Homework 4

Tushar Jain, N12753339
CSCI-GA 1170-003 - Fundamental Algorithms

October 5, 2017

Problem 1.a . What is the height of a nearly complete binary tree containing n elements?

Solution. The relationship between the height h and number of nodes n in a nearly complete binary tree is given by:

$$2^h \leq n \leq 2^{h+1} - 1 \text{ , or } h = \lfloor \log_2 n \rfloor$$

□

Problem 1.b. Give lower and upper bounds for the number of nodes at height h in a nearly complete binary tree.

Solution. For a **nearly complete binary tree** of height h:

- There are 2^d nodes at depth d for $d = 1, 2, ..., h-1$
- The nodes at depth h are as far left as possible with the bounds:

  $$1 \leq \text{Number of Nodes at depth } h \leq 2^h - 1$$

  This is because for a nearly complete binary tree there must be at least 1 leaf node and at least 1 parent with less than 2 children otherwise the nearly complete binary tree will become complete.

□

Problem 1.c. Where in a max-heap might the smallest element reside, assuming that all elements are distinct? Justify your answer.

Solution. Given a max-heap, the smallest element must reside among the leaf nodes of the heap.

This is because in a max-heap a parent must be greater or equal to its children (greater than in our case as it's given that the elements are distinct). This implies that the smallest element can not have any children as that would contradict the above stated max-heap property. Elements which do not have any children are called the leaf nodes and reside at the bottom of the heap.

Thus, in other words, the smallest element must reside among the leaf nodes of the max-heap. □

Problem 1.d. Is an array that is sorted in ascending order a min-heap?

Solution

Yes, if we interpret the array as having an implied binary tree structure then it can be interpreted as a valid min-heap.

Problem 2.a. Give an $O(1)$ algorithm for an operation $Find-2nd-Max(A)$ that returns the value of the second largest element in a max-heap A.

Solution

Pseudo-Code:

Assuming max-heap is represented as an array with 0-indexing where the children of a given index i are located at indices $2i+1$ (left) and $2i+2$(right). If the array has a size less than $2i+1$ or $2i+2$ then those children of i are considered *null*.

```
1 def Find-2nd-Max(A):
2     if A.heapsize < 2: # len(A) < 2 # for python
3         return None
4     if A.heapsize = 2: # len(A) = 2 # for python
5         return A[1]
6     if A[left(0)] > A[right(0)]:
7         return A[left(0)]
8     else:
9         return A[right(0)]
```

Correctness:

Base case: For n=2, there are only 2 elements - root and its child. Following the maxheap property, the value(root) ¿ value(child(root)). Thus, the child is the 2nd largest element in the max-heap.

Induction Step: Using the same max-heap property, we know that the root is the largest element in a max-heap. On similar grounds, the children of the max-heap must also be the largest element of their respective subheaps (sub max-heaps). Thus, the second largest element in a given max heap is one of the children of the root. The algorithm simply compares the values of the children and returns the greater one among them. □

Running Time:

As there is no dependence on the input-size (other than the base case), the algorithm's running time complexity is in the order of $O(1)$. □

Problem 2.b. Give a $O(\log n)$ algorithm for an operation $Extract - 2nd - Max(A)$ that returns the value of the second largest element in a heap A and then removes it from the heap. Do this while only using operations on a heap defined in class: $Max - Heapify$, $Heap - Maximum$, $Heap - Extract - Max$, $Heap - Increase - Key$, and $Max - Heap - Insert$.

Solution

Pseudo-Code:

```
1  def Extract-2nd-Max(A):
2      if A.heapsize < 2:
3          return None
4      max1 = Heap-Extract-Max(A)
5      max2 = Heap-Extract-Max(A)
6
7      Max-Heap-Insert(A, max1)
8      return
```

Correctness:

The algorithm uses an approach of extracting max from the max-heap twice. The first extracted value is stored in a variable while the second time, the value is dumped. And then the first extracted value is inserted back into the max-heap.

If the max-heap has a heapsize greater than 2, then the max of the heap is extracted but stored for later insertion. The heap internally has been max-heapified during the extraction process. Thus, the largest element after the original max becomes the root. We thus extract that (effectively being the second largest element of the original max-heap) from the heap and insert back original maximum (running heapify internally again and again). □

Running Time:

The $Heap-Extract-Max$ method called in lines 4 and 5 have a running time complexity of order $O(\log n)$. So is the time complexity of $Max - Heap - Insert$ method used in line 7. The running time for all the other lines is constant ($O(1)$). Thus, the running time of the above defined method $Extract - 2nd - Max$ is:

$$\begin{aligned} T(n) &= c_1 \log(n) + c_2 \log(n-1) + c_3 \log(n-1) + c_4 \\ &= (c_1 + c_2' + c_3') \log(n) + C \\ &= O(\log(n)) \end{aligned}$$

Thus, the running time for this algorithm is $O(\log(n))$.

□

Problem 2.c. Now give an implementation of $Extract-2nd-Max$ that runs in time close to $\log n$ (notice the lack of the $O()$). This time you can use any operations, and modify the array representation of the heap directly.

Solution

Pseudo-Code:

```
1  def Extract-2nd-Max(A):
2      if A[left(0)] > A[right(0)]:
3          indx = left(0)
4      else:
5          indx = right(0)
6
7      max2 = A[indx]
8      A[indx] = A[A.heapsize - 1] # A[indx] = A[len(A) - 1] # for python
9      A.heapsize = A.heapsize - 1 # A = A[:-1] # for python
10     Max-Heapify(A, indx)
11     return max2
```

Correctness:

The algorithm first identifies the index of the second largest element in the heap simply by comparing the values of the children of the root. Then , it replaces the value at this index with the value of the last node (a leaf node) in the heap. And then, it reduces the size of the heap by 1. Now, it runs the $Max - Heapify$ method at this index and ensures the max-heap property is upheld in the entire heap. □

Running Time:

All lines in this method except line 10 takes constant time ($O(1)$) for execution. The $Heap - Heapify$ method called in line 10 has a running time $\log n$, here $n = \frac{n}{2}$ as we are running it on a sub-heap. Thus, the running time of the above defined method $Extract - 2nd - Max$ is:

$$T(n) = \log(n/2) + c$$

Thus, the running time complexity for this algorithm is still $O(\log(n))$ but it runs in close to $\log n$ time. □

Problem 2.d. Give a high level description of an $O(logn)$ algorithm that extracts the i-th largest element from a heap, and prove its correctness and running time. You may assume that $i < \frac{1}{2} \log \log n$.

Solution

By definition, the children of a node in a max-heap are no larger than the node. So we know that the largest node is at the root, and therefore that's definitely in the i largest. Now, by a similar argument, the next-largest element must be one of the two children of that node, and it's going to be the larger one.

The next-largest is going to be either the other child of the root, or one of the two children of the node we just took.

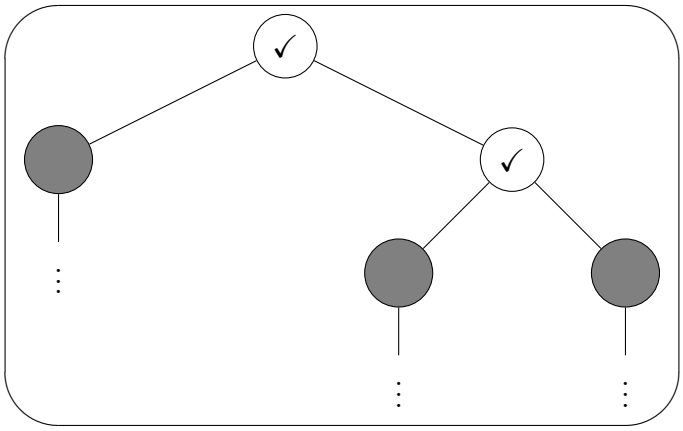

Fig 4.1: Candidates for the 3rd largest element.

The pattern continues as we take the third-largest node:

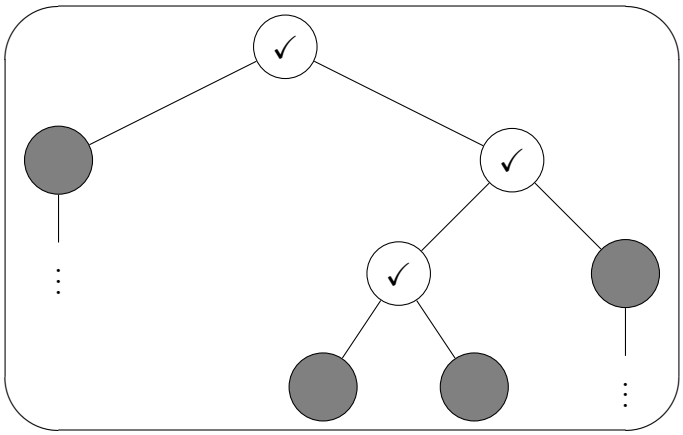

Fig 4.2: Candidates for the 4th largest element.

The next largest node is always one of the children of the nodes we have already included.

That's because, by definition, those child nodes are necessarily no larger than the nodes we've taken already, but they are also at least as big as all of their children.

We can now write down the algorithm that will take the i-largest nodes from the tree:

1. Create a max-heap that will contain the candidates for the next-largest node, and put the root in that max-heap.
2. Pick the largest item out of that max-heap, output it, and remove it from the max-heap.
3. Add the children of that node to the max-heap, and go back to 2.

Pseudo-Code:

```
1  def Extract-ith-Max(A, i):
2      B = list(A[0])
3      Build-Max-Heap(B) # step 1
4      count = 0
5      while count < i:
6          current_max = Heap-Extract-Max(B) # step 2
7          Max-Heap-Insert(B, A[left(index(current_max))]) # step 3
8          Max-Heap-Insert(B, A[right(index(current_max))])
9          count = count + 1
10     return current_max
```

Correctness:

We shall prove the correctness of this algorithm using the following **loop invariant**:

> The root at the start of the while loop, the largest value heap (B) has the i-th largest element of the original max-heap A (which is then extracted to $current\_max$).

Initialization:
For $i = 1$, at the start of the loop, just the root value of A is in the max-heap B, which is then extracted to $current\_max$ and returned as the while loop is not re-entered.

Maintenance:
At the start of the j-th loop, the j-th largest element is the root of the max-heap B which is then extracted and its children are subsequently inserted in B. This helps in maintaining the loop invariant for the next loop.

Termination:
At the end of the i-th loop, i-th largest element has been extracted and placed in the variable $current\_max$ while $i + 1$-th largest element becomes the root of max-heap B and therefore the algorithm is correct. □

Running Time:

Only lines 6-8 in this algorithm needs to be considered carefully as all other operations are of constant time. Each of these line are executed i times in the course of this method execution. As we know, both $Heap - Extract - Max$ and $Max - Heap - Insert$ are of $O(\log n)$ running time complexity. Thus, the running time of the above defined method $Extract - ith - Max$ is:

$$\begin{aligned}
T(n) &= c_0 + c_1 \sum_{i=1}^{2^{i-1}} \log k + 2c_2 \sum_{i=1}^{2^{i-1}} \log k \\
&= c_0 + c_1' \sum_{i=1}^{2^{i-1}} \log k \\
&= c_0 + c_1'(\log 2^0 + \log 2^1 + ... + \log 2^{i-1}) \\
&= c_0 + c_1''(2^0 + 2^1 + ... + 2^{i-1}) \\
&= c_0 + c_1''(\frac{2^0(2^{i-1} - 1)}{2 - 1}) \\
&= c_0 + c_1''(2^{i-1} - 1)
\end{aligned}$$

Also, we're given that $i < \frac{1}{2} \log \log n$, for the case of upper bound:

$$\begin{aligned}
& T(n) < c_0 + c_1''(2^{\frac{1}{2} \log \log n - 1} - 1) \\
\implies & T(n) = O(2^{\frac{1}{2} \log \log n - 1} - 1) \\
& T(n) = O(\log n)
\end{aligned}$$

Thus, this algorithm has a running time of $O(\log n)$.

□

Problem 3. Prove formally that the heap sort algorithm presented in class is correct, and show that in the worst case it does $\Omega(n \log n)$ comparisons.

Solution

Pseudo-Code:

```
1  def BUILD-MAX-HEAP(A):
2      A.heap-size = A.length
3      for i = floor(A.length/2) down to 1:
4          MAX-HEAPIFY(A, i)
```

```
1  def HEAPSORT(A):
2      BUILD-MAX-HEAP(A)
3      for i = A.length down to 2:
4          exchange A[1] with A[i]
5          A.heapsize = A.heapsize - 1
6          MAX-HEAPIFY(A,1)
```

Correctness:
We shall prove the correctness of $HEAPSORT$ using the following **loop invariant**:

> At the start of each iteration of the for loop in $HEAPSORT$, the subarray $A[1...i]$ is a max-heap containing the i smallest elements of $A[1...n]$, and the subarray $A[i+1...n]$ contains the $n-i$ largest elements of $A[1...n]$ in sorted order.

Base case:

During the first iteration of the for loop, $i = A.length = n$. Thus, subarray to be considered $A[1...n]$ is a max-heap since $BUILD-MAX-HEAP(A)$ was just called. Also, it contains the $i = n$ smallest elements, and therefore the empty subarray $A[n+1...n]$ trivially contains the 0 largest elements of A in sorted order.

Induction Step:

Now suppose during the ith iteration of the for loop, the subarray $A[1...i]$ is a max-heap containing the i smallest elements of $A[1...n]$ and the subarray $A[i+1...n]$ contains the $n-i$ largest elements of $A[1...n]$ in sorted order. Since $A[1...i]$ is a max-heap, $A[1]$ is the largest element in $A[1...i]$. Thus, it is the $(n(i1))th$ largest element from the original array since the $n-i$ largest elements are assumed to be at the end of the array. Line 4 swaps $A[1]withA[i]$, so $A[i...n]$ contain the $n-i+1$ largest elements of the array, and $A[1...ii]$ contains the $i-1$ smallest elements. Finally, $MAX-HEAPIFY(A,1)$ is called.

Since $A[1...i]$ was a max-heap prior to the iteration and only the elements in positions 1 and i were swapped, the left and right subtrees of node 1, up to node $i-1$, will be max-heaps. The call to $MAX-HEAPIFY$ will place the element now located at node 1 into the correct position and restore the max-heap property so that $A[1...i-1]$ is a max-heap. This concludes the next iteration, and we have verified each part of the loop invariant. Therefore, by induction, the loop invariant holds for all iterations.

After the final iteration, according to the loop invariant, the subarray $A[2...n]$ contains the $n-1$ largest elements of $A[1...n]$ in sorted order as $i = 2$. Now since $A[1]$ must be the nth largest element, that is the smallest element among the given n numbers, the whole array must be in sorted order.

□

Worst Case:
In the $HEAPSORT$, all comparisons are done in the $MAX-HEAPIFY$ function. Here is it's pseudo-code for reference:

```
1 def MAX-HEAPIFY(A, i):
2     l = left(i)
3     r = right(i)
4     if node i has a larger child
5         then swap A[i] and largest child
6         MAX-HEAPIFY(A, location largest child was in)
```

All the comparisons in the $MAX - HEAPIFY$ function happens in line 4. It decides which is largest of A[i] and its children (if it has children). The worst case occurs when it has two children, and it then takes two comparisons to find the max of the three values. So each execution of line 4 of $MAX - HEAPIFY$ will make at most two comparisons.

Moreover, $MAX - HEAPIFY$ is recursive, so the number of comparisons it makes on n items is $H(n) = H(m) + 2$ where m is the number of items in the recursive call. But that call is to the sub-heap rooted at the location where the large child was; and the worst case will occur when that subheap is as large as possible. How many items m can a subheap of a heap with n items have? It is easy to see that the largest m can be is $2n/3$; this occurs when the bottom level is exactly half full, with all its nodes in the left subheap.

For then the tree breaks into three parts that are almost exactly equal: the right subheap, the left subheap minus the bottom level, and the bottom level. This is because in a binary tree, the maximum number possible of nodes down to level i is $2^{i+1} - 1$ and the maximum possible number of nodes at level $i+1$ is 2^{i+1}, and these differ by only 1. So up to two thirds of the nodes can be in the left subheap.

We then have the recurrence $H(n) = H(2n/3) + 2$, and $H(1) = 0$. Solving by the Master Theorem, we get $H(n) = 2\log_{3/2} n = \Omega(\lg n)$.

Now we turn to the runtime for $BUILD - MAX - HEAP$. Since the only comparisons $MAX - HEAPIFY$ does are in its n calls to $MAX - HEAPIFY$, then it does at least $\Omega(n \lg n)$ comparisons.

Finally, the data comparisons done by $HEAPSORT$ consist of those done by its one call to $BUILD - MAX - HEAP$ (with at least $\Omega(n \lg n)$ comparisons), together with those done by HEAPSORT's own $n-1$ calls to $MAX - HEAPIFY$ (each of which makes $\Omega(\lg n)$ comparisons), for a total of

$$\Omega(n \lg n + (n-1) \lg n) = \Omega(n \lg n)$$

comparisons.

□

Problem 4. Given a max-heap A of size n, let $pos = pos(A, i, n)$ denote the number of positive elements in the sub-heap of A rooted at i. Consider the following recursive procedure that computes *pos*:

```
1 def Positive-Count(A, i, n):
2         if i > n:
3                 return 0
4         if A[i] <= 0:
5                 return 0
6         pos = 1 + Positive-Count(A, Left(i), n)
7         pos = pos + Positive-Count(A, Right(i), n)
8         return pos
```

Problem 4.a. Prove correctness of the above algorithm. Make sure to explain the meaning of each line 2-5. Then argue that the algorithm above runs in time $O(pos)$.

Solution

Correctness:

There will always be three cases for each scenario, $i > n$, $A[i] <= 0$ and otherwise.

Base case: For n=1, there is only 1 element to consider, then the cases are as follows:

- $i > n$ case: As it exceeds the size of the heap, we return 0.
- $A[i] <= 0$ case: As the value is not positive, we return 0.
- Otherwise: As both $Left(i)$ and $Right(i)$ will be greater than n(=1), $Positive-Count(A, Left(i), n)$ and $Positive-Count(A, Right(i), n)$ both will return 0 which can be clearly seen from 1st case above. Thus, $pos = 1$ will be returned.

Induction Step:
Assumption: For $n \leq k$ elements, algorithm works correctly and finds the solution i.e, the number of positive elements.

Now for $n = k + 1$, the algorithm checks if the root value is positive and lies within the heapsize. If so, then it recursively calls on its left and right sub-maxheaps which we know will result in correct values (which are summed and returned with 1) from our assumption. If not, it will return a 0 value.

Thus, it checks constraints and reduces the number of elements to correctly implement the idea of finding positive numbers in a given max-heap. □

Running Time:

Only lines 6-7 in this algorithm needs to be considered carefully as all other operations are of constant time. Each of these line use the concept of divide-and-conquer and thus recursively call this function on a reduced number of elements. Thus, the running time of the above defined method can be calculated as follows:

$$\begin{aligned} T(i) &= c_0 + T(\text{number of elements in } left(i)) + T(\text{number of elements in } right(i)) \\ &= c_0 + T(i') + T(i'') \end{aligned}$$

At each conquer step, we decrease the input into at least half because the max number of elements of subheap of max-heap with i elements is $i/2$.

$$T(i) = c_0 + 2T(i/2)$$

For a call on the complete max-heap of size n:

$$T(i) = O(n) \qquad \text{(Using Master's Theorem)}$$

However, due to the above mentioned checks, the recursive calls with negative values are never made. Thus, due to checks on value, recursive calls are only made on *pos* number of elements

$$\implies T(i) = O(pos)$$

Thus, this algorithm has a running time of $O(pos)$.

□

Problem 4.b. Assume now that we do not really care about the exact value of *pos* when $pos > k$; i.e., if the heap contains more than k positive elements, for some parameter k. More formally, you wish to write a procedure $Positive - Count(A, i, n, k)$ which returns the value $min(pos, k)$, where $pos = Positive - Count(A, i, n)$.

Solution

Pseudo-Code:

```
1  def Positive-Count(A, i, n, k):
2      if k <= 0:
3          return 0
4      if i > n:
5          return 0
6      if A[i] <= 0:
7          return 0
8      pos = 1 + Positive-Count(A, Left(i), n, k - 1)
9      pos = pos + Positive-Count(A, Right(i), n, k - pos)
10     return pos
```

Correctness:

The $k \leq 0$ check acts as a threshold for the number of positive elements. We simple recursively keep on passing how far we are from the threshold. As soon as we have enough *pos* elements, the passed value of k will be less than 0 and all such recursive calls will terminate in constant time returning 0 value.

Now, there will always be four cases for each scenario, $k \leq 0$, $i > n$, $A[i] <= 0$ and otherwise.

Base case: For n=1, there is only 1 element to consider, then the cases are as follows:

- $k \leq 0$ case: As the minimum number of *pos* elements have already been reached, we return 0.
- $i > n$ case: As it exceeds the size of the heap, we return 0.
- $A[i] <= 0$ case: As the value is not positive, we return 0.
- Otherwise: As both $Left(i)$ and $Right(i)$ will be greater than n(=1), $Positive - Count(A, Left(i), n, k-1)$ and $Positive - Count(A, Right(i), n, k-pos)$ both will return 0 which can be clearly seen from 2nd case above. Thus, $pos = 1$ will be returned.

Induction Step:

Assumption: For $n \leq k$ elements, algorithm works correctly and finds the solution i.e, either the number of positive elements or the threshold value k if we reached.

Now for $n = k + 1$, the algorithm checks if the threshold (i.e. $k \leq 0$) has been reached or not and then checks if the root value is positive and lies within the heapsize. If so, then it recursively calls on its left and right sub-maxheaps which we know will result in correct values (which are summed and returned with 1) from our assumption. If not, it will return a 0 value.

Thus, it checks constraints and reduces the number of elements to correctly implement the idea of finding positive numbers in a given max-heap. □

Running Time:

Only lines 8-9 in this algorithm needs to be considered carefully as all other operations are of constant time. Each of these line use the concept of divide-and-conquer and thus recursively call this function on a reduced number of elements. Thus, the running time of the above defined method can be calculated as follows:

$$\begin{aligned} T(i) &= c_0 + T(\text{number of elements in } left(i)) + T(\text{number of elements in } right(i)) \\ &= c_0 + T(i') + T(i'') \end{aligned}$$

At each conquer step, we decrease the input into at least half because the max number of elements of subheap of max-heap with i elements is $i/2$.

$$T(i) = c_0 + 2T(i/2)$$

For a call on the complete max-heap of size n:

$$T(i) = O(n) \qquad \text{(Using Master's Theorem)}$$

However, due to the above mentioned checks, the recursive calls with negative values are never made as well as recursive calls with $k \leq 0$ (i.e. after the minimum limit of positive numbers is reached) are never made. Thus, due to checks on value, recursive calls are only made either on *pos* number of elements or the minimum threshold k.

$$\begin{aligned} & T(i) = O(pos) \\ \implies & T(i) = O(k) \qquad \text{(assuming } pos >> k \text{ and cutting off recursive calls)} \end{aligned}$$

Thus, this algorithm has a running time of $O(k)$. □