Algorithms Lab 6

The in-lab problems are to be solved during the lab time. Work with your team. You do not need to turn in your answers. Ask me for feedback.

The homework problem set is due in one week. Work with your team, but write your solutions individually. List the people with whom you discussed the problems clearly on the first page.

In lab

1. The skyline problem/the upper envelope problem: In this problem we design a divide-and-conquer algorithm for computing the skyline of a set of n buildings.

   A *building* B_i is represented as a triplet $(\mathbf{L_i}, H_i, \mathbf{R_i})$ where $\mathbf{L_i}$ and $\mathbf{R_i}$ denote the left and right x coordinates of the building, and H_i denotes the height of the building (note that the x coordinates are drawn boldfaced.)

   A *skyline* of a set of n buildings is a list of x coordinates and the heights connecting them arranged in order from left to right (note that the list is of length at most $4n$).

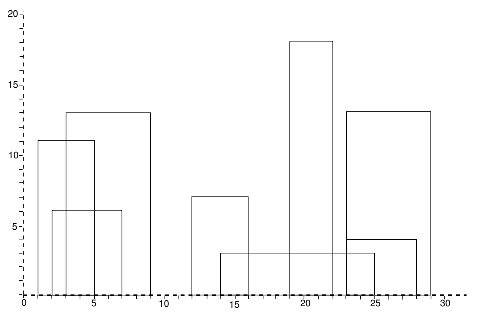
   Example: The skyline of the buildings

   $$\{(\mathbf{3}, 13, \mathbf{9}), (\mathbf{1}, 11, \mathbf{5}), (\mathbf{12}, 7, \mathbf{16}), (\mathbf{14}, 3, \mathbf{25}), (\mathbf{19}, 18, \mathbf{22}), (\mathbf{2}, 6, \mathbf{7}), (\mathbf{23}, 13, \mathbf{29}), (\mathbf{23}, 4, \mathbf{28})\}$$

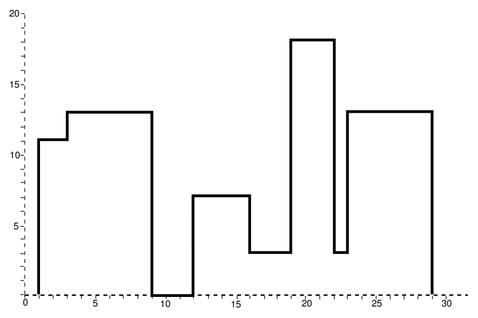
   is

   $$\{(\mathbf{1}, 11), (\mathbf{3}, 13), (\mathbf{9}, 0), (\mathbf{12}, 7), (\mathbf{16}, 3), (\mathbf{19}, 18), (\mathbf{22}, 3), (\mathbf{23}, 13), (\mathbf{29}, 0)\}$$

   (note that the x coordinates in a skyline are sorted).



buildings



skyline

   (a) Let the size of a skyline be the number of elements (tuples) in its list.
   Describe an algorithm for combining a skyline A of size n_1 and a skyline B of size n_2 into one skyline S of size $O(n_1 + n_2)$. Your algorithm should run in time $O(n_1 + n_2)$.

   (b) Describe an $O(n \log n)$ algorithm for finding the skyline of n buildings.

Homework

1. (CLRS 2-4) Let $A[1..n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an inversion of A.

   a. List the inversions of the array $< 2, 3, 8, 6, 1 >$.

   b. What array with elements from the set $\{1, 2, ..., n\}$ has the most inversions? How many does it have?

   c. Give an algorihm that determines the number of inversions in an array in $O(n^2)$ time.

   d. Give an algorihm that determines the number of inversions in an array in $O(n \lg n)$ time worst-case (Hint: modify merge sort).

2. (**0-1 knapsack problem:**) The knapsack problem is the following: You are given n items, numbered 1 to n, and two arrays $w[1..n]$ and $v[1..n]$ that specify how much each item is worth and how much it weighs: item i is worth $v[i]$ and has weight $w[i]$ pounds. Note that we index the arrays from 1 to n just to make it simpler. You are also given a knapsack of capacity W pounds. The goal is to fill the knapsack with items so that the total value of the items in the backpack is maximal. Put differently, you want to find a subset of items such that their total weight is smaller than W (i.e. they fit into the backpack) and their total value is maximized, over all such subsets.

   For simplicity, we'll start by *only* computing the maximal value that we can put in the backpack. Once we know how to do this, we'll show that the solution can be extended to compute not only the value, but the actual set of items that achieve this value.

   Let's try to come up with a solution for this problem.

   (a) Imagine that one of your (algorithms-savy) friends claims that the following greedy strategy always works: pick the items with the biggest value-per-pound. Prove him wrong by showing a counter-example —that is, show that there exist an instance of the knapsack problem where using the greedy strategy will not give the optimal solution.

   (b) You have yet another friend who claims that the following greedy strategy always works: Select the items in order of their values; that is, select with largest value first, and so on. Prove him wrong.

   (c) So...you can't realy on your friends, you have to solve it yourself; great! Let's do it! If you think about it a little bit, you see that the problem is recursive and it has optimal substructure: once you pick an item to put in the backpack, you need to fill the remaining capacity with the remaining items, optimally. That is, you have to find the largest value you can put in the remaining capacity with the remaining items. Do you start seeing the recursive formulation?

   Often the hardest part is coming up with a notation. Let us denote:

   $optknapsack(k, w)$: the maximal value obtainable when filling a knapsack of capacity w using items among items 1 through k.

   To solve our problem we need to compute $optknapsack(n, W)$.

Come up with the recursive formulation for *optknapsack*(k, w). *Hint: The idea is to consider each item, one at a time. Let's take item* k*: either it's part of the optimal solution, or not. We need to compute both options, and chose the best one.*

(d) Show the recursive algorithm for computing *optknapsack*(k, w).

(e) Analysis: Let $T(n, W)$ be the running time of *optknapsack*(k, w).
Write a recurrence relation for $T(n, W)$.

(f) Argue that $T(n, W)$ is exponential.

(g) Describe a dynamic programming solution and analyze its running time.

(h) How can we modify the dynamic programming algorithm for 0-1 Knapsack from simply computing the best value, to actually computing the actual set that gives this value?