = 3$, $S[3] = 2$) et $n = 3$, on peut sauver :

- soit $1 + 1 + 1 = 3$ personnes en coupant le stock en 3 parties de taille 1 ;
- ou bien, 2 personnes en laissant le stock en une seule partie de taille 3 ;
- ou bien, $1 + 3 = 4$ personnes en coupant le stock en 1 partie de taille 1 et 1 partie de taille 2.

1. On note $m[i]$ la meilleure solution (nombre de personnes sauvées) pour un stock de taille i. Exprimer $m[i]$ récursivement, en fonction des valeurs $m[j]$ avec $j < i$, en utilisant aussi S lorsque nécessaire.
2. En déduire un algorithme récursif `RepartitionRec(n,S)` pour calculer la valeur optimale pour une valeur de n et un tableau $S[1...n]$ donné.
3. Quelle est la classe de complexité (approximative) en temps de cet algorithme ?
4. Transformer cet algorithme en algorithme itératif `RepartitionProgDyn(n,S)` en utilisant le principe de la programmation dynamique.
5. Quelle est sa complexité asymptotique en temps ?

Solution.

1. $m[i] = \max\{S[i], \max_{0<j<i} m[j] + S[i-j]\}$
2.

```
RepartitionRec(n,S):

Entrée : un entier n, une liste S d'entiers

Sortie : le nombre maximum de personnes qui peuvent être sauvées avec
     le stock de taille n

        ⋆ max = S[n]
        ⋆ Pour chaque j < n:
            - val = RepartitionRec(j)+S[n-j]
            - Si val > max:
              max = val
        ⋆ Retourner max
```

3. Complexité exponentielle, car on a un arbre des appels récursifs où chaque noeud a potentiellement n enfants et la hauteur est d'ordre n, donc n^n noeuds au total. (Chaque noeud représente un appel récursif.)
4.

```
RepartitionProgDyn(n,S):

Entrée : un entier n, une liste S d'entiers

Sortie : le nombre maximum de personnes qui peuvent être sauvées avec
     le stock de taille n

    • Initialiser un tableau des valeurs optimales m[1..n] rempli de 0
    • Pour i allant de 1 à n:
        ⋆ max = S[i]
        ⋆ Pour j allant de 1 à i-1:
            - val = m[j] + S[i-j]
            - Si val > max:
              max = val
        ⋆ m[i] = max
    • Retourner m[n]
```

5. $\Theta(n^2)$ car pour chaque valeur de i entre 1 et n on fait une boucle sur $1...n$.

Exercice 6 (Plus grand sous-tableau croissant)**.**

Étant donné un tableau de N entiers, on cherche le plus grand sous-tableau d'éléments croissants (pas forcément consécutifs).

Pour le résoudre on va construire les solutions possibles de façon incrémentale.

On suppose qu'on a un tableau d'entiers $TAB[1...N]$. Soit $d[i]$ la taille d'un plus long sous-tableau de TAB croissant dont le dernier élément est $TAB[i]$.

1. Exprimer $d[i]$ en fonction des valeurs $d[j]$ avec $j < i$.
2. En déduire un algorithme récursif pour calculer *la taille* d'un plus grand sous-tableau de TAB croissant.
3. L'algorithme récursif précédent est néanmoins exponentiel. Le transformer en algorithme itératif utilisant la technique de la programmation dynamique.
4. Quelle est sa complexité en temps ?
5. Quelle modification peut-on faire pour aussi calculer le plus long sous-tableau croissant (pas juste sa taille) ?

Solution.

1. $d[i] = \max_{j<i \text{ et } TAB[j] \leq TAB[i]} d[j] + 1$ s'il existe j avec $j < i$ et $TAB[j] \leq TAB[i]$, et sinon, $TAB[i] = 1$.
2.

```
tailleMaxSousTableauCroissantRec(T): taille d'un plus long sous-tableau
croissant du tableau T d'entiers

Entrée : Un tableau d'entiers TAB[1...N]
Sortie : taille d'un plus long sous-tableau croissant de T

• Si N == 1 :
  Retourner 1
• Sinon :
    ⋆ max = 1
    ⋆ Pour i de 1 à N − 1 :
      n = tailleMaxSousTableauCroissantRec(TAB[1...i])+1
      Si TAB[i] ≤ TAB[N] et n > max :
      max = n
    ⋆ Retourner max
```

3.

```
tailleMaxSousTableauCroissant(TAB): taille d'un plus long sous-tableau
croissant du tableau TAB d'entiers

Entrée : Un tableau d'entiers TAB[1...N]
Sortie : taille d'un plus long sous-tableau croissant de T

• Soit d[1...N] = [1, ..., 1] un tableau rempli de 1
• Pour i de 1 à N :
    ⋆ max = 1
    ⋆ Pour j de 1 à i − 1 :
      n = d[j] + 1
      Si TAB[j] ≤ TAB[i] et n > max :
      max = n
• Retourner maximum(d[1], ..., d[N])
```

4. $\Theta(N^2)$
5. Il faut, en plus du calcul de $d[i]$, aussi conserver l'indice $p[i]$ qui précède $TAB[i]$ dans un plus long sous-tableau croissant qui finit avec $TAB[i]$, et "backtracker" (revenir sur ses pas) pour reconstruire la solution.