Chapitre 3

Complexité

Notions	Commentaires
Analyse de la complexité d'un algorithme. Complexité dans le pire cas, dans le cas moyen. Notion de coût amorti.	On limite l'étude de la complexité dans le cas moyen et du coût amorti à quelques exemples simples.

Extrait de la section 1.1 du programme officiel de MP2I : « Algorithmes et programmes ».

Notions	Commentaires
[...] Organisation des activations sous forme d'arbre en cas d'appels multiples.	[...] Les récurrences usuelles : $T(n) = T(n-1) + an$, $T(n) = aT(n/2)+b$, ou $T(n) = 2T(n/2)+f(n)$ sont introduites au fur et à mesure de l'étude de la complexité des différents algorithmes rencontrés. On utilise des encadrements élémentaires *ad hoc* afin de les justifier ; on évite d'appliquer un théorème-maître général.

Extrait de la section 2 du programme officiel de MP2I : « Récursivité et induction ».

Sommaire

0 Complexité temporelle

0.0 Notion de complexité temporelle

Définition 1 (Complexité temporelle).
La complexité temporelle d'un algorithme est une estimation du temps qu'il prend à s'effectuer. On peut la mesurer de deux grandes façons :

- En comptant le nombre de certaines opérations choisies. Par exemple, on peut exprimer la complexité en nombre d'additions effectuées. Si la ou les opérations ne sont pas choisies au hasard
- En comptant le nombre d'opérations élémentaires, c'est à dire du nombre d'étapes que prendra le calcul sur un processeur. Cette méthode impose souvent de faire des approximations pour simplifier.

Elle dépend souvent de l'entrée, on l'exprime souvent comme une fonction de l'entrée ou de la taille de l'entrée. Cette fonction est souvent nommée $C()$ ou $T()$.

Remarque.

- Généralement, ce qui nous intéresse est la complexité sur les grandes entrées. L'objectif est de prévoir (à peu près) combien de temps l'exécution d'un programme prendra, pour savori si'l est raisonnable de le lancer ou s'il faut trouver une autre solution.
- Sauf mention contraire, une complexité est demandée en opérations élémentaires.

Exemple. Considérons la fonction ci-dessous, et calculons sa complexité en nombre d'additions $A()$:

```c
/** Renvoie la somme des entiers de 0 à n */
unsigned somme_entiers(unsigned n) {
  unsigned somme = 0;
  unsigned i = 1;
  while (i <= n) {
    somme = somme + i;
  }
  return somme;
}
```

La fonction n'effectue aucune addition en dehors de sa boucle, ni aucun appel de fonctions. À chaque itération de sa boucle, elle effectue deux additions. La boucle itère n fois (car `i` parcourt l'ensemble $[\![1; n]\!]$). D'où $A(n) = 2n$.

Exemple. Considérons les deux fonctions ci-dessous. On note A_{ep} la complexité en nombre d'additions de `est_premier` , et A_{spi} celle de `somme_premiers_inf` .

```c
/** Teste si n est premier */
bool est_premier(int k) {
  if (k <= 1) { return false; }

  int d = 2;
  while (d*d <= k) {
    if (k % d == 0)  { return false; }
    d = d +1;
  }
  return true;
}
```

```c
/** Renvoie la somme des nombres premiers inférieurs à x */
int somme_premiers_inf(int x) {
  int somme = 0;
  int n = 0;
  while (n < x) {
    if (est_premier(x)) {
      somme = somme + x;
    }
    n = n +1;
  }
  return somme;
}
```

Étudions tout d'abord `est_premier` . Elle ne fait aucune addition en dehors de sa boucle (ni aucun appel de fonction). Elle fait une addition par double. La boucle itère au plus $\lfloor\sqrt{\mathsf{k}}\rfloor$ fois (d varie au plus de $[\![2$ jusqu'à $\lfloor\sqrt{\mathsf{k}}\rfloor + 1$ inclus). Donc $A_{ep}(\mathsf{k}) \leq \lfloor\sqrt{\mathsf{k}}\rfloor$.

Passons maintenant à `somme_premiers_inf` . Elle ne fait aucun appel de fonction ni addition en dehors de sa boucle. Sa boucle itère pour n allant de 0 à $\mathsf{x} - 1$ inclus. À chaque itération, elle effectue au moins 1 addition et au plus $1 + A_{ep}(\mathsf{n})$. Donc au total, on peut encadrer A_{spi} ainsi :

$$\begin{aligned}
\sum_{\mathsf{n}=0}^{\mathsf{x}-1} 1 \leq A_{spi}(\mathsf{x}) &\leq \sum_{\mathsf{n}=0}^{\mathsf{x}-1} \left(1 + A_{ep}(\mathsf{n})\right) \\
\mathsf{x} \leq A_{spi}(\mathsf{x}) &\leq \mathsf{x} + \sum_{\mathsf{n}=0}^{\mathsf{x}-1} \left(\lfloor\sqrt{\mathsf{n}}\rfloor\right) \\
\mathsf{x} \leq A_{spi}(\mathsf{x}) &\leq \mathsf{x}\left(1 + \lfloor\sqrt{\mathsf{x}} - 1\rfloor\right)
\end{aligned}$$

Remarque.

- Ici, on a encadré et non calculé la valeur précise de la complexité. La complexité précise de `est_premier` était en effet difficile à exprimer. C'est un cas très courant ! On va en fait essayer de *borner* les complexités, avec une borne supérieure et une borne inférieure.
- Vous trouvez peut-être que mon passage de l'avant dernière à la dernière ligne majore très grossièrement. Cela aurait pu être la cas [1], mais en l'occurence pas tant que ça. Pour vous en convaincre, utilisez le fait que [2] :

$$\int_{d-1}^{d} \sqrt{x}\,dx \leq \sqrt{d} \leq \int_{d}^{d+1} \sqrt{x}\,dx$$

- L'expression de la complexité dépend de ce en fonction de quoi on l'exprime. Il y a deux grandes options :
  - Exprimer la complexité en fonction de la *valeur* des arguments (ce que l'on a fait ci-dessous).
  - Exprimer la complexité en fonction de la *taille* des arguments, c'est à dire en fonction du nombre de bits nécessaires pour représenter les arguments. Cette deuxième façon de faire prend tout son sens lorsque l'on essaye de comparer la difficulté de différents problèmes (cf MPI).

    Or, la taille des arguments est ne correspond pas à la valeur ! Penser à la taille de l'encodage binaire d'un entier, par exemple.

  À retenir : il faut toujours indiquer en "fonction de quoi" on exprime la complexité.

0.0.0 Opérations élémentaires

Définition 2 (Opération élémentaire).
ne **opération élémentaire** est une opération « de base », au sens où elle n'est pas combinaison d'autres. Cela inclut par exemple :

- La lecture ou l'écriture d'une case mémoire.
- Une opération logique (NON, ET, OU, etc) ou arithmétique (+, -, %, etc) ou une comparaison du contenu de deux variables.
- Démarrer un appel de fonction ou renvoyer une valeur.
- Afficher 1 caractère.
- Et d'autres.

1. Dans beaucoup de calculs de complexité, il est facile d'obtenir une borne supérieure très large mais difficile de la rendre plus précise. Le monde est mal fait.
2. Si c'est un exo de maths trop dur pour l'instant, revenez-y au second semestre.

Remarque. Voici des exemples d'opérations qui *ne* sont *pas* élémentaires :

- Tester l'égalité de deux tableaux (cela demande de tester l'égalité de chacune des cases, c'est à dire de faire plusieurs tests d'égalité de variables).
- Afficher un texte (cela demande plusieurs affichages d'1 caractère).

Proposition 3.
En réalité, toutes les opérataions élémentaires ne prennent pas le même temps, et le temps d'une même opération peut différer selon la machine qui l'exécute. Il est donc inutile d'essayer de compter le nombre exact d'opérations élémentaires. À la place, on chercher à compter **l'ordre de grandeur** du nombre d'opérations élémentaires.
Ainsi, au lieu de dire « on fait 47 opérations élémentaires », on dira « on effectue un nombre constant d'opérations élémentaires ».

Remarque. L'un des buts de la théorie de la complexité est d'analyse un algorithme indépendamment de la machine ou du langage qui l'exécute, afin de permettre des comparaisons des algorithmes et non des implémentations. Cela ne signifie pas que les comparaisons des langages / machines n'est pas pertinente ; ces dernières ce font juste différemment.

Exemple. Reprenons `somme_entiers` . En dehors de sa boucle, elle effectue un nombre constant d'opérations élémentaires. De même, chaque itération effectue un nombre constant d'opérations élémentaires. Or, la boucle itère n fois, donc la complexité $C(n)$ est linéaire en n.

0.1 Notation de Landau

Vous verrez ces notions plus proprement et plus en profondeur en mathématiques. Ici je m'efforce de faire une approche avant tout intuitive, quitte à piétiner certains points de rigueur.[3].

Définition 4 (Notations de Landau).
Soient (u_n) et (v_n) deux suites de réels. On dit que :

- (u_n) est dominé par (v_n) si quand n est grand, on a $|u_n| \leq K.|v_n|$ (avec $K > 0$ une constante). Cela revient à dire que $\frac{|u_n|}{|v_n|}$ est majoré.
  On note alors $u_n = O(v_n)$.
- (v_n) domine (u_n) si quand n est grand, on a $K.|u_n| \leq |v_n|$ (avec $K > 0$ une constante). Cela revient à dire que $\frac{|v_n|}{|u_n|}$ est minoré (par une constante strictement positive).
  On note alors $v_n = \Omega(u_n)$.
  C'est en réalité équivalent au fait que u_n est dominé par v_n.
- (u_n) est de l'ordre de (v_n) si $u_n = O(v_n)$ et $v_n = O(u_n)$. Cela revient à dire que $\frac{|v_n|}{|u_n|}$ tend vers une constante strictement positive.
  On note alors $u_n = \Theta(v_n)$.

En résumé : $O()$ indique l'ordre de grandeur de majoration, $\Omega()$ celui d'une minoration, et Θ les deux à la fois.

Exemple.

- $n - 50 = O(n)$; mais aussi $n + 42 = O(n)$ ou encore $3n + 12 = O(n)$
- $n + 3\sqrt{n} = O(n)$ (prendre $K = 4$ comme constante).
- $n + 3\sqrt{n} = \Omega(n)$; et donc $n + 3\sqrt{n} = \Theta(n)$.
- Pour la fonction `somme_premiers_inf` , on a prouvé que $A_{spi}(x) = O(x\sqrt{x})$ et que $A_{spi}(x) = \Omega(x)$.

Remarque.

3. J'ai essayé l'approche rigoureuse les années précédentes. Cela n'a pas marché.

- **Le but de ces notations est de ne garder que l'ordre de grandeur du terme dominant dans une majoration / minoration.** En effet, comme on l'a vu avec les opérations élémentaires, les calculs de complexité négligent souvent des « détails » de la réalité. Plutôt que de calculer une complexité fausse, on préfère se contenter de calculer l'ordre de grandeur de la vraie complexité (ou d'une borne de celle-ci).
- Il ne faut pas penser que $O(...)$ est une valeur précise. Deux suites différentes peuvent être dominée à l'aide d'un même $O(...)$. Par exemple, $(n+1)_{n\in\mathbb{N}}$ et $\left(\frac{n}{3}+100\right)_{n\in\mathbb{N}}$ sont toutes les deux dominées par $(n)_{n\in\mathbb{N}}$, donc toutes les deux $= O(n)$.
- En maths, vous définirez l'équivalence $\sim$. Ce n'est pas la même chose que le $\Theta()$, ne confondez pas.
- On essaye de mettre une borne la plus « simple » posible dans le $O()$ (ou Ω ou Θ). Par exemple, on préfère $O(n^2)$ plutôt que $O(\pi n^2 - \sqrt{2.71}\log_5 n)$. Au risque de me répéter, le but est de simplifier en ne mémorisant que l'ordre de grandeur du terme dominant.

Proposition 5 (Relation usuelles de domination).
Je note ici $x << y$ le fait que $x = O(y)$. Soient $a, b \in \mathbb{R}_+^*$ et $c \in]1;+\infty[$. On a :

$$\log(n)^a << n^b << c^n << n!$$

Démonstration. Il s'agit des croissances comparées (une version plus faible des puissances comparées, pour être précis). Vous les prouverez en maths. □

Proposition 6 (Opérations sur les dominations).
J'indique ici quelques relations utiles. Vous en verrez plus en maths.

- La domination est transitive, c'est à dire que si $u_n = O(v_n)$ et $v_n = O(z_n)$ alors $u_n = O(z_n)$.
- On peut sommer une quantité finie de dominations, c'est à dire que :
  - Si $u_n = O(y_n)$ et $v_n = O(z_n)$, alors $u_n + v_n = O(|y_n| + |z_n|)$.
  - Idem avec $\Omega()$.
  - Idem avec Θ.

  En particulier, $u_n = O(z_n)$ et $v_n = O(z_n)$ entraine $u_n + v_n = O(z_n)$.
- On peut sommer une quantité variable de dominations uniquement si elles cachent une même constante K. En conséquence, on peut par exemple sommer les domination des itérations successives d'une boucle !
- On peut multiplier une quantité finie de dominations, c'est à dire que :
  - Si $u_n = O(y_n)$ et $v_n = O(z_n)$, alors $u_n.v_n = O(|y_n|.|z_n|)$.
  - Idem avec $\Omega()$.
  - Idem avec Θ.
- On peut multiplier une quantité variable de dominations uniquement si elles cachent la même constante K.

Remarque. Si cette histoire de « quantité variable de domination » vous perturbe (ou que vous ne comprenez pas pourquoi on ne peut pas forcément sommer dans ce cas), attendez le cours de maths sur le sujet.

Exemple. On a en fait déjà utilisé ces propriétés sont le dire dans le calcul de la complexité en opérations élementaires

0.2 Exemples plus avancés

0.2.0 Méthodologie

La méthode utilisée pour ces exemples est toujours la même :

- Calculer d'abord la complexité des appels de fonction.
- Calculer la complexité en dehors des boucles.
- Pour chaque boucle, calculer la compelxité d'une itération (condition + corps), puis le nombre d'itérations [4], et en déduire la complexité de la boucle entière.
  En cas de boucle imbriquées, on commence par les boucles intérieures.

0.2.1 Un exemple avec des boucles imbriquées

On considère la fonction ci-dessous :

```c
/** Échange *a et *b */
void swap(int* a, int* b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
  return;
}


/** Trie tab par ordre croissant en temps quadratique */
void tri_bulle(int tab[], int len) {
  int deja_trie = 0;
  while (deja_trie < len) {

    int indice = 0;
    // Cette boucle fait remonter le plus grand élément
    // non-encore trié et le place à sa place finale
    while (indice +1 < len - deja_trie) {
      if ( tab[indice] > tab[indice+1] ) {
        swap( &tab[indice], &tab[indice+1] );
      }
      indice = indice +1;
    }

    deja_trie = deja_trie +1;
  }
  return;
}
```

On veut calculer sa complexité T en nombre de comparaisons en fonction de la longueur du tableau [5].

Commençons par remarquer que la fonction `swap` n'effectue aucune comparaison. On ne prendra donc pas en compte ses appels dans les calculs qui suivent.

Étudions maintenant La fonction `tri_bulle` :

- Elle n'effectue aucune comparaison en dehors de ses boucles.
- À chaque itération de la boucle intérieure (lignes 60-65), elle effectue deux comparaisons (une dans la condition, une dans le `if` du corps). Elle effectue une dernière comparaison lorsqu'elle quitte la boucle. Comme cette boucle itère `len`−`deja_trie` fois, elle effectue au total $2\,(\texttt{len} - \texttt{deja\_trie})+1$ comparaisons.

4. S'il n'y a pas de difficulté particulière, on peut se contenter d'affirmer le nombre d'itérations sans le justifier.
5. C'est un critère assez courant pour évaluer un tri.

- La boucle principale (ligne 56-69) effectue une comparaison dans sa condition, et contient la boucle précédente. La variable `deja_trie` parcourt $[\![0; \texttt{len}-1]\!]$. On effectuera une dernière comparaison quand on quittera la boucle. Donc au total, cette boucle fait :

$$\begin{aligned}
1 + \sum_{\texttt{deja_trie}=0}^{\texttt{len}-1} (2(\texttt{len} - \texttt{deja_trie}) + 1) &= 1 + \texttt{len} - 2 \sum_{\texttt{deja_trie}=0}^{\texttt{len}-1} (\texttt{len} - \texttt{deja_trie}) \\
&= 1 + \texttt{len} + 2\texttt{len}^2 - 2 \sum_{\texttt{deja_trie}=0}^{\texttt{len}-1} \texttt{deja_trie} \\
&= 1 + \texttt{len} + 2\texttt{len}^2 - (\texttt{len}-1)(\texttt{len}) \\
&= 1 + \texttt{len}^2
\end{aligned}$$

- On peut en déduire que :

$$\boxed{T(\texttt{len}) = \texttt{len}^2 + 1 = \Theta(\texttt{len}^2)}$$

Remarque.

- On aurait pu prouver $T(\texttt{len}) = O(\texttt{len}^2)$ plus facilement : au lieu de donner la valeur précise du nombre de comapraisons de la boucle intérieure effectue, on dit qu'elle effectue $O(\texttt{len})$. Chaque itération de la boucle principale effectue donc $O(\texttt{len})$ comparaisons, or cette boucle itère `len` fois, donc $T(\texttt{len}) = O(\texttt{len}^2)$.
- « Il y a deux boucles imbriquées donc la complexité est quadratique » n'est pas une preuve, et n'est pas toujours vrai.

0.2.2 Un exemple où il faut être très précis sur les itérations

On suppose que l'on dipose d'une fonction `affiche_entiers_l_bits` qui, en temps $\Theta(2^l)$ affiche tous les entiers pouvant s'écrire sur `l` bits. On considère la boucle ci-dessous :

```c
int l = 0;
while (l <= n) {
  printf("Voici les entiers binaires sur %d bits : \n\t", l);
  affiche_entiers_l_bits(buffer, l);
  l = l +1;
}
```

(Vous pouvez ignorer le premier argument de `affiche_entiers_l_bits`, c'est un détail d'implémentation.)

Calculons la complexité de cette boucle. Notons tout d'abord qu'il y a un nombre constant d'opérations élémentaires en dehors de la boucle.*(Pour la suite, je propose deux raisonnements.)*

Version 1 : La boucle itère `l` de 0 à `n` inclus. Chaque itération effectue des opérations en temps constant et un appel en $\Theta(2^{\texttt{l}})$, et est donc en $O(2^{\texttt{n}})$. En sommant sur les itérations, on obtient une complexité T pour la boucle qui vérifie $T(\texttt{n}) = O(\texttt{n}2^{\texttt{n}})$.

Dans cette version, on a donné une borne grossière à chaque itération, afin que la somme des itérations soit simple à calculer.

Version 2 : La boucle itère `l` de 0 à `n` inclus. Chaque itération effectue des opérations en temps constant et un appel en $\Theta(2^{\texttt{l}})$, et l'itération a donc une complexité de l'ordre de $2^{\texttt{l}}$. En sommant sur les itérations, on obtient que le tout est de l'ordre de $\sum_{\texttt{l}=0}^{\texttt{n}} 2^{\texttt{l}} = 2^{\texttt{n}+1} - 1$. D'où $T(\texttt{n}) = \Theta(2^{\texttt{n}})$

Ici, on a donné la valeur exacte de l'ordre de grandeur de chaque itération. Sommer sur les itérations a en conséquence été un peu plus compliqué, mais on a obtenu une valeur plus précise à la fin.

Remarque. L'intérêt des majorations grossières est de simplifier les calculs. On peut ainsi obtenir très rapidement des majorations qui prouvent qu'un code s'exécutera en temps raisonnable [6].

Exercice. Reprendre la preuve de la complexité en nombre de compairaisons du tri bulle. Refaire avec une majoration grossière. Commenter.

0.2.3 Écart incompressible entre les bornes

Revenons à `est_premier` :

```c
/** Teste si n est premier */
bool est_premier(int k) {
  if (k <= 1) { return false; }

  int d = 2;
  while (d*d <= k) {
    if (k % d == 0)  { return false; }
    d = d +1;
  }
  return true;
}
```

Avec les calculs déjà effectués, on conclut que la complexité en additions A_{ep} vérifie $A_{ep} = \Omega(1)$ (car le boucle itère au moins une fois) et $A_{ep} = O(\mathtt{k})$.

On peut prouver qu'il existe des k aussi grand que l'on veut pour lesquels la boucle itère une seule fois (les entiers pairs). Similairement, il existe des k aussi grands que l'on veut pour lesquels la boucle itère $\sqrt{\mathtt{k}}$ fois (les entiers premiers). Il est donc impossible d'obtenir un Θ.

Remarque. Cet exemple peut-être vu comme un cas particulier de la différence entre « meilleur cas » et « pire cas ».

0.3 Pire cas, cas moyen, meilleur cas

Parfois, pour une même taille d'entrée, la complexité peut varier. `est_premier` en était un exemple, mais en voici un encore plus clair (et plus fort) :

```c
/** Renvoie le premeir indice où trouver x dans tab,
 *  et -1 s'il n'en existe pas.
 */
int mem(int x, int const tab[], int len) {
  int indice = 0;
  while (indice < len) {
    if (tab[indice] == x) { return true; }
    indice = indice +1;
  }
  return -1;
}
```

Pour `len` aussi grand que l'on veut, il existe des tableaux de longueur `len` sur lesquels cette fonction s'exécute en temps $\Theta(1)$ et d'autres sur lesquels elle s'exécute en temps $\Theta(\mathtt{len})$.

Autrement dit, on ne peut même pas définir la complexité comme une fonction dépendant uniquement de `len` ! C'est pourtant ce que l'on voudrait : on voudrait pouvoir prévoir le temps d'exécution d'une fonction à partir de critères simples, comme la longueur.

6. Vous ferez une utilisation similaire de majorations grossières en mathématiques en deuxième année, pour prouver qu'une série (ou une série de fonction) est « raisonnable ».

Définition 7 (Pire cas, meilleur cas).

- La **complexité dans le pire des cas** est le nombre *maximal* d'opérations que la fonction exécute sur une entrée d'une taille donnée.
- La **complexité dans le meilleur des cas** est le nombre *minimal* d'opérations que la fonction exécute sur une entrée d'une taille donnée.
- La **complexité dans le cas moyen** est le nombre moyen d'opérations que la fonction exécute sur une entrée d'une taille donnée. La définition de « nombre moyen » demande donc de sommer (et moyenner) sur toutes les entrées possibles de cette taille. On pondère parfois ces entrées par une probabilité afin de mieux coller à une situation pratique.

Exemple. La complexité (pire des cas) de `mem` est $\Theta(\texttt{len})$. Sa complexité meilleur des cas est $\Theta(1)$. Sa complexité en cas moyen... est difficile à définir, car il faut déjà définir ce que signifie moyenner sur tous les tableaux de longueur `len` lorsqu'il y en a une infinité.[7]

Convention 8.
Sauf mention contraire, toute complexité demandée est un pire des cas.

Remarque.

- **Ne confondez pas** « pire cas » avec « majoration de la complexité » et « meilleur cas » avec minoration. Ce qui est vrai, c'est que le meilleur des cas est une minoration du pire des cas. Mais il est souvent possible d'obtenir de meilleurs minorations.
- **Ne confondez pas** « meilleur cas » avec « entrée de petite taille ». On s'intéresse à la complexité sur des *grandes* entrées ![8]

0.4 Ordres de grandeur

On considère une fonction dont la complexité est $C(n)$. On dit que la fonction est :

- de complexité constante si $C(n) = O(1)$.
- polylogarithmique en n si $C(n) = O(poly(\log(n)))$ avec $poly$ une fonction polynomiale.
- linéaire en n si $C(n) = O(n)$. Similaire pour quadratique en n, cubique en n, polynomiale en n.
- exponentielle en n si $C(n) = O\left(e^{poly(n)}\right)$ avec $poly$ une fonction polynomiale.

Notez que je précise à chaque fois « en n », pour bien rappeler en fonction de quoi la complexité est exprimée.

Il est impossible de traduire une complexité en une durée exacte à la nano seconde près : trop de facteurs liés au langage et à la machine entrent en compte. On peut cependant appliquer la méthode suivante pour avoir une approximation très grossière :

1) Évaluer la valeur de la complexité sur l'entrée voulue
2) Diviser par 1Ghz la fréquence du processeur (c'est un arrondi du nombre d'opérations par s de votre processeur).
3) Multiplier le tout par quelque chose entre 2 et 500 (pour prendre en compte la constante cachée du $O()$).

Si le résultat est raisonnable, votre code terminera très vite ; sinon croisez fort les doigts.

7. Les « sommes infinies » posent des difficultés mathématiques, comme vous le verrez en maths.
8. Les petites entrées terminent rapidement en pratique.

Voici un tableau des arrondis des valeurs obtenues par les étapes 1) et 2) de la méthode précédente :

$T(x)$ \ x	10	100	1000	10000	10^6	10^9
$\Theta(\log x)$	1ns	10ns	10ns	10ns	10ns	100ns
$\Theta(\sqrt{x})$	1ns	10ns	100ns	100ns	1µs	1ms
$\Theta(x)$	10ns	100ns	1µs	10µs	1ms	1s
$\Theta(x \log x)$	10ns	100ns	10µs	100µs	10ms	10s
$\Theta(x^2)$	100ns	10µs	1ms	100ms	10min	10 ans
$\Theta(x^3)$	1µs	1ms	1s	10 min	10 ans	∞
$\Theta(2^x)$	1µs	∞	∞	∞	∞	∞

FIGURE 3.1 – Ordre de grandeurs de temps de calculs pour quelques complexités et tailles d'entrées. Non-contractuel.

1 Complexité spatiale

Dans l'idée, la complexité spatiale est l'espace utilisé par une fonction. Cependant, pour des raisons théoriques que vous verrez peut-être plus tard, on utilise un modèle extrêment simplifié. Il a le mérite de permettre des comparaisons entre des algorithmes de manière indépendante du langage et de la machine.

Définition 9 (Complexité spatiale).
La complexité spatiale d'une fonction est le nombre de cases mémoires dont elle a besoin pour s'exécuter. On ne compte pas dedans :

- Les entrées si elles sont en lecture seule.
- La sortie si elle est en lecture seule.

Autrement dit, on compte l'espace *supplémentaire* dont la fonction a besoin.

On la note souvent E ou S. Les notions de pire, moyen et meilleur cas s'appliquent ici aussi.

Remarque.

- Comme pour la complexité temporelle, seul *l'ordre de grandeur* a véritablement un sens en pratique.
- Cette définition a des limites quand on l'applique à un langage pratique. Par exemple, si un programme C alloue n cases mémoires avec `malloc` mais ne les utilise pas, cet espace doit-il compter ? Dans de tels cas, il ne faut pas hésiter à préciser : on peut par exemple distinguer l'espace *utilisé* de l'espace *demandé*.
- Si une même zone mémoire est utilisé par plusieurs appels de fonctions (cela correspond aux arguments passés par pointeur/référence), il ne faut pas la compter plusieurs fois puisque c'est le même espace qui est utilisé !

Exemple. Toutes les fonctions vues jusqu'à présent sont en espace constant car elles créent un nombre fixe de variables, dont la taille est fixe.

En fait, pour avoir une complexité spatiale variable, il faut :

- soit créer/modifier un objet dont la taille est une variable (par exemple un tableau à n cases).
- soit faire de la récurrence.

Théorème 10 (Bornes temps-espace (version simplifiée)).
On considère une fonction qui termine. On note sa complexité temporelle $T(n)$ (en opérations élémentaires) et spatiale $E(n)$ (on compte uniquement l'espace utilisé). On suppose en outre que l'ordre de $E(n)$ n'est pas constant. On a :

$$E(n) \leq T(n) \leq 2^{E(n)}$$

Démonstration. Je vais prouver l'inégalité de gauche, et donner l'intuition de la preuve de celle de droite :

- Utiliser 1 case de la mémoire demande d'y faire une lecture/écriture, et donc au moins 1 opération élémentaire. D'où $E(n) \leq T(n)$.
- L'exécution d'une fonction est déterministe. En partant d'une ligne précise, et de valeurs précises des variables, il y a une seule exécution possible. On nomme (grosso modo) « état mémoire » d'une fonction la donnée de la ligne du code où on en est ainsi que des valeurs des variables. Toutes ces informations sont stockées dans la mémoire. Par déterminisme de l'exécution, si un programme passe deux fois par exactement le même état mémoire, elle est en train de faire une boucle infinie. Or, si on utilise $E(n)$ cases, il n'y a que $2^{E(n)}$ états mémoire possibles car l'unité élementaire est le bit. D'où $T(n) \leq 2^{E(n)}$.

□

Remarque.

- La borne supérieure est hors-programme, mais l'idée de la preuve est très intelligente. On note généralement cette borne supérieure $T(n) \leq 2^{O(E(n))}$, ce qui est plus rigoureux. J'ai ici essayé de simplifier, quitte à tordre un peu la vérité.
- Notez que la preuve de la mjoration n'utilise pas l'hypothèse $E(n)$ non-constant. En fait, l'hypothèse est uniquement là car ce que l'on appelle « espace constant » d'un point de vue pratique n'est pas un espace constant d'un point de vue théorique (notamment à cause des considérations de la taille des entiers [9]). Plus d'infos à ce sujet en MPI.
- Ce théorème n'est en fait généralement même pas écrit à l'aide de E et T, mais à partir des classes de complexité en temps $DTIME$ (plus à ce sujet en MPI) et $DSPACE$ (plus à ce sujet en école).

2 Cas particulier des fonctions récursives

On voudrait analyser la complexité d'une fonction récursive. Par exemple :

Fonction Hanoï

Entrées : `i` : le pic de départ ; `j` : le pic d'arrivée ; `n` : le nombre de disques à déplacer

```
1 si n > 0 alors
2     k ←pic autre que i ou j
3     Hanoï(i, k, n-1)
4     Déplacer 1 disque de i à j
5     Hanoï(k, j, n-1)
```

Cette fonction effectue 1 déplacement par appel, mais il faut prendre en compte tous les appels... On va distinguer deux choses :

9. D'un point de vue théorique, les entiers ne peuvent pas déborder et ont une taille variable. Subséquement, dans la théorie l'espace constant n'existe (quasiment) pas. Ce n'est pas le point de vue que nous utilisons ici.

Définition 11 (Coût local).

- Le **complexité locale à l'appel** : ce sont les ressources utilisées par l'appel en cours.
- Le **complexité des appels récursifs** : ce sont les ressources utiliées par les appels récursifs.

Pour calculer la complexité total, il faudra prendre en compte ces deux éléments.

Remarque. Le terme de « coût » est parfois utilisé comme synonyme de « complexité ». On parle donc aussi de « coût local » et de « coût récursif ».

2.0 Complexité temporelle des fonctions récursives

Proposition 12 (Complexité temporelle récursive).
La complexité temporelle d'une fonction est la somme des complexités temporelles locales de chacun des appels récursifs.

Il « suffit » donc pour la calculer d'être capable de :

- Calculer la complexité locale d'un appel.
- Sommer sur les appels.

C'est cette deuxième étape qui est généralement la plus difficile.

2.0.0 Cas général : formule de récurrence

La complexité temporelle s'exprime naturellement comme une fonction récursive. Si cette suite est une suite facile que l'on sait résoudre grâce au cours de maths, c'est gagné.

Exemple. La complexité en déplacements $D(n)$ de Hanoï vérifie $D(n) = 2D(n-1) + 1$ et $D(0) = 0$. C'est une suite arithmético-géométrique que l'on sait résoudre, et dont la solution est $D(n) = 2^n - 1$.

Remarque.

- Lorsque l'on raisonne en opérations élementaires, on n'a jamais une véritable égalité. Par exemple, pour Hanoï, on obtient $T(n) = 2T(n-1) + O(1)$, c'est à dire qu'il existe $K > 0$ tel que $T(n) \leq 2T(n-1) + K$. Ici, on a une *majoration* par un terme arithémtico-géométrique. On en déduit que $T(n)$ est inférieur à la suite arithmético-géométrique en question, et on obtient $T(n) = O(2^n)$.
  On peut faire de même pour les minorations.
- Lorsque l'on raisonne en opérations élémentaires, on fait souvent ce que j'ai fait dans le point précédent : on donne uniquement la formule de récurrence ($T(n) = 2T(n-1) + O(1)$), mais pas le cas de base. Cela signifie implicitement que le cas de base est est de complexité constante, et qu'il est atteint lorsque n passe en-dessous d'une valeur fixe. On peut prouver que le choix de cette valeur fixe ne change pas l'ordre de grandeur de la complexité.
  Ainsi, dans l'exemple précédent, le cas de base est atteint en $n < 1$ et a un coût $O(1)$. Changer ce seuil à $n < 4$ donnerait $T(n) = O\left((2^{n-2}\right) = O(2^n)$.

Si la complexité n'est pas une suite « gentille » que l'on sait résoudre grâce au cours de maths, on peut tout de même essayer d'observer les premiers termes de la suite et de trouver une logique et de peut-être repérer une complexité qui ressemble à un classique que l'on sait traiter. De manière générale, *essayer de se ramener à ce que l'on maitrise* est un excellent raisonnement en sciences !

2.0.1 Cas particuliers : arbre d'appels

Rappel du cours sur la récursivité : on peut représenter la suite des appels récursifs sous la forme d'un arbre.

On peut parfois utiliser cette représenter pour facilement sommer les complexités locales des appels. Cela revient en fait à réécrire la somme des coûts des appels en faisant des « paquets » d'appels qui ont les mêmes paramètres de complexité (ces paquets sont les lignes de l'arbre si on a bien représenté l'arbre).

Exemple. Voici l'arbre des coûts des appels de Hanoï :

Figure 3.2 – Arbre des coûts en déplacements des appels de Hanoï

Pour calculer à l'aide d'un tel arbre, on procède ainsi :

1) On calcule le coût d'un noeud de l'arbre (= on calcule la complexité locale d'un appel).
2) On compte le nombre de noeuds par ligne (= le nombre d'appels qui ont les mêmes paramètres pour la complexité), et on en déduit le coût d'une ligne de l'arbre.
3) On somme sur les lignes pour conclure.

Les étapes 2) et 3) sont les plus difficiles, car il est aisé d'y affirmer une formule fausse. Il est donc très important d'inclure une ligne au milieu de l'arbre qui montre la forme générale d'une ligne « quelconque », car cette ligne résume le raisonnement.

Exemple. Dans l'arbre précédent, sans compter la ligne du bas (qui effectue 0 déplacements) il y a n lignes. Il y a 2^i appels sur la ligne des appels qui ont $n-i$ en entrée. Ainsi, une ligne associée à i a un coût de 2^i. Les lignes vont de $i = 0$ à $i = n-1$. En sommant sur ces lignes, on obtient $D(n) = \sum_{i=0}^{n-1} 2^i = 2^n - 1$.

Remarque. Parfois, l'arbre obtenu n'a pas une forme simple à décrire. On peut alors essayer de majorer par un arbre « imple à calculer » en « rajoutant des appels là où il en manque » :

Figure 3.3 – Arbre d'appels et majoration pour $T(n) = T(n-1) + T(n-2)$ et cas de base constant.

Diviser pour régner : les arbres d'appels de fonctions « Diviser pour Régner » sont particuliers, mais se résolvent de manière similaire. On se place dans le cas suivant : un problème de taille n est découpé en r sous-problèmes de taille n/c. Le coût local (sépartion/fusion ou cas de base) est donné par la fonction f.

On admet que l'**on peut ignorer les éventuelles parties entières dans l'équation de récurrence sans que cela ne change l'ordre du grandeur du résultat**[10]. L'équation de complexité est donc :

$$T(n) = r.T(n/c) + f(n)$$

Voici un schéma de l'arbre d'appel[11], où L est la hauteur de l'arbre d'appels :

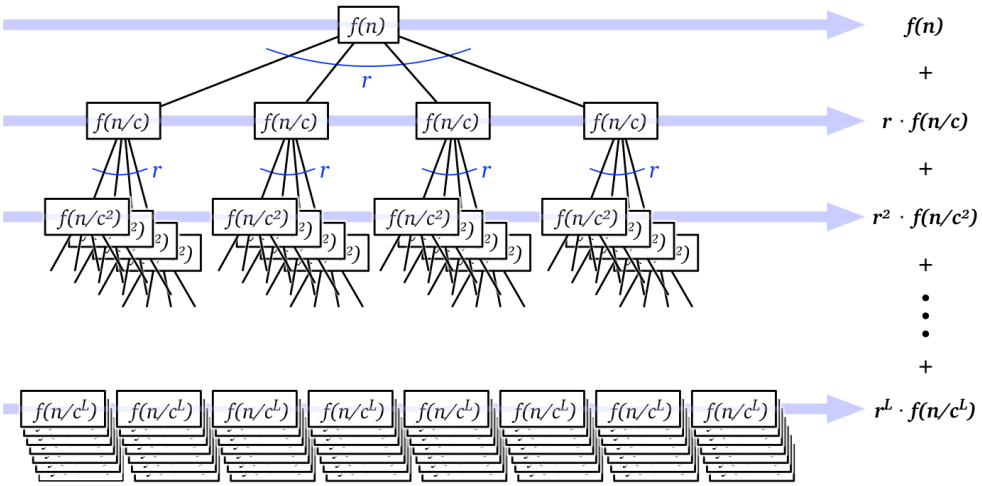

FIGURE 3.4 – Arbre de complexité diviser pour régner

Les coûts T_i de chacune des lignes sont indiqués le long de l'arbre (on a $T_i = r^i f(n/c^i)$). La complexité totale est la somme des coûts de ces lignes. Il existe 3 cas courants dans lesquels une majoration précise de cette somme est simple à calculer :

- Si les T_i décroissent (au moins) géométriquement, c'est à dire s'il existe $K > 1$ tel que $T_{i+1} \leq \frac{T_i}{K}$. On a alors pour tout i, $T_i \leq \frac{T_0}{K^i}$ et donc :

$$\begin{aligned} T(n) &= \sum_{i=0}^{L} T_i \\ &\leq \sum_{i=0}^{L} \frac{T_0}{K^i} \\ &\leq f(n)\left(\sum_{i=0}^{L} \frac{1}{K^i}\right) && \text{en factorisant par } T_0 = f(n) \\ &= O(f(n)) && \text{car le terme entre parenthèses tend vers une constante positive} \end{aligned}$$

  Autrement dit, si les coûts des lignes décroissent (au moins) géométriquement, le coût de la première ligne domine.

- Si les T_i croissent (au moins) géométriquement, c'est à dire s'il existe $K > 1$ tel que $T_{i+1} \geq K.T_i$. On a alors pour tout i, $T_i \leq \frac{T_L}{K^{L-i}}$ et donc :

10. Si vous êtes accroché·es en maths, vous pouvez aller voir le lien suivant pour une preuve de cela : `https://courses.csail.mit.edu/6.046/spring04/handouts/akrabazzi.pdf`
11. Issu de l'excellent *Algorithms* de J. Erikson.

$$\begin{aligned}
T(n) &= \sum_{i=0}^{L} T_i \\
&\leq \sum_{i=0}^{L} \frac{T_L}{K^{L-i}} \\
&\leq r^L f(n/c^L)\left(\sum_{i=0}^{L} \frac{1}{K^{L-i}}\right) && \text{en factorisant par } T_L = r^L f(n/c^L) \\
&= O(r^L f(n/c^L)) && \text{car le terme entre parenthèses tend vers une constante positive}
\end{aligned}$$

Autrement dit, si les coûts des lignes croissent (au moins) géométriquement, le coût de la dernière ligne domine. Notez que dans ce cas, il faut calculer L. C'est généralement un $O(\log_c(n))$, car c'est généralement le premier i tel que n/c^i passe en dessous du seuil du cas de base.

- Si les T_i sont constants, on a :

$$\begin{aligned}
T(n) &= \sum_{i=0}^{L} T_i \\
&= \sum_{i=0}^{L} f(n) && \text{car } \forall i, T_i = T_0 = f(n) \\
&= O(Lf(n))
\end{aligned}$$

Notez qu'ici aussi il faut calculer L.

Résumons :

Proposition 13 (Méthode de calcul Diviser pour Régner).
Pour calculer une majoration du coût temoprel d'un algorithme Diviser pour Régner dont l'équation de récurrence est $T(n) = r.T(n/c) + f(n)$, on utilise la méthode suivante :

- Si les coûts des lignes de l'arbre décroissent (au moins) géométriquement de haut en bas, alors on prouve que le coût de la racine est dominant.
- Si les coûts des lignes croissent (au moins) géométriquement de haut en bas, alors on prouve que le coût des cas de base est dominant.
- Si les coûts des lignes sont constants, on les somme simplement.

Remarque.

- **Il faut savoir identifier et gérer ces 3 cas !!!!!!!!** Et pour ça, pas de secret, il faut connaître les 3 cas et s'entrainer en refaisant ces calculs.
- On peut aussi obtenir les minorations associées, mais généralement on se contente d'un $O()$.
- Il existe un théorème célèbre, appelé le « théorème maître ». C'est une application de la propriété précédente au cas où f est un polynôme. Je vous déconseille fortement d'essayer de l'apprendre par coeur : vous ferez des erreurs en mémorisant ces formules[12]. Vous **n**'avez **pas** le droit de l'utiliser en MP2I/MPI. Je vous le donne ci-dessous à titre informatif :

Proposition 14 (Théorème Maître (hors-programme)).
On considère une formule de récurrence de la forme $T(n) = aT(n/b) + O(n^d)$ avec comme cas de base $T(1) = 1$. Alors :

- Si $a < b^d$, alors $T(n) = O(n^d)$ (« le coût de la racine domine »).
- Si $a > b^d$, alors $T(n) = O(n^{\log_b(a)})$ (« le coût des feuilles domine »).
- Si $a = b^d$, $T(n) = O(n^d \log(n))$ (« le coût des étages est constant »).

12. Le programme officiel est d'accord avec moi, et l'a mis hors-programme pour cette raison.

2.1 Complexité spatiale des fonctions récursives

Rappel du cours sur l'organisation de la mémoire : les variables non-allouées sont crées au début de leur bloc et supprimées à la fin.

En particulier, n'existe à tout moment en mémoire que les variables des appels récursifs qui ont commencé mais n'ont pas encore terminé. Dans l'arbre d'appels, commencer un appel revient à « descendre en suivant un trait » et terminer un appel revient à « remonter en suivant un trait ». Ainsi :

Proposition 15 (Complexité spatiale récursive).
La complexité spatiale d'une fonction récursive est le maximum d'espace nécessaire pour une suite d'appels de l'appel initial à un cas de base.
Autrement dit, c'est la somme maximale des complexités locales d'appels le long d'un chemin de la racine à une feuille de l'arbre d'appels.

Remarque. Ce « Autrement dit » ignore les difficultés liées au partage de la mémoire lors d'un passage par référence/pointeur que l'on a déjà évoquées.

Exemple. La fonction Hanoï a une complexité spatiale en (n). En effet, le coût local de chaque appel est constant, et il y a au plus n appels actifs à tout moment[13]. Comme cette borne sur le nombre d'appels est atteinte, c'est même un $\Theta()$.

Exercice. Calculer la complexité temporelle et spatiale de la fonction `tri_fusion` ci-dessous :

```
(** Fusionne deux listes triées en une seule. *)
let rec fusionne (l0 : 'a list) (l1 : 'a list) : 'a list =
  match (l0,l1) with
  | []    , _      -> l1
  | _     , []     -> l0
  | h0::t0, h1::t1 -> if h0 < h1 then
                        h0 :: (fusionne t0 l1)
                      else
                        h1 :: (fusionne l0 t1)


(** Separe la liste l en deux moitié*)
let rec separe (l : 'a list) : 'a list * 'a list =
  match l with
  | []           -> [] , []
  | h       :: [] -> [h], []
  | h::hbis :: t  -> let (u,v) = separe t in
                     h::u, hbis::v

(** Trie l grâce à l'algorithme du tri fusion *)
let rec tri_fusion (l : 'a list) : 'a list =
  match l with
  | []      -> []
  | h :: [] -> [h]
  | _       -> let (u,v) = separe l in
               fusionne (tri_fusion u) (tri_fusion v)
```

13. Cette preuve est plus convainquante avec un arbre d'appels dessiné. Je dirais même que la preuve est incomplète sans.

3 Complexité amortie

Jusqu'à présent, nous avons étudié la complexité d'*un* appel à une fonction. Le but désormais est d'étudier le coût d'une *suite* d'appels, et plus précisément le coût d'un appel au sein de cette suite.

Considérons l'exemple initial suivant. Un·e MP2I veut aller faire des séances de sport dans une salle. Le tarif[14] est le suivant :

- Si c'est la première fois, les frais de dossiers coutent 30€. La place coûte 10€ de plus.
- Les fois suivantes, la place coûte uniquement 10€.

Question : dans une succession de séances, combien coûte chaque séance ? On peut dire que dans le pire des cas, chaque séance coûte 40€... mais on veut bien que c'est grossier : dans une succession de séances, les frais de dossiers ne sont payés qu'une seule fois. On préfère dire que dans une succession de S séances, le coût de chaque séance est $10 + \frac{30}{S}$. On *amortit* le coût de l'abonnement sur toutes les séances (on dit aussi qu'on *lisse* le coût).

Définition 16 (Complexité amortie).
On considère une structure de données sur laquelle on applique une succession de n opérations. On note C_i les coûts réels de chacun de ces opérations successives. On numérote les opérations à partir de 1. *Un* coût amorti est la donnée de coûts fictifs $\hat{C}_i$ tels que :

$$\sum_{i=1}^{n} \hat{C}_i \geq \sum_{i=1}^{n} C_i$$

Autrement dit, la seule condition sur un coût amorti est que la somme des coûts amorti majore le coût réel.

3.0 Méthodes de calcul

3.0.0 Exemple fil rouge

Nous allons présenter trois méthodes qui cherchent à calculer un coût amorti qui soit une majoration assez précise. Nous les appliquerons à un exemple fil rouge : les tableaux dynamiques. Initialement, un tableau dynamique est un tableau à 1 case, qui n'est pas utilisé : nous appelerons. On ajoute un élément à la fin de celui-ci via la fonction ci-dessous :

Fonction Ajout

Entrées : `T` un tableau dynamique ; `x` un élément à ajouter à la fin

```
1 si T est entièrement plein alors
2 |   T' ←nouveau tableau deux fois plus grand que T
3 |   recopier T dans T'
4 |   remplacer T par T'
5 len ←nombre de cases utilisées de T
6 T[len] ←x
```

L'exemple fil rouge sera donc : calculer un coût amorti d'un appel à Ajout au sein d'une suite d'appels à Ajout. On comptera le coût en nombre d'écriture dans un tableau effectuées.

3.0.1 Méthode du comptable

On imagine qu'au lieu de dépenser du temps ou de l'epsace (ou une autre ressource), la fonction dépense des *pièces*. L'objectif de la méthode du comptable est de montrer comment pré-payer des opérations couteuses en augmentant le prix des opératiosn peu couteuses. On amortit ainsi le coût des opérations couteuses sur les autres.

14. Fictif.

Définition 17 (Méthode du comptable).
On considère une structure de données sur laquelle on applique une succession de n opérations. On note C_i les coûts réels de chacun de ces opérations successives. On numérote les opérations à partir de 1.
La **méthode du comptable** est le fait d'ajouter à chaque opération une **accumulation** a_i et une **dépense** d_i. L'accumulation consiste à à poser une ou plusieurs pièces « sur » la structure, en prévision d'un paiement élevé plus tard. Une dépense consiste à utiliser des pièces déjà déposées pour payer une opération couteuse. On doit garantir qu'à tout moment :

$$\sum_{i=1}^{n} a_i \geq \sum_{i=1}^{n} d_i$$

C'est à dire qu'on ne dépense jamais une pièce que l'on a pas encore accumulée. La méthode du comptable définit alors le coût amorti suivant :

$$\hat{C}_i = C_i + a_i - d_i$$

Démonstration. Prouvons que le coût amorti proposé est bel et bien un coût amorti. On a :

$$\begin{aligned}
\sum_{i=1}^{n} \hat{C}_i &= \sum_{i=1}^{n} (C_i + a_i - d_i) \\
&= \left(\sum_{i=1}^{n} C_i\right) + \left(\sum_{i=1}^{n} a_i\right) - \left(\sum_{i=1}^{n} d_i\right) \\
&\geq \sum_{i=1}^{n} C_i && \text{car } \sum_{i=1}^{n} a_i \geq \sum_{i=1}^{n} d_i
\end{aligned}$$

Il s'agit donc bien d'un coût amorti. □

Remarque. En pratique, il faut donner les accumulations, les dépenses, et une justification convainquante du fait que l'on ne dépense pas plus que ce que l'on a accumulé.

Exemple. Calculons *un* coût amorti pour un Ajout au sein d'une suite d'Ajout sur un tableau dynamique par la méthode du comptable. On fixe les accumulations et dépenses ainsi :

- Si l'appel à Ajout n'a pas beosin de doubler le tableau, on paye 1 pièce (pour l'écriture de x), et on stocke 2 autres pièces sur la case que l'on vient d'écrire, afin d'être plus tard capable de payer un doublement du tableau.

  Ainsi, toutes les cases qui ont été écrites depuis la dernière fois que le tableau a été doublé contiennent 2 pièces posées sur elles. Autrement dit, toutes les cases écrites de la deuxième moitié du tableau contiennent 2 pièces accumulées.

  Le coût amorti pour un tel appel à Ajout est donc $\hat{C}_i = 3$.

- Sinon, il faut payer 1 (pour écrire x) et en plus payer la recopie des cases, donc payer autant de cases que le tableau en a. On va pour cela dépenser les 2 pièces accumulées sur chacune des cases de la deuxième moitié du tableau ! En dépensant ce stock, le coût de la recopie est entièrement couvert. On accumule enfin 2 pièces sur la nouvelle case, pour les mêmes raisons que précédemment.

  Le coût amorti pour un tel appel à Ajout est donc $\hat{C}_i = 3$.

On a ainsi prouvé par la méthode du comptable qu'au sein d'une suite d'appels à Ajout, un appel à Ajout a un coût amorti $\hat{C}_i = 3$.

FIGURE 3.5 – Schéma de l'évolution du tableau dynamique et de l'accumulation de la méthode du comptable

Remarque.

- Dans la preuve précédente, on n'a pas toujours explicitement indiqué les valeurs de d_i, C_i et a_i. Je vous conseille de les indiquer si vous avez besoin de repères clairs pour avancer votre preuve (après tout, le choix de bons a_i et d_i sont tout l'enjeu de la méthode), mais de ne pas les mettre s'ils ne feraient qu'alourdir la rédaction. Par contre, ils ne doivent jamais être ambigus : on doit pouvoir comprendre quelle serait leur valeur !
- La méthode du comptable s'applique quand on arrive à bien prévoir quelle opération peut prépayer quelle autre opération. Ici, lors de l'ajout d'un élément on pré-paye pour la recopie future de cet élément ainsi que pour la recopie d'un élément de la première moitié du tableau.

3.0.2 Méthode du potentiel

La méthode du potentiel revient à faire accumuler et dépenser un potentiel « global », au lieu de petits stocks locaux de pièces. La différence est que l'on définit le potentiel par une formule (à trouver, qui doit répondre à nos besoins), et que c'est de cette formule que l'on déduit les accumulations/dépenses.

Définition 18 (Méthode du potentiel).
On considère une structure de données sur laquelle on applique une succession de n opérations. On note C_i les coûts réels de chacun de ces opérations successives. On numérote les opérations à partir de 1.
La **méthode du potentiel** est le fait d'associer à chaque état de la structure un potentiel ($P(0)$ est le potentiel initial, $P(1)$ le potentiel après la première opération, etc). On doit garantir que :

$$\forall i, P(i) \geq P(0)$$

La méthode du potentiel définit alors le coût amorti suivant :

$$\hat{C_i} = C_i + P(i) - P(i-1)$$

Démonstration. Prouvons que le coût amorti proposé est bel et bien un coût amorti. On a :

$$\begin{aligned}\sum_{i=1}^{n} \hat{C}_i &= \sum_{i=1}^{n} (C_i + P(i) - P(i-1)) \\ &= \left(\sum_{i=1}^{n} C_i\right) + \left(\sum_{i=1}^{n} P(i) - P(i-1)\right) \\ &= \left(\sum_{i=1}^{n} C_i\right) + P(n) - P(0) \\ &\geq \sum_{i=1}^{n} C_i \qquad \text{car } P(n) \geq P(0)\end{aligned}$$

Il s'agit donc bien d'un coût amorti. □

Remarque. Toute la difficulté est de trouver un bon potentiel. On veut une fonction qui :

- Augmente un peu lorsque l'on fait une opération peu couteuse, pour « accumuler du potentiel ».
- Décroit d'un coup lors d'une opération couteuse. Cette décroissance du potentiel va compenser le coût de l'opération : elle revient à « dépenser » le potentiel accumulé.

Figure 3.6 – Évolution typique d'un bon potentiel

Exemple. Calculons *un* coût amorti pour un Ajout au sein d'une suite d'Ajout sur un tableau dynamique (initialement vide) par la méthode du potentiel. Nommons $E_0, \ldots$ les états successifs du tableau, et numérotons les opérations à partir de 1 : ainsi, on passe de l'état E_{i-1} à l'état E_i par l'opération numéro i.

Un état E_i du tableau dynamique contient len_i le nombre de cases utilisées dans le tableau, ainsi que $dispo_i$ le nombre de cases dont le tableau dispose vraiment. On définit le potentiel suivant :

$$P(i) = 2len_i - dispo_i$$

Calculons le coût amorti associé :

- Si l'appel à Ajout n'a pas beosin de doubler le tableau, le coût réel est 1 (on écrit x). Calculons la différence $P(i) - P(i-1)$. Comme le tableau n'a pas été doublé, $dispo_i = dispo_{i-1}$ et donc $P(i) - P(i-1) = 2(len_i - len_{i_1}) = 2$.

  Le coût amorti pour un tel appel à Ajout est donc $\hat{C}_i = 3$.
- Sinon, il faut payer 1 (pour écrire x) et en plus payer la recopie de toutes les cases du tableau, donc un coût total de $1+dispoi-1$. Comme le tableau a été doublé, $dispo_i = 2dispo_{i-1}$ et $len_{i-1} = dispo_{i-1}$. On a donc :

$$\begin{aligned}P(i) - P(i-1) &= dispo_{i-1} - dispo_i + 2(len_i - len_{i-1}) \\ &= dispo_{i-1} - 2dispo_{i-1} + 2 \\ &= 2 - dispo_{i-1}\end{aligned}$$

Le coût amorti pour un tel appel est $1 + dispo_{i-1} + 2 - dispo_{i-1}$, donc : $\hat{C}_i = 3$.

On a ainsi prouvé par la méthode du potentiel qu'au sein d'une suite d'appels à Ajout, un appel à Ajout a un coût amorti $\hat{C}_i = 3$.

Remarque. La méthode du potentiel s'applique lorsque l'on n'arrive pas à prévoir quelle opération prépaye laquelle. À la place, on cherche comment doit évoluer le stock global de « pièces » et ce stock global que l'on définit.

3.0.3 Méthode de l'aggrégat

La méthode de l'aggrégat est plus mathématique. Elle consiste à calculer la somme $\sum_{i=1}^{n} C_i$ en la décomposant en « paquets » simples à calculer. On en déduit le coût total de la suite d'opérations, que l'on peut ensuite lisser sur les opérations.

Définition 19 (Méthode de l'aggrégat).
La **méthode de l'aggrégat** consiste à poser $\hat{C}_i = \frac{1}{n}\sum_{i=1}^{n} C_i$.
Toute la difficulté est donc d'avoir réussi à calculer cette somme. On peut utiliser une majoration très précise à la place de la somme.

Démonstration. Prouvons que le coût amorti proposé est bel et bien un coût amorti. On a :

$$\begin{aligned}\sum_{i=0}^{n-1} \hat{C}_i &= \sum_{i=0}^{n-1}\left(\frac{1}{n}\sum_{j=0}^{n-1} C_j\right)\\ &= n\left(\frac{1}{n}\sum_{j=0}^{n-1} C_j\right) && \text{car le contenu de la seconde somme ne dépend pas de } i\\ &= \sum_{j=0}^{n-1} C_j\end{aligned}$$

Il s'agit donc bien d'un coût amorti. □

Exemple. Calculons *un* coût amorti pour un Ajout au sein d'une suite d'Ajout sur un tableau dynamique (initialement vide) par la méthode de l'aggrégat. On numérote les opérations à partir de 1. Par une récurrence immédiate, on montre qu'après l'Ajout numéro i, le tableau contient i valeurs. On montre de même que l'on double le tableau lorsque $i-1$ est une puissance de 2, et que dans ce cas on recopie toutes les valeurs donc $i-1$ valeurs.

Le coût C_i d'une opération est 1 (écriture de x), plus éventuellement le doublement. Donc :

$$C_i = \begin{cases} 1 + i - 1 & \text{si } i - 1 = 2^p \text{ avec } p \in \mathbb{N} \\ 1 & \text{sinon} \end{cases}$$

Donc :

$$\begin{aligned}\sum_{i=1}^{n} C_i &= \sum_{i=1}^{n} 1 + (i-1).\mathbb{1}_{\text{«}i-1 \text{ est de la forme } 2^p\text{»}}\\ &= n + \sum_{p=0}^{\lfloor \log_2 n \rfloor} 2^p\\ &= n + 2^{\lfloor \log_2 n \rfloor + 1} - 1\\ &\leq 3n\end{aligned}$$

La méthode de l'aggrégat propose alors $\hat{C}_i = \frac{3n}{n} = 3$ comme coût amorti.

Remarque. Cette méthode est à utiliser quand on connait une propriété de la structure qui permet de faire des « paquets » agréables, ou que l'on arrive à réorganiser la somme d'une façon qui lui donne plus de sens.

3.1 Remarques et compléments

- **L'une des applications classiques de la complexité amortie est le calcul de la complexité d'une boucle qui opère à chaque itération sur une structure de données**. Voici un exemple en Python, où append correspond à Ajout :

```python
lst_premiers = []
for x in range(0, n): # pour x allant de 0 inclus à n exclu
    if is_prime(x):
        lst_premiers.append(x)
```

  Le calcul de la complexité de cette boucle est très simple si on utilise l'analyse amortie pour affirmer que l'on peut faire comme si chacun des append est en $O(1)$ (puisque l'on effectue une succession de append sur une même liste), et assez compliqué sinon.
- **Il ne faut pas hésiter à faire un schéma** pour illustrer la structure, les accumulations/dépenses ou le potentiel. Un bon schéma permet d'écrire ensuite une preuve plus concise et plus claire. [15]
- Les trois méthodes de calcul reviennent finalement à trouver une façon de majorer précisément une somme. Elles sont cependant expliquées de manière différente. Prenez la méthode qui vous parle le plus (sauf si dans votre situation précise une des méthodes est clairement plus simple que les autres).
- On cherche surtout à majorer et donc à calculer un $O()$. On pourrait adapter les raisonnements pour minorer et obtenir un $\Omega()$ ou un $\Theta()$, mais c'est plus rare.
- **L'état initial de la structure étudiée n'a pas beaucoup d'importance**. Quitte à majorer, on peut ajouter des opérations initiales « fictives » qui amènent l'état initial dans un état plus agréable, sans que cela ne change l'ordre de grandeur du résultat. En effet, les calculs d'un coût amorti lissent des coûts sur un nombre très grands d'opération. Les opérations fictives auront un coût asymptotiquement nul au sein de ce très grand nombre d'opération [16].
- **Un coût amorti n'a de sens qu'au sein de la suite d'opérations où on l'a calculé.** On a prouvé dans ce cours qu'*au sein d'une suite d'*Ajout, chaque Ajout a un coût amorti constant. Mais on pourrait imaginer une nouvelle opération, Démon qui remplit le tableau dynamique jusqu'à ce que son nombre d'éléments soit une puissance de 2. Avec cette nouvelle opération, chacun des Ajout va doubler le tableau ; et donc le meilleur coût amorti possible pour Ajout au sein d'une suite d'opérations qui alterne Ajout et Démon est un coût linéaire en la longueur du tableau.
- **Un coût amorti n'a rien à voir avec la complexité moyenne** [17]. Ce n'est juste pas la même définition ! Un coût amorti est défini sur *une suite d*'opérations, la complexité moyenne sur *une* opération (mais en moyennant sur les entrées possibles pour cette seule opération).

15. Ce conseil s'applique à l'entiereté de l'informatique (et de la plupart des sciences). Aidez à visualiser !
16. Mathématiquement, on dit que la somme partielle d'une série qui diverge vers $+\infty$ est négligeable devant son reste.
17. J'ai d'ailleurs fait très attention à ne jamais utiliser le mot « moyenne » dans toute la section sur la complexité amortie.