Unity id : amanur
Name - Anusha Gururaja Manur

Homework -2

1. Purpose: Practice solving recurrences. For each of the following recurrences, use the Master Theorem to derive asymptotic bounds for T(n), or argue why the Master Theorem does not apply. You don't have to solve the recurrence if the MT does not apply. If not explicitly stated, please assume that small instances need constant time c. Justify your answers, ie. give the values of a, b, n^log_b(a) f, , for case 3 of the Master Theorem also show that the regularity condition is satisfied. (3 points each)

(a) T(n)=8T(3n/2)+n3.
Solution:
a=8
b=⅔ < 1
Since b<1, Master's theorem does not apply as b > 1 to apply Master's theorem.

(b) T(n)=4T(n/2)+n2sqrt(n).
Solution:
a=4
b=2
$n^{log_b a} = n^{log_2 4} = n^2$
$f(n) = n^{5/2}$
$n^{5/2} \in \Omega(n^{2+\varepsilon}), for\ \varepsilon = 0.1$

Checking regularity condition,
Regularity condition: There is c<1 and n0 ≥ 0 such that af (n /b) ≤ cf (n) for all values of n ≥ n 0

$af(n/b) <= cf(n)$
$4.(n^{5/2}/2^{5/2}) <= c.n^{5/2}$
$1/\sqrt{2} <= c < 1$
=> Case 3 of MT applies
=> $T(n) \in \Theta(n^{5/2})$

(c) T(n)=7T(n/3)+n11/5.

Solution:

a=7

b=3

$n^{log_b a} = n^{log_3 7} = n^{1.77}$

$f(n) = n^{2.2}$

$n^{2.2} \in \ \Omega(n^{1.77+\varepsilon}), \ for \ \varepsilon = 0.1$

Checking regularity condition,

Regularity condition: There is c<1 and n 0 ≥ 0 such that a f (n /b) ≤ cf (n) for all values of n ≥ n 0

$af(n/b) < cf(n)$

$7.(n^{11/5}/3^{11/5}) < c.n^{11/5}$

$7/3^{2.2} \ <= c < \ 1$

=> Case 3 of MT applies

=> $T(n) \in \Theta(n^{5/2})$

(d) T(n)=4T(n/2)+100-sqrt(n).

Solution:

$f(n) = 100 - \sqrt{n}$

Master theorem does not apply as f(n) is not asymptotically positive.

(e) T(n)=½T(n/3)+sqrt(n).

Solution:

a=½

b=3

$n^{log_b a} = n^{log_3 1/2} = n^{-0.63}$

$f(n) = \sqrt{n}$

$n^{0.5} \in \ \Omega(n^{-0.63+\varepsilon}), \ for \ \varepsilon = 0.01$

Checking regularity condition,

Regularity condition: There is c<1 and n0 ≥ 0 such that a f (n /b) ≤ cf (n) for all values of n ≥ n0

$af(n/b) < cf(n)$

$1/2.(\sqrt{n}\,/\sqrt{3}) < c.\sqrt{n}$

$1/(2\sqrt{3}) \ <= \ c \ < \ 1$

=> Case 3 of MT applies

=> $T(n) \in \Theta(\sqrt{n})$

2. Purpose: More practice solving recurrences. Please solve the following recurrences. (3 points each)
(a1 & a2) The two recurrences in lecture 4, page 7. Use T(1)=1.

Solution:
a1) $T(n) = 2T(n/2) + n\, log\, n$
$n = 2^k$

Solving using forward iteration,
$n = 1,\ \ T(2^0) = 1$
$n = 2,\ \ T(2^1) = 2.1 + 2$
$n = 4,\ \ T(2^2) = 2[2 + 2] + 2^2.2 = 2^2 + 2^2 + 2^2.2$
$n = 8,\ \ T(2^3) = 2^3 + 2^3 + 2^3.2 + 2^3.3$
$n = 16,\ \ T(2^4) = 2^4 + 2^4 + 2^4.2 + 2^4.3 + 2^4.4 = = = 2^4 + 2^4(1 + 2 + 3 + 4)$
.
.
$T(2^k) = 2^k + 2^k(1 + 2 + 3 + ... + k)$
=> $T(n) = n(1 + \ logn(logn + 1)/2)$
=> $T(n) \in \Theta(nlog^2n)$

a2) $T(n) = 2T(n/4) + \sqrt{n}/\ log_4\, n$
$n = 4^k$

Solving using forward iteration,
$n = 1,\ \ T(4^0) = 1$
$n = 4,\ \ T(4) = 2.1 + 2/1$
$n = 16,\ \ T(4) = 2[2 + 2] + 2^2/2 = 2^2 + 2^2 + 2^2/2$
$n = 64,\ \ T(4^3) = 2^3 + 2^3 + 2^3/2 + 2^3/3$
$n = 256,\ \ T(4^4) = 2^4 + 2^4 + 2^3/2 + 2^3/3 + 2^4/4 \quad = 2^4 + 2^4(1 + 1/2 + 1/3 + 1/4)$
.
.
$T(4^k) = 2^k + 2^k + 2^k(1/2 + ... + 1/k)$
$$T(4^k) = 2^k + 2^k + 2^k \sum_{i=2}^{k} 1/i$$

$T(4^k) \ = \ 2^{k+1} \ + 2^k(ln\, k \ - \ ln\, 1)$
$T(n) \ = \ 2\sqrt{n} \ + \sqrt{n}(ln\, log_4 n \ )$
$T(n) \ = \ 2\sqrt{n} \ + \sqrt{n}(ln\, \frac{logn}{2} \ )$
$T(n) \ = \ 2\sqrt{n} \ + \sqrt{n}(ln\, log\, \sqrt{n}\ )$

=> $T(n) \in \Theta(\sqrt{n}(ln\, log\sqrt{n})$

(b) The recurrence in lecture 6, page 7. Use T(1)=0.

$T(n) = \ \frac{n+1}{n} T\ (n-1) \ + \ c\frac{2n-1}{n}$

$T(1) = 0$
$T(2) = \frac{3}{2}0 \ + \ \frac{3}{2}c \ = \ \frac{3}{2}c$
$T(3) = \frac{4}{3}.\frac{3}{2}c \ + \ \frac{5}{3}c \ = 4c\left[\frac{3}{2.3} + \frac{5}{3.4}\right]$
$T(4) \ = \frac{5}{4}\left[4c\left[\frac{3}{2.3} + \frac{5}{3.4}\right]\ \right] + \frac{7}{4}c \ = 5c\left[\frac{3}{2.3} + \frac{5}{3.4}\right] \ + \frac{7}{4}c \ = 5c\left[\frac{3}{2.3} + \frac{5}{3.4} + \frac{7}{4.5}\right]$
.
.
.
$T(n) = c.(n+1)\sum_{i=2}^{n} \frac{2i-1}{i(i+1)}$

(c) Assume c>1. T(8)=...=T(0)=c, T(*n*) = 3T(*n*-2) otherwise. Give the exact solution and the asymptotic big-theta bound for T(n). Justify your answers.

Solution:
$T(n) = \{c\ ,\ n = 0, 1, .., 8\}$
3T(n-2) , otherwise
T(0)=c
.
.
T(8)=c
T(9)=3c
T(10) = 3c
T(11)= 3.3.c
T(12) =3.3.c
T(13)=3.3.3.c
T(14)=3.3.3.c

.

.

$T(n) = 3^{\lceil n/2 \rceil - 4}$

=> $T(n) \in \Theta(3^n)$, for n ≥ 9

3. Purpose: Often, recursive function calls use up precious stack space and might lead to stack overflow errors. Tail call optimization is one method to avoid this problem by replacing certain recursive calls with an iterative control structure. Learn how this technique can be applied to QUICKSORT. (2 points each) Solve Problem 7-4, a-c on page 188 of our textbook.

```
TAIL-RECURSIVE-QUICKSORT(A, p, r)
1. while p < r
2.        // Partition and sort left subarray
3         q = PARTITION(A, p, r)
4.        TAIL-RECURSIVE-QUICKSORT(A, p, q-1)
5.        p = q + 1
```

a. Argue that TAIL-RECURSIVE-QUICKSORT (A,1, A.length) correctly sorts the array A

The correctness of quicksort has been proven and can be used to prove the correctness of tail recursive quicksort.

Let us look at quicksort. -

```
QUICKSORT(A, p, r)
1. while p < r
2.        // Partition and sort left subarray
3         q = PARTITION(A, p, r)
4.        QUICKSORT(A, p, q-1)
5.        QUICKSORT(A, q+1, r)
```

Tail recursive quicksort and quicksort do partition, and call themselves recursively with A, p,q-1. But, we see that the difference in the 2 algorithms is in the last line.

While quicksort calls itself recursively on the right subarray with arguments A,q+1, r, Tail recursive quicksort assigns p=q+1 and on the next iteration of the while loop, TAIL-RECURSIVE-QUICKSORT will enable it to sort the right subarray of the original array i.e. it will sort A[q+1 ... r] which is essentially the same thing. This just helps prevent the second recursive call of quicksort.

Since we know that quicksort is correct, we can say the tail recursive quicksort is also correct as they are essentially the same and the difference lies in the way the stack is filled and popped off.

b. Describe a scenario in which TAIL-RECURSIVE-QUICKSORT ' stack depth is theta(n) on an n-element input array.

Stack depth depends on the number of times the **TAIL-RECURSIVE-QUICKSORT** function on line 4 is called. When we consider the scenario of the array already being sorted, the pivot returned will be the last element i.e q=r. Therefore, in each recursive call, pivot will be q=r, and the length of the array will only reduce by 1. The size of the left subarray will be n−1 the first time, the nn−2 the second time, and so on. The right subarray will always have size 0 so there will be n−1 recursive calls before the while-condition p<r is violated. Therefore the stack depth will be theta(n).

c. Modify the code for TAIL-RECURSIVE-QUICKSORT so that the worst-case stack depth is theta(lg n). Maintain the O(n lg n) expected running time of the algorithm.

In order to ensure that the worst case scenario stack depth is $\Theta(logn)$, we have to make sure that the size of the array has been reduced by at least half. Instead of calling the left subarray recursively, we can perform recursive calls on the smaller of the two subarrays. In this way, the size of the recursive subproblem is always <= length/2, and thus will limit the number so stacked up recursive calls, reducing stack depth. If we look at the recursive calls to sort the smaller subproblem as a binary tree, we see that the number of recursive calls to sort the subproblems is the height of the tree which is logn. Therefore the stack depth will be $\Theta(logn)$ in the worst case.

```
MODIFIED-TAIL-RECURSIVE-QUICKSORT(A, p, r):
1. while p < r:
2. q = PARTITION(A, p, r)
3. if q - p < r - q:
4.        MODIFIED-TAIL-RECURSIVE-QUICKSORT(A, p, q-1)
5.        p = q+1
6.else:
7.        MODIFIED-TAIL-RECURSIVE-QUICKSORT(A, q+1, r)
```

```
8.      r = q - 1
```

O(nlgn) expected running time is maintained in MODIFIED-TAIL-RECURSIVE-QUICKSORT, as we call the recursive function and update indices based on a condition, while using the underlying algorithm of TAIL-RECURSIVE-QUICKSORT.

4. (5 points) Purpose: Practice algorithm design and the use of data structures. Design an algorithm that outputs the time-stamps in the correct order and uses only a constant amount of storage, i.e., the memory used should be independent of the number of time-stamps processed. Solve the problem using a heap.

Algorithm:

To output the time-stamps in the correct order, we use a min-heap.

1. Since the time-stamps are at most hundred positions away from its correctly sorted position, we initially build a heap of size 101, with the first 101 time-stamps of the stream.
2. We pop the root of the heap, which will give us the minimum timestamp. We also insert the next element of the stream to the heap.
3. Heapify is called and the timestamp is inserted at the right position.
4. Repeat steps 2-4 for the rest of the timestamps in the stream.

Pseudocode -

```
1.  H = minHeap()  // Initializing min heap
2.  for timestamp=1 to 101 from stream :
3.          H.insert(timestamp)  //building a min heap
4.  for timestamp=101 till end of stream :
5.          min_element = H.Extract-Min()
6.          Print min_element
7.          H.Insert(timestamp)
8.  while(not H.Empty()):
9.          min_element = H.Extract-Min()
10.         Print min_element
```

- **minHeap()** is a standard heap procedure which initializes H as a min heap using a constructor which returns a reference to a min heap.

- **Extract-Min()** returns and deletes the minimum element of the heap.
- **Insert()** inserts an element to the heap.
- **Empty()** checks whether the heap is empty or not.

Example -

Let us assume that the timestamp is at most 3 positions away from the correct sorted position.

Let the stream of number be: 4,21,1,34,67, 9

- Build a min heap of size 4.
  H=[1,21,4,34]
  Output :
- Remove 1 from heap and print it. Also add 67 to the heap.
  H=[4,21,34, 67]
  Output : 1
- Remove 4 from heap and print it. Also add 9 to the heap.
  H=[9,21,34, 67]
  Output : 1 4
- Remove 9 from heap and print it.
  H=[21,34, 67]
  Output : 1 4 9
- Remove 21 from heap and print it.
  H=[34, 67]
  Output : 1 4 9 21
- Remove 34 from heap and print it.
  H=[67]
  Output : 1 4 9 21 34
- Remove 67 from heap and print it.
  H=[]
  Output : 1 4 9 21 34 67

Proof of Correctness:

loop invariant: All elements in the output are smaller than elements in heap and stream elements.

Initialisation:
Initially there is no output and hence is trivially correct.

Maintenance:

Let us consider an element x in the output.
Since this element has been extracted from the root of the min-heap, x is the smallest element in 101 elements of the heapas a min heap always returns the smallest element in the heap.This implies that the printed elements are always smaller than the elements in the heap.

Considering the elements in the stream. We know that the root element which is extracted as the minimum. This element is said to be at most 100 positions away from is correct position. Since we consider the size of heap as 101, we account for the error and hence it will be the minimum in the heap (as it is the root) as well as the elements in the stream . Hence x is smaller than heap and stream elements.

Termination:
At the end of the loop, there are no elements in the stream or in the heap. All the elements are in the sorted order since we're always picking the minimum element from the heap.

Hence we can say loop invariant is holds and ensures that all elements are written in correct order.

Time Complexity Analysis:

- Buildheap -
  Build heap takes linear time i.e $O(n)$ to build a heap. We initially build a heap of 101 elements. Since the size of the heap remains constant and does not vary with n, we take constant time to build the heap. Therefore build heap takes O(1) time.
- Remove minimum-
  Takes constant time to remove the first element of the heap as the first element will always have the minimum element
- Insert and heapify -
  This takes O(1) time, as the size of the heap is constant

- Since, the stream has n elements, we loop through the n elements as shown in line 2-4 of pseudocode, therefore our algorithm has an asymptotic worst case time complexity of **O(1)+O(1)+O(n)**, which is **O(n).**