IENAC – Majeure SITA

Exercices de complexité

Diviser pour régner

2-d Tree

A 2-d tree (see figure 1) is a data structure used to organize a set of points in 2D such that they can be queried efficiently. The building of a 2-d tree over a set P of 2D-points proceeds as follows :

— Choose $p \in P$, remove it from P and start creating a new node containing p.

— Divide the plane in two with the **horizontal** line going through p : the points P_1 belonging to the lower half-plane will be recursively inserted in the left son and the others P_2 in the right one.

— During the next recursive call, a new point is chosen in the corresponding set of points but the half-plane is now divided along the **vertical** line : points belonging to the left half-plane will be inserted in the left son, and the others in the right one.

— Alternate horizontal and vertical divisions at each level until P becomes empty.

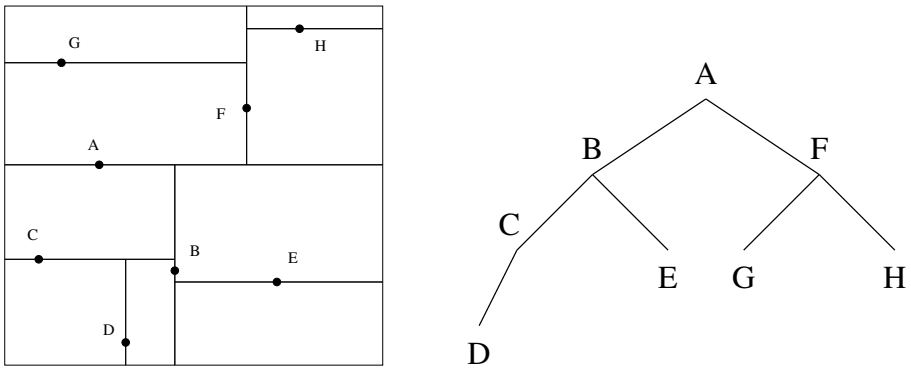

FIGURE 1 – A set of 2D points and its corresponding 2-d tree.

At each step, to divide the set of remaining points in two (almost) equal parts, p must be chosen as the median point among all x-coordinates (i.e. the point with the $\frac{n}{2}^{\text{th}}$ smallest x-coordinate) for vertical divisions, and as the median among y-coordinates for horizontal ones. To compute this step efficiently, we will **sort** the points of P **before starting to build the tree** : the points are sorted (in ascending order) along their x-coordinates in a sequence noted P_x and along their y-coordinates in sequence P_y.

1. During an horizontal step, what is the worst-case time complexity of finding the median point p and dividing P_y in two sorted subsequences of (almost) equal length ?
2. Let $p = (x, y)$, the median point of the preceding question. To be able to recursively call the algorithm (which uses P_y **and** P_x), P_x must also be parted in two sorted (with respect to x-coordinates) subsequences : the points that are below p and the points above respectively. What is the worst-case time complexity of this step ?

3. Write the recurrence relation on the worst-case time complexity of this recursive algorithm, without taking into account the preliminary sorting phase, then give its solution as a function of the number of points $n = |P|$.
4. Give worst-case time and space complexities of the overall algorithm (preliminary sortings included) as functions of n.

Maximum Subsequence Sum

Let $a = [a_1, a_2, \ldots, a_n]$ be an array of integers. We want to compute the maximum sum $\sum_{k=i}^{j} a_k$, $\forall i, j$ such that $1 \leq i \leq j \leq n$. The maximum sum is defined to be 0 if all the integers are negative.

For example, with the array $[1, -2, 1, 2, -1, 3, -2, 1]$, the maximum subsequence corresponds to $i = 3$ and $j = 6$ with the sum equal to $1 + 2 - 1 + 3 = 5$.

1. Write a naive algorithm to compute the value of the maximum subsequence sum, using `for` loops to compute each $\sum_{k=i}^{j} a_k, \forall i, j$ s.t. $1 \leq i \leq j \leq n$.
2. Compute the **time** and **space worst-case complexities** of this algorithm.
3. Write a "divide & conquer" algorithm to compute the maximum subsequence sum, using the following technique for a subarray indexed from i to j (with $1 \leq i \leq j \leq n$) :
   — divide the array in two halves from i to $mid = \frac{i+j}{2}$ and from $mid + 1$ to j (for $i = 1$ and $j = 8$, $mid = \frac{1+8}{2} = 4$ in the example) ;
   — recursively compute the maximum subsequence sum in the left subarray and in the right subarray ;
   — compute the solution for the whole array by considering three cases :
      — either the maximal subsequence is in the left subarray ($\sum_{k=3}^{4} a_k = 1 + 2 = 3$ in the example) ;
      — or the maximal subsequence is in the right subarray ($\sum_{k=6}^{6} a_k = 3$ in the example) ;
      — or the maximal subsequence crosses the middle of the array, which can be obtained by adding :
         — the maximum of the sums $\sum_{k=l}^{mid} a_k, \forall l \in [i, mid]$ in the left part, i.e. over subarrays indexed from mid down to i ($\sum_{k=3}^{4} a_k = 1 + 2 = 3$ in the example) ;
         — and the maximum of the sums $\sum_{k=mid+1}^{l} a_k, \forall l \in [mid+1, j]$ in the right part, i.e. over subarrays indexed from mid to i ($\sum_{k=5}^{6} a_k = -1 + 3 = 2$ in the example).

      which is $\sum_{k=3}^{6} a_k = \sum_{k=3}^{4} a_k + \sum_{k=5}^{6} a_k = 3 + 2 = 5$ in the example.

   Therefore, the maximum subsequence sum is 5 and crosses the middle of the two halves for the example.
4. Compute the **time** and **space worst-case complexities** of this algorithm.

Programmation dynamique

Pyramide de nombres

We want to compute the maximal sum for a path joining the top of a pyramid of numbers to its bottom, such as illustrated in figure 2. Each node of the pyramid is connected to its right and left children (one level below), except for the last level (bottom).

A pyramid of numbers will be represented by a triangular matrix w. The children of cell $w_{i,j}$ will therefore be $w_{i+1,j}$ and $w_{i+1,j+1}$.

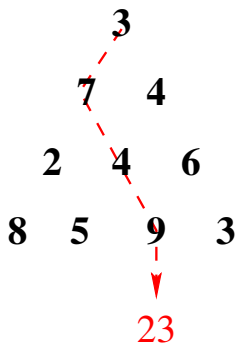

FIGURE 2 – A pyramid of numbers with height 4 and maximal path sum 23.

1. Give the number of different paths in a pyramid of numbers of height n and explain why.
2. Write a **recursive** algorithm that takes as parameter a triangular matrix representing a pyramid of numbers (and possibly its size) and computes its maximal path sum.
   Indication *Use a secondary function with parameters i and j that returns the maximal path sum from number $w_{i,j}$: this sum is equal to $w_{i,j}$ plus the greatest of the maximal path sums of its children. Start from the top.*
3. Give worst-case time **and** space complexities of this algorithm as functions of n.
4. Write an improved version of the previous algorithm by using **dynamic programming**.
   Indication *Matrix w can be directly used to store intermediary results. Start from the bottom.*
5. Give worst-case time **and** space complexities of this algorithm as functions of n.

Knapsack

Le **problème du sac à dos** (*Knapsack*) consiste à choisir un sous-ensemble parmi n objets, de telle manière que la somme des **poids** des objets choisis ne dépasse pas la **capacité** c du sac à dos et que la somme de leurs **valeurs** soit **maximale**. Si on note $w(x)$ le poids de l'objet x et $v(x)$ sa valeur (ces deux fonctions renvoient des valeurs **entières strictement positives**), il faut trouver un sous-ensemble $X' \subseteq X$ de l'ensemble des objets $X = \{x_1, \ldots, x_n\}$ tel que $\sum_{x \in X'} w(x) \leq c$ et de valeur totale $\sum_{x \in X'} v(x)$ maximale.

On peut résoudre ce problème NP-difficile grâce à la relation de récurrence sur $kp(i, j)$ définie comme la valeur totale maximale d'un sous-ensemble d'objets de $\{x_1, \ldots, x_i\}$ ne dépassant pas le poids total j :

$$\forall i \in [1, n],\ \forall j \in [1, c] \quad kp(i, j) = \begin{cases} kp(i-1, j) \text{ si } w(x_i) > j \\ \max(kp(i-1, j), kp(i-1, j - w(x_i)) + v(x_i)) \text{ sinon} \end{cases}$$

avec $\forall j, kp(0, j) = 0$ (l'ensemble vide a une valeur nulle) et $\forall i, kp(i, 0) = 0$ (un sac de poids nul a une valeur nulle). La solution sera ainsi donnée par le calcul de $kp(n, c)$.

1. Écrire une fonction **récursive directe** `knap(w, v, c)` qui calcule $kp(n, c)$ en prenant en paramètres la liste w des poids et la liste v des valeurs des objets ainsi que la capacité maximale c du sac à dos.

2. Calculer des **bornes inférieures** pour les **complexités temporelle et spatiale** en pire cas de cet algorithme.
   Indication : on pourra considérer que les poids des objets valent 1.
3. Améliorer l'efficacité de la fonction précédente en écrivant une fonction `knap_dyn(w, v, c)` qui utilise la technique de **programmation dynamique**.
4. Calculer les **complexités temporelle et spatiale** en pire cas de `knap_dyn`.