Chapitre 10

Décomposition en sous-problèmes

Notions	Commentaires
Diviser pour régner. Rencontre au milieu. Dichotomie.	On peut traiter un ou plusieurs exemples comme : tri par partition-fusion, comptage du nombre d'inversions dans une liste, calcul des deux points les plus proches dans une ensemble de points ; recherche d'un sous-ensemble d'un ensemble d'entiers dont la somme des éléments est donnée ; recherche dichotomique dans un tableau trié. On présente un exemple de dichotomie où son recours n'est pas évident : par exemple, la couverture de n points de la droite par k segments égaux de plus petite longueur.
Programmation dynamique. Propriété de sous-structure optimale. Chevauchement de sous-problèmes. Calcul de bas en haut ou par mémoïsation. Reconstruction d'une solution optimale à partir de l'information calculée.	On souligne les enjeux de complexité en mémoire. On peut traiter un ou plusieurs exemples comme : problème de la somme d'un sous-ensemble, ordonnancement de tâches pondérées, plus longue sous-suite commune, distance d'édition (Levenshtein).

Extrait de la section 4.3 du programme officiel de MP2I : « Décomposition en sous-problèmes ».

Sommaire

0 Diviser pour Régner

0.0 Exemple introductif : le tri fusion

Remarque. En plus de présenter des méthodes algorithmiques **extrêmement** puissantes, ce cours présente aussi des algorithmes de tri usuels à savoir refaire.

On s'intéresse dans cet exemple introductif au problème du Tri, qui consiste à prendre en entrée $x0, ..., x_{n-1}$ une liste linéaire d'éléments munis d'un ordre total, et à ordonner par ordre croissant ces x_i.

0.0.0 Borne inférieure en pire des cas pour un tri par comparaisons

Un théorème très important concerne les solutions à ce problème :

Théorème 1 (Borne inférieure sur les tris).
Tout algorithme qui résout le problème de Tri en ne distinguant les éléments que par des comparaisons entre eux s'exécute en $\Omega(n \log_2 n)$ comparaisons.

Démonstration. Je donne ici une version vulgarisée de la preuve. J'en connais deux versions rigoureuses que vous verrez peut-être en école (ou en MPI ?) : par l'exhibition d'un adversaire, ou par un calcul d'entropie.

Il y a $n!$ façons possibles d'ordonner $x_0, ..., x_{n-1}$. Une comparaison consiste à poser la question $x_i < ? x_j$ et élimine donc toutes les façons d'ordonner qui ne respectent pas cette comparaison. Ainsi, dans le pire des cas, chaque comparaison élimine au plus la moitié ordonnacements possibles restants. Il faut donc $\Omega(\log_2(n!))$ comparaisons. Or, $log_2(n!) \sim nlog_2 n$ donc il faut $\Omega(\log_2(n!))$ comparaisons. □

Remarque.

- La plupart des algorithmes de tri sont en $O(n^2)$.
- Une autre façon de formuler ce théorème est « Sans hypothèse supplémentaire sur les données en entrée, tout algorithme s'exécute de tri s'exécute en $\Omega(n \log_2 n)$.
- Il existe des algorithmes [1] qui passent sous cette borne inférieure : généralement, ils se basent sur le fait que les données sont des entiers/flottants pour pouvoir raisonner autrement que par comparaisons et échapper au théorème.

0.0.1 Tri fusion

Définition 2 (Tri fusion).
Pour trier une liste linéaire, le tri fusion fonctionne ainsi :

(i) Diviser (scinder) la liste en deux sous-listes de longueurs égales (ou presque).

(ii) Trier récursivement les deux sous-listes.

(iii) Effectuer la fusion triée des deux sous-listes triées.

Les cas de base sont la liste linéaire à 0 ou à 1 élément.

Remarque. ⚠ Attention, oublier le cas de base à 1 élément est une erreur classique, qui mène à une boucle infinie.

Exemple. La figure ci-dessous résume l'évolution des divisions et des fusions pour `[38; 27; 43; 3; 9; 82; 10]`. La même de division utilisée consiste à mettre les $\lceil \frac{n}{2} \rceil$ premiers éléments dans une liste, et les $\lfloor \frac{n}{2} \rfloor$ suivants dans l'autre :

1. Par exemple le tri par dénombrement ou le tri par base.

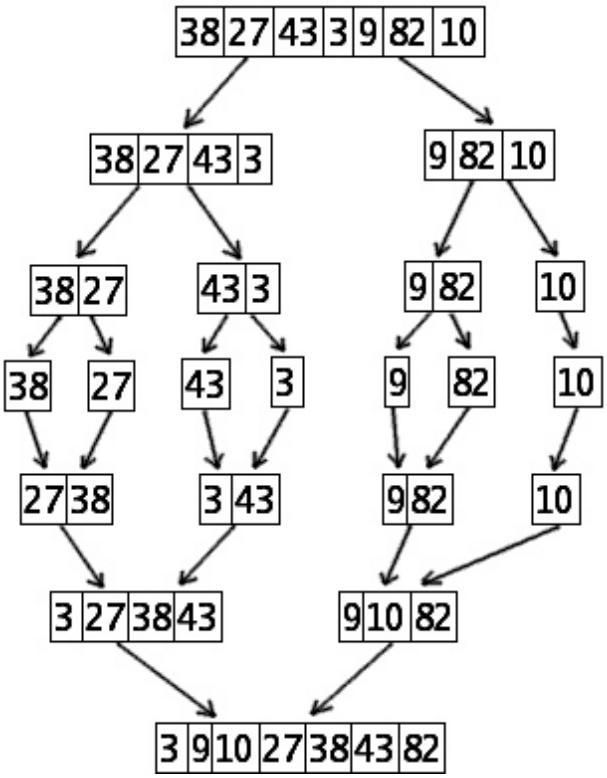

FIGURE 10.1 – Tri fusion de `[38; 27; 43; 3; 9; 82; 10]`. Src : Wikipédia

0.0.2 Implémentation OCaml

En OCaml, c'est la fonction de division `divide` qui est la plus compliquée à écrire. L'idée est de découper la liste selon « la parité de l'indice » : le premier élément va dans la première sous-liste et le second dans la deuxième, le troisième élément dans la première et le quatrième dans la deuxième, le cinquième dans la première et le sixième dans la deuxième, etc etc.

mergesort.ml

```
16 let rec divide lst =
17   match lst with
18   | []
19   | [_] -> lst, []
20   | t0::t1::q ->
21       let (d0, d1) = divide q in
22       t0::d0 , t1::d1
```

La fusion est la fonction `merge` déjà étudiée :

mergesort.ml

```
24 let rec merge lst0 lst1 =
25   match (lst0, lst1) with
26   | [], l
27   | l, [] -> l
28   | t0::q0, t1::q1 ->
29       if t0 < t1 then
30         t0 :: (merge q0 lst1)
31       else
32         t1 :: (merge lst0 q1)
```

L'algorithme s'écrit alors ainsi :

mergesort.ml

```
let rec mergesort lst =
  match lst with
  | []
  | [_] -> lst
  | _ -> let lst0, lst1 = divide lst in
         merge (mergesort lst0) (mergesort lst1)
```

0.0.3 Implémentation en C

La division en deux sous-tableaux est élémentaire en C si l'on se souvient qu'un tableau est affaibli en pointeur et est donc en fait l'adresse de sa première case. Ainsi, on va diviser en deux moitiés :

- La première moitié débute au même endroit que le tableau initial mais est deux fois plus courte.
- La seconde moitié débute à la case d'après la première moitié et prend les éléments restants.

FIGURE 10.2 – Pointeurs pour la division pour `38; 27; 43; 3; 9; 82; 10`

Toutefois, la fonction de fusion est moins agréable à écrire en C qu'en OCaml, à cause des cas où l'on a fini de parcourir un des deux tableaux à fusionner mais pas encore fini l'autre :

mergesort.c

```
int* merge(int const* arr, int len_arr, int const* brr, int len_brr) {
  int len = len_arr + len_brr;
  int* merged = (int*) malloc(len*sizeof(int));

  int j_arr = 0; // prochain élément de arr
  int j_brr = 0; // prochain élément de brr
  for (int i = 0; i < len; i += 1) {
    // On fait merged[i] = min(arr[j_arr], brr[j_brr])
    if (j_arr >= len_arr) {
      merged[i] = brr[j_brr];
      j_brr += 1;
     }
    else if (j_brr >= len_brr
             || arr[j_arr] <= brr[j_brr]) {
      merged[i] = arr[j_arr];
      j_arr += 1;
    }
    else {
      merged[i] = brr[j_brr];
      j_brr += 1;
    }
  }

  return merged;
}
```

FIGURE 10.3 – Premières et dernières étapes de la fusion triée de `[0; 1; 3; 5; 6]` et `[2; 8; 9]`

On en déduit le tri fusion :

mergesort.c

```c
int* mergesort(int const* arr, int len) {
  // Cas de base
  if (len == 0) { return NULL; }
  else if (len == 1) {
    int* sorted = malloc(sizeof(int));
    sorted[0] = arr[0];
    return sorted;
  }

  // Diviser ET trier
  int len_left = len - len/2;
  int len_right = len/2;
  int* left  = mergesort(arr, len_left);
  int* right = mergesort(&arr[len_left], len_right);

  // Fusionner
  int* sorted = merge(left, len_left, right, len_right);

  // Free et return
  free(left);
  free(right);
  return sorted;
}
```

Remarque. Ce tri **n**'est **pas** en place, c'est à dire qu'il crée une nouvelle version triée du tableau au lieu de se limiter à modifier le tableau donné en entrée. Écrire un tri fusion en place efficace est plus compliqué (à cause de l'étape de fusion, qui si elle est mal écrite est trop couteuse).

0.0.4 Analyse de complexité

On va ici prouver la complexité de la version OCaml en fonction de n la longueur de la liste, la version C se traitant de manière similaire.

- `divide` : Chaque appel effectue localement un nombre constant d'opérations, ainsi que 1 appel récursif. À chaque appel récursif, la longueur de la liste diminie de 2 et est minorée par 0 : donc `divide` est en $O(n)$.
- `merge` : Notons n_0 la longueur de `lst0` et n_1 de `lst1`. Chaque appel effectue localcement un nombre, ainsi que 1 appel récursif dans lequel la longeur de l'une des deux listes diminue de 1 et l'autre est inchangée : donc `merge` est en $O(n_0 + n_1)$.
- `mergesort` : Chaque appel effectue localement un nombre d'opérations dominé par les appels à `divide` et `merge`. Le premier est en $O(n)$, et le second fusionne les deux listes renvoyées par `divide` (qui ont donc n élément au total) donc est en $O(n)$ aussi. De plus, chaque appel effectue 2 appels récursifs sur des listes de longueur $\lceil \frac{n}{2} \rceil$ et $\lfloor \frac{n}{2} \rfloor$. Donc l'équation de complexité est :

$$T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + O(n)$$

On peut enlever les parties entières et le $O()$ sans changer l'ordre de grandeur asymptotique du résultat[2]. On résout donc :

$$T(n) = 2T(\frac{n}{2}) + n$$

Par la méthode par arbre :

FIGURE 10.4 – Arbre de complexité de $T(n) = 2T(\frac{n}{2}) + n$

Ici chaque ligne de l'arbre coute n, il y a $1 + \log_2 n$ lignes, donc la complexité $T(n)$ vérifie :

$$T(n) = O(n \log_2 n)$$

Remarque. D'après le théorème 1, l'ordre de grandeur de la complexité du tri fusion est optimal !

2. *Trust me I'm a computer scientist.*

0.1 Principe général d'un algorithme Diviser pour Régner

Définition 3 (Diviser pour Régner).
La méthode Diviser pour Régner (*Divide and Conquer*) permet de résoudre certains problèmes algorithmiques. Pour résoudre une instance I de taille n d'un problème, elle propose de procéder ainsi :

(i) Diviser l'instance I en r instances indépendantes (du même problème) de taille n/c.

(ii) Résoudre récursivement ces r instances.

(iii) Fusionner (« Régner ») les solutions de ces r instances en une solution de I.

Remarque.

- Cette méthode est naturellement récursive.
- Il faut bien sûr traiter les cas de base à part.
- Le tri fusion est un exemple de méthode Diviser Pour Régner.
- Cette méthode n'est pas une recette à infaillible : pour l'appliquer, il faut (et suffit) qu'une solution de l'instance puisse être exprimée comme la fusion de solutions d'une divison de l'instance !!
- Rien n'impose que $r = c$: le nombre de sous-problèmes auxquels on se réduit et la taille de ces sous-problèmes sont, a priori, non-liés. Cependant, en pratique, chaque « élement » de l'instance en entrée va dans au plus une sous-instance et on a donc $r \leq c$.

FIGURE 10.5 – Division d'une instance en sous-instances

0.2 Analyses de complexité

En ignorant [3] [4] les parties entières et les $O()$, l'équation de récurrence vérifiée par la complexité d'un algorithme Diviser pour Régner est de la forme :

$$T(n) = rT(n/c) + f(n)$$

Avec r le nombre de sous-problèmes auxquels on se ramène, n/c la taille des sous-problèmes, et $f(n)$ le coût local (somme du coût de la division et de la fusion).

Nous avions déjà étudié dans le cours sur la complexité comment résoudre de telles équations. Pour rappel, on trace l'arbre d'appels qui est de la forme, on calcule le coût de chaque étage de l'arbre d'appel et on somme les coûts de ces étages. Cette somme est facile à calculer si l'on peut remarquer que :

- Le coût des étages croit (au moins) géométriquement : en exploitant cela, on montre que le coût total est dominé par le coût de l'étage du bas.
- Le coût des étages est constant.
- Le coût des étages décroit (au moins) géométriquement : en exploitant cela, on montra que le coût total est dominé par le coût de la racine.

3. *Trust me !*
4. Si vous voulez vérifier ma source : `https://courses.csail.mit.edu/6.046/spring04/handouts/akrabazzi.pdf`

Remarque.

- Ces trois cas de figure sont des « astuces » à savoir appliquer, pas une propriété invocable. Autrement dit, une copie disant "d'après le cours, le coût des feuilles domine" n'aura pas les points liés au calcul.
- Pour rappel, l'arbre de $T(n) = rT(n/c) + f(n)$ ressemble à ceci :

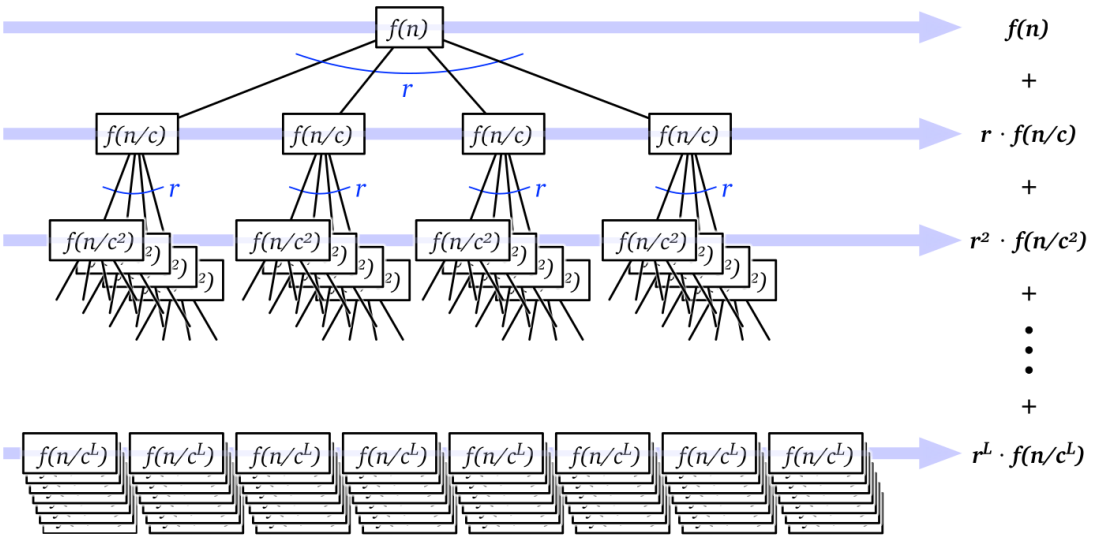

FIGURE 10.6 – Arbre de $T(n) = rT(n/c) + f(n)$. Src : J. Erikson

0.3 Exemple : recherche dichotomique

0.3.0 La recherche dichotomique

La recherche dichotomique d'un élément `x` dans un tableau `arr` est un exemple de méthode Diviser pour Régner, où la Division consiste à se réduire à 1 instance de taille `len/2`. L'étape de Fusion est vide, puisque l'on renvoie immédiatement ce que l'appel récursif renvoie. La voici ici écrite (récursivement) en C, avec des petites manipulations de pointeurs comme dans le tri fusion :

dichotomie.c

```c
/* Recherche x dans arr (à len éléments) */
bool recherche(int x, int const* arr, int len) {
  // Cas de base
  if (len == 0) { return false; }
  if (len == 1) { return arr[0] == x; }

  // Cas récursif
  int mid = len/2;
  if (arr[mid] == x) {      // On a trouvé x
    return true;
  }
  else if (x < arr[mid]) { // On cherche dans la première moitié
    return recherche(x, arr, mid);
  }
  else {                   // On cherche dans la seconde moitié
    return recherche(x, &arr[mid+1], len-mid-1);
  }
}
```

Son équation de récurrence est $T(len) = T(len/2) + O(1)$ qui a pour solution $T(len) = O(\log_2(len))$.

Figure 10.7 – Pointeurs et dichotomie

0.3.1 Une variante : la recherche d'un pic

Dans un tableau *arr* d'entiers distincts (de longueur *len*), un **pic** est une case plus grande que ses deux voisines, c'est à dire un indice $i \in [\![0; len[\![$ tel que $arr[i-1] < arr[i]$ et $arr[i] > arr[i+1]$. Pour gérer le cas particulier des bords, on pose $arr[-1] = -\infty$ et $arr[len] = +\infty$: ainsi la définition de **pic** fonctionne aussi sur les bords :

5	3	**14**	6	2	10	**11**	9

Figure 10.8 – Pics (en gras) dans un tableau

Le problème Peak Finding est le problème de trouver *un* pic (n'importe lequel) dans un tableau donné en entrée.

Il existe une solution dichotomique très élégante à ce problème. Elle consiste à remarquer que si $arr[lo-1] < arr[lo]$ et que $arr[hi] > arr[hi+1]$, alors tout pic de $arr[lo : hi[$ est un pic de *arr* (avec $arr[i : j[$ le sous-tableau de *arr* allant des indices *i* inclus à *j* exclu).

Figure 10.9 – Invariant de la recherche de pic

On en déduit le code récursif suivant :

peak-finding.c

```c
/** Recherche un pic dans arr[lo; hi[ */
int peak_finding_rec(int const arr[], int lo, int hi) {
  int mid = (lo+hi)/2;

  if (lo < mid && arr[mid-1] > arr[mid]) {
    return peak_finding_rec(arr, lo, mid);
  }
  else if (mid + 1 < hi && arr[mid] < arr[mid+1]) {
    return peak_finding_rec(arr, mid+1, hi);
  }
  else {
    return mid;
  }
}
```

En notant $n = hi - lo$, son équation de complexité est $T(n) = T(n/2) + O(n)$ donc $T(n) = O(\log_2 n)$.

Remarque. ⚠ Pour faire fonctionner ce code C, il faut préparer le tableau de sorte à ce que $arr[-1]$ et $arr[len]$ soient valides...

Exercice. Déduire de la fonction précédente une fonction `peak_finding` qui prend en entrée un tableau d'entiers et renvoie l'indice de l'un de ses pics

0.4 Un autre exemple : algorithme de Karatsuba

0.4.0 Présentation du problème

On s'intéresse ici au problème de savoir multiplier deux entiers x et y écrits en binaire, tous les deux sur n bits (quitte à rajouter des 0 à droite au plus court des deux).

Les algorithmes classiques pour la multiplication (par exemple celui vu à l'école primaire) s'exécute en $O(n^2)$. Nous allons ici faire un Diviser pour Régner qui bat cette complexité.

0.4.1 Un Diviser pour Régner inefficace

L'idée de l'étape de division est de couper chacun des entiers en 2 moitiés (bits de poids faible et bits de poids forts). Par exemple :

$$44 = {}^2\overline{0010\ 1100} = {}^2\overline{0010} \cdot 2^4 + {}^2\overline{1100}$$

Pour simplifier, on suppose que n est une puissance de 2. Ainsi, x se décompose en $x = x_{hi} \cdot 2^{n/2} + x_{lo}$ avec x_{hi} et x_{lo} tous deux sur $n/2$ bits. De même pour y avec y_{hi} et y_{lo}.

Reste maintenant le plus difficile : quelles mutiplicatiopns d'entiers de $n/2$ bits faire pour retrouver $x \cdot y$? On veut faire le moins de multiplication possibles. Notons tout d'abord que les multiplication par une puissance de 2 sont peu couteuses : il s'agit d'un simple décalage des bits[5].

Ensuite, développons le produit :

$$\begin{aligned} x \cdot y &= (x_{hi}2^{n/2} + x_{lo}) \cdot (y_{hi}2^{n/2} + y_{lo}) \\ &= x_{hi} \cdot y_{hi}2^n + x_{lo} \cdot y_{lo} + (x_{hi} \cdot y_{lo} + x_{lo} \cdot y_{hi})2^{n/2} \end{aligned}$$

Si on applique cette formule, en admettant que l'addition et la multiplication par une puissance de 2 se fait en $O(n)$, on obtient une complexité vérifiant $T(n) = 4T(n/2) + O(n)$ (le cas de base est la multiplication de deux entiers à 1 bit, qui est triviale). Traçons son arbre :

FIGURE 10.10 – Arbre de complexité de $T(n) = 4T(n/2) + O(n)$

Cet arbre a $1 + \log_2 n$ étage, et l'étage i coute $e_i = 4^i \frac{n}{2^i} = 2^i n$. Donc[6] le cout total vérifie :

5. Pour la même raison que multiplier par 10^k est simple en base 10.

6. On peut remarquer que le coût des étages croît géométriquement : le reste de notre calcul doit donc être de l'ordre du coût de la dernière ligne, c'est à dire de $4^{\log_2 n} \cdot 1 = n^2$.

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_2 n} e_i \\ &= n(2^{\log_2 n+1} - 1) \\ &= O(n^2) \end{aligned}$$

C'est à dire que l'on a rien gagné par rapport à l'algorithme naïf. Le problème ici est soit que le coût de la division/fusion est trop élevé - mais on ne voit pas trop comment l'améliorer - soit que la division n'est pas assez efficace : on a divisé en trop de sous-problèmes, ou en des sous-problèmes trop grands.

0.4.2 Algorithme de Karatsuba

L'algorithme de Karatsuba consiste à procéder comme précedemment, mais à remarquer que le terme $x_{hi} \cdot y_{lo} + x_{lo} \cdot yhi$ peut-être exprimé à l'aide des autres multiplications que l'on effectue et d'une nouvelle multiplication :

$$\begin{aligned} x_{hi} \cdot y_{lo} + x_{lo} \cdot yhi = & \underbrace{x_{hi} \cdot y_{hi}}_{\text{déjà calculé}} + \underbrace{x_{lo} \cdot y_{lo}}_{\text{déjà calculé}} \\ & - (x_{hi} - x_{lo}) \cdot (y_{hi} - y_{lo}) \end{aligned}$$

Ainsi, pour calculer $x \cdot y$, on se ramène aux trois instances suivantes :

- Calculer $x_{hi} \cdot y_{hi}$
- Calculer $x_{lo} \cdot y_{lo}$
- Calculer $(x_{hi} - x_{lo}) \cdot (y_{hi} - y_{lo})$

Le coût de la division est donc $O(n)$ (calcul des *lo* et *hi* ainsi que deux soustractions) et celui de la fusion $O(n)$ (des additions et des multiplications par une puissance de 2). Donc l'équation de complexité est $T(n) = 3T(n/2) + O(n)$. Traçons son arbre :

Figure 10.11 – Arbre de complexité de $T(n) = 3T(n/2) + O(n)$

Il y a $1 + \log_2 n$ étage, et l'étage i coûte $e_i = 3^i \frac{n}{2^i} = \left(\frac{3}{2}\right)^i n$. En notant $h = \log_2 n$, on constate que[7] $e_{i+1} = \frac{3}{2} e_i$ donc que $e_i = \left(\frac{2}{3}\right)^{h-i} e_h$. Il s'ensuit :

7. Croissance géométrique à nouveau !

$$\begin{aligned} T(n) &= \sum_{i=0}^{h} e_i \\ &= e_h \underbrace{\sum_{i=0}^{h} \left(\frac{2}{3}\right)^{h-i}}_{\text{converge}} \\ &= O(e_h) \end{aligned}$$

Or :

$$\begin{aligned} e_h &= 3^{\log_2 n} \cdot 1 \\ &= e^{\ln(3)\frac{\ln(n)}{\ln(2)}} \\ &= n^{\log_2(3)} \\ &\simeq n^{1,585} \end{aligned}$$

Où l'arrondi est fait à la valeur supérieure : d'où $T(n) = O\left(n^{\log_2(3)}\right) = O\left(n^{1,585}\right)$. On a battu les algorithmes naïfs !

Remarque. L'algorithme de Karatsuba est un excellent exemple de la difficulté d'optimiser des Diviser pour Régner : il ne faut pas juste trouver une idée récursive, il faut aussi se demander si on ne peut pas *mieux* diviser ou *mieux* régner. Tout gain sur l'une de ces étapes est significativement récpercuté sur la complexité du tout.

0.5 Pire cas et cas moyen : le tri rapide

0.5.0 Complexité en moyenne

Définition 4 (Complexité moyenne).
La complexité en moyenne d'un algorithme est la fonction T_{moy} qui à n associe la moyenne des complexités des instances de taille n. Autrement dit :

$$T_{moy}(n) = \mathbb{E}(\text{complexité d'une instance de taille } n \text{ prise uniformément au hasard})$$

Proposition 5.
Pour tout algorithme, avec T_{best} sa complexité meilleur cas, T_{moy} sa complexité moyenne, et T_{pire} sa complexité pire des cas, on a :

$$T_{best}(n) \leq T_{moy}(n) \leq T_{pire}(n)$$

Démonstration. Immédiatement en développant l'espérance comme une somme et en minorant/majorant ses termes par le meilleur et le pire cas. □

0.5.1 Borne inférieure en moyenne pour un tri par comparaisons

Le théorème 1 est formulé comme une complexité pire des cas. On peut montrer qu'il est aussi valable dans le cas moyen[8]. C'est à dire que si on prend une liste de longueur n uniformément au hasard, et qu'on la trie par comparaisons, l'espérance du nombre de comparaisons requises sera un $\Omega(n \log_2 n)$.

Autrement dit « le théorème 1 n'est pas seulement valable dans le pire des cas, mais aussi "en pratique", quand on prend une liste à peu près quelconque[9] ».

8. Après formalisation, il s'agit de montrer que pour un arbre binaire à $n!$ feuille, la profondeur moyenne des feuilles est un $\Omega(n \log_2 n)$.
9. Entendre par là « ui n'est ni un meilleur ni un pire cas.

Bien entendu, comme le tri fusion est en $O(n \log 2n)$ dans le pire des cas, il l'est aussi dans le cas moyen. Cependant, en pratique, il existe un autre algorithme qui s'exécute plus rapidement : le tri rapide.

0.5.2 Tri rapide

Définition 6 (Tri rapide).
Pour trier une liste linéaire, le tri rapide fonctionne ainsi :

() Choisir un élément p dans la liste et l'appelmer *pivot*

(i) Diviser la liste en les éléments inférieurs au pivot et les éléments supérieurs au pivot

(ii) Trier ces deux sous-listes

(iii) Concaténer le tout

Le cas de base est celui de la liste à 0 ou 1 élément.

Lorsqu'il est bien implémenté, le tri rapide est... très rapide [10]. Pour sa rapidité, il y a deux facteurs cruciaux :

- Bien choisir le pivot : on veut un pivot qui mène à une division de la liste en deux sous-listes à peu près de longueur égale.
- La division doit s'exécuter rapidement.

On voudrait de plus le réaliser *en place*, c'est à dire que l'on veut modifier le tableau pris en entrée et non créer de nouveaux tableaux.

Pour simplifier, on prendra comme pivot le premier élément du tableau [11]. Pour faire un tri rapide, il faut donc pouvoir résoudre le problème suivant (qui correspond à la phase de Division) :

- Entrée : *arr* un tableau de *len* éléments et p un pivot
- Tâche : réordonner *arr* de sorte à ce que tous les éléments strictement inférieurs à p soit au début, puis que suivent tous les éléments égaux à p, et enfin que terminent tous les éléments strictement supérieurs à p

FIGURE 10.12 – La forme attendue pour *arr* en sortie

0.5.3 Drapeau hollandais

Ce problème peut-être simplifié et abstrait comme le problème du drapeau hollandais [12] :

- Entrée : *arr* un tableau de *len* éléments valant 1 2 ou 3.
- Tâche : réordonner *arr* de sorte à ce qu'il contienne d'abord tous les 1, puis tous les 2, puis tous les 3. La seule modification du tableau autorisée est l'échange du contenu de 2 cases.

On voudrait de plus faire cela efficacement, si possible en temps linéaire en *len* ! Il existe une solution qui fait cela, et qui plus est qui fait cela en un seul parcours du tableau. L'idée est la suivante : la zone du tableau déjà parcourue est découpée en 3 zones comme attendu. Quand on croise un nouvel élément, on fait des échanges au niveau des fins/débuts des zones pour l'insérer efficacemment.

10. J'aime les algorithmes bien nommés.
11. Ce n'est même pas un si mauvais choix : en espérance, cela est équivalent au fait de prendre un élément au hasard dans le tableau.
12. Pourquoi hollandais ? Parce que le drapeau hollandais est tricolore, et que cette abstraction a été popularisée par E. Dijkstra qui est... hollandais.

Montrons les trois cas de figure sur des schémas. On nomme i le nombre d'éléments du tableau déjà lus : ainsi, $arr[0 : i[$ est découpé en 3 zones comme indiqué. Notons `nb_1` le nombre de 1 dans ce début de *arr*, `nb_2` le nombre de 2 dans ce début et `nb_3` le nombre dans ce début.

(a) Situation initiale : on doit insérer $arr[i]$ dans une des trois zones

(b) Insertion d'un 1 en deux échanges

(c) Insertion d'un 2 en un échange

(d) Insertion d'un 3 en un échange

FIGURE 10.13 – Résolution efficace du drapeau hollandais

Avec cette méthode, on maintient les invariants suivants :

- `arr[0 : nb_1[` ne contient que des 1.
- `arr[nb_1 : nb_1+nb_2[` ne contient que des 2
- `arr[nb_1+nb_2 : nb_1+nb_2+nb_3[` ne contient que des 3
- Avec `i` l'indice de la case actuellement lue, `nb_1` est le nombre de 1 entre les indices 0 inclus et `i` exclu dans le tableau initial (de même pour `nb_2` et `nb_3`)

À la fin du processus, les 3 premiers invariants prouve que le tableau est bien découpé en 3 zones comme attendu, et le quatrième prouve que le nombre de 1/2/3 est bien le bon : gagné !

Cette méthode correspond au pseudo-code suivant :

Algorithme 15 : Résolution du drapeau hollandais

Entrées : *arr* un tableau de *len* éléments valant 1 2 ou 3.

```
1  nb_1 ←0
2  nb_2 ←0
3  nb_3 ←0
4  pour i de 0 inclus à len exclu faire
5      si arr[i] = 1 alors
6          Échanger arr[nb_1] et arr[i]
7          Échanger arr[nb_1 + nb_2] et arr[i]
8          nb_1 ← nb_1 + 1
9      sinon si arr[i] = 2 alors
10         Échanger arr[nb_1 + nb_2] et arr[i]
11         nb_2 ← nb_2 + 1
12     sinon
13         nb_3 ← nb_3 + 1
```

0.5.4 Implémentation du tri rapide en C

Pour l'étape de division, on va prendre le premier élément comme pivot puis appliquer l'algorithme du drapeau hollandais. En termes savants, cette façon de faire s'appelle le **partitionnement de Lomuto**[13] [14].

Voici une fonction qui permet d'échanger deux valeurs :

quicksort.c

```c
/** Échange le contenu des cases pointées par a et b */
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
  return;
}
```

Et voici le tri rapide. On utilise les mêmes astuces de pointeur que précedemment. Notez que cette fois, on n'alloue pas de mémoire supplémentaire : tout est fait dans le `arr` d'origine ! Le tri est en place.

quicksort.c

```c
/** Trie arr (qui a len éléments) */
void quicksort(int* arr, int len) {
  // Cas de base
  if (len <= 1) { return; }

  // Diviser
  int pivot = arr[0];
  int nb_pt = 0; // nb d'elem < pivot
  int nb_eq = 0; //           ==
  int nb_gd = 0; //           >
  for (int i = 0; i < len; i += 1) {
    if (arr[i] < pivot) {
      swap(&arr[nb_pt], &arr[i]);
      swap(&arr[nb_pt+nb_eq], &arr[i]);
      nb_pt += 1;
    }
    else if (arr[i] == pivot) {
      swap(&arr[nb_pt + nb_eq], &arr[i]);
      nb_eq += 1;
    }
    else {
      nb_gd += 1;
    }
  }

  // Régner
  quicksort(arr, nb_pt);
  quicksort(&arr[nb_pt+nb_eq], nb_gd);

  return;
}
```

13. Il existe un autre partitionnement plus efficace : le partitionnement de Hoare.
14. En réalité le partitionnement de Lomuto est plus simple : il divise en deux zones, une inférieure ou égale au pivot et l'autre strictement supérieure au pivot.

0.5.5 Analyse de complexité

L'étape de divison est linéaire en n la longueur, et la fusion est instantanée. Dans le pire des cas, le pivot divise très mal (en deux parties de taille 0 et $n-1$) : on obtient l'équation de récurrence pire des cas $T(n) = T(n-1) + O(n)$, qui a pour solution... $O(n^2)$

Pourtant, et je maintiens, ce tri est rapide !! En effet, il n'y a à peu près *que* dans le pire des cas que la division est déséquilibrée. En pratique (en *moyenne*), les deux sous-listes sont à peu près équilibrées.

Proposition 7 (Complexités du tri rapide et du tri fusion).
En notant n le nombre d'éléments :

Complexité \ Algorithme	Tri fusion	Tri rapide
Pire des cas	$O(n\log_2 n)$	$O(n^2)$ rare
Cas moyen	$O(n\log_2 n)$	$O(n\log_2 n)$ avec faible constante

Démonstration. Les complexités pire cas ont été prouvées. Le cas moyen du pire des cas est compris entre le pire cas et le théorème $\Omega(n\log_2 n)$ en complexité moyenne... d'où sa valeur.

Reste la case en bas à droite. C'est un bon exercice de proba de MP2I, que nous ferons peut-être lorsque vous aurez eu le cours de proba. En attendnat, admettons-le. □

Pour accélérer l'algorithme du tri rapide, il faut soit accélérer la division, soit garantir un bon équilibre des deux sous-listes. Pour ce second point, deux façons de faire :

- Utiliser la médiane de *arr* comme pivot : c'est possible (on peut la calculer en temps linéaire, cf `https://fr.wikipedia.org/wiki/M%C3%A9diane_des_m%C3%A9dianes` - cela devrait vous rappeller SLOWSELECT vu en DS), mais en pratique le temps de calcul de la médiane coute plus cher que le gain apportée.
- Utiliser l'élément médian parmi $arr[0], arr[len/2], arr[len-1]$: c'est en pratique incroyablement efficace et fait que le pire des cas n'arrive presque jamais !

Enfin, pour accélérer encore plus, on peut changer le cas de base : il s'avère en effet que le tri rapide n'est pas le plus rapide pour les petits tableaux. Lorsque la longueur du tableau passe en-dessous d'un certain seuil (par exemple $n \leq 4$), on termine alors non pas récursivement mais en utilisant un autre tri (le tri par insertion est populaire à cette fin).

Remarque.

- Vous devriez maintenant avoir tous les éléments théoriques pour comprendre le code source de la fonction `qsort` de C : `https://github.com/lattera/glibc/blob/master/stdlib/qsort.c` . Il vous manque un ou deux éléments pratiques (notamment ce que signifie ajouter/soustraire à un pointeur), mais avec un peu d'acharnement vous devriez finir par comprendre le fonctionnement.
- Le tri rapide est un excellent exemple de Diviser pour Régner qui n'est pas efficace dans le pire des cas mais très efficace dans le cas moyen, donc tout à fait utilisable en pratique.

1 Rencontre au milieu

1.0 Problème d'optimisation associé à SUBSETSUM

Considérons le problème d'optimisation associé à SUBSET-SUM

- Entrée : $E \subset \mathbb{N}$ un ensemble fini d'entiers positifs et $t \in \mathbb{N}$ une « cible » (**t**arget).
- Tâche : Trouver la plus grande somme d'éléments de E inférieure ou égale à t?

Nous pouvons utiliser une solution en retour sur trace[15]. Mais considérons ainsi une solution tout à fait naïve : calculer toutes les sommes possibles de E, et en déduire la plus grande inférieure ou égale à t. Cette solution naïve s'exécute en $O(2^n)$.

Nous allons mélanger cette solution naïve exponentielle avec une étape de division et de fusion pour obtenir une meilleure solution :

(i) Diviser arbitrairement E en F et G deux moitiés de tailles égales.

(ii) Calculer toutes les sommes possibles de F et G à l'aide de l'algorithme naïf.

(iii) Fusionner ces sommes de F et G en toutes les sommes possibles de E et conclure.

Remarquez que l'on ne fait *pas* un Diviser pour Régner, puisque l'on ne continue pas les Divisions/-Fusions récursivement ! Calculons la complexité, avec n le nombre d'éléments de E

- La division se fait en $O(n)$ ou $O(1)$ selon comment l'ensemble est codé. Mettons que l'on utilise des tableaux, donc $O(1)$ (en vrai, on s'en fiche vu l'ordre de grandeur des complexités suivantes).
- Les deux appels récursifs se font en temps $2^{n/2}$.
- Il y a $2^{n/2}$ sommes possibles pour F et G, donc en triant puis faisant une fusion triée l'étape de fusion se fait en temps $O\left(n2^{n/2}\right)$.

Donc la complexité du tout est un $O\left(n2^{n/2}\right)$. On a divisée la complexité du naïf par un facteur $\frac{2^{n/2}}{n}$: c'est énorme !

Remarque. Ni la solution en retour sur trace, ni la solution ci-dessus ne sont en temps polynomial. Mais peut-être y a-t-il une solution intelligente qui nous échappe et réussit à atteindre la complexité polynomiale ? C'est en réalité (sans doute) voué à l'échec : en MPI, vous verrez que l'on pense qu'il n'existe pas de solution polynomiale à ce problème[16][17].

1.1 Principe général d'un algorithme Rencontre au milieu

L'idée est de s'inspirer de la méthode Diviser pour Régner pour gagner en complexité, mais en ne divisant/fusionnant qu'une seule fois (au lieu de le faire récursivement) :

Définition 8 (Rencontre au milieu).
La **rencontre au milieu** (*meet in the middle*) s'applique à des problèmes dont on dispose déjà d'une solution très lente. Elle propose de procéder ainsi pour résoudre une instance I de taille n :

(i) Diviser l'instance I en r sous-instances de taille n/c

(ii) Résoudre les sous-instances à l'aide de la solution très lente

(iii) Fusionner les solutions des sous-instances en une solution de I

Comme la solution naïve est très lente, l'appliquer sur des sous-instances significativement plus petite est un gain de temps qui compense le coût de la division/fusion.

Remarque.

- Si la solution naïve est polynomiale, il n'y a aucun gain d'ordre de grandeur de complexité car $\left(\frac{n}{c}\right)^k = O\left(n^k\right)$.
- Les c peuvent être différentes entre les sous-instances.
- À vrai dire, la rencontre au milieu n'est pas, à ma connaissance, une méthode générale comme le Diviser pour Régner. C'est plus une astuce qui sert parfois, notamment et surtout pour Subsetsum.

15. Et l'élaguer efficacemment par la méthode de Séparation&Évaluation, cf MPI.
16. Il en existe une si et seulement $P = NP$; ce qui est certes une question ouverte mais la communauté des chercheur·ses penche plutôt vers $P \neq NP$.
17. Si vous n'avez pas compris la note de bas de page précédente, c'est normal : vous êtes en MP2I alors que les classes de complexité P et NP sont au programme de MPI.

2 Programmation dynamique

2.0 Exemple fil rouge : plus lourd chemin dans une pyramide

2.0.0 Présentation du problème

On considère une pyramide d'entiers comme sur la figure ci-dessous, dans laquelle on veut trouver un chemin descendant de poids maximal (en partant du sommet) :

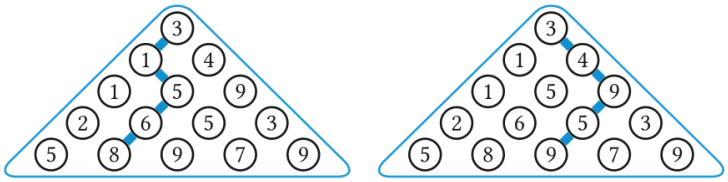

FIGURE 10.14 – Une pyramide d'entiers et deux chemins descendants dans celle-ci

Donnons-nous un peu de vocabulaire :

- Les *enfants* d'un noeud de la pyramide sont les deux noeuds qui sont respectivement juste en dessous à gauche et juste en dessous à droite.
- Un *chemin* descendant est une succession d'enfants depuis un point de départ donné et jusqu'à la ligne du bas.
- Le *poids* d'un chemin est la somme des valeurs des noeuds du chemin.
- Pour un noeud, on définit ses sous-pyramides gauche et droite :

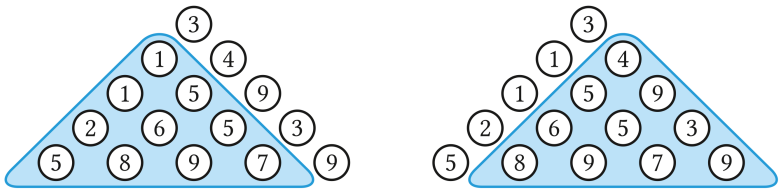

FIGURE 10.15 – Les deux sous-pyramides (ici du sommet). On remarque qu'elles s'intersectent (beaucoup).

Remarque. Les pyramides ne sont *pas* des arbres, car les sous-arbres ne sont pas disjoints. Par exemple, le noeud 5 de la troisième ligne est à la fois l'enfant droit du 1 et l'enfant gauche du 4 de la ligne d'au-dessus.

Dans un premier temps, on va uniquement chercher $s_{i,j}$, le poids d'un plus long chemin descendant depuis le noeud de coordonnées (i, j).

2.0.1 Solution naïve en OCaml

On va utiliser un tableau de tableaux pour représenter une pyramide en OCaml. Par exemple, la pyramide ci-dessous correspond à :

pyramide.ml

```
let pyramide_cours =
  [|     [|3|];
        [|1; 4|];
       [|1; 5; 9|];
      [|2; 6; 5; 3|];
     [|5; 8; 9; 7; 9|]
  |]
```

Ainsi, la première ligne est `pyramide_cours.(0)` , la seconde ligne `pyramide_cours.(1)` , etc. Remarquez que les enfants du noeud de coordonnée (i, j) sont les noeuds de coordonnées $(i + 1, j)$ et $(i + 1, j + 1)$. D'où :

pyramide.ml

```
44 let gauche j = j
45 let droite j = j+1
```

Débutons par une solution naïve au problème : comme pour le calcul de la hauteur d'un arbre binaire, on a $s_{i,j} = pyra_{i,j} + \max(s_{i+1,gauche(j)}, s_{i+1,droite(j)})$ (avec $pyra$ la pyramide).

FIGURE 10.16 – Solution récursive pour le plus lourd chemin

pyramide.ml

```
48 (** Renvoie le poids d'un plus lourd chemin descendant
49     dans la pyramide [pyra] qui débute en [(lgn,col)] *)
50 let rec poids_naif pyra lgn col =
51   let nb_lgn = Array.length pyra in
52   if lgn >= nb_lgn then
53     0
54   else
55     let p_gauche = poids_naif pyra (lgn+1) (gauche col) in
56     let p_droite = poids_naif pyra (lgn+1) (droite col) in
57     pyra.(lgn).(col) + max p_gauche p_droite
```

Cependant, si l'on calcule la complexité temporelle de cette fonction avec n le nombre de ligne, l'équation associée est $T(n) = 2T(n-1) + O(1) = O(2^n)$.

Pourquoi est-ce autant ? C'est pourtant un beau diviser pour régner, non ? *NON !!* Les sous-problèmes auquel on se réduit ne sont pas indépendants : on cherche un chemin dans la « sous-pyramide gauche », et un chemin dans la « sous-pyramide droite », mais celles-ci s'intersectent ! Cela mène à des appels récursifs qui ont lieu plusieurs fois :

FIGURE 10.17 – (Début de) l'arbre d'appel de `poids_naif pyramide_cours 0 0`

Exercice. Dans une exécution de `poids_naif p 0 0`, déterminer combien de fois est calculée $s_{i,j}$

Pour corriger ce défaut, nous allons utiliser de la programmation dynamique !

Remarque. De son côté, la complexité spatiale se calcule comme suit : chaque appel est localement en espace constant, l'arbre d'appel est de hauteur n, d'où une complexité spatiale linéaire en n le nombre de lignes.

2.1 Principe général de la Programmation Dynamique

Définition 9 (Cadre d'application de la programmation dynamique).
La **programmation dynamique** s'applique lorsque l'on résout un problème à l'aide d'une idée récursive ; mais que cette résolution mène à recalculer plusieurs fois les mêmes valeurs.
La programmation dynamique consiste alors à mémoriser les valeurs déjà calculées pour ne pas les recalculer.

Remarque. Le fait que le code soit basé sur une idée récursive ne signifie pas que la fonction en elle-même est récursive. Vous avez par exemple déjà codée la dichotomie en itératif alors qu'elle est basée sur une idée récursive.

2.2 Méthode de haut en bas

2.2.0 Pour les pyramides

Nous allons modifier la fonction `poids_naif` pour qu'elle mémorise les valeurs déjà calculées. Plus précisément, on va maintenir un tableau de tableaux `s` tel que `s[i][j]` vaut `None` si $s_{i,j}$ n'a pas déjà été calculée, et `Some s` avec `s` le poids associé sinon :

pyramide.ml

```
(** Renvoie le poids d'un plus lourd chemin descendant
    dans la pyramide [pyra] qui débute en [(lgn,col)].

    [s] est la matrice des valeurs déjà calculées
*)
let rec poids_memo s pyra lgn col =
  (* Si s.(i).(j) n'est pas déjà calculée : la calculer *)
  if s.(lgn).(col) = None then begin
    let nb_lgn = Array.length pyra in
    if lgn = nb_lgn-1 then (* la ligne du bas *)
      s.(lgn).(col) <- Some pyra.(lgn).(col)
    else (* pas sur la ligne du bas *)
      let g = poids_memo s pyra (lgn+1) (gauche col) in
      let d = poids_memo s pyra (lgn+1) (droite col) in
      s.(lgn).(col) <- Some (pyra.(lgn).(col) + max g d)
  end
  ;
  (* Ensuite, dans tous les cas : renvoyer la valeur mémorisée *)
  Option.get s.(lgn).(col)
```

Remarque.

- `s` est un tableau, donc son contenu est passé (comme) par référence : c'est pour cela que ça marche ! Quand un appel récursif modifie `s.(lgn).(col)`, cette modification reste là pour les appels suivants !
- À la fin, j'utilise `Option.get` : à une valeur de type `'a option` de la forme `Some p`, elle associe `p` (et sinon lève une exception).

- Pourquoi utiliser des Option ? Une alternative est d'utiliser une valeur particulière qui signifie « la réponse n'a pas encore été calculée ». Par exemple, si j'admets que tous les noeuds contiennent une valeur positive, -1 est une valeur possible pour signifier « pas encore calculé ». Cela a cependant plusieurs inconvénients majeurs :
  - Il faut trouver une telle valeur particulière. En effet, se tromper sur elle plante tout l'algorithme (l'algo croit qu'une valeur n'est pas calculée alors qu'elle l'est...).
  - Utiliser des options permet d'obtenir des erreurs *de type*, *à la compilation* si jamais on les manipule mal et que l'on confond valeur déjà calculée et valeur non encore calulée. C'est *extrêmement* appréciable : le débogage se fait à la compilation (avec un message qui indique la ligne précise de l'erreur) et non à l'exécution [18] !!

Exemple. Déroulons cette algorithme sur la pyramide précédente, en partant de $(0, 1)$:

FIGURE 10.18 – Arbre d'appels `poids_memo` en partant de $(0, 1)$. La mémoïsation revient à élaguer l'arbre.

18. De manière générale c'est, et à raison, un argument majeur des langages fortement typés : le compilateur détecte énormément d'erreurs qui seraient détectés à l'exécution dans d'autres langages.

2.2.1 Calcul de la complexité

Calculer la complexité de `poids_memo` . L'analyse usuelle des fonctions récursives n'est pas très pratique ici, car l'idée de cette fonction est juste de faire en sorte que le pire des cas n'arrive qu'une fois.

À la place, comptons combien de fois chaque valeur est appelée. Remarquons pour cela un lemme central :

Lemme 10.
Dans un algorithme de programmation dynamique mémoïsé (de haut en bas), le calcul de la solution d'une instance se fait récursivement au plus une fois.

Démonstration. Cf le code : si la valeur est déjà calculée, on se contente de la renvoyer (et donc on ne continue pas récursivement). Si elle n'est pas calculée, on la calcule récursivement... et on la stocke pour les appels suivants qui ne vont donc pas la recalculer récursivement ! □

Or, pour `pyra_memo` , l'appel sur (i, j) peut-être provenir d'un appel sur $(i-1, j-1)$ (le parent a appelé son enfant droit) ou $(i-1, j)$ (le parent a appelé son enfant gauche). Chacune de ces deux façons d'appeler (i, j) a lieu a plus une fois (cf le lemme). Ainsi, le nombre d'appels qui a lieu est au plus le double du nombre de cases de `s`.

En notant n le nombre de lignes, il y a $O(n^2)$ cases. Or, chaque appel effectivement localement un nombre constant d'opérations. Donc la complexité temporelle est de la forme $T(n) = O(n^2)$; c'est à dire linéaire en la taille de la pyramide !!

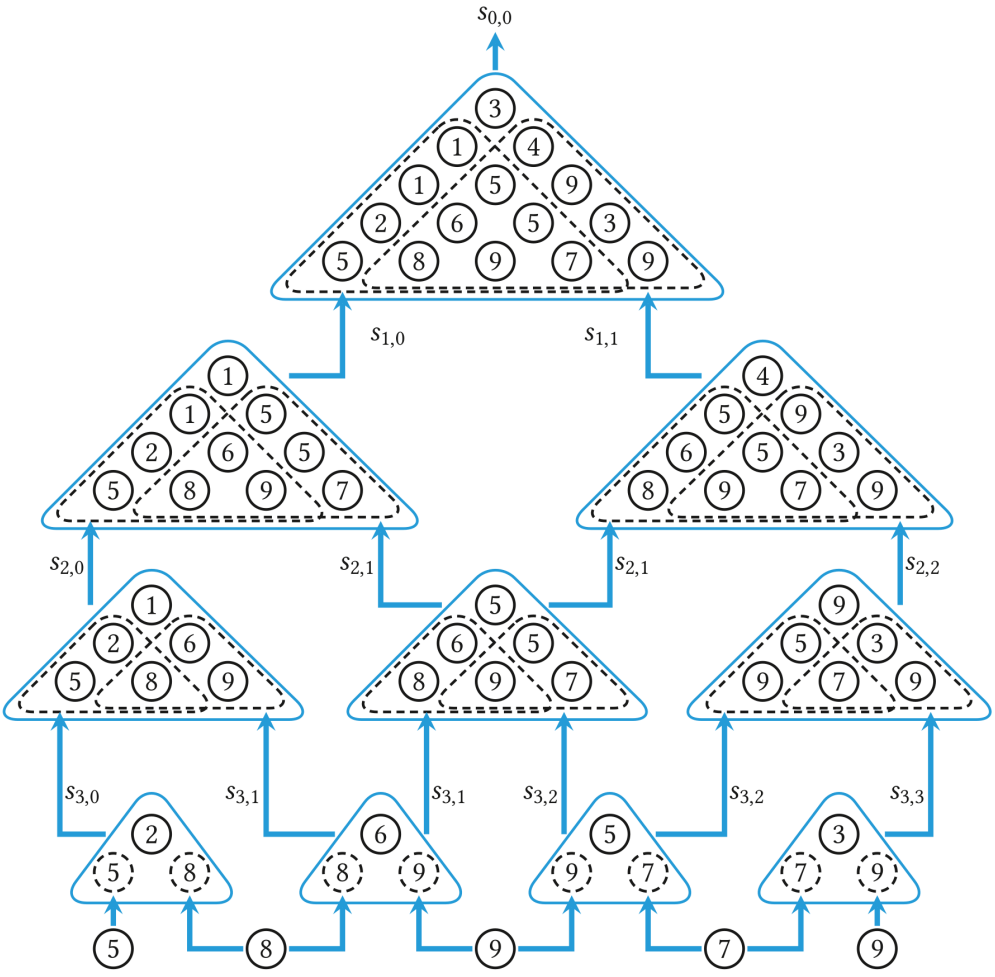

FIGURE 10.19 – Arbre des dépendances entre les instances.

Passons maintenant à la complexité spatiale : il y a les mêmes appels récursifs que dans la solution naïve (qui coûte $O(n)$ en mémoire), mais il faut en plus disposer de la mémoire `s` qui a les mêmes dimensions que la pyramide : d'où un coût spatial en $O(n^2)$.

La programmation dynamique est un *tradeoff* : on gagne (beaucoup) de temps en payant (un peu) de mémoire.

2.2.2 Pseudo-code générique

Définition 11 (Haut en bas).
En programmation dynamique, la méthode **de haut en bas** consiste à modifier une solution récursive pour qu'elle mémorise les valeurs déjà calculées afin de ne pas les recalculer.
Cette mémorisation est aussi appelée la **mémoïsation**.

Voici un pseudo-code générique d'une solution mémoïsée :

Fonction HautEnBas
Entrées : *mem* une mémoire, x une entrée
Sorties : y, la solution à l'entrée x

```
si mem[x] n'est pas déjà calculé alors
    si x est un cas de base alors
        mem[x] ←— solution y du cas de base
    sinon
        y ←— solution pour x calculée récursivement par HautEnBas(mem, ...)
        mem[x] ←— y
renvoyer mem[x]
```

Remarque.

- Il faut déjà disposer d'une solution récursive au problème pour pouvoir construire une fonction comme ci-dessus.

  En fait, on ne s'attaque pas à un problème en se disant « je vais faire de la programmation dynamique ! ». On s'attaque à un problème en se disant « Je cherche une formule de récurrence, et si (lorsque) j'en trouve une je me demande si elle est accélérable par programmation dynamique. »
- Il est *capital* que le calcul récursif de la solution y soit fait avec la fonction mémoïsé : si on utilise la solution naïve... bah elle est très lente, donc notre code aussi [19].
- En pratique, il faut commencer par initialiser la mémoire *mem* (souvent en marquant toutes les valeurs comme n'étant pas encore calculées). Ainsi, une solution de haut en bas est composée d'une fonction mémoïsée comme ci-dessus ; et d'une fonction principale qui initialise la mémoire puis appelle la précédente pour qu'elle calcule la solution.

Exemple. Voici une telle « fonction principale » pour `poids_memo` :

pyramide.ml

```
83 let poids pyramide =
84   let nb_lgn = Array.length pyramide in
85
86   (* La mémoire est un triangle, comme la pyramide  *)
87   let create_lgn lgn =
88     Array.make (lgn+1) None in
89   let s = Array.init nb_lgn create_lgn in
90
91   (* On peut maintenant utiliser [poids_memo s] *)
92   poids_memo s pyramide 0 0
```

19. J'aime quand une subtilité s'explique aussi trivialement.

2.3 Méthode de bas en haut

2.3.0 Pour les pyramides

Repartons de la figure 10.19 : on note que les instances qui correspondent à la ligne du bas n'ont aucun pré-requis : on peut les calculer directement. Une fois celles-ci calculées, on peut calculer celles de la ligne d'au-dessus. Et ainsi de suite.

On en déduit la solution non-récursive suivante. Cette fois-ci, je n'utilise pas des `Option`.[20] À la place, vous me voyez utiliser `0` pour remplir les cases non encore calculées. Notez également qu'une ligne numéro `lgn` a `lgn+1` colonnes, indexées de `0` à `lgn`.

pyramide.ml

```
97  let poids_bashaut pyramide =
98    let nb_lgn = Array.length pyramide in
99
100   (* On crée la mémoire *)
101   let create_lgn lgn =
102     Array.make (lgn+1) 0 in
103   let s = Array.init nb_lgn create_lgn in
104
105   (* On peut remplir la ligne du bas. *)
106   s.(nb_lgn-1) <- Array.copy pyramide.(nb_lgn-1);
107
108   (* Ensuite on remplir les lignes supérieurs
109      les unes après les autres *)
110   for lgn = nb_lgn -2 downto 0 do
111     for col = 0 to lgn do
112       let g = s.(lgn+1).(gauche col) in
113       let d = s.(lgn+1).(droite col) in
114       s.(lgn).(col) <- pyramide.(lgn).(col) + max g d
115     done
116   done;
117
118   (* Plus qu'à renvoyer le contenu de s.(0).(0) *)
119   s.(0).(0)
```

Remarque.

- On peut créer une ligne fictive en dessous de la ligne du bas, qui stocke les plus lourds chemins vides. Cela simplifie le code en évitant les lignes 107 à 111.
- Cet exemple des pyramides est tiré de Informatique MP2I/MPI de Balabonski, Conchon, Filiatre, Nguyen et Sarte. Dans le livre, vous trouverez des codes plus concis que les miens - mais les miens ont des noms plus explicites pour les variables et les quantités intermédiaires.[21]

2.3.1 Calcul de la complexité

Cette analyse de complexité-ci est bien plus standarde. Notons $n = nb\_lgn$ pour abréger.

Les lignes 101-103 effectuent n fois la création d'une ligne, donc $O(n^2)$ opérations. La double bloucle (110-116) a un corps en $O(1)$ donc la boucle interne est en $O(lgn)$ donc le tout en $O\left(\sum_{l=0}^{n-2} l\right) = O\left(n^2\right)$. Les autres lignes de code sont de complexité inférieure. D'où une complexité temporelle $T(n) = O\left(n^2\right)$, comme pour la version de haut en bas...

Sauf que dans ce $O\left(n^2\right)$, la composante linéaire liée au coût de la récursivité a disparu ! Or, comme la pile mémoire (où sont empilés les appels) est généralement plus limitée que le tas mémoire (où sont généralement allouées les mémoires de la programmation dynamique), ce n'est pas négligeable.

20. Je *devrais* - encore une fois, cela me protège efficacemment des erreurs de programmation - ; mais cela rend le code un peu plus intimidant à lire.

21. Disons que les leurs ont une longueur moins intimidante ; mais j'espère que les miens sont plus simples à comprendre.

2.3.2 Pseudo-code générique

Définition 12 (Bas en haut).
En programmation dynamique, la méthode **de bas en haut** consiste à calculer les instances les unes après les autres dans un ordre fixé qui soit compatible avec les dépendances de la formule récursive.
Ainsi, il n'y a pas besoin de récursivité puisque l'ordonnancement assure que les solutions des sous-instances dont on a besoin ont déjà été calculées.

Voici un pseudo-code récursif d'une solution de bas en haut. On se donne un ordre (bien fondé) $\preccurlyeq$ sur les instances qui soit compatible avec l'ordre de calcul :

Fonction BasEnHaut
Entrées : x une instance
Sorties : La solution y de cette instance
1 $mem \longleftarrow$mémoire qui stockera les résultats
2 **pour chaque** *instance* $x' \preccurlyeq x$, *par ordre croissant* **faire**
3 $\quad y' \longleftarrow$solution de x' (calculée à partir de $mem[x'']$ pour $x'' \preccurlyeq x'$)
4 $\quad mem[x'] \longleftarrow y'$
5 **renvoyer** $mem[x]$

Remarque.

- Pour cette façon de faire, il faut à la fois disposer d'une formule de récurrence (pour la ligne 7 du pseudo-code) ; et d'un ordre de calcul compatible avec cette formule.
- Pour certain·es auteur·ices, le terme « programmation dynamique » désigne spécifiquement la méthode de bas en haut, l'autre étant la « mémoïsation ». À titre personnel[22], je considère qu'il est plus pertinent d'appeler « programmation dynamique » toute solution basée sur une idée récursive et sur la mémorisation des résultats intermédiaires pour ne pas les recalculer. Mais sachez que certains sujets de concours pourront être en désaccord avec moi.

2.3.3 Optimisation de la mémoire

La méthode de bas en haut permet d'optimiser la mémoire : on peut « oublier » les solutions des instances qui ne serviront plus. On peut parfois réussir à gagner ainsi un ou plusieurs ordres de grandeur de complexité spatiale !

Exemple. Pour les pyramides, dans la version de bas en haut, pour calculer une ligne lgn, on utilise la ligne $lgn + 1$ du dessous. Mais une fois lgn calculée, la ligne $lgn + 1$ ne servira plus !

En d'autres termes, on peut modifier le calcul de bas en haut pour qu'il n'utilise que deux lignes comme mémoire : la ligne du dessous (calculée), et la ligne actuelle que l'on est en train de calculée.

FIGURE 10.20 – Les deux lignes de la mémoire dont on a besoin

22. Mais pas que, beaucoup de collègues de MP2I/MPI partagent mon avis.

On obtient le code ci-dessous :

pyramide.ml

```
let poids_2lgn pyramide =
  let nb_lgn = Array.length pyramide in

  let ligne_dessous = Array.copy (pyramide.(nb_lgn -1)) in
  let ligne = Array.make (nb_lgn+1) 0 in

  (* Invariant : ligne_dessous.(col) est le poids
     d'un plus lourd chemin descendant (lgn,col) *)
  for lgn = nb_lgn -2 downto 0 do
    for col = 0 to lgn do
      let g = ligne_dessous.(gauche col) in
      let d = ligne_dessous.(droite col) in
      ligne.(col) <- pyramide.(lgn).(col) + max g d
    done;

    (* lgn va augmenter : ligne devient la ligne du dessous *)
    for col = 0 to nb_lgn do
      ligne_dessous.(col) <- ligne.(col)
    done
  done;

  ligne.(0)
```

Mais on peut même faire encore mieux ! Remarquons que *ligne_dessous*[*col*] est utilisé pour *ligne*[*col*−1] et *ligne*[*col*]. Autrement dit, on peut fusionner les deux lignes en une seule ; dont le début vaut *ligne*[*col*] et la fin vaut *ligne_dessous*[*col*] :

pyramide.ml

```
let poids_opt pyramide =
  let nb_lgn = Array.length pyramide in
  let s = Array.copy pyramide.(nb_lgn-1) in
  for lgn = nb_lgn-2 downto 0 do
    for col = 0 to lgn do
      s.(col) <- pyramide.(lgn).(col) + max s.(gauche col) s.(droite col)
    done
  done;
  s.(0)
```

Exercice. Formaliser l'explication précédente en un (ou des) invariant(s) qui prouve la correction de `poids_opt` .

La fonction `poids_opt` (ainsi que `poids_2lgn`) a la même complexité temporelle que les programmations dynamiques précédentes... mais est en espace $O(nb\_lgn)$! On a gagné un ordre de grandeur ! Notez que c'est le même espace que la solution naïve : nous avons donc amélioré significativement le temps sans plus rien payer en espace !

2.4 Comparaison des deux méthodes

Résumons les avantages et inconvénients :

Haut en Bas (mémoïsation)	Bas en Haut
Simple à écrire à partir d'une version récursive naïve	Demande de déterminer un ordre de calcul (et de ne pas se tromper dans la manipulation des indices des boucles).
Simple à prouver par induction	Simple à prouver avec des invariants ; en tous cas jusqu'à un certain niveau d'optimisation de la complexité spatiale...
Ne calcule que les valeurs nécessaires	Selon la qualité de l'ordre de calcul, peut calculer des valeurs inutiles
	Peut optimiser son coût en espace

Remarque. Mentionnons tout de même que la version naïve récursive a l'avantage d'être la plus simple à coder et de pouvoir aider à déboguer une programmation dynamique.

2.5 Message d'utilité publique...

LA PROGRAMMATION DYNAMIQUE, CE N'EST PAS REMPLIR DES TABLEAUX. C'EST OPTIMISER UNE SOLUTION RÉCURSIVE.

Pourquoi est-ce que j'insiste ? Parce que :

- La mémoire utilisée n'est pas obligatoirement un tableau.
- Et surtout... pour concevoir une solution en programmation dynamique, il faut d'abord et avant tout concevoir une solution récursive. Le tableau, s'il apparait, vient seulement ensuite et uniquement comme réponse à « dans quoi vais-je stocker mes résultats intermédiaires ? ».

2.6 (Re)construction de la solution optimale

2.6.0 Fonctionnement générique

Jusqu'à présent, nous avons vu comment calculer la *valeur* d'une solution optimale. Mais quid de la solution optimale elle-même ?

Exemple. On sait calculer le *poids* d'un plus lourd chemin descendant dans une pyramide, mais quel est le plus lourd chemin associé ?

Définition 13 ((Re)construction de la solution optimale en programmation dynamique).
Il y a deux grandes façons de calculer une solution optimale par programmation dynamique :

- En plus de stocker les valeurs des solutions optimales des sous-problèmes, stocker une solution optimale associée. On peut par exemple stocker dans la mémoire des paires (*valeur*, *solution*) ; ou utiliser deux mémoires.
- À la fin du calcul de la valeur de la solution optimale, reconstruire la solution optimale : on sait quel sous-problème de l'entrée est optimal donc on connait le « premier choix » de la solution optimale, et ainsi de suite.

2.6.1 Pour les pyramides : mémorisation des solutions optimales

Le code que je présente ici repart de la solution de bas en haut (optimisée), mais on peut tout à fait le faire de haut en bas.

On va représenter un chemin comme la liste des indices (lgn, col). On va utiliser deux tableaux :

- `s` qui stocke $s_{i,j}$ comme précédemment
- `sol` qui stocke les solutions partielles associées aux valeurs de `s`.

pyramide.ml

```
let chemin_opt pyramide =
  let nb_lgn = Array.length pyramide in
  let s = Array.copy pyramide.(nb_lgn-1) in
  let sol = Array.init (nb_lgn+1) (fun col -> [nb_lgn-1, col]) in
  for lgn = nb_lgn-2 downto 0 do
    for col = 0 to lgn do
      let g, d = s.(gauche col), s.(droite col) in
      s.(col) <- pyramide.(lgn).(col) + max g d;
      let sol' = if g >= d then sol.(gauche col) else sol.(droite col) in
      sol.(col) <- (lgn,col) :: sol'
    done
  done;
  sol.(0)
```

La complexité temporelle de cette méthode est en $O(n^2)$ comme précédemment. La complexité spatiale, elle, change : en plus du coût spatial $O(n)$ que l'on avait précédemment, il y a désormais le coût des différentes listes. Dans le pire des cas, celle-ci sont un $O(n^2)$ donc le coût spatial redevient $O(n^2)$.

2.6.2 Pour les pyramides : reconstruction de la solution optimale

Le code que je présente ici prend en argument une mémoire `s` telle que `s.(i).(j)` $= s_{i,j}$ et renvoie un chemin optimal (représenté par une liste comme précédemment). Il peut donc être effectué à la fin de la méthode de haut en bas, ou de bas en haut non-optimisé. Il n'est par contre pas compatible avec les méthodes de bas en haut optimisée en espace (puisque l'optimisation consiste à « oublier » des solutions des sous-problèmes, dont on a besoin ici).

pyramide.ml

```
let rec reconstruit s lgn col =
  let nb_lgn = Array.length s in
  if lgn = nb_lgn -1 then
    [(lgn, col)]
  else
    if s.(lgn+1).(gauche col) >= s.(lgn+1).(droite col) then
      (lgn, col) :: reconstruit s (lgn+1) (gauche col)
    else
      (lgn, col) :: reconstruit s (lgn+1) (droite col)
```

La complexité temporelle et spatiale de cette fonction est $O(n)$, mais elle suit l'exécution d'une fonction qui remplit `s` en temps et espace $O(n^2)$.