

Homework #2

UCSD Winter 2002, CSE 101, 1/31/02

Question 1. Heap Data Structures. Consider a min-heap storing the values 1, 2, 3, ..., 15. Answer each of the following questions, and justify your answers.

Part a) Where in the heap can the value 1 possibly go?

Since it is a min-heap the minimum element has to go to the root of the tree, which is at location $A[1]$.

Part b) Which values can possibly be stored in entry $A[2]$?

We have exactly 15 elements, which is (2^4-1) . This means we have a full heap of height 3. There must be 6 elements below $A[2]$, thus we can have 2-9 at this position, since for these numbers it is true that there are at least 6 elements that are greater.

Part c) Where in the heap can the value 15 possibly go?

15 must go in a leaf since there are no elements that are greater than it. Leaves are at positions $A[8]$ - $A[15]$.

Part d) Where in the heap can the value 6 possibly go?

Anywhere but the root ($A[1]$).

Question2. Binary Tree Isomorphism

Solution:

The straightforward way one could think of is to divide each binary tree into two subtrees, and then check if the subtrees are isomorphic. If two subtrees of a node in a tree are respectively isomorphic to two subtrees of a node in the other tree, then the parent trees should also be isomorphic. By doing this recursively, we could figure out the solution to our original problem. So let's consider the following DQ algorithm:

Assumptions:

- x, y are the roots of two binary trees, T_x and T_y .
- $Left(z)$ is a pointer to the left child of node z in either tree, and $Right(z)$ points to the right child. If the node doesn't have a left or right child, the pointer returns NIL .
- Each node z also has a field $Size(z)$ which returns the number of nodes i in the subtree rooted at z . $Size(NIL)$ is defined to be 0. (Note: We can compute $Size(x)$ recursively in linear time by recursively computing and storing all sizes in the two subtrees and then defining $Size(x) = 1 + Size(left(x)) + Size(right(x))$.)
- Then algorithm $SameTree(x,y)$ returns a Boolean answer that says whether or not the trees rooted at x and y are isomorphic, i.e., the same if you ignore the difference between left and right pointers.

Program:

```

SameTree(x, y: Nodes): Boolean;
  IF Size(x) ≠ Size(y) THEN return False; halt.
  IF x = NIL THEN return True; halt.
  IF SameTree(Left(x), Left(y))
  THEN return SameTree(Right(x), Right(y))
  ELSE return (SameTree(Right(x), Left(y)) AND SameTree(Left(x), Right(y)));

```

Time analysis:

- If the two trees are of different sizes, the program immediately returns false.
- If T_x and T_y are of both size n :

If the subtrees of T_x and T_y are of different size l and $r = n-l-1$, then we do at most one recursive call (suppose it doesn't return false immediately) of size l and one of size r . So we have $T(n) \leq T(l) + T(n-l-1) + O(1)$.

If all subtrees are of equal size, then all of them should be size $(n-1)/2$, thus we have $T(n) \leq 3T(n/2) + O(1)$, because we will make at most 3 recursive sub-calls of size less than $n/2$. This seems to be the worst case, and we get $T(n) = O(n^{\log_2 3})$ according to the Master method.

Now we conjecture this is true, that is, this is also true with the case when the subtrees are of different size l and r . We prove it by induction.

Claim: $T(n) \leq c n^{\log_2 3}$

Base: choose appropriate c , the claim is true when $n = 1$.

Induction: hypothesis: assume the claim is true for all $1 \leq n \leq k$

Now let $n = k+1$, apply the induction hypothesis, we have

$$\begin{aligned}
 T(n) &\leq cl^{\log_2 3} + c(n-l-1)^{\log_2 3} + c \\
 &\leq cl^{\log_2 3-1} + c(n-l-1)(n-l-1)^{\log_2 3-1} + c \\
 &\leq (cl + c(n-l-1)) n^{\log_2 3-1} + c \\
 &\leq (c(n-1)) n^{\log_2 3-1} + c \\
 &\leq c n^{\log_2 3} - c n^{\log_2 3-1} + c \\
 &\leq c n^{\log_2 3} \quad (\text{because } n \geq 1)
 \end{aligned}$$

Therefore, by induction, $T(n) \leq c n^{\log_2 3}$ is true for all n . So the runtime of this DQ algorithm should be $T(n) = O(n^{\log_2 3})$.

By the way, there is another non-DQ solution to this problem in linear time. Briefly, first transform both trees to *left-heavy* (or *right-heavy*) form, that is, the bigger subtree is always on the left (or right). This can be done using several $O(n)$ DFS traversals. Then by using DFS compare the size of subtrees in all levels, we will get the answer.

Question 3. Median of two sorted lists: CLRS 9.3-8 (2nd edition pg. 193).

Solution:

We have to take advantage of the fact that we know that X and Y are sorted. If $X[5]$ is the median it means that 4 elements in X are less than the median and $n-5$ elements are greater. This then implies that the remainder of the elements less than the median are in $Y[]$, so we know that in $Y[]$ the first $n-4$ elements are less than the median and the last 5 elements are greater than the median. What we

need to do is to perform a binary search for the median in one of the arrays. The search is guided by the contents of the other array. In particular, if we claim that the median is the i -th element in X , then we have three cases. Case 1) If $X[i] = Y[n-i+1]$ then the claim is true. Case 2) If $X[i] < Y[n-i+1]$ then the claim is false and the actual median is to the right of i in X or to the left of $(n-i+1)$ in Y . Case 3) If $X[i] > Y[n-i+1]$ then the claim is false and the actual median is to the left of i in X , or to the right of $(n-i+1)$ in Y . The following is a recursive DQ algorithm, which essentially is a modified binary search in a sorted list.

```

FindMedian (X[1..n], Y[1..n])
{
  n = length(X)
  if (n ≤ 2)
    Z = Merge(X[1..n], Y[1..n])
    return Z[2]
  end
  c = floor((n+1)/2)
  if (X[c] == Y[n-c+1])
    return X[c];
  if (X[c] > Y[n-c+1])
    return findMedian(X[1..c] , Y[n-c+1..n]);
  return findMedian(X[c..n], Y[1..n-c+1]);
}

```

The algorithm is essentially the same as binary search. The recurrence relation for it is: $T(n) = T(n/2) + O(1)$. Using the Master Method we get that the running time of this proposed algorithm is $\Theta(\lg n)$. Since the running time of the Merge function is $O(n)$, the call to it in the base case does not affect the overall running time complexity of the main algorithm since the arguments of Merge are arrays of constant length (i.e., length ≤ 2).

Question4. Skyline problem:

Solution:

We divide the inputs into smaller subset (generally of equal size), then solve (conquer) each subset recursively, and merge the solutions together. Assume n is a power of 2. (If not, add at most n buildings of height 0 to the list to make the total a power of 2. This at most doubles n .) Now we give the pseudo code:

Skyline(Buildings[1...n]: Array of triples of real numbers(start, end, height))::List of points(x, y) sorted by first co-ordinate x);

1. IF $n=1$ set *Keypoints* to the list containing: $(B[1].start, B[1].height)$ and $(B[1].end, 0)$, and return *Keypoints*.
2. Initialize *Keypoints* as an empty list.
3. $ListA \leftarrow Skyline(Buildings[1..n/2]);$
4. $ListB \leftarrow Skyline(Buildings[n/2+1..n]);$
5. $CurrentHeightA \leftarrow 0;$
6. $CurrentHeightB \leftarrow 0;$
7. WHILE $ListA \neq NUL$ and $ListB \neq NUL$ DO:

```

BEGIN
{  IF    the  $x$  field of the head of  $ListA$  is less than that of  $ListB$ 
  THEN  $Currentx \leftarrow$  the  $x$  field of the head of  $ListA$ ,
       $CurrentHeightA \leftarrow$  the  $y$ -field of the head of  $ListA$ ,
      Append ( $Currentx$ ,  $\text{Max}(CurrentHeightA, CurrentHeightB)$ ) to  $Keypoints$ ,
      Delete the head of  $ListA$ .

  ELSE  $Currentx \leftarrow$  the  $x$  field of the head of  $ListB$ ,
       $CurrentHeightB \leftarrow$  the  $y$ -field of the head of  $ListB$ ,
      Append ( $Currentx$ ,  $\text{Max}(CurrentHeightA, CurrentHeightB)$ ) to  $Keypoints$ ,
      Delete the head of  $ListB$ .

}
8. IF  $ListA = NUL$ 
   THEN append  $ListB$  to  $Keypoints$ .
   ELSE append  $ListA$  to  $Keypoints$ 
9. RETURN  $Keypoints$ .

```

In this algorithm above, line 3 and 4 is the “divide” part, while line 5 to 8 is the merge (conquer) part. The merging is like the merge sort, just go through the already sorted $ListA$ and $ListB$, so the runtime of this part is $O(k)$ (k is the minimum of the sizes of the two list). Therefore, according to the Master Theorem, $T(n) = 2T(n/2) + O(n)$, so the time complexity of this DQ algorithm is $T(n) = O(n \log n)$.

Question 5. Stooage sort: CLRS 7-3 (2nd edition pg. 161).

Solution:

Part a) We will prove by induction that Stooage sort correctly sorts the input array $A[1 \dots n]$. The base case is for $n=2$. In this case the first two lines of Stooage sort guarantee that a bigger element will be returned in a position following that of a smaller element. Thus, Stooage sort correctly sorts A when $n=2$. The inductive case happens when $n \geq 3$. Then, assume by inductive hypothesis that the recursive call to Stooage-sort in line 6 correctly sorts the elements in the first two-thirds of the input array A , and let A' be the array thus obtained. Also, assume by inductive hypothesis that the recursive call to Stooage-sort in line 7 correctly sorts the elements in the last two-thirds of the array A' , and let A'' be the array thus obtained. Finally, assume by inductive hypothesis that the recursive call to Stooage-sort in line 8 correctly sorts the elements in the first two-thirds of the array A'' , and let B be the (final) array thus obtained. Now we need to prove that at the end of the algorithm, the array B is sorted. To prove this, we just need to prove that for any two elements $A[i] < A[j]$, the position of $A[i]$ in B will be smaller than the position of $A[j]$, no matter which is their position in the input array A . Since this will be true for any pair $A[i], A[j]$, then the correctness of Stooage sort will follow.

We divide the proof in two cases. 1) Consider the simple case in which the position of $A[i]$ is smaller than the position of $A[j]$ in the input array A . Then, clearly, from the three facts that we have established using the inductive hypothesis, we have that the position of $A[i]$ in B will be smaller than the position of $A[j]$. 2) Consider the more involved case in which the position of $A[j]$ is smaller than the position of $A[i]$ in the input array A . We need to prove that their order in B

will be swapped. Here we have some other cases. First, if $A[j]$ and $A[i]$ are both in the first two-thirds of the input array A , then they will be swapped in the recursive call in line 6 (which is correct by inductive hypothesis) and will not be swapped again in the following recursive calls (again, by the inductive hypothesis). Second, if $A[j]$ and $A[i]$ are both in the last two-thirds of the input array A , then they will be swapped in the recursive call in line 7 (which is correct by inductive hypothesis) and will not be swapped again in the following recursive call (again, by the inductive hypothesis). Finally, if $A[j]$ is in the first one-third of the input array A and $A[i]$ is in the last one-third of the input array A , then we claim that their positions are swapped either in the recursive call in line 7 or in the recursive call in line 8. In fact, assume their positions are not swapped in the recursive call in line 7. This means that after the execution of the recursive call in line 6 (which is correct by inductive hypothesis) the position of $A[j]$ in array A' is still in the first one-third and moreover, at least all the elements in the second third of A' are greater than $A[j]$. Then we have that since these elements are greater than $A[j]$, then they are greater than $A[i]$ (since we assumed $A[i] < A[j]$), and after the execution of the recursive call in line 7 (which is correct by inductive hypothesis), $A[j]$ will appear in the first two-thirds of A'' . Now, since also $A[i]$ will remain in the first third of A'' , their positions will be swapped by the execution of the recursive call in step 8.

Part b and c) The recurrence is clearly $T(n) = 3T(2n/3) + O(1)$. Its solution is, by case 1 of the Master Theorem, $T(n) = \Theta(n^{\log_{3/2} 3}) = \Omega(n^{2.7})$. Clearly, the worst-case running time of Stooge-sort is worse than all worst-case running times of insertion sort, merge sort, quicksort and heapsort. Hence professor Stooge should not be rewarded.

Question 6. DQ Maximal Elements.

Solution:

Now that we know the solution of the skyline problem, we could reduce this “maximal elements” problem to the skyline problem by transform each point (x, y) to a “building” $(0, y, x)$. Here is the algorithm:

Step 1: Find the minimum of both x-coordinate x_m and y- coordinate y_m , then shift the original XY-plane’s coordinates so that the point (x_m, y_m) become $(0, 0)$ in the shifted plane. This “shift” operation takes $O(n)$ time.

Step 2: For every point (x, y) in the shifted plane, regard it as a “building” specified as $(0, y, x)$ in the skyline problem. Then run the DQ algorithm of skyline and get the output list of as the form *Keypoints* $((x_1=0, y_1), (x_2, y_2), \dots, (x_{n+1}, y_{n+1}=0))$. So the runtime of this step should be $O(n \log n)$.

Step 3: Do a refine operation the list *Keypoints*:

FOR $j = 1$ TO n DO

IF $y_{j+1} = y_j$ THEN delete both x_{j+1} and y_{j+1}

It’s clear to see this operation can be done in $O(n)$ time.

Step 4: For the refined list *Keypoints*, delete x_1 and y_k , which both are 0. Then match y_i with x_{i-1} , so the new list will look like $((y_1, x_2), (y_2, x_3), \dots, (y_{k-1}, x_k))$. Swap x with y for each pair, and output the list. This step runtime: $O(n)$.

Step 5: Shift all the points in the list back to the original plane by $x = x + |x_m|$, $y = y + |y_m|$, and output this final list of maximal elements. The time complexity is $O(n)$.

Therefore, the total of this algorithm runtime $T(n) = O(n \log n)$.

Question 7. Water Jugs: CLRS 8-4 (2nd edition pg. 179).

Solution:

Part a) A simple deterministic algorithm to group the jugs in pairs is the following. Select a jug from the blue set. Compare it with every red jug and find its pair. Do this for all the blue jugs. Clearly, the algorithm will find the correct grouping, but the total number of comparisons performed by it is $\Theta(n^2)$.

Part b) Consider the red jugs in some arbitrary order. The correct matching will identify one permutation of blue jugs out of $n!$ possible permutations of blue jugs. A comparison tree that does this will have $\geq n!$ leaves, and hence height $\Omega(n \lg n)$.

Part c) Consider the following algorithm for grouping the jugs into pairs, which works much like the randomized version of quicksort. Select a random blue jug and partition the red jugs with it into two groups: one group that contains red jugs that are smaller than the selected blue jug, and one group that contains red jugs that are larger than the selected blue jug. While doing the partitioning find the red jug which is equal to the selected blue jug. Now, using the red jug, which is equal to the selected blue jug, partition the blue jugs. So we have obtained 2 sets of partitions that match up. Now, recursively perform the same operations on these new partitions of jugs until the size of the partitions > 1 . If during the execution of this algorithm we make sure that the partition, which contains jugs that are smaller than the selected jug is placed to the left of the selected jug, while the other partition is placed to the right the algorithms will not only pair the jugs but will also sort them. Since this algorithm closely reassembles quicksort the same running time analysis applies to it. Thus the expected running time of this algorithm will be $O(n \lg n)$, while the worst case running time will be $O(n^2)$.