Programmation et algorithmique

IN101
ENSTA Paris - TC 1ère année

François Pessaux

U2IS

2022-2023

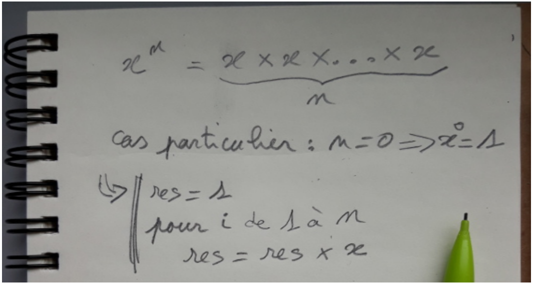
prenom.nom@ensta-paristech.fr

Diviser pour régner

Le problème à résoudre

- Calculer x^n.
- Première question : $x, n \in ?$
- $\Rightarrow$ Choix de $x, n \in \mathbb{N}$.
- Seconde question : « comment le faire ? ».
- $\Rightarrow$ Papier, crayon et éventuellement réflexion.

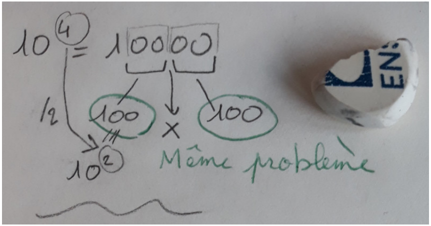
x^n : première idée




```
int power (int x, int n) {
  int res = 1 ;
  for (int i = 1; i <= n; i++)
      res = res * x ;
  return res;
}
```

- Une boucle ⇒ complexité temporelle : $O(n)$.
- Complexité spatiale : $O(1)$.

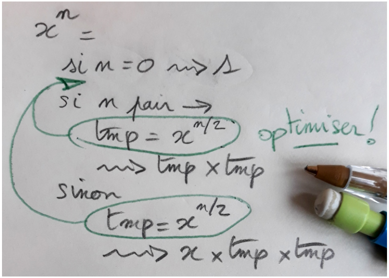
x^n : autre idée



- Cas particulier d'utilisation de $x^{a+b} = x^a \times x^b$.
- Sous-problèmes de même forme.
- Essayer de prendre $a = b$ pour ne calculer qu'une fois la puissance.
- Facile : on prend $a = \frac{n}{2}$.

- Et si n impair ?
- Exemple : $10^5 \ldots = 10 \times 10^4$.
- Or 10^4 ($x^{\text{nombre pair}}$), on sait faire efficacement.

x^n : l'algorithme efficace



- Fonctionne car `n` est un entier.
- Si flottant : $1/2 = 0.5$ et non $0 \Rightarrow$ boucle infinie.

- Complexité temporelle : $\theta(log(n))$.
- *Quid* complexité spatiale ?

```
int power (int x, int n) {
  if (n == 0) return 1 ;
  int tmp = power (x, n / 2) ;
  if (n % 2 == 0)
    return (tmp * tmp) ;
  else return (x * tmp * tmp) ;
}
```

Paradigme « Diviser pour régner »

- Partitionner le problème d'origine en sous-problèmes de même forme
  - voire identiques
  - $\Rightarrow$ 1 gros problème $\leadsto$ 1 ou plusieurs problèmes plus petits.
- Résoudre les sous-problèmes (récursivement).
- Recombiner les solutions des sous-problèmes $\leadsto$ solution du problème initial.

Programmation dynamique

Le problème à résoudre

- Barre de longueur fixe à couper pour revente.
- Prix de revente connus pour chaque longueur.
- But : maximiser le gain.

Longueur	0	1	2	3	4	5	6	7	8
Prix	0	2	3	8	10	13	15	16	21

- Une solution optimale : bout de 3 + bout de 5 → prix : 21.
- Autre solution optimale : bout de 8 → prix : 21.

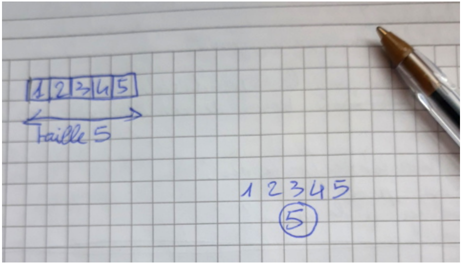
Trouver un algorithme

- Trouver toutes les découpes possibles.
- Calculer pour chacune le gain.
- Retourner le plus grand.

⇒ Comment calculer toutes les découpes possibles ?

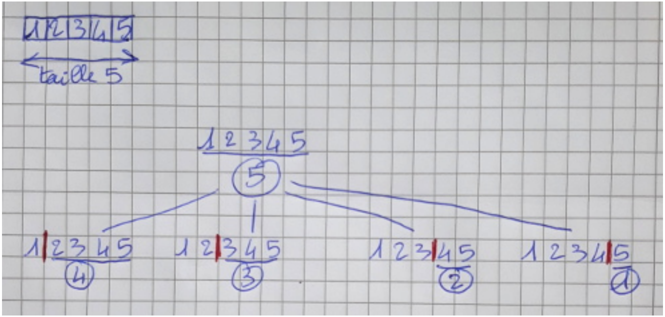
- Commencer par le cas de la barre entière.
- Puis, faire 1 découpe...
- ...faire ça pour toutes les positions possibles de la découpe.
- Recommencer sur de qui reste de chacune des découpes.

Exploration : 0 découpe



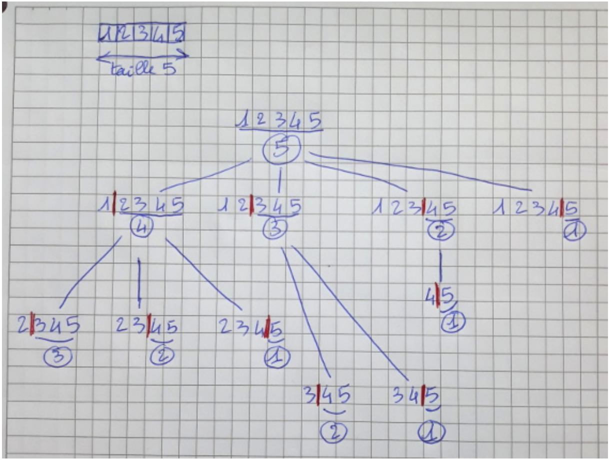
- Barre représentée avec les morceaux entre lesquels couper numérotés.
- Pas de découpe ⇒ 1 morceau de longueur d'origine.

Exploration : 1 découpe

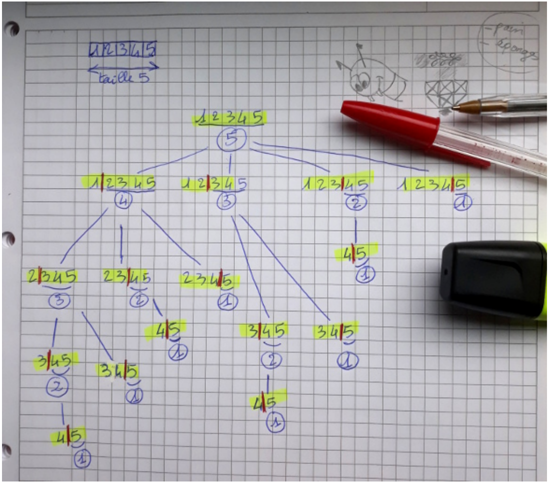


- Taille $n \Rightarrow n-1$ positions où découper.
- Prochaine étape, recommencer sur le reste de chaque découpe.

Exploration : 2, découpes

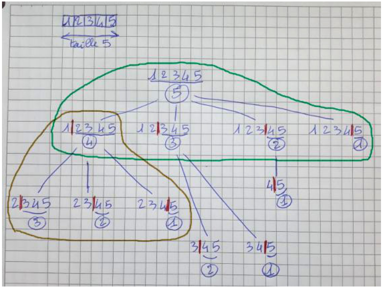


Exploration : 3 découpes etc.



- Longueur initiale $n \Rightarrow 2^{n-1}$ possibilités.

Calcul du meilleur gain



- À chaque étage, meilleur gain =

max {
Pour chaque découpe, (taille barre après découpe $\in [1; n]$)
prix de la partie coupée +
meilleur gain de ce que l'on peut tirer du reste
de la découpe.

Calcul du meilleur gain

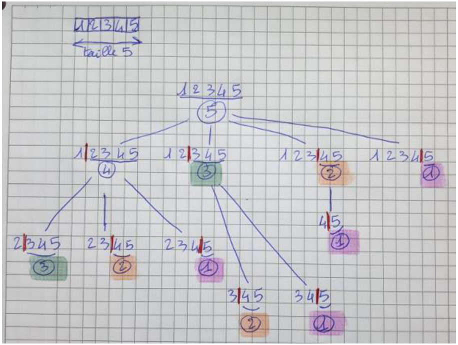
- À chaque étage, meilleur gain =

$$max \begin{cases} \text{Pour chaque découpe, (taille barre après découpe} \in [1; n]) \\ \qquad \text{prix de la partie coupée +} \\ \qquad\quad \text{meilleur gain de ce que l'on peut tirer du reste} \\ \qquad\quad \text{de la découpe.} \end{cases}$$

```c
#define PSIZE (9)
/* Size = 0 => price = 0 => prices[0] = 0 */
int prices[PSIZE] = { 0, 2, 3, 8, 10, 13, 15, 16, 21 } ;

int cut (int size) {
  if (size <= 0) return 0 ;
  int best = 0 ;           /* Meilleur prix, récursivement. */
  int cut_size = 1 ;       /* Taille de la partie retirée. */
  while (cut_size <= size) {  /* <= size : cas sans découpe. */
    best = max (best, prices[cut_size] + cut (size - cut_size)) ;
    cut_size++ ;
  }
  return best ;
}
```

Quelle efficacité ?



- Problème de taille 3 calculé 2 fois.
- Problème de taille 2 calculé 3 fois.
- Problème de taille 1 calculé 4 fois.
- Complexité exponentielle.
- ⇒ Éviter de recalculer.

Éviter de recalculer : version haut-bas

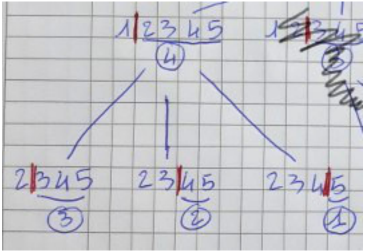
- Idée : mémoriser les calculs déjà faits pour les occurrences futures.
- Tableau de taille n.
- Avant de recalculer, regarder si résultat déjà dans le tableau.
- Si non, calculer et mémoriser dans le tableau. ⇒ Complexité : $O(n^2)$.

```
#define PSIZE (9)
int prices[PSIZE] = { 0, 2, 3, 8, 10, 13, 15, 16, 21 } ;
# Initially, no intermediate prices are recorded.
int memo_prices[PSIZE] = { -1, -1, -1, -1, -1, -1, -1, -1, -1 };

int cut (int size) {
  if (size <= 0) return 0 ;
  if (memo_prices[size] >= 0) return memo_prices[size];
  int best = 0 ;           /* Meilleur prix, récursivement. */
  int cut_size = 1 ;       /* Taille de la partie retirée. */
  while (cut_size <= size) {  /* <= size : cas sans découpe. */
    best = max (best, prices[cut_size] + cut (size - cut_size));
    cut_size++ ;
  }
  memo_prices[size] = best;
  return best ;
}
```

Éviter de recalculer : version bas-haut

- Version précédente : encore une récursion.
- Consommation mémoire au travers de la « pile d'exécution ».



$$\texttt{memo\_prices[4]} = max \begin{cases} \texttt{prices[1]} + \texttt{memo\_prices[3]} \\ \texttt{prices[2]} + \texttt{memo\_prices[2]} \\ \texttt{prices[3]} + \texttt{memo\_prices[1]} \\ \texttt{prices[4]} + \texttt{memo\_prices[0]} \end{cases}$$

- $\texttt{memo\_prices}[i] = max_{j=1 \text{ à } i} \left\{ \texttt{prices}[j] + \texttt{memo\_prices}[i-j] \right\}$
- Idée : remplir `memo_prices` pour i de 0 à n.
- $\Rightarrow$ 2 boucles ($O(n^2)$) mais récursion disparue.

Éviter de recalculer : version bas-haut

- $\texttt{memo\_prices}[i] = max_{j=1 \text{ à } i} \left\{ \texttt{prices}[j] + \texttt{memo\_prices}[i-j] \right\}$

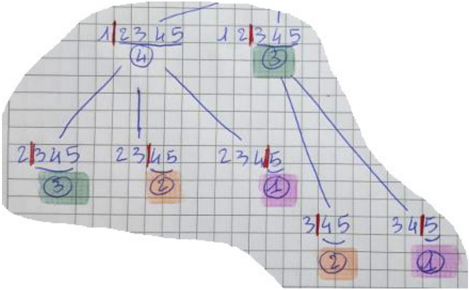
```
int prices[PSIZE] = { 0, 2, 3, 8, 10, 13, 15, 16, 21 } ;
int memo_prices[PSIZE] = { -1, -1, -1, -1, -1, -1, -1, -1, -1 };

int cut (int size) {
  for (int i = 0 ; i <= size; i++) {
    int best = 0 ;
    for (int j = 1; j <= i; j++)
      best = max (best, prices[j] + memo_prices[i - j]) ;
    memo_prices[i] = best ;
  }
  return memo_prices[size] ;
}
```

Programmation dynamique

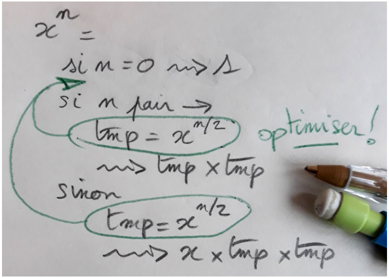
- Décomposer un problème en problèmes plus petits de même forme.
- Mémoriser les calculs intermédiaires.

- Exploration de toutes les configurations possibles.
- Fonctionne même si sous-problèmes pas indépendants.
  - Calculs pour taille = 4 et pour taille 3 non indépendants.



  - Chacun nécessite le calcul pour taille = 2.

Remarque sur l'algorithme x^n



- tmp = power (x, n / 2) puis retourner tmp ∗ tmp.
- On a directement évité de recalculer plusieurs fois la puissance.
- Le sous-problème pouvait être vu comme 2 dépendants (égaux).

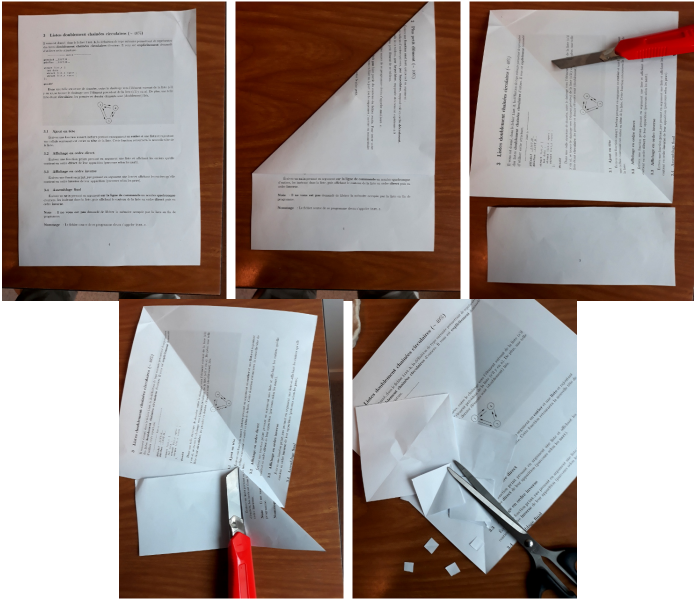
Programmation gloutonne

Le problème à résoudre

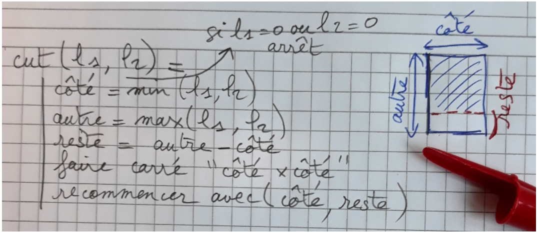
- Soit une feuille de dimensions $l_1 \times l_2$ (entières).
- Découper la feuille en carrés.
- Minimiser le nombre de carrés à découper.

- Explorer toutes les découpes possibles ? Non !
- Essayer à chaque fois de faire le plus grand carré.
- À chaque étape, c'est la solution optimale.

L'algorithme . . . sur papier



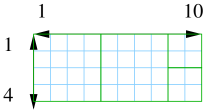
L'algorithme sans papier




```c
void cut (int dim1, int dim2) {
  if ((dim1 != 0) && (dim2 != 0)) {
    int sq_size, rem ;
    if (dim1 < dim2) {
      sq_size = dim1 ; rem = dim2 - dim1 ;
    }
    else {
      sq_size = dim2 ; rem = dim1 - dim2 ;
    }
    printf ("Carré %d x %d.\n", sq_size, sq_size) ;
    cut (sq_size, rem) ;
  }
}
```

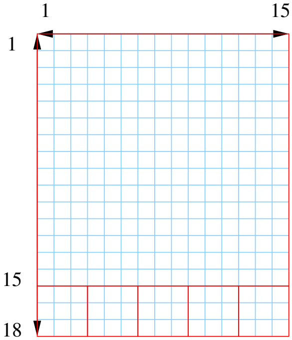
Optimalité . . .ou pas

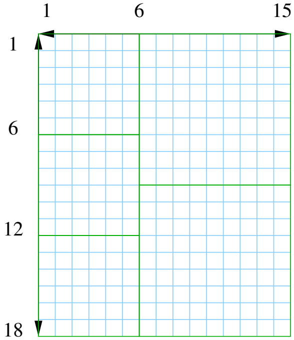
```
./cutsquare 10 4
Carré  4 x 4
Carré  4 x 4
Carré  2 x 2
Carré  2 x 2
```



- 10×4 : solution optimale trouvée par l'algorithme.

```
./cutsquare 15 18
Carré  15 x 15
Carré  3 x 3
Carré  3 x 3
Carré  3 x 3
Carré  3 x 3
Carré  3 x 3
```





- 15×18 : solution meilleure non trouvée par l'algorithme.

Algorithmes gloutons

- Problème posé, exploration de toutes les solutions : impossible.
- $\Rightarrow$ Stratégie pour trouver une « bonne » solution au problème.
- À chaque étape, choix d'une solution optimale au problème local.
- Espoir : combinaison des optima locaux $\leadsto$ « bonne » solution au problème global.
- Rarement obtention d'une solution optimale globale.
- Algorithme glouton = heuristique.