Using Data Structures in Algorithms, lecture notes
The Skylines Problem

We have been examining how to use data structures and restructuring of the algorithm to improve efficiency. The Skylines problem was introduced to illustrate two ideas: 1. Doing some pre-processing, frequently sorting the input in a certain way, allows you to restructure computations to make them smoother, allowing smaller amounts of work at each step. 2. Using an appropriate data structure can help take advantage of this smoothness, since the small changes at each step can be simulated using a small number of data structure operations.

The Skyline Problem Let $B_1, ...B_n$ with each B_i having three real fields, $s_i < f_i$, and $h_i > 0$ represent a set of rectangular buildings in a city, with B_i stretching along the horizontal axis from $x = s_i$ to $x = f_i$ at height $y = h_i$. We want to trace the outline of all the buildings together; more precisely, we want to graph the curve $y = maxheight(x) = max_{i|s_i \leq x < f_i} h_i$ as x ranges from the smallest s_i to the largest f_i. Note that the only points where $maxheight(x)$ changes is when $x = s_i$ or $x = f_i$. So the curve is determined by the set of $2n$ points $x, maxheight(x)$ for $x = s_1, ...s_n, f_1, ..f_n$.

The obvious algorithm We could compute the $2n$ key points above by brute force. For each i, set $x = s_i$, and, keeping an element with the maximum height max so far, run through all the buildings B_j. For each, if $s_j \leq x < f_j$, compare h_j to max; if $h_j > max$, set max to h_j. At the end, add the point (x, max) to our list of key points. Repeat for $x = f_i$. This takes time $O(n)$ for each of n buildings, so the total time is $O(n^2)$.

Improving the time complexity of algorithm We want to improve the efficiency of this algorithm. We follow the outline from the Using Data Structures Summary Sheet on the web page. The Basic Design Steps are:

- Is there some useful pre-processing that can be done? In other words, would it be easier to perform the strategy if the input is in some format?

  Well, we can see that the algorithm would be much easier to visualize if we did a sweep from the smallest value of x to the largest. It would make computing $maxheight(x)$ for the new value of x repeat much of the work for the previous value, since only one building would change status (active to inactive or vice versa) at any one step. So let's add a pre-processing stage where we sort the list of crucial x-values from lowest to highest, and then run through the sorted list in order.

- Re-structure the algorithm taking advantage of the format.

  As we said above, now that we have preprocessed, changing the order, we can view the algorithm in terms of the set of buildings that stretch across a given x. Let $Active(x) = \{B_i | s_i \leq x < f_i\}$ be this set of buildings. At each stage, if $x = s_i$ we add building B_i to $Active(x)$ and if $x = f_i$, we delete B_i from $Active(x)$. So once we've sorted, the new strategy is something like: Go through the crucial values of x from smallest to largest. Maintain a set of buildings $Active$, initially empty. For each crucial value of x, if $x = s_i$, we first add B_i to $Active$, and if $x = f_i$, we delete B_i from $Active$. Then we find the largest h_j for $B_j \in Active$ and add (x, y) to our list of crucial points.

- Identify structures that come out of the strategy. What is the strategy defined in terms of?

  The structure that is maintained above is the set of buildings $Active$ that stretch over x.

- Identify what information we need to know at each step of the strategy. Use this to define Access operations to the data structure.

  We need to find the element of $Active$ with the largest height, so we'll need some kind of FindMax operation, keyed by height.

- Identify how the structure changes in each step. This defines Update queries.

  In each stage, we either add B_i to $Active$ or delete it from $Active$. So we'll need a data structure that supports $Insert$ and $Delete$ operations.

- Find a data structure that supports these operations.

  The heap data structure supports all three operations. This data structure is reviewed on pages 275-281 of the text, but should be familiar to you from CSE 100. Some people were wondering about the Delete operation for heaps. So let me review it. Let's maintain a pointer to the rightmost element r on the bottom level of our heap. We can delete this element without violating the property that the heap be a complete binary tree, up to the last level and that the leaves at the bottom level be a consecutive group from left to right. So to delete a node a anywhere in the heap, do the following: Swap a's contents with r's contents. Delete r. If the heap order is violated at a, repeatedly swap the violating node with its parent or child until the heap property is maintained. This involves at most $logn$ swaps total.

- Re-write the algorithm in terms of these operations. How many times is each operation used?

  The pseudo-code would look like {comments in brackets }:

  $Skyline[B[1..n]$: ARRAY of TRIPLES [S:real,F: real, H: real]]: LIST of PAIRS [X: real, Y:real]. {The skyline algorithm takes in an array of descriptions of rectangular buildings and outputs the list of crucial points in plotting the curve $y = maxheight(x)$}

  1. Create an array $Crucial[1..2n]$ of pairs (s_i, i), (f_i, i). {The crucial values of x. The second co-ordinates give a pointer back to the corresponding building }
  2. Heapsort(Crucial) {Sort this array by its first co-ordinates.}
  3. Initialize an empty max-heap of pairs (h, i) where h is a real number that orders the heap and $i \in \{1, ..n\}$ is a pointer to a building. We also create an array of pointers H so that $H[i]$ is a pointer to the node in the heap storing (h_i, i) if such a point exists (initially, all are NUL). This creates a double-link between the building and the node where the building is stored in the heap. This can be used to delete items in the heap, given the building name, and will at most double the time of heap operations to maintain. Initailize Points to be an empty list of points.
  4. For $j = 1to2n$ do:
  5. begin;
  6. $(x, i) \leftarrow Crucial(j)$;
  7. IF $x = s_i$ THEN HeapInsert (h_i, i) ELSE HeapDelete(H(i));
  8. $y \leftarrow FindMax$;
  9. Append (x, y) to $List$.
  10. end; {for}
  11. Return $Points$.

- Use the time analysis for the data structure to give the time analysis for the algorithm. Compute costs for each operation in the data structure, and multiply these costs with the number of times each operation is performed. Be sure to include the pre-processing stages in the time analysis! Identify which parts are taking the most time, and delete the other parts in the O notation.

  Insert and Delete operations take time $O(\log n)$ in a heap; FindMax is constant time. As mentioned above, the added time to update the pointers in H just changes the hidden constants, at most doubling the number of steps by adding two more steps in each swap.

  The preprocessing stage is $O(n + 2n\log(2n)) = O(n + 2n\log n + 2n) = O(n\log n)$ to create and sort the array $Crucial$.

  Then it takes constant time to intialize an empty heap, but linear time $O(n)$ to initialize the array of pointers H, setting each to NUL.

  The main loop is repeated $2n$ times. Each time, one Insert or Delete operation is carried out, and one Find Max operation, as well as a few other $O(1)$ commands, such as appending a point to $List$. Thus the total time is $O(\log n + 1) = O(\log n)$ for each of the $2n$ iterations, for a total time of $O(n\log n)$.

  There are several parts that are $O(n\log n)$ and no part larger, so the overall time is $O(n\log n)$.