Informatique *Cours*
Complexité Ipesup

Objectifs

— analyser la complexité temporelle et spatiale d'un algorithme ;
— distinguer et calculer complexité dans le pire cas, en moyenne et amortie ;
— résoudre des équations de complexité issues d'algorithmes récursifs ;
— comprendre les compromis espace/temps et leurs implications pratiques.

Différents algorithmes pouvaient avoir des performances très différentes, même si les deux algorithmes résolvent le même problème ! La *complexité* est l'étude des performances des algorithmes. On l'aborde selon deux critères principaux.

— La *complexité temporelle* : temps de calcul nécessaire à l'exécution de l'algorithme.
— La *complexité spatiale* : espace mémoire utilisé pour les données de travail de l'algorithme.

1 Complexité temporelle

En général, les performances d'un algorithme dépendent étroitement des entrées considérées.

Définition

La *complexité temporelle* d'une exécution sur une entrée donnée correspond au nombre d'opérations atomiques effectuées.

1.1 Complexité en fonction de la taille de l'entrée.

On exprime classiquement la complexité en fonction de la *taille* des entrées. Ainsi, pour un algorithme opérant sur un tableau de N entiers, le temps de calcul et l'espace mémoire sont exprimés en fonction de N.

Exemple

Considérons un algorithme déterminant si deux tableaux a_1 et a_2, de tailles respectives N_1 et N_2, possèdent un élément commun : chaque élément de a_1 est comparé successivement à tous les éléments de a_2 jusqu'à détection d'un élément commun ou épuisement des comparaisons possibles.

```
1 bool intersect(int a1[], int n1, int a2[], int n2) {
2     for (int i1 = 0; i1 < n1; i1++) {
3         for (int i2 = 0; i2 < n2; i2++) {
4             if (a1[i1] == a2[i2]) { return true; }
5         }
6     }
7     return false;
8 }
```

Les comparaisons sont effectuées uniquement dans le test `if (a1[i1] == a2[i2])`, placé dans deux boucles imbriquées. En l'absence d'interruption par un `return true`, la boucle interne réalise N_2 comparaisons à chaque itération de la boucle externe, exécutée N_1 fois, soit un total de $N_1 \times N_2$ comparaisons lorsque les tableaux n'ont aucun élément commun. À l'inverse, si un élément commun existe, l'exécution s'interrompt après un nombre de comparaisons dépendant de sa position, compris entre 1 et $N_1 \times N_2$.

1.2 Pire cas, meilleur cas, complexité moyenne

On obtient ainsi un encadrement du nombre C de comparaisons en fonction des tailles N_1 et N_2 des tableaux d'entrée :

$$1 \leq C \leq N_1 \times N_2.$$

Toutes les valeurs intermédiaires étant possibles, la complexité d'une exécution donnée ne dépend pas uniquement de la taille des entrées. On introduit donc trois notions servant de repères pour caractériser la complexité d'un algorithme.

Définition

Pour une taille d'entrée donnée, on distingue :
- **le pire cas**, correspondant à la complexité maximale possible ;
- **le meilleur cas**, correspondant à la complexité minimale possible ;
- **le cas moyen**, défini comme la moyenne des complexités sur l'ensemble des entrées possibles de cette taille.

Dans le cas de la version simple de `intersect`, le pire cas est $N_1 \times N_2$, lorsque les tableaux n'ont aucun élément commun ou lorsque l'unique élément commun apparaît en dernière position dans chacun d'eux, tandis que le meilleur cas est 1, lorsque les deux tableaux commencent par le même élément.

La complexité moyenne ne se déduit pas directement : elle dépend de la distribution des valeurs possibles dans les tableaux. Si le nombre de valeurs distinctes est très grand devant N_1 et N_2, la probabilité d'un élément commun est faible et la complexité moyenne est proche du pire cas $N_1 \times N_2$. À l'inverse, si les valeurs possibles sont peu nombreuses, la présence d'un élément commun devient probable, ce qui réduit la complexité moyenne.

1.3 Profils de complexité.

Complexité	Appellation	Cas typique
1	constante	opération de base
$\log(N)$	logarithmique	dichotomie
N	linéaire	boucle simple, recherche séquentielle
$N\log(N)$	linéarithmique	diviser pour régner, tri fusion
N^2	quadratique	deux boucles imbriquées, tri par insertion
N^3	cubique	trois boucles imbriquées, produit de matrices
2^N	exponentielle	recherche exhaustive

Les complexités N^2 et N^3 sont des cas particuliers de complexité *polynomiale*, c'est-à-dire de la forme N^k.

1.4 Ordres de grandeur.

Dans l'analyse de la complexité d'un algorithme, on s'intéresse principalement à la *complexité asymptotique*, qui décrit l'évolution de la complexité pour de grandes valeurs de N. On cherche généralement non pas une expression exacte, mais une indication de l'ordre de grandeur.

Les notations de Landau fournissent des outils pour formaliser ces ordres de grandeur.

Notations de Landau

Étant donnée une fonction $f : \mathbb{N} \to \mathbb{N}$, on a les notations suivantes.

- $O(f(n))$ désigne les fonctions majorées par f, à un facteur constant près. On note $g(n) = O(f(n))$ s'il existe un facteur $k \in \mathbb{R}^+$ et un rang $n_0 \in \mathbb{N}$ tels que pour tout $n \geq n_0$, on a $g(n) \leq kf(n)$.
- $\Omega(f(n))$ désigne les fonctions minorées par f, à un facteur constant près. On note $g(n) = \Omega(f(n))$ s'il existe un facteur $k \in \mathbb{R}^+$ et un rang $n_0 \in \mathbb{N}$ tels que pour tout $n \geq n_0$, on a $kf(n) \leq g(n)$.
- $\Theta(f(n))$ désigne les fonctions du même ordre de grandeur que f. On note $g(n) = \Theta(f(n))$ s'il existe deux facteurs $k_1, k_2 \in \mathbb{R}^+$ et un rang $n_0 \in \mathbb{N}$ tels que pour tout $n \geq n_0$, on a $k_1 f(n) \leq g(n) \leq k_2 f(n)$.
- $\sim f(n)$ désigne les fonctions équivalentes à f. On note $g(n) \sim f(n)$ si $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 1$.

Les définitions des notations de Landau permettent de déduire directement des règles de combinaison, notamment pour les ordres de grandeur Θ.

Règles de calcul sur les ordres de grandeur

L'ordre de grandeur d'un produit est le produit des ordres de grandeur, à une constante près.

- Si $C(n) = \Theta(f(n))$, alors $kC(n) = \Theta(f(n))$.
- Si $C(n) = \Theta(f(n))$ et $D(n) = \Theta(g(n))$, alors $C(n)D(n) = \Theta(f(n)g(n))$.

L'ordre de grandeur d'une somme est celui du terme dominant.

- Si $C(n) = O(D(n))$, alors $C(n) + D(n) = \Theta(D(n))$.

Que compter ? Le temps d'exécution d'un programme dépend des opérations réalisées, dont les coûts élémentaires (arithmétique, tests, accès mémoire, etc.) ne sont pas nécessairement comparables.
Un décompte exhaustif de toutes les opérations est à la fois difficile et peu pertinent, car il reviendrait à additionner des coûts hétérogènes. Le raisonnement en termes d'ordres de grandeur, via les notations O ou Θ, est volontairement imprécis : il décrit des vitesses de croissance sans fournir de valeur exacte pour une taille donnée, mais constitue souvent l'analyse la plus pertinente en l'absence d'hypothèses sur la machine d'exécution.
Pour des estimations plus fines, on peut utiliser la notation d'équivalence $\sim$, qui impose de préciser les constantes multiplicatives. Dans ce cadre, on ne comptabilise pas toutes les opérations, mais seulement celles jugées représentatives du temps d'exécution réel.
Pour les programmes manipulant des tableaux, le nombre d'accès mémoire (lecture ou écriture) fournit souvent une approximation réaliste.
Deux modèles d'analyse sont utilisés selon les situations.

- Le plus souvent, on se limite à un ordre de grandeur global, exprimé par un O ou

un Θ, en considérant chaque groupe d'opérations élémentaires comme une unité de complexité.
— Lorsque cela est pertinent, on effectue un décompte précis portant sur une opération représentative, comme les accès mémoire, les multiplications ou les comparaisons d'éléments.

2 Complexité des boucles

Une boucle répète une suite d'instructions ; pour en analyser la complexité, on détermine le nombre de tours exécutés par son corps.

2.1 Boucles simples.

Pour une boucle `for` simple, l'entête permet de connaître directement le nombre d'itérations, en particulier lorsque l'indice est incrémenté ou décrémenté d'une unité à chaque tour. Ce cas fréquent correspond notamment au parcours complet d'un tableau. Les deux boucles suivantes effectuent chacune n tours, l'indice prenant successivement les valeurs de 0 à $n-1$, dans l'ordre croissant ou décroissant.

```
for (int i = 0; i < n; i++) { ... }
for (int j = n - 1; j >= 0; j--) { ... }
```

Exemple

Un appel `power_n(a,n)` à la fonction d'exponentiation naïve effectue systématiquement n tours de boucle, et donc n multiplications.

Lorsque l'indice de boucle est incrémenté d'une valeur constante $k > 1$, le nombre de tours est égal à la longueur de l'intervalle divisée par k. Par exemple, la boucle suivante effectue $\lfloor n/2 \rfloor$ tours.

```
for (int i = 0; i < n; i += 2) { ... }
```

Si l'indice de boucle est multiplié ou divisé à chaque itération, le nombre de tours devient logarithmique. Les boucles suivantes effectuent $\lfloor \log(n) \rfloor$ tours.

```
for (int i = 0; i < n; i *= 2) { ... }
for (int i = n; i > 0; i /= 2) { ... }
```

2.2 Boucles conditionnelles

Le nombre d'itérations d'une boucle `while` se détermine en analysant la condition d'arrêt et l'évolution des variables impliquées.

```
i = n;
while (i > 0) {
    ...
    i -= 1;
```

```
}
```

On retrouve des schémas analogues à ceux des boucles `for`, mais aussi des évolutions moins directement prévisibles.

Exemple

```
int power_b(int a, int n) {
    int r = 1;
    while (n > 0) {
        if (n % 2 == 1) r *= a;
        a = a * a;
        n = n / 2;
    }
    return r;
}
```

Dans la fonction `power_b(a,n)` precedente, la variable n est divisée par deux (avec arrondi inférieur) à chaque itération, ce qui entraîne un nombre de tours logarithmique en n. Plus précisément, si $2^k \leq n < 2^{k+1}$, la boucle réalise $k+1$ itérations.
Chaque itération effectue une ou deux multiplications ; l'appel `power_b(a,n)` réalise donc entre $\lfloor \log(n) \rfloor + 1$ et $2\lfloor \log(n) \rfloor + 2$ multiplications, soit une complexité $\Theta(\log(n))$.

Exemple

La boucle suivante calcule les nombres de Fibonacci jusqu'à atteindre ou dépasser une valeur n.

```
int a = 0, b = 1;
while (b < n) {
    b = a + b;
    a = b - a;
}
```

Le nombre d'itérations est égal au nombre de termes de la suite de Fibonacci appartenant à l'intervalle $[0, n[$, quantité calculable mais nécessitant une analyse mathématique non triviale.

2.3 Boucles emboîtées.

Dans des boucles imbriquées, la boucle interne est exécutée intégralement à chaque itération de la boucle externe. Lorsque le nombre d'itérations internes est constant, le nombre total d'exécutions est donné par un produit ; par exemple, le corps suivant est exécuté n^2 fois.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        ...
    }
}
```

Si le nombre d'itérations de la boucle interne varie, le nombre total d'exécutions s'obtient en additionnant, pour chaque itération externe, le nombre correspondant d'itérations internes.

Exemple

Le tri par sélection consiste à choisir successivement le plus petit élément restant du tableau.

```
void selection_sort(int a[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int j_min = i;
        for (int j = i + 1; j < n; j++) {
            if (a[j] < a[j_min]) { j_min = j; }
        }
        swap(a, i, j_min);
    }
}
```

La boucle externe effectue $n-2$ itérations et, pour chaque i, la boucle interne en réalise $n-i-1$. Le nombre total d'exécutions du test `if (a[j] < a[j_min])` est donc

$$\sum_{i=0}^{n-2}(n-i-1) = \sum_{i'=1}^{n-1} i' = \frac{n(n-1)}{2}.$$

3 Complexité en moyenne

La complexité moyenne d'un algorithme prend en compte l'ensemble des entrées possibles.

Complexité moyenne

Pour une taille d'entrée N, la complexité moyenne est la moyenne des complexités associées à toutes les entrées possibles de cette taille.

Cette notion est directement exploitable lorsque l'ensemble des entrées de taille N est fini ; dans le cas contraire, elle nécessite une modélisation finie.

3.1 Complexité moyenne par dénombrement.

Lorsque le domaine des entrées est fini, la complexité moyenne se calcule en sommant les complexités de toutes les entrées possibles, puis en divisant par leur nombre. Les tableaux booléens constituent un exemple typique : pour une taille N, il existe 2^N tableaux distincts.

Exemple

Dans le programme `first_one`, qui recherche la première occurrence de 1 dans un tableau de 0 et de 1, le nombre de cases consultées est égal à $k+1$ pour un tableau contenant k zéros initiaux suivis d'un 1, varie de 1 dans le meilleur cas à n dans le pire cas, et peut être analysé en dénombrant les configurations correspondantes parmi les 2^N tableaux possibles.
On calcule la complexité moyenne de `first_one` en dénombrant les tableaux booléens de taille N selon le nombre k de cases consultées.

Pour $k = N$, seuls un tableau est possibles (un unique 1 en dernière position). Pour $1 \leq k < N$, les tableaux commencent par $k-1$ zéros, suivis d'un 1, les $N-k$ cases restantes étant arbitraires, soit 2^{N-k} tableaux.
La complexité totale sur l'ensemble des 2^N entrées est alors

$$C_T(N) = N + \sum_{k=1}^{N} k 2^{N-k},$$

d'où la complexité moyenne

$$C_m(N) = \frac{N}{2^N} + \sum_{k=1}^{N} \frac{k}{2^k}.$$

On obtient en particulier

$$C_m(N+1) - C_m(N) = \frac{1}{2^N},$$

ce qui montre que $C_m(N)$ est une somme d'inverses de puissances de 2.
Une approche alternative consiste à raisonner par case : la case d'indice k est consultée pour 2^{N-k} tableaux. En sommant sur toutes les cases, on obtient

$$C_T(N) = \sum_{k=0}^{N-1} 2^{N-k}, \qquad C_m(N) = \sum_{k=0}^{N-1} \frac{1}{2^k} = 2 - \frac{1}{2^{N-1}}.$$

La formule obtenue est cohérente : on a $C_m(1) = 1$ et, pour tout N, la complexité moyenne est comprise entre le meilleur cas 1 et le pire cas N.

3.2 Modèle des tableaux aléatoires.

Lorsque l'ensemble des entrées possibles est infini, la complexité moyenne dépend du choix d'un *modèle* probabiliste, défini par des classes d'entrées pondérées.

Pour les tableaux d'entiers non bornés, même à taille N fixée, le nombre d'entrées est infini. On adopte alors un modèle abstrait qui ignore les valeurs exactes et ne considère que les relations d'ordre entre les éléments. Pour deux cases i et j, les cas $a[i] < a[j]$ et $a[j] > a[i]$ ont chacun une probabilité $\frac{1}{2}$, tandis que l'égalité a une probabilité nulle.

Ainsi, seules les configurations où tous les éléments sont distincts ont un poids non nul. Les classes d'entrées de taille N correspondent alors aux $N!$ ordres relatifs possibles, tous équiprobables, chacun de poids $\frac{1}{N!}$. Ce modèle est celui des *tableaux aléatoirement ordonnés sans doublons*.

Il constitue un cadre naturel pour l'étude de la complexité moyenne des algorithmes de tri.

4 Complexité des fonctions récursives

Dans le cas d'algorithmes récursifs, la complexité peut elle-même être caractérisée par des équations récursives.

Exemple

On veut calculer $n! = 1 \times 2 \times 3 \times \cdots \times n$. Voici une définition immédiate en OCaml.

```
let rec fact n =
  if n < 2 then 1 else n * fact (n-1)
```

Le nombre $C(n)$ de multiplications réalisées pour calculer `fact(n)` suit l'une des deux formules suivantes.

— pour $n < 2$, zéro,

— pour $n \geq 2$, une multiplication en plus du coût du calcul de `fact(n-1)`.

Autrement dit :

$$\begin{cases} C(0) = 0 \\ C(1) = 0 \\ C(n+1) = 1 + C(n) & \text{si } n \geq 1 \end{cases}$$

4.1 Résolution des suites récursives simples

Les équations récursives de complexité définissent des fonctions $C : \mathbb{N} \to \mathbb{R}$, que l'on peut interpréter comme des suites numériques $(C(n))_{n\in\mathbb{N}}$.

Suite numérique

Une suite numérique est une fonction $\mathbb{N} \to \mathbb{R}$, notée $(u_n)_{n\in\mathbb{N}}$.

Certaines formes classiques de suites récursives admettent des expressions fermées connues.

— Une suite définie par $u_{n+1} = a + u_n$ est arithmétique et vérifie $u_n = an + b$.

— Une suite définie par $u_{n+1} = a\,u_n$ est géométrique et vérifie $u_n = b\,a^n$.

Ces résultats permettent de résoudre directement certaines équations de complexité. Ainsi, la complexité de `fact` définit une suite arithmétique et vérifie $C(n) = n - 1$ pour $n \geq 1$. Pour l'exponentiation rapide, la relation devient arithmétique en se restreignant aux puissances de 2, ce qui donne $C(2^k) = k + 2$.

Dans des cas plus généraux, on peut utiliser un raisonnement par *télescopage* : si $u_{n+1} = u_n + f(n)$, alors

$$u_n = u_{n_0} + \sum_{n_0 \leq k < n} f(k),$$

ce qui ramène le calcul de la suite à l'étude d'une somme.

Enfin, lorsqu'aucune expression explicite simple n'est accessible, on peut encadrer les valeurs de la suite par récurrence.

Exemple

Pour l'exponentiation rapide,

```
let rec power a n =
  if n = 0 then 1
  else
    let b = power a (n / 2) in
    if n mod 2 = 0 then b * b
    else a * b * b
```

si $2^k \leq n < 2^{k+1}$ alors

$$k+1 \leq C(n) \leq 2(k+1),$$

d'où, pour tout n, un encadrement logarithmique

$$\log(n) \leq C(n) \leq 2\log(n).$$

5 Complexité amortie

La *complexité amortie* d'un algorithme correspond à une analyse lissée de sa complexité sur une séquence d'invocations successives. Elle vise à garantir un équilibre global entre des appels peu coûteux et des appels ponctuellement très coûteux, en s'intéressant au coût total d'une séquence plutôt qu'au coût d'une exécution isolée.

5.1 Réalisation d'un compteur binaire.

Pour énumérer tous les tableaux de n booléens, on peut les interpréter comme les écritures binaires des entiers de 0 à $2^n - 1$. L'énumération consiste alors à passer d'un tableau au suivant en effectuant un incrément binaire, c'est-à-dire en reproduisant les opérations sur les bits correspondant à l'ajout de 1.

La fonction C suivante réalise un tel incrément, en supposant que le bit de poids faible est à l'indice 0.

```
1 void incr(bool c[], int n) {
2   int i = 0;
3   while (i < n && c[i]) {
4     c[i] = 0;
5     i++;
6   }
7   if (i < n) c[i] = 1;
8 }
```

Le coût d'un appel à `incr` dépend du nombre de bits consécutifs égaux à 1 à partir de l'indice 0, c'est-à-dire du nombre de retenues à propager. Dans le meilleur cas, lorsque $c[0] = 0$, la boucle `while` n'est pas exécutée et l'on effectue seulement deux accès mémoire. Dans le pire cas, lorsque tous les bits valent 1, la boucle parcourt l'ensemble du tableau et lit puis modifie chacune des n cases, soit $2n$ accès mémoire. Le pire cas d'un incrément est donc linéaire en n, ce qui donne une borne grossière en $O(n2^n)$ pour l'énumération complète.

Équilibre à long terme. Cependant, les 2^n appels successifs à `incr` nécessaires à l'énumération complète ne portent pas sur des entrées arbitraires. En partant du tableau représentant 0, on connaît exactement la suite des tableaux manipulés. L'observation des premiers appels met en évidence une structure régulière, illustrée par le tableau suivant.

Numéro	Paramètre	Nombre de tours
0	00000...0	0
1	10000...0	1
2	01000...0	0
3	11000...0	2
4	00100...0	0
5	10100...0	1
6	01100...0	0
7	11100...0	3
8	00010...0	0
9	10010...0	1
10	01010...0	0
11	11010...0	2
12	00110...0	0
13	10110...0	1
14	01110...0	0
15	11110...0	4

On observe qu'un appel sur deux n'effectue aucun tour de boucle, qu'un appel sur quatre en effectue un, un appel sur huit en effectue deux, etc. Les appels coûteux sont donc de plus en plus rares. On peut montrer que le nombre total de tours de boucle sur l'ensemble des 2^n appels est proportionnel au nombre d'appels, et non à la taille n des tableaux.

La complexité amortie s'applique précisément à ce type de situation : des algorithmes généralement peu coûteux, mais pouvant être très chers sur certaines entrées, à condition que ces entrées défavorables apparaissent suffisamment rarement pour garantir une borne globale sur une séquence d'exécutions.

L'analyse nécessite d'identifier un état évolutif (ici le tableau de booléens) et de comprendre comment son évolution conditionne l'apparition des opérations coûteuses.

5.2 Méthode du potentiel.

La méthode du potentiel, aussi appelée méthode du physicien, formalise cette intuition en associant à chaque état possible x de l'algorithme une valeur non négative $\Phi(x)$, appelée *potentiel*, représentant un coût latent.

Potentiel

Une fonction de potentiel associe à chaque entrée possible x d'un algorithme une valeur $\Phi(x) \geq 0$.

Les algorithmes à bonne complexité amortie alternent typiquement entre des opérations peu coûteuses qui font croître le potentiel, et des opérations coûteuses associées à une forte diminution du potentiel. L'analyse amortie tient compte à la fois du coût réel d'une opération et de la variation de potentiel qu'elle induit.

Coût amorti

Pour une opération transformant une entrée x_e en une sortie x_s, le coût amorti est

défini par

$$A = C + \Phi(x_s) - \Phi(x_e),$$

où C désigne le coût réel de l'opération.

Les notions d'entrée et de sortie recouvrent ici l'ensemble de l'état manipulé par l'algorithme : paramètres, structures de données et état mémoire avant et après l'appel.

Theoreme de amortissement

Considérons une suite de n opérations consécutives

$$x_0 \xrightarrow{op_1} x_1 \xrightarrow{op_2} \dots \xrightarrow{op_n} x_n$$

telle que $\Phi(x_0) = 0$. En notant C_i le coût réel et A_i le coût amorti de op_i, on a

$$\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} A_i.$$

La démonstration repose sur une somme télescopique faisant apparaître la variation globale de potentiel. En particulier, si le coût amorti est borné par une constante k, alors toute séquence de n opérations a un coût réel total inférieur ou égal à kn, ce qui garantit un coût moyen constant.

5.3 Analyse amortie du compteur binaire.

Le coût réel d'un incrément du compteur binaire dépend du nombre k d'occurrences consécutives de 1 à partir de l'indice 0. Pour un tableau `c`, un appel à `incr(c,n)` réalise :

— $2(k+1)$ accès mémoire si $k < n$,
— $2k$ accès mémoire si $k = n$.

Pour l'analyse amortie, on introduit une fonction de potentiel mesurant la présence de 1 dans le tableau, définie par

$$\Phi(c) = 2 \times (\text{nombre de 1 dans } c).$$

Lors d'un appel à `incr`, les k premiers 1 sont remplacés par des 0, puis, si $k < n$, le 0 suivant devient 1. La variation de potentiel entre le tableau initial `c` et le tableau obtenu `c'` est alors

$$\Phi(c') - \Phi(c) = \begin{cases} -2k+2 & \text{si } k < n, \\ -2k & \text{si } k = n. \end{cases}$$

Le coût amorti A_c d'un appel à `incr` sur un tableau ayant k occurrences consécutives de 1 à partir de l'indice 0 vaut donc

$$A_c = \begin{cases} 2(k+1) - 2k + 2 = 4 & \text{si } k < n, \\ 2k - 2k = 0 & \text{si } k = n. \end{cases}$$

En conclusion, la fonction d'incrément du compteur binaire possède une complexité amortie bornée par 4. Toute séquence d'incréments partant du tableau nul effectue ainsi en moyenne moins de 4 accès mémoire par incrément, malgré l'existence d'appels de coût linéaire en n.

6 Différence entre complexité moyenne et complexité amortie

On peut trouver des similitudes entre les notions de complexité en moyenne et de complexité amortie. Ces deux concepts sont cependant bien différents.

— Par définition, la complexité en moyenne est une moyenne : elle nous donne une complexité supposée représentative du plus grand nombre d'entrées, mais sans apporter aucune garantie sur la complexité d'une opération particulière, ni même d'une séquence d'opérations particulières. Rien n'empêche qu'une séquence mal choisie d'invocations d'un algorithme donné enchaîne les pires cas, au mépris de la valeur moyenne.
— La complexité amortie donne une borne garantie pour une séquence d'opérations : elle ne dit rien de la complexité pour une entrée particulière, mais assure un équilibre à toute séquence d'opérations, même la moins favorable. Cette notion présuppose que les invocations successives de l'algorithme ne sont pas indépendantes les unes des autres, et qu'un état ou une structure de données évolue avec chaque nouvelle invocation.

7 Complexité spatiale

La *complexité spatiale* d'un algorithme mesure la quantité de mémoire qu'il utilise. Comme pour la complexité temporelle, on l'exprime généralement de façon asymptotique en fonction de la taille de l'entrée, et l'on peut distinguer meilleurs cas, pires cas et complexité moyenne.

L'évaluation de la complexité spatiale consiste à identifier les données allouées en mémoire, en se référant aux modèles d'exécution des langages OCaml et C.

On retient notamment les règles suivantes :

— chaque variable déclarée compte pour une unité ;
— chaque structure en C compte pour une constante, correspondant à la taille de sa représentation mémoire, que l'on peut ramener à 1 lorsqu'on raisonne en ordre de grandeur ;
— un tableau de taille n compte pour n fois la taille de ses éléments, ou simplement pour n si ceux-ci ont une taille constante.

La somme des tailles des données allouées fournit une borne sur la complexité spatiale. Toutefois, toutes les données ne sont pas nécessairement présentes simultanément en mémoire : la complexité spatiale correspond donc uniquement à la quantité maximale de mémoire utilisée à un instant donné.

Complexité spatiale

La *complexité spatiale* d'une exécution est la quantité maximale de mémoire occupée au cours de cette exécution.

Dans de nombreux cas, cette quantité est facile à estimer. Par exemple, le tri par insertion n'utilise que des variables locales et a donc une complexité spatiale constante, tandis que le tri par fusion requiert un tableau auxiliaire de taille n et possède une complexité spatiale linéaire.

7.1 Complexité spatiale cachée : la pile d'appels

Un algorithme peut présenter une complexité spatiale non négligeable sans allocation explicite de mémoire, du fait de la pile d'appels récursifs.

Exemple

Dans le tri rapide, aucune structure supplémentaire n'est créée et chaque appel à `quickrec` n'utilise que quelques variables locales. Cependant, lors des appels récursifs emboîtés, les variables locales de tous les appels présents dans la pile coexistent en mémoire et leurs coûts s'additionnent.
La complexité spatiale du tri rapide est donc proportionnelle à la profondeur maximale de récursion : logarithmique en moyenne en la taille du tableau, mais linéaire dans le pire cas. Avec une variante exploitant l'optimisation des appels terminaux, elle devient logarithmique même dans le pire cas.

7.2 Compromis spatio-temporel

La complexité spatiale et la complexité temporelle étant distinctes, un algorithme peut privilégier l'une au détriment de l'autre.

Exemple

Considérons le calcul du coefficient binomial $\binom{n}{k}$ à partir de la relation

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}.$$

Cette formule conduit directement à l'implémentation récursive suivante.

```
1 int binom_rec(int k, int n) {
2     assert(0 <= k && k <= n);
3     if (k == 0 || k == n)
4         return 1;
5     else
6         return binom_rec(k-1, n-1) + binom_rec(k, n-1);
7 }
```

Cette fonction n'utilise aucune structure de données autre que la pile d'appels, mais possède une complexité temporelle exponentielle en n, due aux recalculs répétés des mêmes coefficients.

Mémorisation : dépenser de l'espace pour gagner du temps. La complexité temporelle du calcul des coefficients binomiaux peut être fortement réduite en mémorisant les valeurs déjà calculées. La fonction `binom_rec` utilise un tableau bidimensionnel pour stocker les coefficients connus, et ne calcule que ceux qui ne sont pas encore renseignés.
Le calcul de $\binom{7}{3}$ illustre cette approche : seuls les coefficients nécessaires du triangle de Pascal sont effectivement évalués.

$k=$	0	1	2	3	4	5	6	7
$n=0$	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	–			
5	–	5	10	10	–	–		
6	–	–	15	20	–	–	–	
7	–	–	–	35	–	–	–	–

La fonction `binom_memo` initialise le tableau avant d'appeler `binom_rec`. Les coefficients égaux à 1 ne sont pas stockés, car ils ne nécessitent aucun calcul. Cette mémorisation évite tout recalcul : le nombre de coefficients

```c
//Coefficients du binome dynamiques
int binom(int k, int n) {
    int *row = calloc(n + 1, sizeof(int));
    for (int i = 0; i <= n; i++) {
        row[i] = 1;
        for (int j = i - 1; j > 0; j--)
            row[j] += row[j-1];
    }
    return row[k];
}
```

Savoir oublier. On peut encore ajuster le compromis espace–temps avec une troisième version, qui calcule légèrement plus de coefficients que la précédente (tout en restant sous une borne quadratique), mais se contente d'une complexité spatiale linéaire. L'idée consiste à calculer successivement les lignes du triangle de Pascal jusqu'à la $(n+1)$-ème, en ne conservant en mémoire que la dernière ligne calculée (programme 6.12).
Pour éviter d'écraser des valeurs encore nécessaires, la mise à jour se fait en partant de la fin de la ligne. En reprenant $n=7$, la ligne obtenue à la fin de l'étape $i=5$ est la suivante, les deux dernières cases étant déjà réservées mais encore inutilisées.

$$i=5 \qquad \boxed{1}\boxed{5}\boxed{10}\boxed{10}\boxed{5}\boxed{1}\boxed{-}\boxed{-}$$

À l'étape suivante, pour $i=6$, après deux itérations de la boucle interne, les cases d'indices 4 à 6 ont été mises à jour. La prochaine opération consiste à mettre à jour `row[3]` avec la somme `row[2] + row[3]`.

$$\underset{\downarrow}{j=3} \qquad i=6 \qquad \boxed{1}\boxed{5}\boxed{10}\boxed{10}\boxed{15}\boxed{6}\boxed{1}\boxed{-}$$

à traiter *déjà à jour*

Cette méthode calcule davantage de coefficients que `binom_memo`, puisqu'elle évalue entièrement chaque ligne du triangle. La complexité temporelle reste néanmoins quadratique en n, tandis que la complexité spatiale est réduite à un seul tableau de taille proportionnelle à n.

8 Liens entre complexité spatiale et complexité temporelle

Chaque accès à un mot mémoire étant une opération, la complexité temporelle est toujours supérieure ou égale à la quantité de mémoire à laquelle un programme accède. Ceci permet d'établir un lien entre complexité spatiale et complexité temporelle, dont le détail est cependant légèrement différent d'un langage à l'autre.

— En OCaml, la complexité spatiale est toujours bornée par la complexité temporelle. En effet toute zone de mémoire utilisée en OCaml est initialisée, et le coût temporel de la création d'une structure ne peut donc pas être inférieur à la taille de cette structure.
— En C, l'allocation de mémoire avec `malloc` n'initialise pas la zone de mémoire utilisée. On peut donc avoir une complexité spatiale dépassant la complexité temporelle dans le cas où l'on réserve avec `malloc` plus de mémoire que ce à quoi on accédera effectivement.

9 Tri rapide

Les boucles et la récursion *ne sont pas deux techniques antagonistes*. Il est tout à fait possible d'utiliser les deux à l'intérieur d'une même fonction. On peut observer ceci par exemple avec l'algorithme de *tri rapide*, pour l'analyse duquel nous allons devoir combiner les techniques des sections précédentes.

Présentation de l'algorithme. Le cœur de l'algorithme de *tri rapide* est le suivant : après avoir choisi un élément « pivot » on trie séparément les éléments inférieurs au pivot et les éléments supérieurs au pivot. Dans le code C du programme 6.7 la fonction principale `quickrec` trie le segment $[l, r[$ du tableau `a`, c'est-à-dire entre les indices l (inclus) et r (exclu).

Le pivot est `a[l]`, le premier élément du segment à trier. La première étape consiste à réarranger les éléments en trois groupes : à gauche les éléments plus petits que le pivot, à droite les éléments plus grands, et au milieu le pivot et les éventuels autres éléments qui lui seraient égaux.

L'algorithme conclut alors en triant récursivement, et surtout séparément, les groupes gauche et droit.

Le réarrangement en trois groupes est opéré par la boucle `for`. Durant l'exécution de cette boucle les trois groupes se forment progressivement avec les éléments d'un quatrième groupe : celui des éléments non encore répartis. Ce quatrième groupe est situé dans le tableau entre le groupe des éléments égaux au pivot et celui des éléments plus grands que le pivot.

La boucle progresse en consultant le premier élément de la zone à répartir, et en l'intervertissant au besoin avec un autre élément pour le placer dans le bon groupe. Le segment $[i, hi[$ des éléments à répartir diminue à chaque tour, soit par incrément de i soit par décrément de hi.

Listing 1 – Programme 6.7 – tri rapide d'un tableau

```
1  void swap(int a[], int i, int j) {
2      int tmp = a[i];
3      a[i] = a[j];
4      a[j] = tmp;
5  }
6
7  // trie uniquement a[l..r[
8  void quickrec(int a[], int l, int r) {
9      if (r - l <= 1) return;
10     int p = a[l], lo = l, hi = r;
11     for (int i = l+1; i < hi; ) {
12         if (a[i] == p) { i++; }
13         else if (a[i] < p) { swap(a, i++, lo++); }
14         else /* a[i] > p */ { swap(a, i, --hi); }
15     }
16     quickrec(a, l, lo);
17     quickrec(a, hi, r);
18 }
19
20 void quicksort(int a[], int n) {
```

```
21      knuth_shuffle(a, n);
22      quickrec(a, 0, n);
23  }
```