CMPUT204
Department of Computing Science
University of Alberta

Solutions for Homework Assignment #2

Question 1. (29 marks) Prove asymptotic upper and lower bounds for each of the following recurrences. Assume that in each case, the function is some positive constant for sufficiently small n. To make it more convenient, you may assume that $n = c^k$ for some constant c that you choose. Make your bounds as tight as you can.

a. (4 marks) $T(n) = 7T(\frac{n}{3}) + n\sqrt{n}$.

Solution: Using the master theorem with $a = 7$, $b = 3$ and $c = \frac{3}{2}$ we have: $f(n) = \Theta(n^c)$ and $c < \log_3 7$. Thus $T(n) \in \Theta(n^{\log_3 7})$.

b. (4 marks) $T(n) = T(n-1) + \frac{1}{n}$

Solution: Since the definition of $T(n)$ does not fit into the requirements of the master theorem we use iterated substitution.

$$\begin{aligned} T(n) &= T(n-1) + \frac{1}{n} \\ &= T(n-2) + \frac{1}{n-1} + \frac{1}{n} \\ &\vdots \\ &= T(1) + \frac{1}{2} + \ldots + \frac{1}{n} \\ &\in \Theta(H(n)) = \Theta(\log n). \end{aligned}$$

c. (5 marks) $T(n) = 2T(\frac{n}{2}) + \frac{n}{\lg n}$.

Solution: Again we use iterated substitution.

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + \frac{n}{\lg n} \\ &= 2^2 T(\frac{n}{2^2}) + 2 \times \frac{n/2}{\lg(n/2)} + \frac{n}{\lg n} \\ &= 2^2 T(\frac{n}{2^2}) + \frac{n}{\lg n - 1} + \frac{n}{\lg n} \\ &= 2^3 T(\frac{n}{2^3}) + \frac{n}{\lg n - 2} + \frac{n}{\lg n - 1} + \frac{n}{\lg n} \\ &\vdots \\ &= 2^i T(\frac{n}{2^i}) + \frac{n}{\lg n - (i-1)} + \ldots + \frac{n}{\lg n} \end{aligned}$$

Assuming that $n = 2^k$ for some integer k and $T(1) = c$ we get:

$$\begin{aligned} T(n) &= 2^{\lg n} T(1) + \frac{n}{1} + \frac{n}{2} + \ldots + \frac{n}{\lg n} \\ &= c.n + n.H(\lg n) \\ &\in \Theta(n \log\log n) \end{aligned}$$

d. (5 marks) $T(n) = T(n-1) + n \lg n$.

Solution: Again, we use iterated substitution.

$$\begin{aligned} T(n) &= T(n-1) + n \lg n \\ &= T(n-2) + (n-1)\log(n-1) + n \log n \\ &\vdots \\ &= T(1) + 2\lg 2 + 3 \lg 3 + \ldots + n \lg n \end{aligned}$$

Each term in the summation above is at most $n \log n$ and there are n of them. So $T(n) \leq T(1) + n^2 \lg n \in O(n^2 \log n)$. Also if we consider the last $\frac{n}{2}$ term in the last equation above we get:

$$\begin{aligned} T(n) &\geq \sum_{i=\frac{n}{2}}^{n} i \lg i \\ &\geq \frac{n}{2}\left(\frac{n}{2} \lg \frac{n}{2}\right) \text{ So} \\ T(n) &\in \Omega(n^2 \log n) \end{aligned}$$

Thus $T(n) \in \Theta(n^2 \log n)$.

e. (5 marks) $T(n) = 3T(\frac{n}{3} + 1) + \frac{n}{2}$.

Solution:

$$\begin{aligned} T(n) &= 3T(\frac{n}{3} + 1) + \frac{n}{2} \\ &= 3^2 T(\frac{n}{3^2} + \frac{1}{3} + 1) + \frac{1}{2}(n+3) + \frac{n}{2} \\ &= 3^3 T(\frac{n}{3^3} + \frac{1}{3^2} + \frac{1}{3} + 1) + \frac{1}{2}(n + 3 + 3^2) + \frac{1}{2}(n+3) + \frac{n}{2} \\ &\vdots \\ &= 3^i T(\frac{n}{3^i} + \frac{1}{3^{i-1}} + \ldots + \frac{1}{3} + 1) + \frac{in}{2} + \frac{1}{2}\left((i-1)3 + (i-2)3^2 + \ldots + 3^{i-1}\right) \end{aligned}$$

Assume that $n = 3^k$ and let $i = k$ in the last equation. Noting that $0 < \sum_{j=0}^{\infty} \frac{1}{3^j} < 1$ and that $T(n)$ is constant for small values of n we get: $T(n) = c.3^k + \frac{kn}{2} + \frac{1}{2}((k-1)3 + (k-2)3^2 + \ldots + 3^{k-1})$. So $T(n) \leq c.n + n \log_3 n + n \log_3 n \in O(n \log n)$ and $T(n) \geq \frac{n \log_3 n}{2} \in \Omega(n \log n)$, i.e. $T(n) \in \Theta(n \log n)$.

f. (6 marks) $T(n) = T(\frac{n}{2}) + 2T(\frac{n}{4}) + 2n$.

Solution: We prove that $T(n) \in \Theta(n \log n)$ by showing that there are constants $c, d > 0$ such that: (i) $T(n) \leq cn \lg n$ and (ii) $T(n) \geq dn \lg n$.

(i) This is trivially true for small values of n by the assumption if we take large enough c. So assume that $T(i) \leq ci \lg i$ for all values of $i < n$. Thus:

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + 2T(\frac{n}{4}) + 2n \\ &\leq c\frac{n}{2} \lg \frac{n}{2} + 2c\frac{n}{4} \lg \frac{n}{4} + 2n \\ &\leq c\frac{n}{2}(\lg n - 1) + c\frac{n}{2}(\lg n - 2) + 2n \\ &= cn \lg n - \frac{3cn}{2} + 2n \leq cn \lg n, \end{aligned}$$

if we take $c \geq \frac{2}{3}$.

(ii) Again, this is true for small values of n provided that we take d sufficiently small. So assume that $T(i) \geq di \lg i$ for all values of $i < n$. Thus:

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + 2T(\frac{n}{4}) + 2n \\ &\geq d\frac{n}{2}\lg\frac{n}{2} + 2d\frac{n}{4}\lg\frac{n}{4} + 2n \\ &\geq dn\lg n - \frac{3dn}{2} + 2n \\ &\geq dn\lg n, \end{aligned}$$

if we take $d < \frac{2}{3}$.

Question 2. (6 marks) Suppose that we want to sort a list A consisting of a sorted list with many elements (say $10,000,000$) followed by a few (say 7) "random" elements. Which of the sorting methods we have seen so far would be especially suitable for such a task? Explain why and justify your answer.

Solution: Assume that the size of A (including those few random elements) is n. Regardless of the values of those few random elements, we know that merge sort, heap-sort, and quick-sort each take $\Omega(n \log n)$ time in best case. For insertion sort, the pass over the sorted part of A takes only $O(n)$ (because it is a sorted list). For each of those few random elements, insertion sort takes at most $O(n)$ to put them in their correct locations. Therefore, the total running time of insertion sort will be a few times $O(n)$ which is $O(n)$. Thus insertion sort is the better algorithm among the ones we have seen in this course.

Question 3. (25 marks) A d-ary heap is like a binary heap, except that non-leaf nodes have d children instead of 2 children (with one possible exception for the last leaf).

a. (2 marks) Explain how to implement a d-ary heap in an array and implement the corresponding Parent and Child operation as we had for the binary heap.

Solution: We can use an array to represent a d-ary heap. The root is $A[1]$ and for every node $A[i]$ its d children are in $A[d(i-1)+2], \ldots, A[di+1]$. The parten and jth child of node i can be computed by:

```
Parent (i)
    return ⌊i/d⌋
Child (i, j)
    return d(i − 1) + j + 1
```

b. (2 marks) What is the height of a d-ary heap of n elements in terms of n and d?

Solution: Since every node (other than leaves) in a d-ary tree has d children, the number of nodes in a d-ary tree of height h is at most $1+d+d^2+\ldots+d^h = \frac{d^{h+1}-1}{d-1}$ and smaller than $1+d+d^2+\ldots+d^{h-1}+1 = \frac{d^h-1}{d-1}+1$. Therefore $h = \lceil \log_d(1+n(d-1))-1 \rceil \in \Theta(\log_d n)$.

c. (4 marks) Write a version of Max-Heapify for a d-ary max-heap and analyze its running time.

Solution: The algorithm is very similar to Max-Heapify for the binary case, except that in this case we compare the key with each of its d children.

```
Max-Heapify (A, i)
    largest ⟵ i
    for j ⟵ 1 to d do
        j ⟵ Child(A, i, j)
        if j ≤ heapsize[A] and A[j] > A[largest] then
            largest ⟵ j
```

end if
if $largest \neq i$ **then**
exchange $A[i]$ and $A[largest]$
Max-Heapify $(A, largest)$
end if
end for

The number of comparisons in each iteration is $\Theta(d)$ and the number of recursive calls is at most $O(\log_d n)$ (proportional to the height of the tree). Thus the running time of this version of Max-Heapify is $O(d \log_d n)$.

d. (5 marks) Write a version of Build-Heap for a d-ary max-heap and analyze its running time.

Solution: Similar to the binary case, we can call Max-Heapify to turn A into a heap. The leaves are already a heap. The parent of the last element is $\lfloor n/d \rfloor$.

Build-Max-Heap (A)
$heapsize[A] \longleftarrow length[A]$
for $i \longleftarrow \lfloor length[A]/d \rfloor$ **downto** 1 **do**
Max-Heapify (A, i)
end for

The proof of correctness of Build-Max-Heap in CLRS for binary heaps can be easily modified to work for this case (we skip the details). A naive analysis of running time yields $O(nd \lg_d n)$ as each Max-Heapify takes $O(d \log_d n)$. But here is a more careful analysis. The number nodes at the root level (level 0) of a d-ary heap is $d^0 = 1$, at level 2 is d, at level 3 is d^2, and in general at level i which has height $h - i$ is d^i. By part (c) the time required for Max-Heapify at height i is $O(di)$. Thus the running time of Build-Max-Heap is:

$$\begin{aligned} T(n) &\leq \sum_{i=0}^{i=h} d^{h-i} O(di) \\ &\leq O(d^{h+1}) \sum_{i=0}^{i=h} \frac{i}{d^i} \\ &\leq O(nd) \sum_{i=0}^{\infty} \frac{i}{d^i}. \end{aligned}$$

From A.8 in CLRS, the summation above is at most $\frac{1/d}{(1-1/d)^2} = \frac{d}{(d-1)^2}$. Thus $T(n) \in O(n \frac{d^2}{(d-1)^2})$ and since $d \geq 2$: $\frac{d^2}{(d-1)^2} \leq 2$. Therefore $T(n) \in O(n)$.

e. (4 marks) Give an efficient implementation of Extract-Max in a d-ary heap and analyze its running time.

Solution: Again, this will be similar to the Extract-Max for the binary heap.

Extract-Max (A)
if $heapsize[A] < 1$ **then**
return error
end if
$max \longleftarrow A[1]$
$A[1] \longleftarrow A[heapsize[A]]$
$heapsize[A] \longleftarrow heapsize[A] - 1$
Max-Heapify $(A, 1)$
return max

It is easy to see that Extract-Max has the same running time as Max-Heapify which is $O(d \log_d n)$

f. (4 marks) Give an efficient implementation of Increase-Key(A, i, k), which first sets $A[i] \longleftarrow \max(A[i], k)$ and then updates the d-ary heap appropriately. Analyze the running time.

Solution:

Increase-Key (A, i, k)
  if $k < A[i]$ **then**
    error
  end if
  $A[i] \longleftarrow k$
  while $i > 1$ and $A[Parent(i)] < A[i]$ **do**
    exchange $A[i]$ and $A[Parent(i)]$
    $i \longleftarrow Parent(i)$
  end while

The running time of this algorithm is $O(\log_d n)$ as the path traced from the node updated to the root has length $O(\log_d n)$ and it takes constant time to do each iteration of the Increase-Key procedure.

g. (4 marks) Write a version of Insert in a d-ary heap and analyze its running time.

Solution: It is similar to the Insert for the binary case:

Max-Heap-Insert (A, k)
  $heapsize[A] \longleftarrow heapsize[A] + 1$
  $A[heapsize[A]] \longleftarrow -\infty$
  Heap-Increase-Key $(A, heapsize[A], k)$

The running time is the same as Heap-Increase-Key, i.e. $O(\log_d n)$ since the first two operations take constant time.

Question 4. (12 marks)

a. (10 marks) Give an algorithm (in pseudo-code) for the following problem: given an array A of size n of integers, find the $\sqrt{n}$ largest elements of A and return them. Your algorithm must run in $O(n)$. Explain your algorithm and analyze its running time.

Solution: Given array A first we make a max-heap on this array by calling the Build-Max-Heap procedure we have seen for the heap-sort algorithm. By definition of heap, the largest element of A is located at $A[1]$. So Heap-Extract-Max returns the largest element of A. We keep calling Heap-Extract-Max on A iteratively for $\sqrt{n}$ times to find the $\sqrt{n}$ largest elements of A and return them. So the algorithm will be as follows:

Find-Largest (A, n)
  ** $Answer[1 \ldots \sqrt{n}]$ will hold the solution
  Build-Max-Heap (A)
  for $i \longleftarrow 1$ **to** $\lceil \sqrt{n} \rceil$ **do**
    $Answer[i] \longleftarrow$ Heap-Extract-Max (A)
  end for

We have seen that Build-Max-Heap takes $O(n)$ time and each call to Heap-Extract-Max takes $O(\lg n)$. The We call Heap-Extract-Max $\sqrt{n}$ times. Therefore the running time of the loop is at most $O(\sqrt{n} \lg n)$ which is at most $O(n)$ (because $\lg n \in O(\sqrt{n})$. Thus the total running time is $O(n)$.

b. (2 marks) Suppose that you modify your algorithm of part (a) to find the p largest elements of A (instead of $\sqrt{n}$). What is the largest value of p for which your algorithm still runs in $O(n)$ time?

Solution: If we change the "for loop" to range from 1 to p instead of 1 to $\lceil \sqrt{n} \rceil$, then the algorithm finds the p largest elements of A by the same argument and it will take $O(n)$ (for Build-Max-Heap) plus $O(p \lg n)$. So as long as $p \in O(\frac{n}{\lg n})$.

Question 5. (10 marks) Consider the following very simple and elegant(?) sorting algorithm:

```
SomeSort (A, b, e)
if e = b + 1 then
    if A[b] > A[e] then
        exchange A[b] and A[e]
    end if
else if e > b + 1 then
    p ⟵ ⌊(e−b+1)/3⌋
    SomeSort (A, b, e − p)
    SomeSort (A, b + p, e)
    SomeSort (A, b, e − p)
end if
```

a. (4 marks) Explain why SomeSort correctly sorts its input array A, assuming that $n = e - b + 1$ is the length of the array.

Solution: let $n = e - b + 1$ be the number of elements in the array A. We use an inductive argument to show that for any value of n this algorithm sorts array A. The claim is obvious for $n \leq 2$. So let's assume that SomeSort works for arrays of size smaller than $n > 2$ and assume that A is an array of size n. To make the arguments easier assume that $e - b + 1 = 3p$ (to give a formal proof we cannot make this assumption). By induction hypothesis and because $e - p - b + 1 < n$, the first recursive call to SomeSort sorts the first $2k$ elements. Therefore, any element from the top p largest elements of A that was in locations b to $e - p$ in A is now located between $e - 2p$ to $e - p$ in A. Therefore, all the top p largest elements of A are now at indicies between $e - 2p$ and e in A. Since $b + p \leq e - 2p$ this implies that the top p largest elements of A are at locations $b + p$ to e. Therefore, the second recursive call to SomeSort puts these top p largest elements of A at their correct locations (at the end part of A) by induction hypothesis. Now that the last p locations in A have their correct numbers in, the last call to SomeSort puts the rest of the numbers into their correct locations between b and $e - p$, by induction hypothesis.

b. (4 marks) Find a recurrence for the worst-case running time of SomeSort. Give a tight (i.e. Θ) asymptotic bound for the worse-case running time of SomeSort (Hint: for simplicity assume that $n = 3^k$ for some constant k).

solution: The running time function $T(n)$ is constant for small values of n and $T(n) = 3T(\lceil \frac{2n}{3} \rceil) + c$ For simplicity, we assume that $n = 3^k$ for some $k > 0$. In this case, $\lceil \frac{2n}{3} \rceil = \frac{2n}{3}$. Thus the recurrence relation can be written as $T(n) = 3T(\frac{2n}{3}) + c$. And now using the master theorem we get $T(n) \in \Theta(n^{\log_{3/2} 3})$.

c. (2 marks) By comparing SomeSort with insertion sort, merge sort, heap-sort, and quicksort argue if this simple algorithm is efficient.

Solution: Since $\log_{3/2} 3 > 2.7$ the running time of SomeSort is $\Omega(n^{2.7})$ which is worse than all of insertion sort, merge sort, heapsort and quicksort. So this algorithm is not so elegant!.