Homework 2 Solutions
CSI 503 Spring 2021

RAHUL BHENJALIA

March 8 2021

1 Problems

Note: When solving the problems, you must include your calculation and/or justification that shows how you derived the answers. You will not earn points if you simply provide the final answers without sufficient details.

1. We want to test if there is a repeated number in an array. A naive algorithm for solving the task would take $\Theta(n^2)$. [20pt.]

   (a) Design a *divide-and-conquer* algorithm for the task that has better time complexity than $\Theta(n^2)$. The main function in your pseudocode must be named Test-Repeat(A, p, r). Your function needs to return a boolean value that indicates the existence of a repeated number (true) or not (false). We will initially run Test-Repeat(A, 1, n). Hint: You can rename and update the Merge-Sort algorithm to achieve this (see Section 2.3.1 in the textbook). However, you are not allowed to call an external sorting algorithm from your pseudocode (such a solution will receive no points).

   .

   .

   .

   $ANSWER: 1(a)$

```
Test-Repeat(A, p, r)
     if( (r-p) > 0 ) then
          q =  floor( (p { r)/2 )
          n1 =  q - p + 1
          n2 = r-q
          let L(1 ....n1+1) and R(1 ....n2+1) be new arrays
          for i=1 to n1 do
              L[i] = A[p+i-1]
          for j=1 to n2 do
               R[i] = A[q+j]
```

```
         if(Test-Repeat(L, p, n1)) then
              return True
         if(Test-Repeat(R, n2, q)) then
              return True
          i=1
          j=1
          k=1
          while i<n1 and j<n2 do
              if L[i] = R[i] then
                  return True
              if L[i]<R[j] then
                  A[k] = L[i]
                  i=i+1
              else
                  A[k] = R[j]
                  j=j+1
                  k=k+1
          while i<n1 do
              A[k] = L[i]
              i=i+1
              k=k+1
          while j<n2 do
              A[k] = R[j]
              j=j+1
              k=k+1
    return False
```

(b) Derive the recurrence for your algorithm and determine its asymptotically tight bound in the worst case.

.

$ANSWER:1(b)$

.

.

Since, the algorithm is based on Merge-Sort, we can derive the recurrence relation based on that algorithm:

.

The array is divided in two after every iteration, we can write it as:

.

$T(n) = T(n/2) + T(n/2) + n$

$T(n) = 2T(n/2) + n$

.

Substituting the T(n):

$T(n) = 2(2T(n/4) + (n/2)) + n$

$T(n) = 4T(n/4) + n$

.
From the above equation we can basically formulate a generalized equation based on array length and progression:

.
$T(n) = 2^i T(n/2^i) + in$
.
Now, since we have a base case as n=1, evaluating T(1)

.
$1 = n/2^i$
$2^i = n$
$i = log_2 n$
.
Now, since we have a value of i we can put the values in original equation we got above and evaluate asymptotically tight bound worst case:

.
$T(n) = nT(1) + (log_2 n)n$
$T(n) = n + nlog_2 n$
Hence the asymptotically tight bound for the given algorithm is $O(nlog_2 n)$ which is better than $O(n^2)$

2. Consider a binary min-heap data structure that supports the Insert and Extract-Min operations. In summary, Insert adds the element at the left-most available position in the last level of the heap, and performs upward swaps as needed. Extract-Min extracts the root, replaces it with the rightmost element in the last level of the heap, and performs downward swaps as needed. (See Chapter 6 of the textbook if you need a refresher.) Insert and Extract-Min each take $O(\log n)$ in the worst case. [15pt.]

   (a) Show a *potential* function Φ that results in an amortized cost of $O(\log n)$ for Insert and $O(1)$ for Extract-Min.

   (b) Fill out the following table to derive the amortized cost of Insert and Extract-Min based on the potential function. Show your calculations.

Operation	Actual Cost	$\Delta\Phi$	Amortized Cost
Insert	?	?	?
Extract-Min	?	?	?

3. In the following randomized function, A is the input array with size n. Random(n) produces a uniformly random number between 1 and n.

   (a) Determine its asymptotic time complexity in the worst case.
   .
   .
   .

$ANSWER : 3(a)$

.

From the algorithm, we can clearly predict that only way this algorithm reaches its worst case time complexity is when $STEP : 4$ results in TRUE and $STEPS : 5to9$ are executed.

.

Let's assume that RANDOM(n) produces k which makes $STEP : 4$, TRUE.

.

In that case, (ignoring probabilisitc analysis for now), we can take time cost for $STEP : 3$ as 1

.

For and after $STEP : 5$, it would be n, since its a for loop from 1 to n

.

Now, for $STEP : 7,8,9$, the time cost would be $nlog_2n$, since they're inside the for loop and the while loop is iterating over j, which given the update condition iterates as $j = 1,2,4,8$.....

```
1. function RAND-ALG(A,n):
2.     s = A[1]                        COST: 1
3.     k=RANDOM(n)                     COST: 1
4.     if k<log_2{n} then              COST: 1
5.        for i=1 to n do              COST: n
6.            j=1                      COST: n
7.            while j<n do             COST: n * log_2{n}
8.                  s=s+A[i]*A[j]      COST: n * log_2{n}
9.                  j=j*2              COST: n * log_2{n}
```

.

Total Cost, $T(n) = 3 + 2n + 3nlog_2n$

.

Based on above time cost, we can say that asymptotic time complexity in the worst case for the algorithm is $O(nlog_2n)$

(b) Using probabilistic analysis, determine its asymptotic time complexity on average.

.

$ANSWER : 3(b)$

Now, for the given algorithm, the average analysis depends on $STEP : (3)$ and $STEP : (4)$, especially $STEP : (4)$, because if $STEP : (4)$ condition gets TRUE, the worst time case is considered and if its false it goes best which would be just O(1) and at cost $T(n) = 3$.

.

We can test out some iterations to get an idea of what kind probability we can expect based on n

.

$log_2(1) = 0$ TRUE=1 FALSE=0
(0), 1
$log_2(2) = 1$ TRUE=1/2 FALSE=1/2
(1), 2
$log_2(3) = 1.58$ TRUE=2/3 FALSE=1/3
1, (1.58), 2, 3
$log_2(4) = 2$ TRUE=2/4 FALSE=2/4
1, 2, (2), 3, 4
$log_2(5) = 2.32$ TRUE=2/5 FALSE=3/5
1, 2, (2.32), 3, 4, 5
$log_2(6) = 2.58$ TRUE=2/6 FALSE=4/6
1, 2, (2.58), 3, 4, 5, 6
$log_2(7) = 2.8$ TRUE=2/7 FALSE=5/7
1, 2, (2.58), 3, 4, 5, 6, 7
$log_2(8) = 3$ TRUE=3/8 FALSE=5/8
1, 2, 3, (3), 4, 5, 6, 7, 8
$log_2(9) = 3.16$ TRUE=3/9 FALSE=6/9
1, 2, 3, (3.16), 4, 5, 6, 7, 8, 9
$log_2(10) = 3.32$ TRUE=3/10 FALSE=7/10
1, 2, 3, (3.16), 4, 5, 6, 7, 8, 9, 10
.
.
Based on above test iterations, we can see a pattern, where the probability of TRUE of $STEP:(4)$ condition to be TRUE is always more than it to be FALSE
.
We can create an arbitrary case where we can take probability of TRUE to be 2/3 and FALSE to be 1/3 and do a probabilistic analysis based on that.
.
Hence, the probability of Best-Case time complexity kicking in is 1/3 and Worst-Case time complexity would be 2/3
.
$T(n) = 1/3(Best - CaseTimeCost) + 2/3(Worst - CaseTimeCost)$
.
$T(n) = 1/3(3) + 2/3(3 + 2n + 3nlog_2 n)$
.
$T(n) = 1 + 2 + 4n/3 + 2nlog_2 n$
.
$T(n) = 3 + 4n/3 + 2nlog_2 n$
.
Hence, asymptotic time complexity on average for the algorithm would be $\theta(nlog_2 n)$

4. Suppose that we have a data structure that holds a list of numbers (internally stored as an array). The data structure only supports the append()

operation which by default stores the given number at the end of the array. However, every time that the size of the array including the appended item becomes an exact power of 2 it performs a circular right shift on the whole array (right after storing the appended number at the end of the array). It moves final entry to the first position while shifting all other numbers to the next position.

(a) Using the *aggregate analysis* method, show the average cost of each append() operation.

(b) Repeat the above using the *accounting method*. Clearly describe the costs, payments, and credits.