CS 473UG: Algorithms, Spring 2006
Midterm 1 Solutions

1. *Minimum Spanning Forest with k-edges:* For a graph $G = (V, E)$ with each edge e having cost $c(e)$ ($c(e) > 0$ for every e), a minimum spanning forest of k-edges is $F \subseteq E$ of k edges (i.e., $|F| = k$) such that the graph (V, F) is a forest (i.e., has no cycles) and $\sum_{e \in F} c(e)$ is minimum. Describe an algorithm to compute the minimum spanning forest of a graph.

   Solution
   The algorithm we will use is a modified version of one of the MST algorithms from the book. Sort all the edges based on their weight. Then iteratively add edges in order of increasing weight as long as their addition does not create a cycle in the graph. Stop when there are k edges in the graph.

   The running time for this algorithm is $O(e \log n)$ for the sorting stage, plus $O(e \log k)$ since for any edge we add, we look for cycles in a graph with $\leq k$ edges. This can be done in $O(e \log k)$ by using the union-find structure to check if two vertices are already in the same component in $O(\log k)$ amortized time.

   So the total running time is $O(e \log n)$

   To show that this returns the minimum spanning forest, we do a proof by contradiction. Consider some other minimum spanning forest F^* with k edges. Let our spanning forest from the algorithm above be F. If F and F^* have the same components, then they must be the same by the proof of optimality of the MST algorithm, since we have made an MST on each component.

   So suppose F and F^* are different in at least one component. Let e be an edge in F^* which goes between two components of F. Consider $F^* - e$, which has one more component than F (since we removed an edge). Since F has fewer components, there is another edge e' in F which goes between two components of $F^* - e$.

   Now $F^* - e + e'$ is a forest with k edges. Since our algorithm did not use e, it must be larger than every edge in F or else we would have added it (since it does not create a cycle in F). Therefore, $w(e) > w(e')$, so $F^* - e + e'$ is a smaller forest with k edges, which contradicts F^* having minimum possible weight.

2. Consider the following sorting algorithm

   - Divide the array A into $\sqrt{n}$ equal sized sub-arrays
   - Sort each sub-array recursively
   - Merge the sorted sub-arrays

   (a) Taking $T(n)$ to be the time taken by the algorithm to sort an array of size n, we get the following recursive bound on the running time

   $$T(n) = \sqrt{n}T(\sqrt{n}) + O(n)$$

   Solve the above recurrence to get a bound on the running time. **[4 points]**

   Solution
   Use a recursion tree. On level i of the tree, there are $n^{1-\frac{1}{2^i}}$ nodes, each of size $\Theta(n^{\frac{1}{2^i}})$. So there is $\Theta(n)$ "work" done at each level.

   Since we are taking the square root each time, there are $O(\log \log n)$ levels (until $n^{\frac{1}{2^i}} = O(1)$). So the bound on this recurrence is $O(n \log \log n)$.

   (b) What is wrong with the analysis in the previous part? **[1 point]**

   Solution
   Since we are merging $O(\sqrt{n})$ arrays, we can no longer perform the merge in $O(n)$ time.

(c) Give the correct analysis and the correct assymptotic running time. **[5 points]**

Solution

The correct recurrence is $T(n) = \sqrt{n}T(\sqrt{n}) + O(n^{3/2})$, since we must find the minimum of $\sqrt{n}$ things each time we merge one element, and there are n elements to merge.

Then consider the recursion tree. Each level has $n^{1-\frac{1}{2^i}}$ nodes on it. The amount of work at each node is a bit more complicated. A level 0 node requires $O(n^{3/2})$ work. A level 1 node requires $(n^{1/2})^{3/2} = n^{3/4}$ work. A level 2 node requires $(n^{1/4})^{3/2} = n^{3/8}$ work.

In general, a leve i node requires $(n^{\frac{1}{2^i}})^{\frac{3}{2}} = n^{\frac{3}{2^{i+1}}}$ work. In general, the number of nodes at level i is $(n^{\frac{1}{2^i}})^{\frac{3}{2}} = n^{\frac{3}{2^{i+1}}}$. This means that the work done at level i is $n^{\frac{2^i-1}{2^i}} \cdot n^{\frac{3}{2^{i+1}}} = n^{\frac{2(2^i-1)+3}{2^{i+1}}} = n^{\frac{2^{i+1}+1}{2^{i+1}}}$. At this point, bounding each term by a geometric series will show that the recurrence is dominated by $O(n^{3/2})$.

Solving this recurrence turned out to be more difficult than we realized, so we were generous with partial credit on this one. You also got partial credit if you came up with the wrong recurrence but solved it correctly.

3. For a set S of points on the real-line, describe an algorithm to find the fewest number of intervals of size 1 that cover the points in S, i.e., every point in S belongs to some interval chosen by the algorithm.

Solution

Let T be the set of intervals in our solution. We give a greedy solution that repeatedly adds to T the rightmost interval that covers the leftmost uncovered point in S. In order to implement the algorithm efficiently, we sort the points before adding any intervals to T.

```
Sort S from left to right
Let T = {}
While S is nonempty
  Let s be the leftmost point in S
  Let t be a unit interval with s as its left endpoint
  T = T ∪ {t}
  While (leftmost point s in S is covered by t)
    remove s from S
  End While
End While
Return T
```

It takes $\Theta(|S| \log |S|)$ time to sort S. Each time through the outer While loop, the inner While loop runs at least once. The inner While loop runs exactly $|S|$ times. Thus, the total running time for the inner and outer While loops is $\Theta(|S|)$. Let S' be the set of points that are the left endpoints of the intervals in T. No two points in S' can be covered by a single interval. Thus, any solution must contain at least $|S'| = |T|$ intervals.

4. Given a sorted array A of distinct integers, describe an algorithm that finds i such that $A[i] = i$, if such an i exists.

Solution

We will use a binary search to look for the entry 0 in the array $B[i] = A[i] - i$. First we show that $B[i]$ is a sorted array. Since $A[i]$ is sorted and consists of distinct integers, we know that $A[i+1] \geq A[i] + 1$. Thus, $B[i+1] = A[i+1] - (i+1) \geq A[i] - i = B[i]$. The correctness of our algorithm follows directly from the correctness of binary search and the fact that $A[i] = i$ if and only if $B[i] = 0$. Note that we only need to compute an entry of $B[i]$ if the binary search requires us to examine that entry. Thus, we

get the following solution, which runs in $O(\log n)$ time and uses constant extra space. We assume the indexing of the array is from 1 to n. (We intialize low to 0 and $high$ to $n+1$ since we are maintaining the invariant that if there exists an i with $A[i] = i$, then $low < i < high$.)

```
low = 0; high = n + 1;
While low < high
  mid = ⌊(low+high)/2⌋
  If A[mid] − mid < 0
    low = mid + 1
  Elseif A[mid] − mid > 0
    high = mid − 1
  Else
    print "A[mid] = mid"
    break
End While
```