Complexité Techniques de calcul et de réduction

Unité : IN311

Lilian BUZER

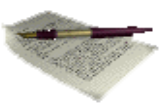
version β

année 2003-2004

Table des matières

Légende :

exercice avancé



à savoir refaire

à connaître par coeur

A Introduction au calcul de complexité

1- La notion de coût d'exécution

Lors de l'étude d'un problème, nous pouvons être amenés à estimer et comparer les temps d'exécution requis par différentes méthodes. Avant d'entrer directement dans une phase de développement et de test, nous pouvons déjà en amont chercher à évaluer de manière théorique la quantité d'appels aux opérations les plus coûteuses en temps. Nous pouvons alors estimer le temps d'exécution suivant deux optiques distinctes. Il s'agit de l'étude en moyenne et de l'étude dans le pire cas. L'approche en moyenne consiste à évaluer la durée moyenne d'exécution d'un algorithme donné. Elle se base sur des modèles probabilistes complexes. Par conséquent elle se révèle souvent très difficile à mettre en oeuvre et sort du cadre de cette introduction. L'approche dans le pire cas consiste à estimer la durée maximale d'exécution d'une méthode donnée. Ce type d'étude est tout à fait à la portée d'un cycle ingénieur. L'évaluation dans le pire cas est par exemple utile pour les procédés en temps réel. En effet, il faut pouvoir contrôler le déroulement d'un programme afin qu'au début de chaque nouvelle étape prévue à un instant t les résultats nécessaires à cette partie aient été calculés dans le temps précédemment imparti.

Problème 1 : Placement des invités

Un traiteur organise un colloque. La seule hôtesse d'accueil présente ne dispose pas de la liste des invités. Elle n'a ni feuille ni crayon. Les cartons d'invitation ont été disposés aux places respectives des invités. Malheureusement les noms ont été écrits dans une police trop petite et l'hôtesse doit donc prendre à chaque fois quelques secondes pour les lire. Sachant qu'il y a n invités et m tables. Exprimez dans le pire cas le nombre de fois où l'hôtesse devra examiner les cartons. Nous négligerons les parcours entre les chaises et nous supposerons que l'hôtesse est tellement stressée qu'elle ne parvient pas à retenir l'emplacement des noms précédemment lus.

Réponse : $n\cdot(n+1)/2-1$

Problème 2 : les boules de bowling

J'ai n boules de bowling numérotées de 1 à n disposées sur des emplacements portant leurs numéros. A la fin de la journée, les clients ont remis les boules complètement dans le désordre. Combien dois-je effectuer de déplacement dans le pire cas pour que les boules soient toutes à leur place ?

Réponse : n

2- La méthode d'évaluation

2.1- Exemple

Prenons une fonction calculant le minimum des valeurs présentes dans un tableau de n éléments :

```
int Min(int *T, int n)
{
 int m=T[0];
 for (int i=1 ; i < n ; i++)
  if ( T[i] < m )
    m=T[i];
 return m;
}
```

Le temps d'exécution est fonction du nombre de lecture en mémoire et du nombre de comparaisons effectuées. Dans le pire cas, nous effectuons n lectures et $n-1$ comparaisons. La durée nécessaire à l'appel de cette fonction est donc proportionnelle aux nombres de valeurs présentes. La constante de proportionnalité dépendra du coût en temps d'une lecture en mémoire et d'une comparaison. Ces paramètres dépendent entièrement de l'architecture utilisée.

Remarque : nous notons que dans cet exemple trivial, la quantité d'opérations effectuées est toujours égale à n quelles que soient les valeurs présentes. Ainsi en moyenne la quantité d'opérations est égale à n .

2.2- La méthode

Dans l'étude du pire cas, nous pouvons écrire :

- Appel séquentiel :

```
f(...)
{
 g(...);
 h(...);
}
```

$$Coût(f(...)) = Coût(g(...)) + Coût(h(...))$$

- Appel conditionnel

```
f(...)
{
 if (...)
   g(...)
 else
   h(...);
}
```

$$Coût(f(...)) = max(Coût(g(...)), Coût(h(...)))$$

Remarque : attention aux appels itératifs

```
f(..., int n)
{
 for (i = 1 ; i < n ; i++)
   g(...);
}
```

Nous aurons $Coût(f(..., n)) = n \cdot Coût(g(...))$. Cependant ce n'est pas toujours le cas, c'est-à-dire que lorsque l'appel à la fonction g() est dépendant de l'indice de boucle :

```
f(..., int n)
{
  for (i = 1 ; i < n ; i++)
    g(...,i);
}
```

Nous devons réécrire le coût en comptabilisant les différents appels, ainsi nous avons :

$$Coût(f(\dots,n))=\sum_{i=1}^{n} Coût(g(\dots,i))$$

3- Notation grand O et classe de complexité

3.1- Principe général

Supposons qu'une méthode effectue $C(n)=3n^2+6n+18$ opérations coûteuses dans le pire cas. Lorsque $C(n)$ est grand, cette valeur devient équivalente à celle du monôme de plus haut degré. Ainsi $C(n)\underset{+\infty}{\sim}3\cdot n^2$. Si nous travaillons sur un processeur pouvant effectuer 10^9 de ces opérations par seconde, nous obtenons les résultats suivants :

n	$f(n)=n$	$f(n)=n^2$	$f(n)=n^4$
10^1	~ 0	~0	~0
10^3	~ 0	~0	∞
10^6	~ 0	∞	∞
10^9	~ 1 s	∞	∞
10^{12}	∞	∞	∞

Nous remarquons que dans certains cas, l'algorithme est réalisable dans un temps quasi-immédiat (inférieur à une seconde : ~0). Dans d'autres cas, la durée dépasse largement l'heure et sauf cas particulier, la méthode peut être considérée comme trop lente (∞). Uniquement dans le cas $f(n)=n$ lorsque la quantité d'opérations à effectuer est de l'ordre de la puissance du processeur, nous nous trouvons dans une tranche où la durée du programme s'avère acceptable (ni immédiat, ni trop long). Nous sommes amenés à penser qu'il existe une zone de cassure à partir de laquelle une méthode ne peut plus être utilisée. Ainsi lorsque $f(n)=n$ nous pouvons traiter de l'ordre de 10^9 données, seulement 10^4 lorsque $f(n)=n^2$ et uniquement 179 lorsque $f(n)=n^4$. Au final, l'importance du coefficient multiplicateur est relative. D'une part, la valeur de ce dernier est quasiment du même ordre de grandeur d'une méthode à l'autre. D'autre part, lorsque n est grand, le polynôme $C(n)$ de plus haut degré finira toujours par l'emporter.

3.2- Définition de la classe de complexité

Définition : notation grand O

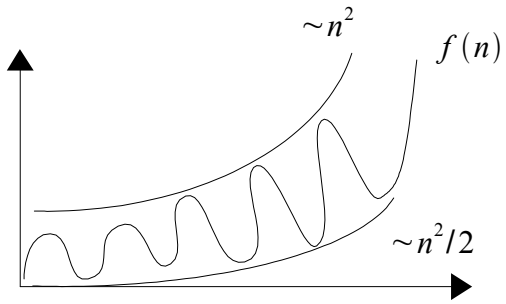
Une fonction $T(n)$ est en grand O de $f(n)$: $T(n)=O(f(n))$ si et seulement si :

$\exists N,\alpha,\beta$ *t.q.* $\forall n>N, \alpha\cdot f(n)\leqslant T(n)\leqslant\beta\cdot f(n)$.

Remarque : cette définition est plus forte que la définition usuelle du grand O mathématique. En effet, ce dernier définit uniquement une borne supérieure. Ainsi on peut écrire dans ce contexte : $x^2 = O_{math}(x^3)$. Cependant dans notre étude, cette possibilité n'est pas intéressante car il nous sera inutile d'estimer une fonction de coût $C(n)=n^2$ en $O(n^3)$. Ainsi nous surchargeons la définition usuelle afin qu'elle modélise à la fois une borne supérieure et une borne inférieure.

Remarque : nous avons $T(n)\sim f(n)\Rightarrow T(n)=O(f(n))\Rightarrow T(n)=O_{math}f(n)$

En effet, l'équivalence entre deux fonctions est une définition plus forte que notre notation grand O. Pour cela il suffit de regarder l'exemple ci-dessous qui présente une fonction en $O(n^2)$ qui ne peut être équivalente à une fonction $f(n)=c\cdot n^2$.



Remarque : l'utilisation de cette notation grand O est fortement reliée aux cas pratiques rencontrés lors de l'étude des algorithmes. En effet, si nous obtenions un programme ayant un coût du type $C(n)=n\cdot\sin(n)$, le modèle proposé ne serait plus utilisable dans ce contexte.

3.3- Classes de complexité les plus usuelles

$O(n)$	*linéaire*	$O(n^2)$	*quadratique*
$O(n^3)$	*cubique*	$O(\log n)$	*logarithmique*
$O(c^n)$	*exponentielle*		

4- Manipulation des classes de complexité

4.1- Règles d'écriture

1– $f(n)=O(f(n))$
2– $g=O(f), f=O(h)\Rightarrow g=O(h)$
3– $c\cdot O(g(n))=O(c\cdot g(n))=O(g(n))$
4– $f(n)\cdot O(g(n))=O(f(n)\cdot g(n))$
5– $\lim_{+\infty} g/f\to 0 \quad \Rightarrow O(f+g)=O(f)$
6– $O(f)+O(g)=O(f+g)$

Remarque : l'écriture $O(f+g)$ n'est utilisée que par commodité

4.2- Définition d'un ordre

Nous pouvons définir un ordre entre les différentes classes de complexité. Si nous avons deux classes de complexité $O(f(n))$ et $O(g(n))$ et si $\lim\limits_{+\infty} \frac{f(n)}{g(n)} \to 0$ alors nous pouvons écrire : $O(f(n)) < O(g(n))$.

4.3- Les classes polynomiales et logarithmiques

Nous cherchons à trier l'ensemble des classes de complexité du type : $O(n^{\alpha} \cdot \log^{\beta} n)$. Pour cela nous rappelons d'abord :

$$\lim_{+\infty} \frac{x^{\beta>0}}{e^x} \to 0 \qquad \lim_{+\infty} \frac{\log x^{\alpha>0}}{x^{\beta>0}} \to 0 \qquad x^y = e^{y \cdot \ln x} \qquad \log x = \frac{\ln x}{\ln 10} \qquad \ln_2 = \frac{\ln x}{\ln 2} = \frac{\log x}{\log 2}$$

Montrez que : $2^{\ln_2 x} = x$

Nous travaillons toujours dans un cadre où n tend vers l'infini. Pour rappel mnémotechnique, l'exponentielle l'emporte toujours sur le polynôme qui l'emporte toujours sur le logarithme. Pour ordonner les $O(n^{\alpha} \cdot \log^{\beta} n)$, il suffit de considérer d'abord les polynômes : $O(n^{0.5}) < O(n) < O(n^2)$ et ensuite d'insérer les facteurs logarithmiques entre chacun. Au final nous obtenons par exemple :

$$O(n^{0.5}) < \underbrace{\qquad}_{O(n^{0.5} \cdot \log n) < O(n^{0.5} \cdot \log^{1.5} n)} < O(n) < \underbrace{\qquad}_{O(n \cdot \log n) < O(n \cdot \log^{1.5} n)} < O(n^{1.5}) < \underbrace{\qquad}_{O(n^{1.5} \cdot \log n) < O(n^{1.5} \cdot \log^{1.5} n)} < O(n^2)$$

4.4- Intérêt pratique

Soient deux méthodes de coût respectif : $f(n) = 5 \cdot n^2 + 6 \cdot \log n + 7$ et $g(n) = \sqrt{n} + \log n$. Calculez la classe de complexité de $O(f(n) \cdot g(n))$.

Résolution :

$f(n) = 5 \cdot n^2 + 6 \cdot \log n + 7 \rightarrow f(n) = O(n^2)$

$g(n) = \sqrt{n} + \log n \rightarrow g(n) = O(n^{0.5})$

$\rightarrow O(f(n) \cdot g(n)) = O(n^2 \cdot n^{0.5})$

5- D'autres estimations

Nous avons cherché à évaluer le comportement d'une méthode dans le pire cas. De manière symétrique, il est possible de chercher à estimer le meilleur cas. La classe de complexité sera alors notée : $\Omega(f(n))$ (grand oméga). Si un algorithme est tel que $\Omega(f(n)) = O(f(n))$ alors nous dirons qu'il est aussi en $\Theta(f(n))$ (grand thêta).

Exemple 1 : la recherche de la position d'un élément dans une liste est dans le pire cas en $O(n)$ (il suffit que l'élément recherché soit le dernier de la liste ou qu'il ne soit pas présent) et dans le meilleur cas en $\Omega(1)$ (il suffit que l'élément recherché soit le premier). Comme $\Omega(1) \neq O(n)$ il n'y a pas d'écriture en grand thêta.

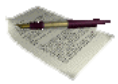
Exemple 2 : le calcul du minimum d'un ensemble de valeurs est en $O(n)$ et en $\Omega(n)$. En effet dans le pire cas comme dans le meilleur n éléments doivent être lus et comparés. Par conséquent la méthode est en $\Theta(n)$.

Remarque : on parle parfois de la complexité d'un problème. Elle ne correspond pas forcément à la complexité d'un algorithme résolvant ce problème. La complexité d'un problème est reliée à la meilleure complexité des méthodes connues. Ainsi nous avons vu des méthodes de tri de complexité quadratique alors que le problème du tri lui est de complexité $O(n \cdot \log n)$.

6- Exemples de calcul de complexité

6.1- Exercices : quelques fonctions standards

o Recherche dichotomique - récursif : $O(\log n)$
o Recherche dichotomique - itératif : $O(\log n)$
o Recherche dans une liste non triée : $O(n)$
o Minimum : $O(n)$
o Tri par sélection : $O(n^2)$
o Fusion de deux listes triées : $O(n+m)$
o Recherche de doublons – listes non triées : $O(n \cdot m)$
o Recherche de doublons – listes triées : $O(n \cdot \log m)$



Remarque : dans l'exercice sur le tri par sélection, il faut faire attention à bien manipuler les différents paramètres pour construire une démonstration exacte. Attention pour la fonction de fusion à prendre des paramètres supplémentaires pour résoudre correctement l'exercice.

6.2- Les incontournables à connaître

Recherche dichotomique	$O(\log n)$
Min, max, moyenne	$O(n)$
Tri	$O(n \cdot \log n)$
Fusion de deux listes triées	$O(n+m)$

6.3- Recherche des premiers

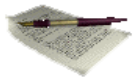
Suivant une liste de n scores, on veut connaître les k premiers. Plusieurs approches sont possibles :

1- On prend le premier on le retire de la liste et on recommence k fois : $O(k \cdot n)$
2- On tri les joueurs et on affiche les k premiers : $O(n \cdot \log n)$

insérez les exos TD corrigés

Problème : quelle méthode faut-il choisir suivant la variable k ?
Réponse : si $k < O(\log n)$ alors il faut choisir la première méthode
si $k = O(\log n)$ les deux méthodes sont de même classe de complexité
si $k > O(\log n)$ la deuxième approche est plus intéressante

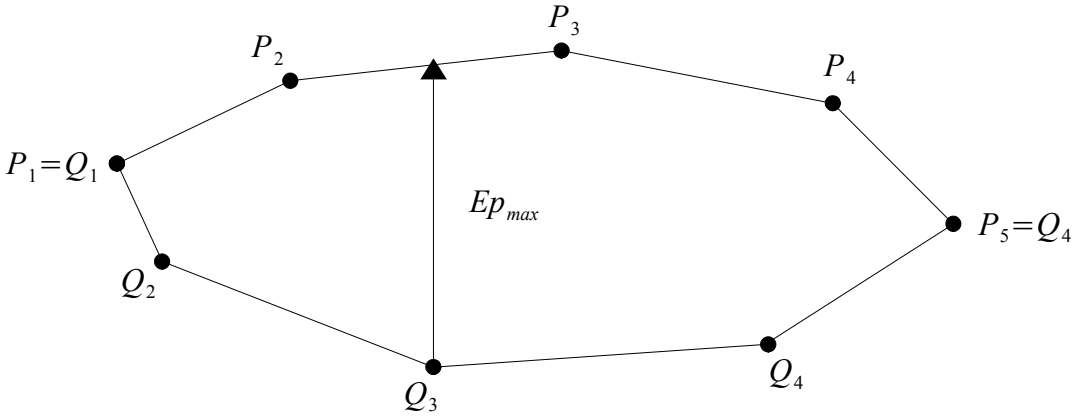
Question : quelle méthode faut-il choisir lorsque je veux



- les trois premiers ?
- 10% des admis ?
- Organiser des 8èmes de finale

6.4- Le calcul de l'épaisseur verticale maximale

Nous étudions l'épaisseur maximale d'un convexe bidimensionnel. L'intérêt de cette application est par exemple de savoir si un objet est en mesure d'être saisi par une pince mécanique. Le polygone convexe est représenté en mémoire suivant deux listes de sommets correspondant respectivement à la bordure supérieure et inférieure du convexe. Pour supprimer les cas particuliers ces deux listes ont pour même point de départ (resp. de fin) le sommet le plus à gauche (resp. le plus à droite), voir figure ci-dessous.



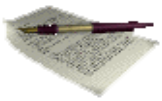
Nous remarquons que l'épaisseur maximale est forcément localisée sur un sommet du polygone. Une fonction calculant la distance verticale entre un point et un segment est donnée : CalDist(Point,Segment). Comme cette fonction a un ensemble de données à traiter de taille bornée, sa complexité est obligatoirement en $O(1)$. Il faut donc trouver une paire segment-point qui correspond à la solution. Le segment et le sommet étant forcément par définition du problème situés sur deux bordures différentes, deux cas symétriques apparaissent : le sommet est présent sur la bordure supérieure ou sur la bordure inférieure. Pour simplifier, nous considérons uniquement le cas où le sommet est sur la bordure supérieure, l'autre cas se déduisant de manière triviale.

Méthode 1 : double parcours

```
Ep = 0
Pour i = 1 à n
  Pour j = 1 à m
    Si ( Q[j].X <= P[i].X <= Q[j+1].X )
      Ep = max(Ep,CalDist(P[i],Q[j],Q[j+1])
```

Complexité : O(n²)

Méthode 2 : parcours et recherche dichotomique



```
Ep = 0
Pour i = 1 à n
  j = RechDich(Q,m,P[i].X)
  Ep = max(Ep,CalDist(P[i],Q[j],Q[j+1])
```

Complexité : O(n.log n)

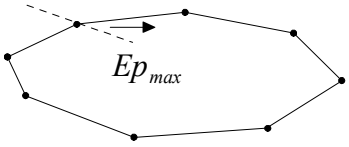
Méthode 3 : parcours unique mais en parallèle

```
Ep = 0
j=1
Pour i = 1 à n
  Tant que (Q[j].X < P[i].X) alors j++
  Ep = max(Ep,CalDist(P[i],Q[j],Q[j+1])
```

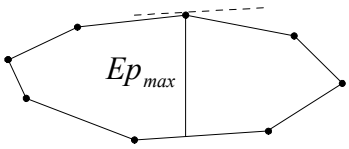
Complexité : O(n)

Méthode 4 : double parcours dichotomique

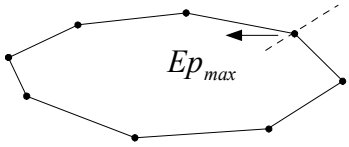
Cette partie correspond à un niveau recherche et la construction d'un tel résultat ne peut être demandée en examen. Il faut remarquer, que si nous notons f une fonction dessinant la bordure supérieure et g la bordure inférieure alors, la fonction g-f est une fonction convexe. Son minimum est atteint à la position de l'épaisseur maximale. Pour une abscisse donnée, nous pouvons savoir si ce minimum est situé à gauche ou à droite de cette abscisse. Pour cela il suffit de faire passer par le sommet étudié une parallèle au segment associé. Les configurations obtenues donnent une réponse sur la localisation du minimum :



Minimum de g-f à droite



Minimum de g-f atteint



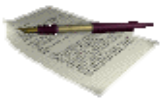
Minimum de g-f à gauche

La fonction Gradient qui retourne la réponse sur la localisation reçoit un nombre fini de données, elle est donc en O(1). Nous appliquons une méthode de coupe dichotomique permettant de réduire par deux à chaque fois l'ensemble des sommets considérés. Le segment faisant face au sommet courant est trouvé par une recherche dichotomique suivant l'abscisse du sommet courant.

```
i=1  j=n
Répète
  p = (i+j)/2
  k = RechDich(Q,m,P[p].X)
  Grad = Gradient(p,k)
  si ( Grad == LEFT )  j = p;
  sinon si ( Grad == RIGHT) i=p;
          sinon retourner CalDist(P[p],Q[k],Q[k+1])
```

Complexité : $O(\log^2 n)$ – meilleure complexité atteinte

6.5- Calcul de l'enveloppe convexe d'un ensemble de points

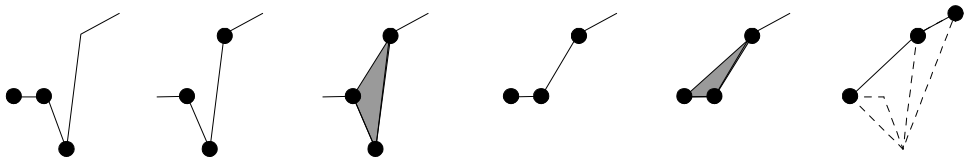


Exemple 1: la méthode des trois pièces (3 coins)

Nous nous proposons de construire l'enveloppe convexe associée à un ensemble de n points. Pour cela nous allons créer indépendamment la bordure supérieure et la bordure inférieure. Les points présents sont d'abord triés par abscisses croissantes. Ensuite nous appliquons la méthode suivante pour construire la bordure supérieure :

0- Positionner les trois pièces sur les trois premiers points
1- Si les pièces forment un angle ouvert
    alors fermer le trou, faire reculer les deux dernières pièces et retourner en 1
2- Avancer les 3 pièces
3- Tant que les 3 pièces ne sont pas au bout, retourner en 1.

Exemple :



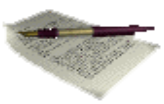
a- départ b- avancée c- angle ouvert d- recul e- angle ouvert f- bordure supérieure

Exercice : Ecrire l'algorithme correspondant. Exhiber les deux cas particuliers pouvant poser problème et proposer une modification les traitant de manière élégante.

Remarque : Les techniques de calcul de complexité vues précédemment ne s'appliquent plus ici car il est impossible d'exprimer de manière exacte les actions effectuées. En effet, les trois pièces peuvent avancer et reculer à n'importe quel moment.

Exercice : Montrer que cette méthode est linéaire. Pour cela démontrer que le nombre d'avancées et le nombre de retours sont bornés par n. N'oublier pas d'après la définition du grand O, de montrer qu'il y a un minimum de $\alpha \cdot n$ opérations effectuées.

Conclusion : Il est alors possible de calculer l'enveloppe convexe de toute ligne polygonale simple (qui ne s'auto-intersecte pas) en temps linéaire (algorithme de Melkmann). La complexité du calcul de l'enveloppe convexe d'un ensemble de n points quelconques est en $O(n \log n)$. L'algorithme construit ici correspond bien à ce résultat puisque le facteur en $O(n \log n)$ apparaît dans la méthode de tri employée au départ.



Exemple 2 : l'algorithme de l'emballage de paquet cadeau (Gift wrapping)

Présentation : Nous partons d'un point extrême, par exemple le plus à gauche. Ce point est forcément situé sur l'enveloppe convexe. Prenons un papier cadeau dont on fixe l'extrémité sur ce point. Nous étirons le papier vers le haut et ensuite nous tournons dans le sens des aiguilles d'une montre. Lorsque le papier rencontre un point, ce dernier est forcément un sommet de l'enveloppe finale. En effectuant le tour complet, l'enveloppe convexe est construite.

Exercice : Construire l'algorithme correspondant. Pour cela, vous disposez d'une fonction $angle(P_1, P_2, P_3)$ retournant la valeur (signée) de l'angle $\widehat{P_1 P_2 P_3}$. Montrer que cette méthode est quadratique.

7- Calculs de complexité avancés

7.1- Comparaison de classes

- $O(2^n)$ et $O(e^n)$
- $O(e^{x^2})$ et $O(e^x)$
- $O(2^{2^n})$ et $O(e^n)$

7.2- Approche récursive avec décimation

Nous cherchons à déterminer la classe de complexité $T(n)$ qui vérifie la propriété suivante : $T(n)=f(n)+T(\alpha \cdot n)$ avec $\alpha>1$. Nous notons $\beta=1/\alpha$.

Cas $f(n)=O(n)$

Nous obtenons par développement de la relation de récursivité :

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_\beta n} O(\alpha^i \cdot n) \\ &= O(n \cdot \sum_{i=0}^{\log_\beta n} \alpha^i) \quad \text{or} \quad \sum_{i=0}^{\log_\beta n} \alpha^i \text{ est une constante, série positive bornée par } \frac{1}{1-\alpha} \\ &= O(c \cdot n) = O(n) \end{aligned}$$

Cas $f(n)=O(n^2)$

Nous obtenons par développement de la relation de récursion :

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_\beta n} O((\alpha^i \cdot n)^2) \\ &= O(\sum_{i=0}^{\log_\beta n} \alpha^{2i} n^2) \\ &= O(n^2 \cdot \sum_{i=0}^{\log_\beta n} \alpha^{2i}) \quad \text{or} \quad \sum_{i=0}^{\log_\beta n} \alpha^{2i} \text{ est une constante} \\ &= O(c \cdot n^2) = O(n^2) \end{aligned}$$

Ainsi nous remarquons $f(n) \in O(n^k) \Rightarrow T(n) \in O(n^k)$

Cas $f(n)=O(\log n)$

$$\begin{aligned}T(n) &= \sum_{i=0}^{\log_\beta n} O(\log \alpha^i n)\\ &= O(\sum_{i=0}^{\log_\beta n} (i\log\alpha+\log n))\\ &= O(\log\alpha\cdot\sum_{i=0}^{\log_\beta n} i)+O(\sum_{i=0}^{\log_\beta n}\log n)\\ &= O(\sum_{i=0}^{\log_\beta n} i)+O(\log_\beta n\cdot\log n)\\ &= O(\log_\beta^2 n)+O(\log_\beta n\cdot\log n)=O(\log^2 n)\end{aligned}$$

Cas $f(n)=O(\log^2 n)$

$$\begin{aligned}T(n) &= \sum_{i=0}^{\log_\beta n} O(\log^2 \alpha^i n)\\ &= O(\sum_{i=0}^{\log_\beta n} i^2\log^2\alpha)+O(\sum_{i=0}^{\log_\beta n} 2\ i\log\alpha\log n)+O(\sum_{i=0}^{\log_\beta n}\log^2 n)\\ &= O(\log_\beta^3 n)+O(\log_\beta^2 n\log n)+O(\log_\beta n\log^2 n)\\ &= O(\log^3 n)\end{aligned}$$

Ainsi nous remarquons $f(n)\in O(\log^k n)\Rightarrow T(n)\in O(\log^{k+1} n)$

7.3- Les erreurs à ne pas commettre

Remarque 1 :

Il serait maladroit de transformer directement l'expression $T(n)=O(n)+T(\alpha\cdot n)$ en écrivant : $T(n)=O(n)+O(\alpha n)+...=O(n)+O(n)...=O(n\log n)$. En effet si le terme en α^i peut faire penser à une constante, ce n'en est pas le cas car il est insidieusement dépendant de n. En effet, nous pouvons écrire :

$$\sum_{i=0}^{4} O(\alpha^i n)=\sum_{i=0}^{4} O(n)=O(n)$$

car $O(\alpha n), O(\alpha^2 n), O(\alpha^3 n), O(\alpha^4 n)$ sont toutes des fonctions en $O(n)$. Cependant dans le cas précédent l'indice i atteint la valeur $\log_\beta n$, pour le dernier membre nous avons $O(\alpha^{\log_\beta n} n)$ et cette classe est plus faible que $O(n)$ et ne doit donc pas être remplacée abusivement par un $O(n)$ qui après sommation fournirait une complexité erronée en $O(n\log n)$.

Remarque 2 :

Rien ne nous empêche d'appeler une fonction log(n) fois, fonction qui va exécuter au final n, n/2, n/4, n/8 ... opérations. Nous écrirons pourtant cette fois que le coût global est en $O(n\log n)$. Cela est juste, car nous travaillons dans le modèle du pire cas, et si cette fonction peut exécuter n

opérations, elle est donc en $O(n)$. Même si cet exemple donne une somme linéaire en n, il se pourrait très bien que la prochaine fois les log(n) appels génèrent n puis n, ... et n calculs, ce qui amènerait un coût total en $O(n\log n)$.

7.4- Etude des doubles appels récursifs

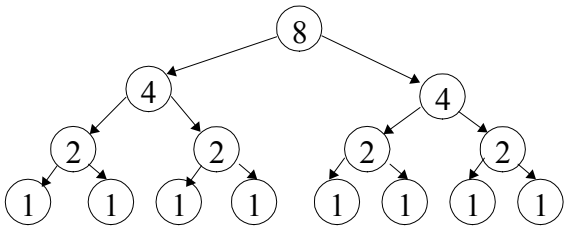
Nous allons étudier l'équation suivante :

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

Cas $f(n)=O(n)$

Prenons par exemple le déroulement pour n=8

Nombre de noeuds	*Hauteur*	*Travail à chaque noeud*
1	3	8
2	2	4
4	1	2
8	0	1



Si j'ai n feuilles, alors $n-1$ noeuds sont présents. La hauteur de l'arbre est égale à $\log_2 n$. Ainsi nous avons successivement :

$$\begin{aligned} T(n) &= \sum_{h=0}^{\log_2 n} \sum_{k=0}^{2^{\log_2 n-h}} O(2^h) \\ &= \sum_{h=0}^{\log_2 n} O(2^{\log_2 n}) = \sum_{h=0}^{\log_2 n} O(n) = O(n\log n) \end{aligned}$$

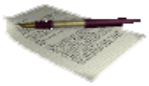
8- La complexité mémoire

8.1- Introduction

Nous avons estimé avec le modèle précédent le temps d'exécution des programmes. Nous pouvons étendre ce principe à l'évaluation de la quantité de mémoire nécessaire pour exécuter un algorithme.

Remarque : le nombre d'opérations effectuées en mémoire est une borne inférieure pour la complexité en temps. Par exemple, si nous trions n nombres, la méthode de tri devra forcément donner un résultat comportant ces n nombres et ainsi la classe de complexité de tous les algorithmes de tri sera $\geqslant O(n)$. Cependant la taille mémoire de stockage des données du problème n'est pas une borne pour la complexité en temps. En effet, pour une recherche dichotomique, la complexité mémoire est en O(n) et la complexité en temps est seulement en O(log n).

8.2- Attaque de l'algorithme El-Gamal



Nous nous proposons dans cet exercice de discuter la qualité de la méthode de cryptographie d'El-Gamal en évaluant les classes de complexité des fonctions nécessaires pour casser son codage. Nous précisons que cette activité est licite. En effet, elle occupe des centaines de mathématiciens et d'informaticiens de part le monde, car vérifier qu'il n'existe pas de méthodes triviales pouvant décoder une méthode de cryptographie c'est aussi garantir son efficacité.

Introduction : le logarithme discret.

Nous travaillons dans un corps $\mathbb{Z}/p\mathbb{Z}$ avec p premier. L'exponentielle discrète est définie à partir de la multiplication du corps. Ainsi nous avons dans $\mathbb{Z}/13\ \mathbb{Z}$:

$$2^1=2\ [13] \quad 2^2=4\ [13] \quad 2^3=8\ [13] \quad 2^4=3[13]$$
$$2^5=6\ [13] \quad 2^6=12\ [13] \quad 2^7=11\ [13] \quad 2^8=9\ [13]$$
$$2^9=5\ [13] \quad 2^{10}=10\ [13] \quad 2^{11}=7\ [13] \quad 2^{12}=1\ [13]$$

Nous avons construit une bijection de $\mathbb{Z}/p\mathbb{Z}$. Nous pouvons ainsi définir le logarithme discret comme l'opération inverse. Ainsi $\log_2 7 = 11\ [13]$ et $\log_2 9 = 8\ [13]$.

Calcul :

Deux approches sont possibles. Soit nous précalculons toutes les valeurs du logarithme discret dans un tableau et nous accédons directement aux valeurs résultats. Complexité en mémoire : O(p), complexité en temps O(1) (hors précalcul). Soit nous calculons itérativement les valeurs x^i jusqu'à trouver la valeur recherchée. Complexité en mémoire : O(1), complexité en temps O(p).

Pour des clefs de 128 bits, p est de l'ordre de $2^{128}=3{,}4.10^{38}$. Sur une machine récente, la puissance processeur est de l'ordre de 10^9 et la taille mémoire de l'ordre de 10^11. Dans tous les cas, avec les deux méthodes précédentes nous ne pouvons calculer un logarithme discret.

Le principe :

Futur receveur :
p : public
x , y : choisis

$\xleftarrow[\text{y caché}]{x\ ,\ x^y}$

Emetteur :
a_i : message privé
k : choisi

$\xrightarrow[\text{k caché}]{x^k\ ,\ a_i(x^y)^k}$

Décodage :
$a_i \cdot \frac{1}{(x^k)^y}$

Nous supposons que les bandits ont accès à toutes les informations qui transitent entre les deux interlocuteurs. Les paramètres x et k sont des clefs privées. La connaissance de l'une ou de l'autre permet de décoder le message en intégralité. On peut considérer ces valeurs comme cachées. En effet, pour pouvoir déterminer ces dernières à partir des informations qui transitent, il faudrait calculer un logarithme discret ce qui comme nous l'avons vu reste très complexe. La fonction puissance est par contre simple à calculer : complexité en O(log p) . Ce procédé de cryptographie se base donc sur la difficulté calculatoire de déterminer la valeur inverse de l'exponentielle dans $\mathbb{Z}/p\mathbb{Z}$. Si demain, un mathématicien trouvait un théorème qui rendrait triviaux les calculs de logarithme discret, cette méthode de cryptographie serait rendue instantanément obsolète.

Une méthode intermédiaire

Dans les deux méthodes précédentes, soit la mémoire soit la puissance calcul était mise à rude épreuve. Cependant dans ces deux circonstances, l'autre partie était complètement inutilisée (complexité en O(1)). Nous allons maintenant présenter une méthode intermédiaire qui équilibre la complexité entre la mémoire et la puissance processeur. Il s'agit de la méthode « Baby steps, Giant Steps » de Shanks.

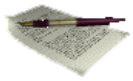
Mise en place :

Prenons le cas où p=97. Nous pouvons écrire tout nombre y soit la forme $y = y_1\, y_2$ avec y_1 dizaine et y_2 unité . Nous avons alors :

$$x^y = z[p] \Leftrightarrow x^{y_1\, y_2} = z[p] \Leftrightarrow x^{y_1 0} = z \cdot x^{-y_2}[p]$$

Il nous suffit de calculer les valeurs $z \cdot x^{-y_2}[p]$, de les trier en vue de pouvoir appliquer une recherche dichotomique. Ensuite nous calculons successivement les valeurs $x^{y_1 0}$ et nous les recherchons dans l'ensemble des $z \cdot x^{-y_2}[p]$. Si une correspondance est trouvée nous avons déterminé une valeur *y* correspondant au logarithme discret cherché. Dans notre cas ($10 \sim \sqrt{p}$) nous avons pour le calcul de $\log_x z$:

Génération des $z \cdot x^{-y_2}[p]$:	$O(\sqrt{p})$
Tri des $z \cdot x^{-y_2}[p]$:	$O(\sqrt{p}\log\sqrt{p})$
Génération des $x^{y_1 0}$:	$O(\sqrt{p})$
Recherche parmi les $z \cdot x^{-y_2}[p]$:	$O(\sqrt{p}) \cdot O(\log\sqrt{p})$
Complexité en temps :	$O(\sqrt{p}\log p)$



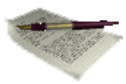
Complexité en mémoire : $O(\sqrt{p})$

Conclusion :

Cette méthode permet d'équilibrer l'utilisation mémoire et l'utilisation processeur. Pour des clefs de l'ordre de 64 bits, nous avons maintenant : une taille mémoire de l'ordre de $2^{32} = 4{,}3.10^9$ et une quantité de calcul de l'ordre de $64 * 2^{32} = 2{,}8.10^{11}$ ce qui rend la méthode réalisable sur un PC actuel. En mars 1999, le gouvernement français a autorisé l'utilisation de clefs de 128 bits. Nous confirmons que pour cet algorithme, il faut une telle taille pour être correctement protégé. Nous comprenons ainsi l'inquiétude des principaux utilisateurs (banques...) qui avant 1999 ne pouvaient utiliser que des clefs de taille 32 bits.

Question : Quelle est la meilleure méthode de répartition pour l'algorithme de Shanks qui minimise la classe de complexité ? Prenons p = ab, a le nombre de baby steps (les unités dans l'exemple précédent) et b le nombre de giant steps (les dizaines). Nous avons :

Génération des $z \cdot x^{-y_2}[p]$:	$O(a)$
Tri des $z \cdot x^{-y_2}[p]$:	$O(a\log a)$
Génération des $x^{y_1 0}$:	$O(b)$
Recherche parmi les $z \cdot x^{-y_2}[p]$:	$O(b) \cdot O(\log a)$
Complexité en temps :	$O((a+b)\log a)$



Complexité en mémoire : $O(a)$

Supposons que a est un polynôme de p : $a = p^{\alpha}$. La complexité en temps est alors égale à : $O((p^{\alpha} + p^{1-\alpha}) \cdot \log p)$. La complexité minimale est atteinte pour $\alpha = 1/2$, ainsi la meilleure complexité en temps atteignable par l'algorithme de Shanks sous ces hypothèses est $O(\sqrt{p} \log p)$. Effectivement en pratique, si nous tenons à réaliser cette méthode sur une machine donnée, nous positionnerons la quantité a sur la taille mémoire disponible et nous obtiendrons ensuite la complexité en temps correspondante.

B Les techniques de réduction

Nous avons remarqué dans les exercices précédents que la classe de complexité d'un algorithme pouvait être diminuée en utilisant diverses techniques : structures de données, notions mathématiques et surtout adaptations algorithmiques. Ces diverses réductions ont été présentées au cas par cas. Néanmoins, historiquement, on constate que certaines de ces techniques se sont regroupées autour de principes généraux adaptés à la construction et à l'optimisation des algorithmes. Nous allons exposer trois de ces grandes méthodes : diviser pour régner, approche gloutonne et programmation dynamique.

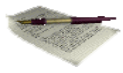
1- Diviser pour régner (divide & conquer)

La construction de la solution d'un problème par cette méthode s'effectue en trois phases :

- Décomposition du problème courant en sous-problèmes (divide)
- Résolution de chaque sous-problème (conquer)
- Construction de la solution à partir des solutions des sous-problèmes (combine)

Diviser pour régner fut une célèbre stratégie militaire avant de devenir une technique de conception algorithmique. Les généraux remarquèrent qu'il était plus facile de vaincre une armée de 50000 hommes puis une autre de la même taille que de directement affronter une seule armée de 100000 hommes. Cette technique appliquée à l'informatique a souvent permis pour un problème donné de construire des algorithmes de plus basse complexité.

1.1- Tri par fusion (MergeSort)



Nous avons précédemment vu les méthodes quadratiques de tri par sélection et insertion. Cependant nous n'avons pas su construire d'algorithme de complexité optimale dans le pire cas. Appliquons notre nouvelle technique pour mettre en place un tri par fusion (MergeSort). Pour cela, nous allons répartir les nombres à trier en deux groupes (divide), trier ces deux groupes séparément (conquer) et réunir les deux listes triées pour n'en avoir plus qu'une (combine).

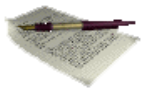
```
MergeSort(int * T, int n)
{
   if ( n == 1 ) return;                  // divide   O(1)
   m = n/2;
   MergeSort(T,m);                        // conquer  2·T(n/2)
   MergeSort(&T[m],n-m);                  // combine  O(n)
    int * D = new int[n];
   Fusion(T,m,&T[m],n-m,D);
   Recopie(T,D,n);                        // fusion dans un tableau temporaire
   delete [] D;
}
```

Soit T(n) la classe de complexité de cette méthode. L'étape de divide est en $O(1)$ et l'étape de conquer est en $2 \cdot T(n/2)$. Le combine consiste à construire par fusion une liste triée à partir de deux listes triées. Cet exercice a été vu dans la première partie. Nous rappelons que sa complexité est en $O(n)$. Au final nous obtenons une équation du type :

$$T(n)=O(1)+2\cdot T(n/2)+O(n)=O(n)+2\cdot T(n/2)$$

Cette équation a déjà été résolue au chapitre précédent, nous savons qu'elle admet pour solution : $T(n)=O(n\cdot\log n)$. Nous avons donc construit un algorithme de complexité optimale pour ce problème.

1.2- Tri par pivot médian



La construction d'un algorithme par la méthode diviser pour régner n'est pas unique pour un problème donné. En effet, cette technique peut amener à diverses constructions même si ces dernières sont toutes basées sur une approche divide&conquer. De ce fait, nous présentons une autre méthode de tri de type diviser pour régner mais très différente de la précédente.

Cette fois, nous choisissons une valeur pivot, qui va séparer l'ensemble des valeurs en deux parties, celles qui lui sont inférieures et celles qui lui sont supérieures. S'il y a k éléments dans la liste des inférieurs, la valeur pivot sera positionnée au final dans la $k+1$ *ème* case du tableau trié. Comme valeur pivot, nous prenons la valeur médiane de la liste des éléments courants. La fonction $Mediane(n)$ est donnée et elle admet une complexité en $O(n)$. Cette valeur particulière nous permet de diviser la liste courante en deux sous-listes de taille égale. Ensuite nous utilisons une fonction linéaire *Split* , qui répartit les éléments du tableau autour de la valeur pivot. L'algorithme correspondant est :

```
MedianSort(int * T, int n)
{
   int i, int s,                    // nombre d'éléments inférieurs (resp. supérieurs)
   if ( n <= 1 ) return;
   int med=Median(T,n,i,s);     // Divide  O(n)     (i+s=n-1)
   Split(T,n,i,s,med);
   T[i+1]=med;
   MedianSort(T,i);             // Conquer  2·T(n/2)
   MedianSort(&T[k+1],s)
}
```

Cette fois l'étape de divide est en $O(n)$. L'étape conquer est de la même manière basée sur un double appel récursif. L'étape combine est absente car il n'y a tout simplement rien à faire. L'équation de la fonction de complexité est identique à la précédente et cette méthode est donc aussi en $O(n\cdot\log n)$.

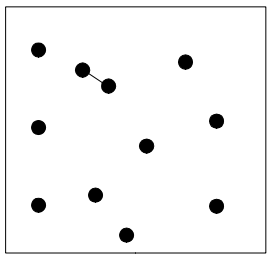
Cette méthode est donc optimale. Pour contourner le surcoût induit par le calcul de la valeur médiane (qui reste important en pratique), la valeur pivot peut être choisie au hasard. Cette approche est à l'origine du célèbre algorithme de tri QuickSort. Ce dernier reste en moyenne très efficace. En effet, il effectue quelques étapes supplémentaires (en fonction de sa malchance), mais en moyenne cette perte est rentabilisée par la disparition des calculs de valeurs médianes.

Exercice : Montrez que la méthode QuickSort est quadratique dans le pire cas.

1.3- La paire de points la plus proche (closest pair)

Nous proposons de résoudre un problème de géométrie classique. Soient n points répartis dans le plan. Nous voulons trouver dans cet ensemble les deux points les plus proches. Pour cela, nous pouvons construire un algorithme naïf de complexité quadratique :

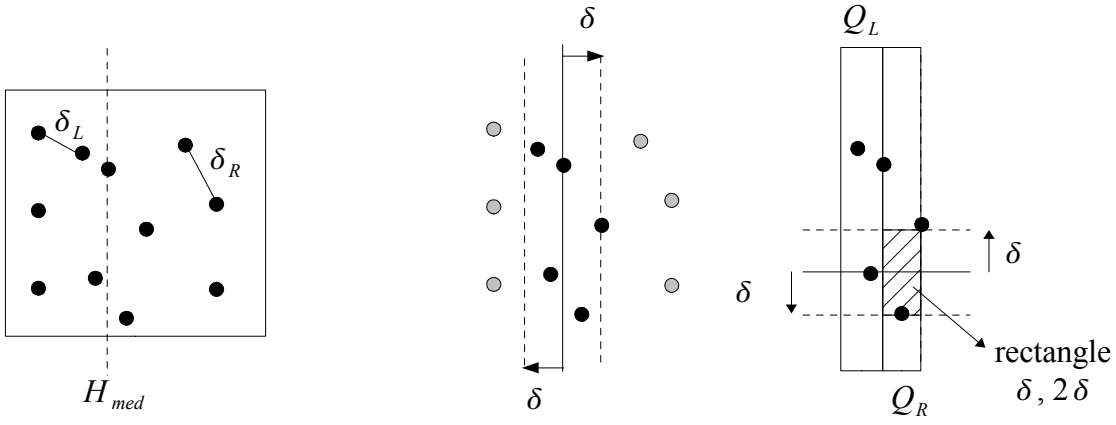
```
min =  +∞  ;
S =   P0 P1  ;
for (int i = 0 ; i<n ; i++)
   for (int j = i+1 ; j<n ; j++)
      t =  d(Pi, Pj)
      if ( t < min )
         { min = t; S =  Pi Pj  ;}
```



Cette méthode a longtemps été considérée comme la meilleure. Seulement plus tard, après l'apparition des techniques de réduction, une autre méthode basée sur une approche divide & conquer a permis d'atteindre une complexité optimale en $O(n \log n)$. En premier lieu, les points sont triés par ordonnées croissantes dans une première liste V_i et par abscisses croissantes dans une deuxième liste H_i . On sélectionne le point d'abscisse médian et on sépare l'ensemble des points courants en deux parties définies par la droite verticale passant par ce point. Le problème est relancé dans la partie gauche et droite et fournit deux réponses : δ_L et δ_R . Il reste à gérer pour fournir la distance minimale, la longueur des segments dont les extrémités se trouvent chacune d'un côté différent. Notons δ_{RL} la fonction qui traite ce cas et δ_m son résultat. La solution finale est le minimum entre ces trois valeurs : $min(\delta_R, \delta_L, \delta_{RL})$.

```
CP( Hi, Vi, n )
{
  si (n < 2) retourner  +∞
  si ( n == 2) retourner  d(V0, V1)
  med = n/2;                                  // divide O(n)
   VL = Select(Vi, Hmed.X)
   VR = Select(Vi, Hmed.X)
   δL  = CP( H0 ... Hmed−1, VL, med )         // conquer
   δR  = CP( Hmed ... Hn, VR, n−med )
   δ = min(δL, δR)                            // combine
   QL = Select(Vi, Hmed.X − δ, Hmed.X)
   QR = Select(Vi, Hmed.X , Hmed.X + δ)
   δm = δRL(QL, QR, δ)
  retourner  min(δ, δm)
}
```

L'étape divide consiste à répartir les points de la liste triée V_i dans deux sous-listes V_L et V_R correspondant à la partie gauche et droite. Ceci s'effectue par un simple parcours de la liste V_i et la complexité de cette partie est en $O(n)$. L'étape conquer est effectuée par deux appels récursifs. Pour l'étape combine, au lieu de prendre tous les segments manquant possibles, nous sélectionnons uniquement ceux présents dans une bande d'épaisseur $\delta = min(\delta_L, \delta_R)$ autour de l'axe vertical. En effet, si un sommet de la partie gauche se trouve hors de cette bande, quel que soit le sommet choisi dans la partie droite, le segment engendré aura forcément une longueur supérieure à δ_L et δ_R et sera par conséquent sans intérêt. Les deux listes obtenues : Q_L et Q_R sont utilisées par la fonction δ_{RL} . Nous supposons pour l'instant que cette fonction est linéaire. L'équation de complexité vérifie ainsi la propriété suivante $T(n) = O(n) + 2 \cdot T(n/2) + O(n)$, nous avons donc $T(n) = O(n \log n)$. Par conséquent, les deux tris utilisés au départ pour construire les deux listes V_i et H_i ne générant qu'une complexité en $O(n \log n)$, la classe de complexité de l'algorithme global est bien en $O(n \log n)$.



Il reste à exhiber la fonction δ_{RL} . Nous disposons de deux listes de points triés par ordonnées croissantes : Q_L et Q_R . Pour cela nous allons effectuer un parcours en parallèle comme dans l'exercice de l'épaisseur verticale. Nous obtenons sans difficulté particulière la fonction suivante :

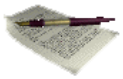
```
δRL(QL, QR, δ)
{
  j=0
   dmin = δ
  pour  i = 0  à  cardinal(QL) − 1
     tant que (  QR[ j ].Y < QL[i].Y − δ  )  j++;
     t = j
     tant que  QR[t].Y < QL[i].Y + δ
        dmin = min(dmin, d(QR[t], QL[i]))
        t++;
}
```

La méthode consiste à parcourir les points de la liste Q_L . Pour chaque point, nous cherchons uniquement des points dans Q_R qui ont une ordonnée à $\overline{+}\delta$ du point courant. Effectivement les autres points seraient inutiles car les segments générés auraient forcément une longueur supérieure à δ . Pour éviter de rechercher le premier point au dessus de $Q_L[i].Y - \delta$, nous conservons l'indice *j* du dernier point recherché. Ainsi comme Q_L et Q_R sont triées par ordonnées croissantes, le prochain point cherché sera un successeur de $Q_R[j]$. Nous évitons ainsi l'apparition d'un facteur en $O(n \log n)$ dû à une recherche. Cependant nous avons deux boucles imbriquées et jusqu'à nouvel ordre ces deux dernières impliquent une complexité en $O(n^2)$. A ce niveau nous devons constater une propriété particulière. En effet, pour tout point de Q_L nous sélectionnons dans Q_R les points se trouvant dans un rectangle de longueurs de cotés : $\delta, 2 \cdot \delta$. Dans un tel rectangle et d'après la définition de δ , il ne peut se trouver au plus que 6 points de Q_R . Par l'absurde, si ce n'était pas le cas, il y aurait forcément dans ce rectangle de H_R un segment de longueur strictement inférieure à δ ce qui contredirait la construction même de δ . En conséquence, la deuxième boucle imbriquée est bornée par 6. Ainsi la complexité de cette fonction est en $O(6 \cdot n) = O(n)$.

2- Approches gloutonnes (greedy methods)

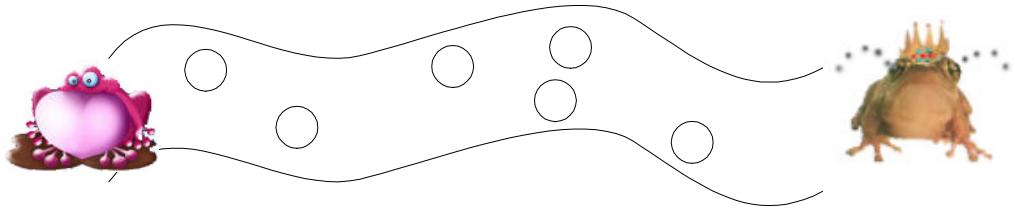
Il se peut parfois que la solution optimale à construire se fasse par insertion immédiate d'un élément de manière triviale. Les algorithmes qui en découlent sont extrêmement simples à mettre en oeuvre et très efficaces. Nous présentons ci-dessous deux exemples.

2.1- Monsieur Grenouille est amoureux



Le problème :

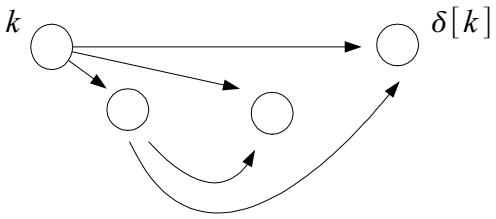
Monsieur Grenouille se trouve sur une pierre au milieu d'un cours d'eau. Sa belle et tendre promise se trouve assez loin de lui et il aimerait la rejoindre au plus vite. Monsieur Grenouille ne peut pas sauter sur la terre ferme de risque d'être dévoré tout cru par un prédateur. Il est contraint de sauter de pierre en pierre jusqu'à atteindre sa bien-aimée. Construisez un chemin optimal qui permettra à Monsieur Grenouille de rejoindre sa belle en un minimum de sauts.



Pour cela vous disposez d'un tableau $\delta[i]$ indiquant pour chaque pierre *i* l'indice de la pierre la plus loin qu'il peut atteindre de cet endroit. Nous supposons donc que si Monsieur Grenouille est sur la pierre k et qu'il peut rejoindre $\delta[k]$ il peut aussi décider de sauter moins loin sur $k+1,\ k+2\ \ldots \delta[k]-1$. Nous supposons aussi que s'il se trouve sur une pierre entre k et $\delta[k]$ alors Monsieur Grenouille peut aussi sauter sur $\delta[k]$, autrement dit :

$$\forall k, j, k<j<\delta[k] \text{ on a } \delta[j] \geqslant \delta[k] \qquad (*)$$

Soit graphiquement



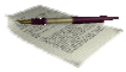
Solution : il suffit à Monsieur Grenouille de sauter à chaque fois au plus loin. Le chemin construit comporte un minimum de sauts. L'algorithme construit donc la suite : $1,\ \delta(1), \delta(\delta(1)), \ldots$. Cette solution triviale n'est pas forcément toujours évidente à exhiber lorsque le titre du chapitre « algorithmes gloutons » n'est pas donné.

Démonstration : Les démonstrations des méthodes gloutonnes sont courtes mais cependant non triviales. Elles sont fréquemment basées sur un développement par l'absurde. Essayez d'abord de construire la votre avant de lire la suite.

Supposons qu'il existe une solution *S* avec un saut de moins que la solution *C* construite par notre algorithme. Si tel est le cas, cela implique qu'il existe forcément un saut de *S* qui débute avant ou au même endroit qu'un saut de *C* mais finit strictement après. Cette condition est obligatoire pour réduire le chemin d'un saut. Ceci contredit la propriété (*) et le fait que Monsieur Grenouille a toujours sauté au plus loin.

Historique : si nous choisissons une ligne polygonale et un critère de simplification géométrique, nous sommes amenés à construire un graph ayant les mêmes propriétés que celui des sauts de Monsieur Grenouille. Cette méthode fut le point de départ de toutes les méthodes de simplification bidimensionnelle (Méthodes de Imai&Iri). Aujourd'hui largement employées en cartographie (simplification des frontières, des routes, des cours d'eau pour des affichages à échelle variable), ces méthodes se sont généralisées à la 3D où elles sont actuellement un domaine de recherche extrêmement actif.

2.2- Le sélecteur d'activité



Je dispose d'une machine capable d'effectuer des opérations. J'ai une liste d'actions devant être effectuées, mais pour des raisons extérieures, elles ne peuvent débuter qu'à certains moments s_i . Leurs durées étant aussi imposées, nous connaissons pour chaque action l'intervalle de temps pendant lequel elle doit être effectuée : $[s_i, f_i[$. Nous voulons réaliser un maximum d'opérations. Construisez un séquencement qui permet d'obtenir une liste de tâches indépendantes (dont les intervalles d'action ne se superposent pas). Les tâches sont triées par ordre de fin f_i croissant.

Solution :

j=1
$Sol = \{1\}$
pour i = 2 à n
si $f_j \leqslant s_i$ alors
$Sol = Sol \cup \{i\}$
j = i

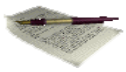
Nous sélectionnons la première tâche puis ensuite nous prenons la première tâche commençant après la fin de la tâche courante et finissant au plus tôt.

Démonstration : 

Il suffit de démontrer que la première tâche appartient à une solution optimale. Si c'est le cas, les tâches en collision sont retirées et nous nous ramenons au cas précédent en sélectionnant la tâche finissant au plus tôt c'est-à-dire (comme la liste est triée par ordre croissant) la première tâche de la nouvelle liste.

Par l'absurde, supposons qu'aucune solution optimale ne contient la première tâche. Prenons *S={i,j,k,l,...}* une de ces solutions. Comme la liste est triée par f_i croissant, la tâche 1 finit avant la tâche i, et par conséquent elle est compatible avec les tâches j,k,l.... Ainsi, si l'on remplace la tâche i dans *S* par la tâche 1, nous obtenons une nouvelle solution réalisable, ayant un nombre maximal d'opérations et contenant la tâche 1. Ceci contredit l'hypothèse de départ.

3- La programmation dynamique (dynamic programming)

Maximum de vraisemblance entre deux chaînes 

Le problème : Nous avons deux chaînes de caractères que nous mettons en correspondance face à face ceci en insérant éventuellement des blancs. Il est inutile de mettre deux blancs en vis à vis. Le score final est donné par la somme des scores des symboles mis deux à deux en correspondance. Nous avons :

- Deux lettres identiques : $\begin{matrix} A \\ A \end{matrix}$ Match : $+1$
- Deux lettres différentes : $\begin{matrix} A \\ C \end{matrix}$ Mismatch : -1
- Lettre en face d'un blanc : $\begin{matrix} A \\ \_ \end{matrix}$ Gap : -2

Sans méthode particulière, il faudrait évaluer toutes les superpositions possibles. Par exemple, pour deux mots de n lettres, s'il y a k vides en haut, il y en aura autant en bas et cela fait pour ce cas : $C_{n+k}^{k} \cdot C_{n}^{k}$ possibilités. Nous nous proposons de résoudre ce problème par une approche de type programmation dynamique qui nous permettra d'obtenir une complexité bien plus faible que la méthode exhaustive ci-dessus.

La méthode proposée se développe en $\boxed{6}$ étapes :

1- Expression de la fonction à évaluer
2- Ecriture de l'optimisation à effectuer
3- Partitionnement de l'ensemble des possibles
4- Définition des sous-problèmes (prouvez qu'ils préservent l'optimum)
5- Expression des dépendances
6- Construction de l'algorithme (initialisation et remplissage des données)

3.1- Expression de la fonction à évaluer

Soit $\begin{pmatrix} M' \\ P' \end{pmatrix}$ la mise en correspondance de deux mots M et P de longueurs m et p. Exprimons le score de la vraisemblance :

$$\begin{array}{lccccccc} M': & A & C & \_ & A & A & B & \_ \\ P': & A & T & A & \_ & \_ & B & B \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \delta: & 1 & -1 & -2 & -2 & -2 & 1 & -2 \quad = -7 \end{array}$$

Nous avons donc :

$$\delta\begin{pmatrix} M' \\ P' \end{pmatrix} = \delta\begin{pmatrix} A \\ A \end{pmatrix} + \delta\begin{pmatrix} C \\ T \end{pmatrix} + \ldots + \delta\begin{pmatrix} \_ \\ B \end{pmatrix}$$

Ainsi

$$\delta\begin{pmatrix} M' \\ P' \end{pmatrix} = \sum_{i=1}^{l} \delta\begin{pmatrix} M'[i] \\ P'[i] \end{pmatrix}$$

avec l longueur de la correspondance et $\delta\begin{pmatrix} X \\ X \end{pmatrix} = 1$, $\delta\begin{pmatrix} X \\ Y \end{pmatrix} = -1$ et $\delta\begin{pmatrix} X \\ \_ \end{pmatrix} = -2$.

3.2- Ecriture de l'optimisation à effectuer

Appelons Ω , l'ensemble des mises en correspondance du mot M (de longueur m) et

du mot P (de longueur p). Nous cherchons :

$$\underset{(M',P')\in\Omega}{Max}\ \delta\begin{pmatrix}M'\\P'\end{pmatrix}$$

3.3- Partitionnement de l'ensemble des possibles

Il s'agit de l'étape critique. Il faut intuitivement repérer une bonne partition de Ω permettant de redécouper notre problème en sous-problèmes similaires au problème initial. Nous proposons :

Ω_X^X : Ensemble des superpositions finissant par deux lettres.
$\Omega_{\bar{X}}^{\bar{}}$: Ensemble des superpositions finissant par un blanc en haut et une lettre en bas.
$\Omega_{\bar{}}^X$: Ensemble des superpositions finissant par un blanc en bas et une lettre en haut.

Nous avons bien construit une partition de Ω :

$$\Omega=\Omega_X^X\cup\Omega_{\bar{}}^X\cup\Omega_{\bar{X}}^{\bar{}}\ ,\ \Omega_X^X\cap\Omega_{\bar{}}^X=\emptyset\ ,\ \Omega_X^X\cap\Omega_{\bar{X}}^{\bar{}}=\emptyset\ \text{ et }\ \Omega_{\bar{X}}^{\bar{}}\cap\Omega_{\bar{}}^X=\emptyset$$

3.4- Définition des sous-problèmes (prouvez la préservation de l'optimum)

$$\underset{(M',P')\in\Omega}{Max}\ \delta\begin{pmatrix}M'\\P'\end{pmatrix}\ =\ \underset{(M',P')\in\Omega_X^X\cup\Omega_{\bar{}}^X\cup\Omega_{\bar{X}}^{\bar{}}}{Max}\ \delta\begin{pmatrix}M'\\P'\end{pmatrix}$$

$$=Max\left(\underset{(M',P')\in\Omega_X^X}{Max}\ \delta\begin{pmatrix}M'\\P'\end{pmatrix},\ \underset{(M',P')\in\Omega_{\bar{}}^X}{Max}\ \delta\begin{pmatrix}M'\\P'\end{pmatrix},\ \underset{(M',P')\in\Omega_{\bar{X}}^{\bar{}}}{Max}\ \delta\begin{pmatrix}M'\\P'\end{pmatrix}\right)\ (*)$$

or

$$\underset{(M',P')\in\Omega_X^X}{Max}\ \delta\begin{pmatrix}M'\\P'\end{pmatrix}\ =\ \underset{(M',P')\in\Omega_X^X}{Max}\ \delta\begin{pmatrix}M'[1]\dots M'[l-1]\\P'[1]\dots P'[l-1]\end{pmatrix}+\delta\begin{pmatrix}M'[l]\\P'[l]\end{pmatrix}$$

$$=\ \underset{(M',P')\in\Omega_X^X}{Max}\ \delta\begin{pmatrix}M'[1]\dots M'[l-1]\\P'[1]\dots P'[l-1]\end{pmatrix}+\begin{cases}1 \text{ si } M[m]=P[p]\\-1 \text{ si } M[m]\neq P[p]\end{cases}$$

$\underset{(M',P')\in\Omega_X^X}{Max}\ \delta\begin{pmatrix}M'[1]\dots M'[l-1]\\P'[1]\dots P'[l-1]\end{pmatrix}$ correspond au score maximal obtenu par la superposition des $m-1$ premières lettres de M et les $p-1$ premières lettres de P.

Nous pouvons ainsi donner la définition des sous-problèmes :
T[i,j] : Maximum de vraisemblance issu des superpositions des i premières lettres de M et des j premières lettres de P.

L'écriture (*) nous garantit que les résultats des sous-problèmes sont suffisants pour obtenir

la solution optimale du problème courant.

3.5- Expression des dépendances

Nous avons simplement :

$$T[i,j]=Max\begin{cases} T[i-1,j-1]+\delta_{P[j]}^{M[i]} \\ T[i-1,j]+\delta_{GAP} \\ T[i,j-1]+\delta_{GAP} \end{cases}$$

3.6- Construction de l'algorithme (initialisation et remplissage des données)

On remplit la première ligne et la première colonne du tableau bidimensionnel. Le reste du tableau se construit de gauche à droite et de haut en bas. Pour simplifier le calcul de la première ligne, il suffit de considérer le cas où l'on confronte i lettres avec une chaîne vide. Ainsi $T[i,0]=\delta_{GAP}\cdot i$. Nous avons de la même manière $T[0,j]=\delta_{GAP}\cdot j$.

Nous pouvons maintenant écrire l'algorithme :

for ($i=0$; $i\leqslant m$; i++) T[i,0] = $\delta_{GAP}\cdot i$
for ($i=0$; $i\leqslant p$; i++) T[0,j] = $\delta_{GAP}\cdot i$

for($i=1$; $i\leqslant m$; i++)
  for($j=1$; $j\leqslant p$; j++)
  {
    if ($M[i]==P[j]$) $\delta=\delta_{Match}$ else $\delta=\delta_{Mismatch}$
    T[i][j] = $max(T[i-1,j-1]+\delta, T[i-1,j]+\delta_{Gap}, T[i,j-1]+\delta_{Gap})$;
  }

Remarque 1 : Pourquoi part-on à l'envers pour finalement remplir le tableau à l'endroit. Dans tous les cas, le tableau se construit de la même manière. Seulement si l'on opère avec :

Ω_X^X ensemble des superpositions qui commencent par deux lettres

Il faut ensuite penser à l'envers car nous définissions T[i,j] comme la superposition des i et j dernières lettres des mots M et P . Donc, la question consiste à savoir à quel moment vous préférez penser à l'envers.

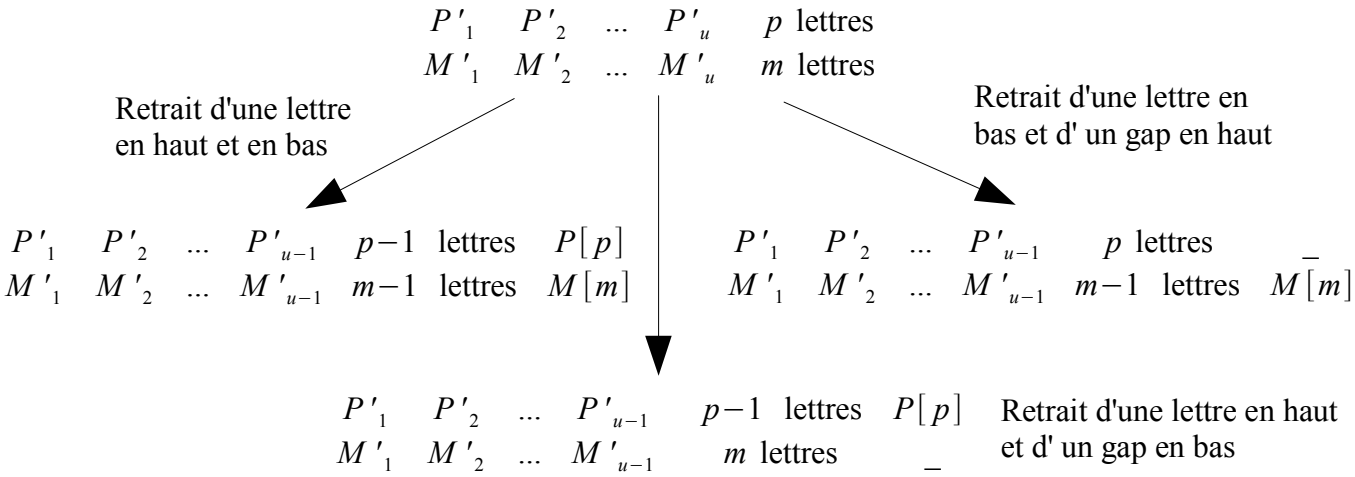
Remarque 2 : Que sont devenues les différentes configurations des superpositions ? Toujours d'après (*) il est inutile de connaître la meilleure superposition courante pour déterminer le meilleur score final. De plus il n'y a pas forcément une unique superposition optimale. Cependant, si besoin est, nous pouvons en reconstruire une en marquant le chemin qui a conduit à la valeur stockée dans T[i,j].

Remarque 3 : Si les deux mots commencent par la même lettre, en est-il de même pour la superposition optimale ? Réponse : oui et non. Cela dépend du chemin mis en mémoire (il peut y avoir des chemins équivalents). Ainsi avec AAB et AB, nous avons :

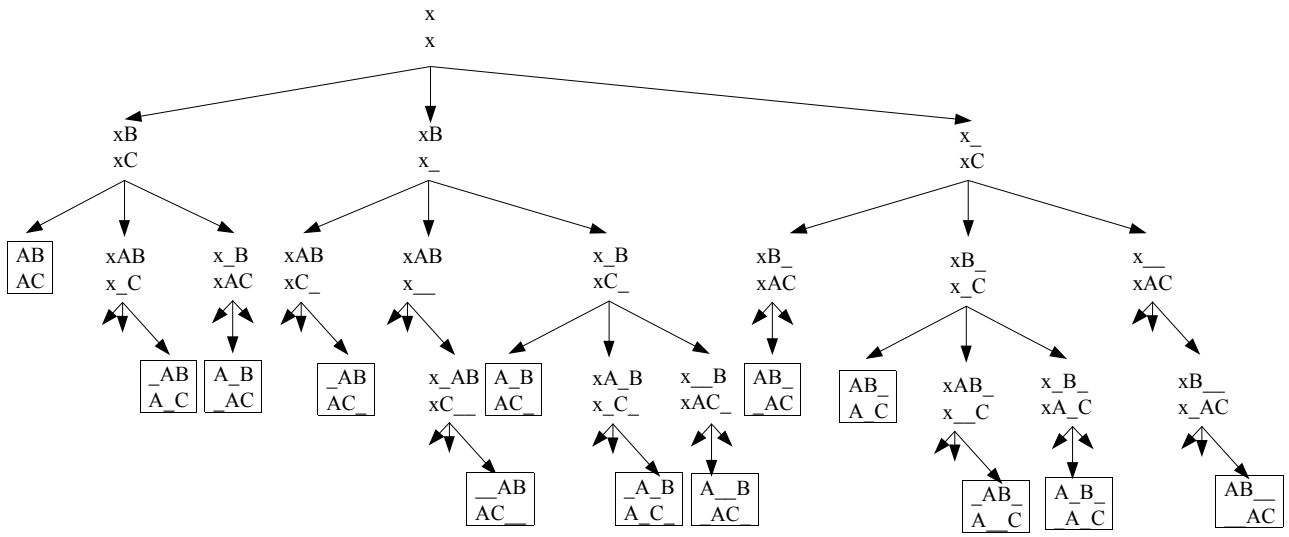
$$\delta\begin{pmatrix} A & A & B \\ A & \_ & B \end{pmatrix}=\delta\begin{pmatrix} A & A & B \\ \_ & \text{A} & B \end{pmatrix}$$

Exemple : M = AB P= AC

D'après $\Omega = \Omega_X^X \cup \Omega_{\_}^X \cup \Omega_{\bar{X}}$:



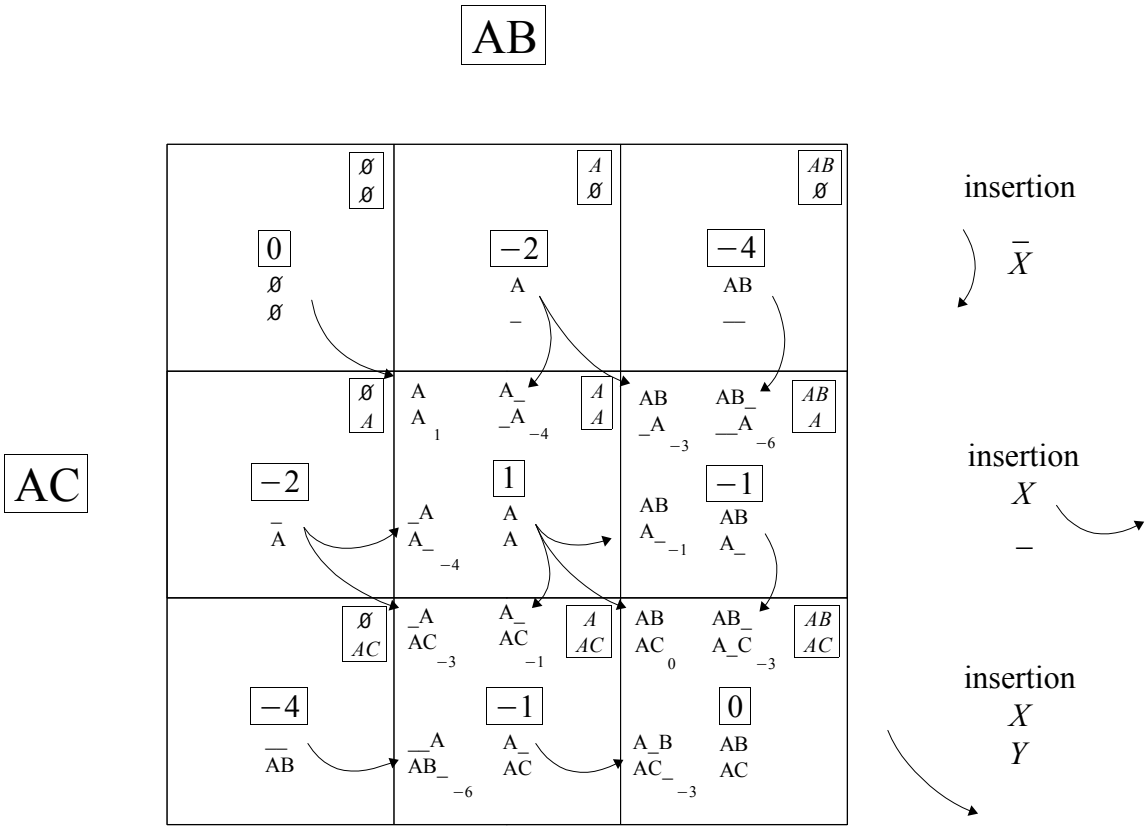
Nous obtenons bien à partir de cette répartition la reconstruction de l'ensemble Ω .



Vérifions qu'aucune d'entre elles n'a été oubliée :

- aucun gap, que des matchs : 1 seule possibilité AB / AC
- un gap en haut (donc un gap en bas) :
  $A_3^3 = 6$ possibilités : _AB/A_C A_B/_AC _AB/AC_ A_B/AC_ AB_/_AC AB_/A_C
- deux gaps en haut (donc deux gaps en bas) :
  $C_4^2 \cdot C_2^2 = 6$ possibilités : __AB/AC__ _A_B/A_C_ A__B/_AC_ _AB_/A__C A_B_/_A_C AB__/__AC

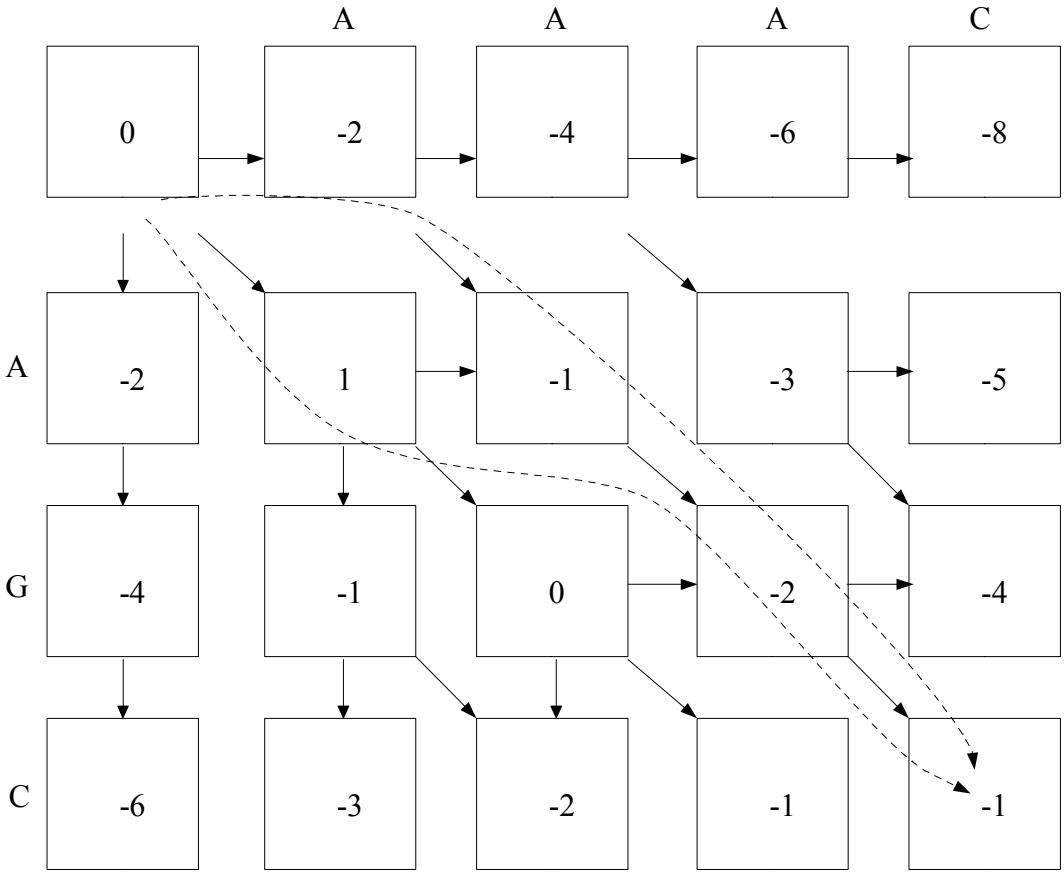
Déroulement de l'algorithme :



Les meilleures configurations locales sont comparées et seule la meilleure est retenue. Cependant, le fait que certaines configurations disparaissent à jamais peut parfois gêner les esprits, même si d'après (*) cela est mathématiquement justifié. En effet, ceci restreint fortement l'ensemble des combinaisons (voir arbre de la page précédente).

Prenons la case : $\boxed{\begin{matrix}AB\\A\end{matrix}}$, la superposition $\begin{matrix}AB\\A\_\end{matrix}$ l'emporte sur $\begin{matrix}AB\\\_A\end{matrix}$ et sur $\begin{matrix}AB\_\\\_\_A\end{matrix}$. Cela n'implique pas que la superposition $\begin{matrix}AB\\A\_\end{matrix}$ sera retenue pour la solution optimale, cela implique que deux configurations sont écartées et qu'elles n'apparaîtront jamais dans la solution optimale. Comment pouvons-nous, sans savoir ce qui se passe après, retirer dès maintenant des possibilités qui éventuellement auraient pu s'avérer meilleures au final ? Raisonnons par l'absurde, supposons que la solution optimale s'écrive : $\begin{matrix}AB\$\\A\_X\end{matrix}$. Gardons la partie $\begin{matrix}\$\\X\end{matrix}$ et insérons lui en début notre meilleure superposition locale : $\begin{matrix}AB\\A\_\end{matrix}$. Alors $\begin{matrix}AB\$\\A\_X\end{matrix}$ est une superposition valide et son score est strictement supérieur à l'optimum. Ceci contredit le fait que $\begin{matrix}AB\\A\_\end{matrix}$ puisse réapparaître dans la solution optimale.

Test sur : AAAC et AGC



Voici le score maximal de la superposition de AAAC et AGC. Pour reconstruire une superposition optimale, il suffit de remonter les différents chemins possibles. Par exemple :

$\begin{matrix}\text{AAAC}\\ \text{\_AGC}\end{matrix}$ pour le chemin du haut et $\begin{matrix}\text{AAAC}\\ \text{A\_GC}\end{matrix}$ pour celui du bas.

Je tenais à remercier les élèves de mon groupe pour leur présence, leur attention, leur participation et leur bonne humeur tout au long de cet enseignement. J'espère que cette version relookée du module IN311 leur a plu. J'espère qu'ils apprécieront ce polycopié et qu'ils pardonneront les erreurs qui s'y trouvent (ps: si vous pouviez me faire remonter les coquilles que vous trouvez)...