FSAB 1402: Informatique 2
Complexité Calculatoire

Pierre Dupont et Peter Van Roy

Département d'Ingénierie Informatique, UCL

Pierre.Dupont@uclouvain.be

Peter.Vanroy@uclouvain.be

Plan

- Une brève introduction aux tuples
- Comment caractériser l'efficacité d'un programme ?
  - Approche expérimentale : temps d'exécution
  - Approche théorique : complexité temporelle
- Outils mathématiques : notations O, Ω et Θ (équations de récurrence)
  - Complexité spatiale
  - Complexité en moyenne
- Quelques réflections sur la performance
  - La loi de Moore
  - Les problèmes NP-complets
  - L'optimisation

Lecture pour le quatrième cours

- Transparents sur le site Web du cours
- Dans le livre
  - Chapitre 1 (section 1.7)
    - La complexité calculatoire
  - Chapitre 3 (section 3.6)
    - L'efficacité en temps et en espace

Tuples

Tuples

Un **tuple** : une collection *séquentielle* de *taille fixe* avec *accès rapide* à chaque élément

declare

X=montuple(1 3 5 7 25)

{Browse {Width X}}

{Browse {Label X}}

{Browse X.3}

Temps d'exécution et espace mémoire

Approche expérimentale

Comment caractériser l’efficacité d’un programme?

- Le ***temps*** que met le programme à produire un résultat
  $\Rightarrow$ lien avec la ***complexité temporelle*** de l’algorithme
- L’***espace*** utilisé (mémoire, espace disque) par le programme
  $\Rightarrow$ lien avec la ***complexité spatiale*** de l’algorithme

Quels sont les facteurs influençant le temps (ou l’espace)?

Un facteur prépondérant : les données du problème

```
% Input: T un tuple de n entiers (n>0)
% Output: la valeur maximale dans T
fun {TupleMax T}
   N={Width T}
   fun {Loop I CurrentMax}
      if I=<N then
         {Loop I+1
          if CurrentMax<T.I then T.I else CurrentMax end}
      else
         CurrentMax
      end
   end
in
   {Loop 2 T.1}
end
```

- La ***taille*** du problème (ici, la taille *n* du tuple)
- Les ***valeurs spécifiques*** définissant une instance particulière du problème (ici, les valeurs mémorisées dans le tuple)
- La taille du problème **est** le nombre de valeurs à spécifier pour définir une instance particulière du problème

Meilleur cas, pire cas et cas moyen

- Il y a souvent un nombre infini d'***instances possibles*** (ici, toutes les valeurs possibles d'un tuple de taille *n* contenant des entiers)
- Selon l'instance particulière considérée, un algorithme ***peut*** prendre plus ou moins de temps
- Les instances possibles peuvent alors être classées en ***meilleur(s)*** cas, ***pire(s)*** cas ou cas ***moyens***
- Nous nous intéressons généralement au temps pris dans le **pire cas** car
  - Nous voulons une borne supérieure du temps d'exécution
  - Le meilleur cas donne lieu à une estimation optimiste
  - Un cas représentatif "moyen" est souvent difficile à définir

Les facteurs influençant le temps d'exécution

- Les ***données*** du problème (l'instance particulière : taille + valeurs)
- L'***algorithme*** utilisé pour résoudre le problème

mais aussi …

- Le ***matériel*** (vitesse du processeur, taille et vitesse d'accès à la mémoire, temps de transfert disque, *etc*)
- Le ***logiciel*** (langage de programmation, compilateur/interpréteur, *etc*)
- La ***charge de la machine*** (nombre de processus qui s'exécutent, *etc*)
- Le ***système d'exploitation*** (gestion des différents processus, *etc*)
- La ***charge du réseau*** (accès aux données, écriture des résultats, *etc*)
- *Etc*

Mesure expérimentale du temps d'exécution

- Écrire un ***programme*** implémentant l'algorithme à étudier
- Exécuter le programme pour ***différentes instances*** du problème (taille + valeurs spécifiques)
- Utiliser une méthode comme ***System.currentTimeMillis()*** (en Java) ou la fonction ***OS.time*** (en Oz) pour mesurer le temps effectif d'exécution

Limitations de l'approche expérimentale

- Nécessité d'***implémenter*** les différents algorithmes que l'on veut comparer
- ***Nombre limité*** (et forcément fini) d'***instances testées***
- Ces instances ne sont ***pas forcément représentatives*** de tous les cas
- Outre l'algorithme et les instances testées, tous les ***autres facteurs*** (logiciel, matériel, …) influencent la mesure du temps d'exécution

Complexités temporelle et spatiale

Analyse asymptotique

Analyse asymptotique

- Objectif: analyser le temps (ou l'espace) en se concentrant sur l'algorithme et l'influence de la taille du problème, généralement dans le pire cas
- Complexité temporelle = analyse ***asymptotique*** du nombre d'opérations effectuées
- Complexité spatiale = analyse ***asymptotique*** de l'espace utilisé

- *L'analyse asymptotique s'intéresse à l'évolution de la complexité lorsque la taille du problème augmente (i.e. tend vers l'infini)*

La vraie question: comment évolue le temps d'exécution en fonction de la taille du problème?

- ***Par exemple***, si la taille *n* du problème est multipliée par ***10*** comment évolue le temps $T = f(n)$?
- Si $f(n) = c \Rightarrow f(10n) = c$ — T est inchangé
- Si $f(n) = c.n \Rightarrow f(10n) = c.(10n) = 10f(n)$ — T x 10
- Si $f(n) = c.n^2 \Rightarrow f(10n) = c.(10n)^2 = 100f(n)$ — T x 100

- La vitesse du processeur est ***un*** des facteurs qui conditionnent la valeur de la constante c. La vitesse du processeur ***ne*** change ***rien*** au rapport f(10n)/f(n).
- Une ***constante*** est donc tout ce qui ne dépend pas de la taille du problème
- Si l'on s'intéresse à l'influence de la taille du problème, on peut donc ***négliger*** les constantes, c'est-à-dire ignorer l'influence de tous les facteurs constants (processeur, langage de programmation, compilateur, *etc*)

Opérations primitives

Une ***opération primitive***

- est une instruction en ***langage de haut niveau*** (par exemple Java ou Oz ou une description en pseudo-code)
- représente un ***nombre constant d'opérations élémentaires*** effectivement exécutées sur le processeur une fois le programme compilé ou interprété dans un environnement donné
- est une opération du langage noyau, comme par exemple:
  - une affectation d'une valeur à une variable
  - la comparaison de deux nombres
  - une opération arithmétique élémentaire (p.ex. addition de deux entiers petits)
  - un accès à un élément d'un tableau ou d'un tuple
  - le renvoi d'une valeur par une fonction
  - une instruction d'appel d'une fonction (≠ l'exécution de l'ensemble de la fonction!)

Comme les constantes disparaissent dans l'analyse asymptotique, il suffit de compter les *opérations primitives* plutôt que les *opérations élémentaires*.

Pourquoi se soucier du temps d'exécution en pratique? (1)

- Hypothèse: on peut traiter **une** opération primitive en 1 μs
- f(n) désigne le nombre d'opérations primitives effectuées en fonction de la taille du problème
- **Combien de temps** prend le programme pour terminer son exécution si n=1000 selon f(n)?

f(n)	Temps
n	1 ms
$400n$	0.4 s
$2n^2$	2 s
n^4	~11.5 jours
2^n	3.4 x 10^{287} années!!

Pourquoi se soucier du temps d'exécution en pratique? (2)

- Quelle est la taille **maximale** du problème que l'on peut traiter?

$f(n)$	En 1 seconde	En 1 minute	En 1 heure
n	1×10^6	6×10^7	3.6×10^9
$400n$	2500	150 000	9×10^6
$2n^2$	707	5477	42426
n^4	31	88	244
2^n	19	25	31

- Si m est la taille maximale que l'on pouvait traiter en un temps donné, que devient m si l'on reçoit de notre sponsor favori un processeur **256** fois plus rapide?

$f(n)$	Nouvelle taille maximale
n	$256m$
$400n$	$256m$
$2n^2$	$16m$
n^4	$4m$
2^n	$m+8$

Calcul du nombre d'opérations primitives

```
% Input: T un tuple de n entiers (n>0)
% Output: la valeur maximale dans T
fun {TupleMax T}                                                  1
   N={Width T}                                                    2
   fun {Loop I CurrentMax}                               m (appels)
      if I=<N then                                               2m
         {Loop I+1
          if CurrentMax<T.I then T.I else CurrentMax end}  4m ou 5m
      else
         CurrentMax
                                                                  m
      end
   end                                                  m (retours)
in
   {Loop 2 T.1}                                                   2
end                                                               1
```

- Note:
  - **if** I=<N **then** $\Rightarrow$ 2 opérations primitives (comparaison, branchement)
  - **If** CurrentMax<T.I **then** $\Rightarrow$ 3 opérations primitives (accès, comparaison, branchement)
- Dans le pire cas, on exécute *10(n-1)+6=10n-4* opérations primitives (avec *m=n-1*)
- *Ce calcul introduit de nouvelles constantes (p.ex. 10) que l'on peut négliger pour les mêmes raisons que précédemment!*

Notations *O*, Ω et Θ

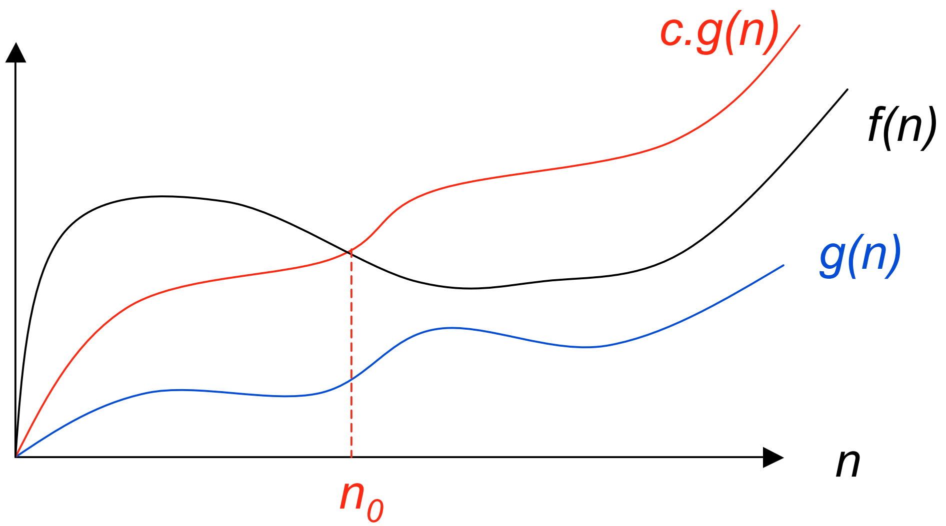
Bornes de complexité

Un outil mathématique: la notation *O*

- Une mesure de l'"*ordre de grandeur*" d'une fonction *f(n)* :

  trouver une fonction *g(n)* qui constitue une ***borne supérieure*** de *f(n)* à une constante multiplicative *c* près et pour autant que *n* soit *suffisamment grand*

$$f(n) \in O(g(n)) \quad \textbf{\textit{si}} \quad \exists c > 0, \exists n_0 \geq 1 \text{ tels que } f(n) \leq c.g(n), \forall n \geq n_0$$



Utilisation de la notation O

- $2n+10 \in O(n)$ car $2n+10 \leq 4.n$ pour $n \geq 5$
- $2n+10 \in O(n^2)$ car $2n+10 \leq 1.n^2$ pour $n \geq 5$
- $2^{100} \in O(1)$ car $2^{100} \leq 2^{100}.1$ pour $n \geq 1$
- $3n^2 + 10n \log_{10} n + 125n + 100 \in O(n^2)$

  car $3n^2 + 10n \log_{10} n + 125n + 100 \leq 4.n^2$ pour $n \geq 148$

- On s'intéresse à la borne ***la plus stricte possible*** $\Rightarrow 2n+10 \in O(n)$
- Il suffit de ***garder les termes dominants*** et ***supprimer les constantes***

Retour à notre exemple

```
% Input: T un tuple de n entiers (n>0)
% Output: la valeur maximale dans T
fun {TupleMax T}
   N={Width T}                                              O(1)
   fun {Loop I CurrentMax}                O(n) appels (et retours)
      if I=<N then                                          O(1)
         {Loop I+1
          if CurrentMax<T.I then T.I else CurrentMax end}   O(1)
      else
         CurrentMax                                         O(1)
      end
   end
in
   {Loop 2 T.1}                                             O(1)
end                                                         O(1)
```

- La complexité temporelle de l'algorithme est globalement *O(n)*

Notations Ω, Θ

- Ω désigne une ***borne inférieure***:

$$\boxed{f(n) \in \Omega(g(n)) \;\; \textbf{\textit{si}} \;\; g(n) \in O(f(n))}$$

Par exemple, $n^3 \in \Omega(n^2)$ car $n^2 \in O(n^3)$

- Θ désigne une fonction ***asymptotiquement équivalente***:

$$\boxed{f(n) \in \Theta(g(n)) \;\; \textbf{\textit{si}} \;\; f(n) \in O(g(n)) \;\; \textbf{\textit{et}} \;\; f(n) \in \Omega(g(n))}$$

Par exemple, $400n\text{-}3 \in \Theta(n)$

Pourquoi distinguer *O* et Θ?

```
% Input: T un tuple de n entiers (n>0)
% Output: l'indice du premier entier négatif dans T (renvoie -1 si aucun entier négatif)
fun {TupleFirstNegative T}
   N={Width T}
   fun {Loop I}
      if I>N then ~1 elseif T.I<0 then I
      else {Loop I+1} end
   end
in
   {Loop 1}
end
```

- La complexité temporelle de TupleFirstNegative est $O(n)$
- Sa complexité temporelle dans le *meilleur* cas est $\Theta(1)$
- Sa complexité temporelle dans le *pire* cas est $\Theta(n)$
- *La complexité dans le meilleur cas n'est pas toujours inférieure à la complexité en géneral.* Par exemple, la complexité dans tous les cas de TupleMax est $\Theta(n)$.

Complexité en moyenne

- Le problème est qu'il est difficile de définir un cas moyen
- Nécessité de connaître la ***distribution de probabilités*** des cas
- La ***complexité en moyenne*** est souvent équivalente à la complexité dans le pire cas

```
fun {TupleFirstNegative T}
   N={Width T}
   fun {Loop I}
      if I>N then ~1 elseif T.I<0 then I
      else {Loop I+1} end
   end
in
   {Loop 1}
end
```

- ***Sous l'hypothèse*** que l'indice du premier entier négatif suit une ***distribution uniforme***, sa valeur est en moyenne $n/2$
- La complexité en moyenne de TupleFirstNegative est $\Theta(n/2) = \Theta(n)$

Complexité spatiale (1)

- ***Raisonnement analogue*** à celui utilisé pour la complexité temporelle
- On s'intéresse ici à l'espace utilisé: ***la consommation de mémoire***
- On s'intéresse aux ***termes dominants*** (analyse asymptotique)

```
% Input: T un tuple de n entiers (n>0)                      Θ(n)
% Output: l'indice du premier entier négatif dans T (renvoie -1 si aucun entier négatif)
fun {TupleFirstNegative T}
   N={Width T}                                              Θ(1)
   fun {Loop I}                                             Θ(1)
      if I>N then ~1 elseif T.I<0 then I
      else {Loop I+1} end                                   Θ(1)  (récursion terminale !!)
   end
in
   {Loop 1}
end
```

- La complexité spatiale de TupleFirstNegative est $\Theta(1)$
  - Parce que le tuple existe déjà à l'entrée, la consommation est donc une constante

Complexité spatiale (2)

- Dans l'utilisation mémoire d'un programme, il y a deux concepts
  - La ***taille instantanée de mémoire active*** $m_a(t)$, en mots
  - La ***consommation instantanée de mémoire*** $m_c(t)$, en mots par seconde
- Il ne faut pas confondre ces deux nombres!
  - Une base de données en mémoire vive: une grande taille instantanée avec une petite consommation
  - Une simulation: une petite taille instantanée avec une grande consommation

Estimation expérimentale de la complexité asymptotique

- Sélection (délicate…) d'***instances représentatives*** du pire cas
- ***Répétition*** de la mesure de temps pour chaque instance (calcul d'un temps moyen pour lisser l'influence des autres facteurs)
- ***Répétition*** de la mesure de temps pour plusieurs instances de la même taille
- Mesure du temps pour des ***valeurs croissantes*** de la ***taille*** des instances

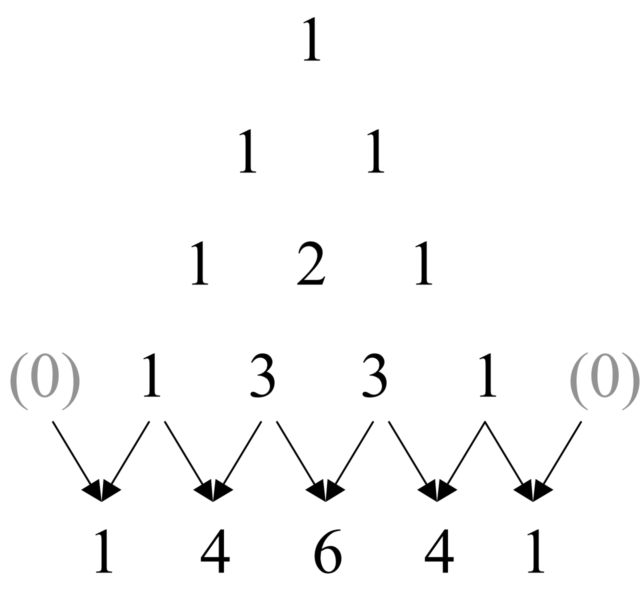
- *Peu importe la valeur absolue du temps. Pour rappel, la question centrale est: comment évolue le temps lorsque la taille du problème augmente?*

Un exemple d'analyse de complexité

Le triangle de Pascal

Le triangle de Pascal (1)

- Nous allons définir la fonction {Pascal N}
- Cette fonction prend un entier N et donne la Nième rangée du triangle de Pascal, représentée comme une liste d’entiers
- Une définition classique est que {Pascal N} est la liste des coefficients dans l’expansion de $(a+b)^n$



Le triangle de Pascal (2)

- Calculez la fonction {Pascal N}

1. Pour la rangée 1, cela donne [1]
2. Pour la rangée N, déplacez à gauche la rangée N-1 et déplacez à droite la rangée N-1
3. Alignez les deux rangées déplacées et additionnez-les élément par élément pour obtenir la rangée N

1

1 1

1 2 1

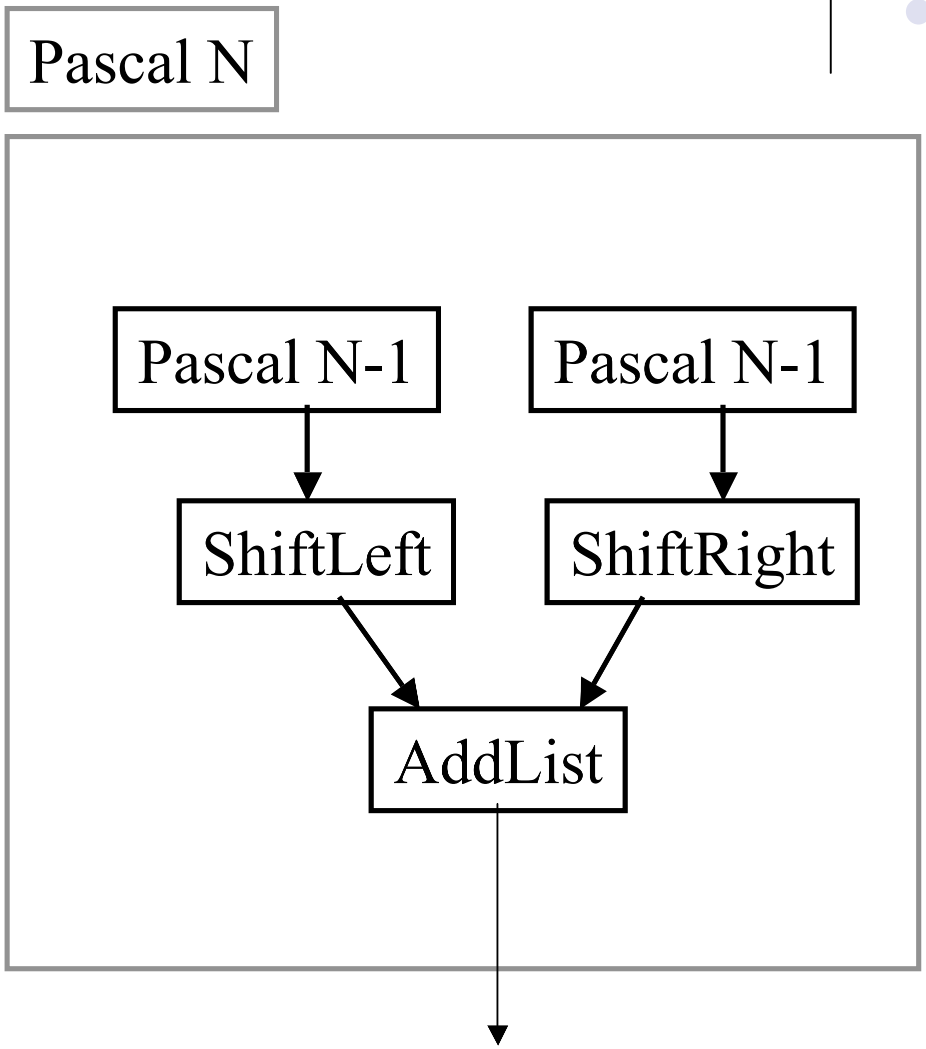
(0) 1 3 3 1 (0)

1 4 6 4 1

Déplacez à droite: [0 1 3 3 1]

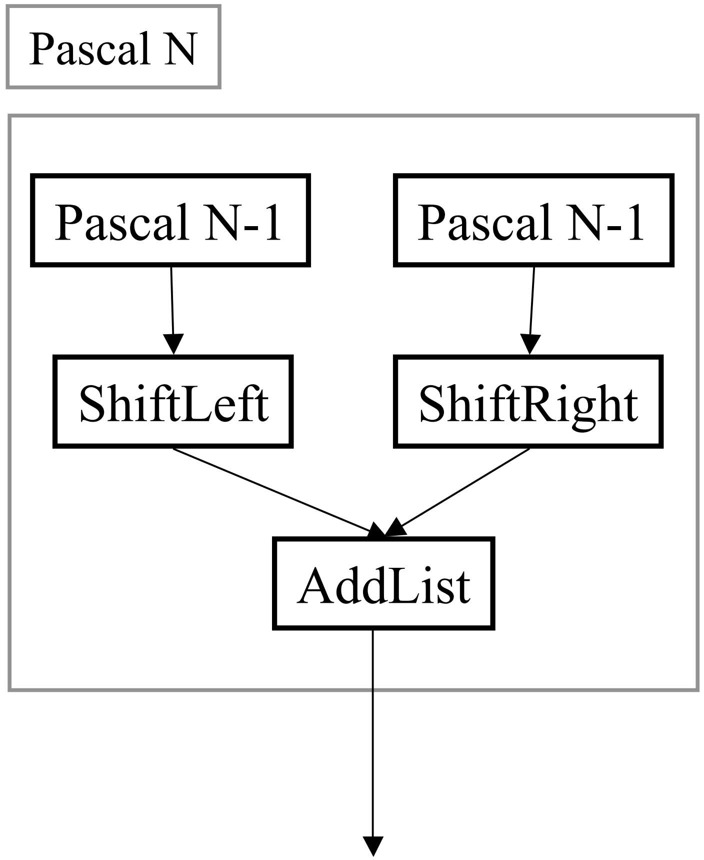
Déplacez à gauche: [1 3 3 1 0]

Schéma de la fonction



Code de la fonction

```
declare
fun {Pascal N}
   if N==1 then [1]
   else
      {AddList
        {ShiftLeft
            {Pascal N-1}}
        {ShiftRight
            {Pascal N-1}}}
   end
end
```



Fonctions auxiliaires (1)

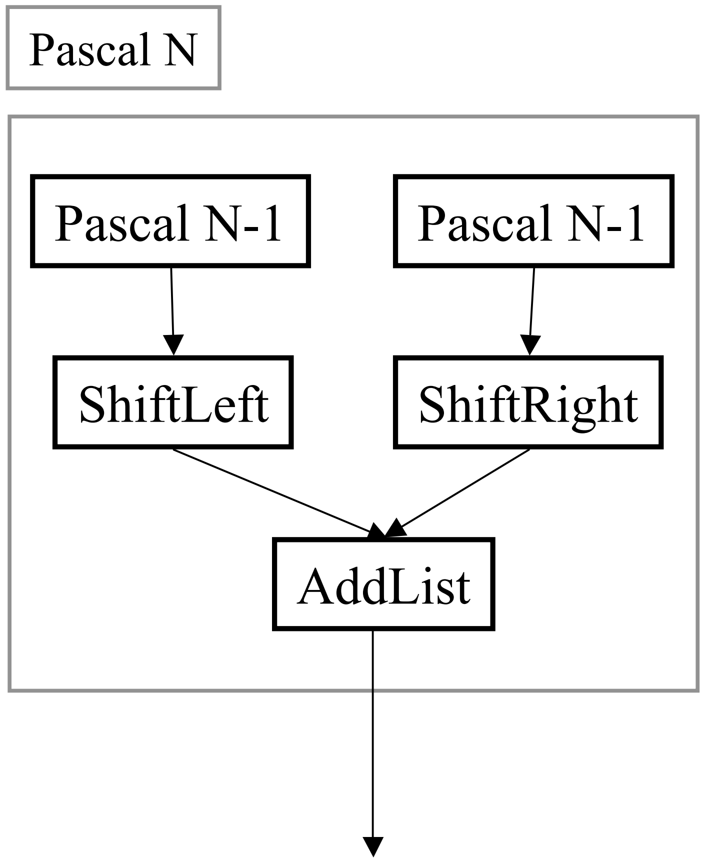
```
fun {ShiftLeft L}
   case L of H|T then
      H|{ShiftLeft T}
   else [0]
   end
end

fun {ShiftRight L}  0|L end
```

Fonctions auxiliaires (2)

```
fun {AddList L1 L2}
   case L1 of H1|T1 then
      case L2 of H2|T2 then
                 H1+H2|{AddList T1 T2}
      end
   else nil end
end
```

Complexité temporelle de Pascal

```
declare
fun {Pascal N}
  if N==1 then [1]
  else
    {AddList
      {ShiftLeft
          {Pascal N-1}}
      {ShiftRight
          {Pascal N-1}}}
  end
end
```



Analyse simplifiée

- {Pascal N}

  fait **2** appels à {Pascal N-1},

  qui font **4** appels à {Pascal N-2},

  …,

  qui font $\mathbf{2^{(N-1)}}$ appels à {Pascal 1}.

- La complexité temporelle est donc au moins :

$1+2+2^2+\ldots+2^{(N-1)} \in \Theta(2^N)$

Équations de récurrence

$f(n)$ désigne le nombre d'opérations primitives effectuées par {Pascal N}

- $f(n) = c_1$, *si* $n=1$ $\qquad$ ($c_1, c_2 \in$ $O(1)$)
- $f(n) = c_2.n + 2\ f(n-1)$, *sinon*

$\Rightarrow f(n) \in \Theta(2^n)$ [voir section 3.6 du livre]

FastPascal

- On peut faire un seul appel récursif si on garde le résultat dans une variable locale L

```
fun {FastPascal N}
   if N==1 then [1]
   else L in
      L={FastPascal N-1}
      {AddList {ShiftLeft L} {ShiftRight L}}
   end
end
```

- La complexité devient $O(n+(n-1)+\ldots+1) = O(n^2)$

Quelques informations supplémentaires

- Tableau 3.4 (p.151):
  - Quelques équations de récurrence classiques et leurs solutions
- Tableau 3.3 (p.150):
  - Les temps d'exécution d'instructions en langage noyau
- Tableau 3.5 (p.155):
  - L'utilisation mémoire associée aux instructions en langage noyau

Quelques réflections sur la performance

Réflections sur la performance

- L’augmentation exponentielle de la vitesse
  - La loi de Moore
- Les problèmes NP et NP-complets
  - Vivre avec les problèmes NP-complets
- L’optimisation
  - “L’optimisation prématurée est la source de tous les maux”

La loi de Moore

- La densité des circuits intégrés double environ tous les 18 mois
  - Observé pour la première fois par Gordon Moore en 1965
  - Ce phénomène se vérifie jusqu'à ce jour!
  - L'origine de cette loi est technologique et économique
- "La performance double environ tous les deux ans"
  - Interpretation fausse mais courante de la loi de Moore
  - Cette interpretation semble se vérifier aussi
- Par contre, la vitesse horloge n'augmente pas forcément de la même façon!
  - D'ailleurs, nous sommes actuellement sur un plateau avec une vitesse d'environ 3 GHz qui n'augmente plus

Les problèmes NP

- Certains problèmes semblent être insolubles en pratique
  - Pas parce qu'ils ont beaucoup de travail à faire, mais pour des raisons plus fondamentales
  - Il existe des algorithmes, mais ces algorithmes ont une complexité trop élevée (par exemple, exponentielle)
- Par exemple, les problèmes NP
  - Un problème est dans la classe NP si on peut vérifier un candidat solution en temps polynômial
    - NP veut dire "***en temps Non-déterministe Polynomial***"
  - Mais trouver une solution peut être beaucoup plus coûteux (souvent exponentiel)!

Satisfaisabilité des circuits digitaux

- Soit un circuit digital combinatoire (sans mémoire) construit avec des portes Et, Ou et Non
  - Existe-t-il un ensemble d'entrées qui rend vraie la sortie?
- Ce problème est dans NP: il est simple de vérifier un candidat solution
- Mais il est beaucoup plus compliqué de trouver une solution
  - Après des décennies de travail, aucun chercheur en informatique n'a trouvé un algorithme qui est meilleur (dans le cas général) que simplement d'essayer toutes les possibilités!
  - Le meilleur algorithme connu a une complexité exponentielle
  - On soupçonne qu'il n'existe pas d'algorithme polynomial (mais on n'a pas de preuve)
- La gloire éternelle attend la personne qui (1) prouve qu'il n'existe pas d'algorithme polynomial ou (2) trouve un algorithme polynomial

Les problèmes NP-complets

- Certains problèmes dans la classe NP ont la propriété, que si on trouve un algorithme efficace pour résoudre le problème, on peut dériver un algorithme efficace pour tous les problèmes NP
- Ces problèmes s'appellent les problèmes NP-complets
- La satisfaisabilité des circuits digitaux est un problème NP-complet

Vivre avec les problèmes NP-complets

- On rencontre souvent des problèmes NP-complets en pratique
- Comment on fait, si le meilleur algorithme connu pour ces problèmes est exponentiel?
- Souvent on peut modifier le problème pour éviter le cas exponentiel
  - Par exemple, on se contente d'une bonne approximation ou d'un algorithme qui parfois ne donne pas de résultat
  - Exemple: ***problème du voyageur de commerce*** ("Traveling Salesman Problem"): quel est l'itinéraire du voyageur qui visite toutes les villes avec une distance totale minimale?
    - C'est un problème NP-complet
    - La variante où l'on est satisfait d'une distance à 10% de la distance minimale est polynomial

L’optimisation

- Dans certains cas, la performance d’un algorithme peut être insuffisante même si le problème est soluble en pratique
- Il existe alors des techniques pour améliorer des performances
  - Par exemple, la ***mémoisation***: garder les résultats des anciens calculs pour ne plus les refaire
  - Cette technique suffit pour convertir la version exponentielle de la fonction Fibonacci en version polynomiale
- Généralement, on peut améliorer jusqu’à un certain point, après duquel le programme devient rapidement plus compliqué pour des améliorations de plus en plus petites
- “L’optimisation prématurée est la source de tous les maux”
  - Ne jamais optimiser avant que le besoin se manifeste

Résumé

Résumé

- L'***efficacité*** d'un programme se caractérise par son ***temps d'exécution*** et son ***espace mémoire***
  - Ces notions dépendent de ***beaucoup de facteurs*** (CPU, charge du réseau, qualité de l'algo, langage de programmation, etc.)
- La ***complexité asymptotique*** (temporelle et spatiale) permet d'analyser la qualité des algorithmes, indépendamment des autres facteurs et ***sans*** devoir les implémenter
  - Il suffit de compter le nombre d'***opérations primitives*** (ignorer des grandeurs qui ne dépendent pas de la taille du problème)
  - La ***complexité spatiale*** est déterminée par un raisonnement analogue
- Les notations ***O*** et **Θ** sont utiles pour exprimer des bornes sur les complexités et facilitent l'analyse
- L'amélioration des performances du matériel suit la ***loi de Moore***
- Certains problèmes sont insolubles pour des raisons fondamentales, par exemple les ***problèmes NP-complets***, pour lesquels les meilleurs algorithmes connus sont exponentiels