CS-510 Problem Set 2

Hassan Ali Khan Student ID: 15030044

May 14, 2017

Problem 1. Suppose you are given an array $A[1...n]$ of sorted integers that has been circularly shifted k positions to the right. For example, $[35, 42, 5, 15, 27, 29]$ is a sorted array that has been shifted $k = 2$ positions. We can obviously find the largest element in $O(n)$ time. Describe an $O(\log n)$ time algorithm to find the maximum in A.

Algorithm 1

```
procedure FindMax($A, i1, i2$)        ▷ Find the max in sorted and shifted array A[1..n],i1 =1,i2=n
    if $A[i1] > A[i2]$ then
        return $A[i1]$
    else

        if $A[i2/2] < A[i2]$ then
            FindMax($A, i1, i2/2$)
        else
            FindMax($A, i2/2, i2$)
```

Problem 2. Given a sorted array of distinct integers $A[1...n]$, you want to find out whether there is an index i for which $A[i] = i$. Give a divide and conquer algorithm that runs in $O(\log n)$ time to find out an index i if it exists.

Algorithm 2

```
procedure FindInd($A, start, end$)          ▷ Find the max in sorted and shifted array A[1..n],mid=n/2
    $mid = (start + end)/2$
    if $start = mid$ then

        if $start = mid$ then
            return $mid$
        else
            return $-1$
    else
        if $A[mid] < mid$ then
            FindInd($A, mid, end$)
        else
            FindInd($A, start, mid$)
```

Problem 3. Suppose you are choosing between the following 3 algorithms:

- Algorithm A solves the problem of size n by dividing it into 5 subproblems of size $n/2$, recursively solving each subproblem, and then combining the solutions in linear time.
- Algorithm B solves the problem of size n by recursively solving two subproblems of size $n-1$ and then combining the solutions in constant time
- Algorithm C solves the problem of size n by dividing it into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

What are the running times of each algorithm and which would you choose and why?

Problem 4. You are given an array of n elements, and you notice that some of the elements are duplicates; that is, they appear more than once in the array. Show how to remove all duplicates from the array in $O(n \log n)$ time.

Algorithm 3

```
$repCount = 0$
**procedure** RemoveDuplicate($A, low, high$)          ▷ Find the max in sorted and shifted array
A[1..n],mid=n/2
    **if** $low < high$ **then**
        $mid = (low + high)/2$
        RemoveDuplicate($A, mid + 1, high$)
        FindInd($A, low, mid$)
        Merge($A, low, mid, high$)
    **else**
        Return

**procedure** Merge($A, low, mid, high$)
    **for** $(l1 = low, l2 = mid + 1, i = low; l1 <= mid, l2 < high; i + +)$ **do**

        **if** $A[l1] < A[l2]$ **then**
            $b[i] = A[l1 + +]$
        **if** $A[l1] > A[l2]$ **then**
            $b[i] = A[l2 + +]$
    $tempInd = 0$
    **for** $(i = low; l1 <= high; i + +)$ **do**
        **if** $A[i + 1] = A[i]$ **then**
            $repCount + +;$
            $A[A.length - repCount] = b[i]$
        **else**
            $A[tempInd + +] = b[i]$
```

Problem 5. Assume you have an array $A[1..n]$ of n elements. A majority element of A is any element occurring in more than $n/2$ positions (so if $n = 6$ or $n = 7$, any majority element will occur in at least 4 positions). Assume that elements cannot be ordered or sorted, but can be compared for equality. (You might think of the elements as chips, and there is a tester that can be used to determine whether or not two chips are identical.) Design an efficient divide and conquer algorithm to find a majority element in A (or determine that no majority element exists). Aim for an algorithm that does $O(n \log n)$ equality comparisons between the elements. A more difficult $O(n)$ algorithm is possible, but may be difficult to find.

Algorithm 4

```
procedure FindMajor($A$)
    $B[Max(A)] = 0$
    for $(i = 1; l1 <= n; i++)$ do
        $B[A[i]++$
    $MAX = 0, MAXind = -1$
    for $(i = 1; l1 <= n; i++)$ do
        if $B[i] > MAX$ then
            $MAX = B[i]$
            $MAXind = i$
    return $MAXind$;
```

Problem 6. Given the following recurrence relations, find the running time (in big-O notation) using the recursion tree method, substitution method and the Master Theorem.

- $$T(n) = \begin{cases} 8T\left(\frac{n}{2}\right) + \Theta(n^2) & \text{if } n \geq 2 \\ 1 & \text{else} \end{cases}$$
- $$T(n) = \begin{cases} 3T\left(\frac{n}{4}\right) + n \log n & \text{if } n \geq 2 \\ 1 & \text{else} \end{cases}$$
- $$T(n) = \begin{cases} 2T\left(\frac{n}{4}\right) + \sqrt{n} & \text{if } n \geq 2 \\ 1 & \text{else} \end{cases}$$

Answer:
1-Substitution:
= 8 T(n/2) + n^2
$= 8(8T(n/4) + (n^2)/4) + n^2$
$= 8 * log(n) + 8(n^2) * log(n) + n^2$

$2 - Substitution :$
$= 3T(n/4) + n * log(n)$
$= 3(3T(n/8) + (n/2) * log(n/2) + n * log(n)$
$= (3 * log(n))/2 + (3(n) * log(n))/2 + n * log(n)$

Problem 7. Suppose you have an unsorted array A of all integers in the range 0 to n except for one integer, denoted the *missing number*. Assume $n = 2^k - 1$. Design a $O(n)$ Divide and Conquer algorithm to find the missing number. Argue (informally) that your algorithm is correct and analyze its running time and space requirements.

Algorithm 5

```
procedure FINDMISSING($A[n-1]$)
    $B[n] = 0$
    for ($i = 1; i <= n - 1; i++$) do
        $B[A[i]] = 1$
    for ($i = 1; i <= n; i++$) do
        if $B[i] == 0$ then
            return $B[i]$;
```

The run-time of above given algorithm is
= 1 + (4(n-1)) + 4(n) + 1
= 8n - 2

Problem 8. Devise a Divide and Conquer procedure for computing the kth largest integer in an array of integers. Analyse the asymptotic time complexity of your algorithm. (*Hint*: Use the Partition procedure discussed in class)

Algorithm 6

```
procedure FindKmax($A$,$n$,$k$)
    if $n <= k$ then
        $A = Sort(A)$;
        $returnA[1]$;
    $losers[n/2] = 0$;
    $winners[n/2] = 0$;
    $loserInd = 0$;
    $winnerInd = 0$;
    for $(i = 1; l1 <= n; i+ = 2)$ do
        if $A[i] > A[i+1]$ then
            $losers[loserInd + +] = A[i+1]$;
            $winners[winnerInd + +] = A[i]$;
        if $A[i] < A[i+1]$ then
            $losers[loserInd + +] = A[i]$;
            $winners[winnerInd + +] = A[i+1]$;
    $FindKmax(winners, n/2, k)$;
```

The run-time of above given algorithm is
= K*log(k) + 8*log(n)

Problem 9. Given a binary string of type$\{1^m 0^n\}$, devise an algorithm that finds the number of zeroes in $O(\log n)$ time.

Algorithm 7

```
$l = 1, h = n, An = n$
procedure BinarySearch($A$,$An$,$l$,,$h$)
    if $A[(l+h)/2 == 1A[(l+h)/2+1 == 0$ then
        $returnAn - (l+h)/2+1$
    if $A[(l+h)/2 == 0A[(l+h)/2+1 == 0$ then
        $BinarySearch(A, An, l, (l+h)/2)$
    if $A[(l+h)/2 == 1A[(l+h)/2+1 == 1$ then
        $BinarySearch(A, An, (l+h)/2, h)$
```

Problem 10. Flavius Josephus was a famous historian of the first century. During the Jewish-Roman war, he was captured along with 40 other jewish rebels by the Romans. The rebels preferred suicide to capture and so decided that they would form a circle and kill every third person in the circle until no one was left. Josephus and one of his friend didn't want suicide so Josephus quickly calculated where he and his friend should stand in the circle so that they would be the last one's left.

- Calculate where Josephus and his friend would have stood.
- Come up with a more general solution where instead of 41, there are n people in the circle.

Problem 11. The Tower of Hanoi is a classic puzzle invented by the French Mathematician Edouard Lucas in 1883. In this puzzle there are 8 disks, initially stacked in decreasing order of size one of three pegs. We have to move all of the disks to another peg while following these two rules

- Only one disk can be moved at a time
- A larger disk can't be put on top of a smaller disk

Design an algorithm to solve this problem and analyze the running time of your algorithm.
Hint : Think how you can solve the problem if there were only 2 disks instead of 8 and see if you can build from that.

Algorithm 8

```
procedure MoveTower($disk, source, dest, spare$)
    if $disk == 0$ then
        $movediskfromsourcetodest$
    if $disk != 0$ then
        $MoveTower(disk - 1, source, spare, dest)$
        $movediskfromsourcetodest$
        $MoveTower(disk - 1, spare, dest, source)$
```

Problem 12. [**K-way Merge**] Suppose you have k sorted arrays $A_1, A_2, ...A_k$ each with n elements. You want to combine them into a single sorted array of size kn. One way to do this would be to use the merge operation we discussed in class. First merge arrays A_1, A_2 then merge the result with A_3 and so on

- Figure out how many steps this algorithm would take.
- Design a better algorithm for this problem.

Answer:

- $2n + 3n + 4n + ...(k-1)n = n(2 + 3 + 4 + ...k) = n(k^2)$

Algorithm 9 :

```
• function MERGE($A[k][n], k, n$)
      if $k == 0$ then
          return 0
          for ($i = 1; i <= k; i+ = 2$) do
              $Merge2in1(A[i], A[i+1], A[i], n);$
          $Merge(A[][], k/2, n * 2);$
```

Problem 13. [Fibonacci Numbers]
Given below is a recursive algorithm for finding the n_{th} fibonacci number

Algorithm 10 :

```
function FIB(n)
    if $n == 0$ or $n == 1$ then
        return 1
    else
        return FIB(n-1) + FIB(n-2)
=0
```

- Analyze the running time of this algorithm
- A lot of computation seems to be repeated over and over in this algorithm. For example $fib(4)$ would compute $fib(3)$ and $fib(2)$ and then the called $fib(3)$ would also compute $fib(2)$. Is there a way we could avoid computing the same thing over and over again while still using a recursive approach?

$T(n) = T(n-1) + T(n-2) + O(1)$
$T(n) = 2^n + 2^n + O(1)$

$If we use bottom up approach we can save the extra work, as Fibonacci of numbers less than current n will be calculated only once$

Problem 14. [GCD]
We studied the following recursive alogrithm for finding the gcd of two numbers p and q in discrete mathematics.

Algorithm 11 :

```
function GCD(p,q)
    if $q == 0$ then
        return p
    else
        return GCD(q, $p \mod q$)
```

It turns out that the worst case input for this algorithm is $p = fib(n)$, $q = fib(n-1)$. Analyze the running time of the algorithm for this input.

Problem 15. In each row of the following table, write whether $f = O(g)$, or $f = \Omega(g)$, or both i.e. $f = \Theta(g)$

$f(n)$	$g(n)$	Your Answer
100	17	
10^{200}	$n - 10^{200}$	
$n^2 \log 300$	$n \log n$	
n^{100}	2^n	
$n \log n$	$n - 100$	
$\sqrt{n}$	$\log n$	
$n^{1.01}$	$n + 100$	
$2 \log n$	$\log(n^2)$	
$\log(n^2)$	$(\log n)^2$	

Problem 16. Order the following functions of n from the smallest to largest complexity. If two have the same complexity put them in one line next to each other.

$$n \log n, \quad (n-5)(n), \quad 10 \log 300, \quad n + 7\log(n^2),$$

$$2^n, \quad 10n^2 + 15n, \quad 17, \quad 15n^3 + n\log n, \quad \frac{3n^8}{n^6}$$

Problem 17. [Complexity Classes]

- Order the following functions of n from the smallest to largest complexity. If two have the same complexity put them in one line next to each other.

$$10^{200}, \quad n^2 \log 300, \quad 2\log_{10} 10^{200}, \quad n^2 + 7\log(n^2),$$

$$3^{n^2}, \quad n^2 + \sqrt{n}, \quad n^2 \log n, \quad n^4 + n^2 + n^2 \log n, \quad n^4$$

- Briefly explain next to each line why these functions are in the same complexity class.

Problem 18. [Find Max and Min]
Show that $\frac{3n}{2} - 2$ comparisons are necessary in the worst case to find both the minimum and maximum of A.
Hint: Count how many numbers are potentially either the maximum or minimum, and see how a comparison affects these counts. Note that at the end of the algorithm both these counts should be exactly 1.

Problem 19. Inversions
Suppose that you somehow know that the number of inversions in an array of size N is 10 where $10 \ll N$. Which of the following sorting algorithms should you use to sort this array completely

- Merge Sort
- Selection Sort
- Insertion Sort

Give an appropriate reason for your choice.

When little number of inversions the Insertion sort will perform in the memory operations better because of very few swaps. The merge and selection sort are independent of the number of inversions in an array as they don not consider the presented structure of array when memory operations are concerned, so, we will not see any improvement in later two sorts even if array is partially sorted