Analyse asymptotique des algorithmes glouton, programmation dynamique et backtracking

Frédéric Carpentier-Lapointe et Imane Abdillahi

November 20, 2019

1 Algorithme glouton

```
Solution glouton(Roll roll)
{
  Solution sol;
  int sum = 0;

  // Première partie : Calcul de la rentabilité pour chaque coupe
  vector<Rentability> rentabilities;
  for (int i = 0; i < roll.sizeCuts.size(); i++)
  {
    Rentability ri;
    ri.cutSize = roll.sizeCuts.at(i);
    ri.cutProfit = roll.priceCuts.at(i);
    ri.ratio = (double)ri.cutProfit / (double)ri.cutSize;
    rentabilities.push_back(ri);
  }
  sort(rentabilities.begin(), rentabilities.end(), compareRentability);


  // Deuxième partie : critère glouton
  int = 0;
  while (i < rentabilities.size())
  {
    Rentability r = rentabilities.at(i);

    int sumTemp = sum;
    if (sumTemp + r.cutSize <= roll.sizeRoll) {
      sum += r.cutSize;
      sol.sizeCuts.push_back(r.cutSize);
      sol.profit += r.cutProfit;
    } else {
      i++;
```

```
    }
  }
  return sol;
}
```

Tout d'abord, nous savons que les opérations élémentaires ont une complexité de $\Theta(1)$. Ces opérations peuvent être, par exemple, la déclaration et l'initialisation de variables. Par la suite, nous arrivons à la *première partie* de l'algorithme glouton où il faut calculer le vecteur de rentabilités pour chaque coupe du rouleau. Cette partie est de complexité $\Theta(n)$. En effet, on va exécuter cette boucle n fois. À la suite de cette boucle, on trie également le vecteur avec la fonction *sort* qui est de complexité $\Theta(nlogn)$.

Après le calcul de notre vecteur de rentabilités, on arrive à la *deuxième partie* de l'algorithme qui débute avec notre boucle while. On débute chaque tour de boucle avec la rentabilité maximum avec l'index i. Par la suite, nous avons notre critère glouton qui détermine si on peut continuer avec la rentabilité actuelle. Cette condition est une opération élémentaire et peut se faire en temps constant. Il y a alors deux cas de figure : la coupe avec la rentabilité actuelle est acceptée dans la solution ou elle ne l'est pas. Dans les deux cas, on aura des opérations élémentaires à réaliser (soit incrémenter i, ou ajouter à notre vecteur de solution). Ainsi, notre boucle dépend du vecteur de rentabilités qui est parcouru au fil des tours de boucles. Il est également intéressant de noter que dés les premiers tours de boucle, nous sommes presque assurés que les rentabilités soient conservées dans notre solution puisque notre rouleau est à sa longueur maximale. C'est lorsqu'il ne restera que de petites coupes qu'il sera probable que l'on refuse des coupes dans notre solution. Ainsi, on aura au plus n itérations dans la boucle while. Ce qui donne une complexité de $\Theta(n)$ pour cette dernière. Au final, ce qui domine dans l'algorithme est le tri initial de notre vecteur de rentabilité. Ce qui nous donne une complexité total de $\Theta(nlogn)$.

2 Algorithme de programmation dynamique

```
Solution progDyn(Roll roll)
{
  vector<int> sizeCuts = roll.sizeCuts;
  vector<int> priceCuts = roll.priceCuts;
  Solution sol[roll.sizeRoll + 1];
  int sizeCutsStep[roll.sizeRoll + 1];
  std::fill_n(sizeCutsStep, roll.sizeRoll + 1, 1);

  sol[0].profit = 0;
  int max_profit = -1;
  for (int i = 1; i <= roll.sizeRoll; i++)
  {
```

```
    for (int j = 0; j < i; j++)
    {
      int new_profit = priceCuts[j] + sol[i-j-1].profit;
      if (new_profit > max_profit) {
        max_profit = new_profit;
        sizeCutsStep[i] = j + 1;
      }
    }
    sol[i].profit = max_profit;
  }

    int total_lenght = roll.sizeRoll;
    vector<int> cutsSolution;
    while (total_lenght > 0) {
      cutsSolution.push_back(sizeCutsStep[total_lenght]);
      total_lenght = total_lenght - sizeCutsStep[total_lenght];
    }
    sol[roll.sizeRoll].sizeCuts = cutsSolution;
  return sol[roll.sizeRoll];
}
```

Pour cet algorithme, on débute avec quelques opérations élémentaires sauf une qui est le remplissage du tableau de coupes qui nous permettra de récupérer les coupes pour la solution. Ce remplissage est de complexité $\Theta(n)$, puisque l'on parcourt tous les éléments du tableau.

Par la suite, nous avons deux boucles imbriquées, une qui va de i à n et une autre qui débute de j à i. Ainsi, on aura une complexité pour ces deux boucles de $\Theta(n^2)$. L'intérieur de la boucle ne contient que des opérations élémentaires.

Après les deux boucles successives, on retrouvera les coupes utilisées pour la solution que l'on souhaite obtenir à l'aide de la boucle while. Cette dernière est implicite et dépend des longueurs des coupes, mais elle diminue au fil des tours de boucle. On peut donc dire qu'elle a une complexité de $\Theta(nlogn)$.

Finalement, ce qui domine est évidemment les deux boucles imbriqués. Ainsi, l'algorithme aura une complexité de $\Theta(n^2)$.

3 Algorithme de recherche avec retour en arrière (backtracking)

```
void cutSet(vector<Rentability>& rentabilities, int sum,
                vector<Rentability>& rentabilitiesTemp, int node, Solution& sol)
{
    if (sum < 0)
        return;
    if (sum == 0)
    {
```

```
      int sommeTot = 0;
      for (int i = 0; i<rentabilitiesTemp.size(); i++){
                sommeTot += rentabilitiesTemp[i].cutProfit;
            }
        if (sommeTot > profitmax){
            profitmax = sommeTot;
            sol.profit = sommeTot;
            cutfinal=rentabilitiesTemp;
        }
        return;
    }
    for (int i = node; i < rentabilities.size() && sum - rentabilities[i].cutSize >= 0; i++
    {
        rentabilitiesTemp.push_back(rentabilities[i]);
        cutSet(rentabilities, sum - rentabilities[i].cutSize, rentabilitiesTemp, i,sol);
        rentabilitiesTemp.pop_back();
    }
}

Solution backtrack(Roll roll)
{
         Solution sol;

  vector<Rentability> rentabilities;
  for (int i = 0; i < roll.sizeCuts.size(); i++)
  {
    Rentability ri;
    ri.cutSize = roll.sizeCuts.at(i);
    ri.cutProfit = roll.priceCuts.at(i);
    ri.ratio = (double)ri.cutProfit / (double)ri.cutSize;
    rentabilities.push_back(ri);
  }

    vector<Rentability> r;
    cutSet(rentabilities, roll.sizeRoll,r, 0,sol);

    for (int i = 0; i<cutfinal.size(); i++){
                sol.sizeCuts.push_back(cutfinal[i].cutSize);
            }
        return sol;

}
```

En premier lieu, la fonction backtrack remplit le vecteur "rentabilities". Puisque l'operation effectué est un pushback, la complexité est de $\Theta(1)$. Cependant la boucle est effectué N fois. La complexité total de cette boucle est donc

de $\Theta(n)$). La fonction cutSet() est ensuite appelé. Cette fonction permet de regrouper les meilleures coupes d'un ensemble. Dans cette methode, les deux premiers if ont pour conditions des opérations élémentaires d'une complexité de $\Theta(1)$. La boucle à l'interieur du second if est d'une complexité de $\Theta(n)$, car il s'agit d'une boucle, contenant des operation élémentaires, effectué n fois.

La boucle effectuée dans la fonction cutSet() est aussi effectué n fois, elle ne contient cependant pas d'opérations élémentaires. L'intérieur de la boucle appelle récursivement la fonction CutSet(). Il s'agit là du principe de backtrack qui teste toutes les possibilités d'une branche et, lorsque ces possibilités sont testées, retourne en arrière pour tester toutes les possibilités pour la branche suivantes. La compléxité de cette opération est $\Theta(2^n)$. Ainsi la complexité finale de cet algorithme est exponentielle, soit $\Theta(2^n)$