Basic Algorithms CSCI-UA.0310

Additional Problems

Remark: For most of the problems, only a pointer to the solution is provided. You need to complete the solutions on your own.

Problem 1. *Complete the following divide & conquer algorithm to determine if the n elements of the array $A[1 \dots n]$ are all equal. The initial call is* ALLEQUAL*(A,1,n).*
Note: There is much easier algorithm to do this! The following algorithm is just an example of the divide & conquer technique.

```
allEqual(A, i, j)
    If i == j   Return TRUE
    If A[i] != A[j]   Return FALSE
    ...
    ...
```

Write a recursion for the time complexity of your algorithm and solve it to obtain the worst-case asymptotic time complexity for your algorithm. You do NOT need to prove your result!

Here is a complete version:

```
allEqual(A, i, j)
    If i == j   Return TRUE
    If A[i] != A[j]   Return FALSE
    m = (i + j)/2
    Return allEqual(A, i, m) && allEqual(A, m + 1, j)
```

For the time complexity, try to find the time complexity of each line, then you will obtain the following recursion for the time complexity $T(n)$:

$$T(n) \leq 2T(n/2) + O(1)$$

Draw the recursion tree for the worst case, then you get that $T(n) = \Theta(n)$.

Problem 2. *For each of the following functions, indicate the most accurate asymptotic bound that $f(n)$ satisfies among the following options.*

(a) $O(g(n))$

(b) $\Omega(g(n))$

(c) Both (i.e., $\Theta(g(n))$*)*

If $f(n) = O(g(n))$, then find the constant $c > 0$ and the positive integer n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. Similarly, if $f(n) = \Omega(g(n))$, find c and n_0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

- $f(n) = 3n^2 \qquad g(n) = n^2$
- $f(n) = 2n^4 - 3n^2 + 7 \qquad g(n) = n^5$
- $f(n) = \frac{\log_2 n}{n} \qquad g(n) = \frac{1}{n}$
- $f(n) = \log_2 n \qquad g(n) = \log_2 n + \frac{1}{n}$
- $f(n) = 2^{k \log_2 n} \qquad g(n) = n^k$

- $f(n) = 2^n \qquad g(n) = 2^{2n}$

- Θ: $c_\Omega = c_O = 3, n_0 = 1$
- O: $c_O = 2, n_0 = 7$
- Ω: $c_\Omega = 1, n_0 = 2$
- Θ: $c_\Omega = 1/3, c_O = 1, n_0 = 2$
- Θ: $c_\Omega = c_O = 1, n_0 = 2$
- O: $c_O = 1, n_0 = 1$

Problem 3. *(a) Given an array A of $2n$ distinct elements, we have seen a naive algorithm to find the minimum element using $2n - 1$ comparisons. Write the pseudo-code!*

(b) Similarly, we can find the maximum element using $2n - 1$ comparisons. So you can simply merge the two algorithms to find the maximum and minimum elements using $2(2n - 1)$ comparisons. Write down the pseudo-code for this algorithm.

(c) Develop an algorithm that finds both the minimum and maximum elements using at most $3n - 2$ comparisons.

(c) POINTER TO THE SOLUTION: Group $2n$ elements into n pairs and compare elements within each pair. Within each pair, call the bigger element the "winner" and the smaller one the "loser". We get n winners and n losers. The maximum element is among the winners and the minimum is among the losers. Then, using Part (a), we need $n - 1$ comparisons to find the maximum element among the winners and $n - 1$ comparisons to find the minimum element among the losers. This way, we get the maximum and minimum elements using

$$n + (n - 1) + (n - 1) = 3n - 2$$

comparisons.

Problem 4. *(a) Given an array $A[1 \dots n]$ of n distinct integers, give an $O(n^2)$ algorithm to find the number of pairs (x, y) such that $x < y$.*

(b) Improve the former algorithm to an $O(1)$ algorithm.

(a) Find every possible pair and check whether it satisfies the given condition. Count the number of such pairs. Write the pseudo-code!

(b) Note that the smallest element in the array appears in $n - 1$ such pairs, the second smallest element appears in $n - 2$ such pairs, etc.
So, in total, there are $\sum_{i=1}^{n-1} i = n(n - 1)/2$ such pairs. You only need to return this number!
(We do not even need this reasoning; There are $\binom{n}{2} = n(n - 1)/2$ such pairs!)

Problem 5. *Let $A = \{a_1, a_2, \dots, a_n\}$ be a set of n positive integers. You may assume that all basic arithmetic operations, i.e., addition and multiplication, and comparisons, can be executed in $O(1)$ time.*

(a) Develop an $O(n \log n)$ algorithm to check whether for all subsets $T \subset A$, the sum of all elements in T is at least $|T|^2$. ($|T|$ stands for the cardinality of T)

(b) *Now suppose that in addition to A and n, you are also given another integer $k \leq n$. Give a more efficient algorithm to check whether the former statement holds* **only** *for all subsets $T \subset A$ of cardinality k.*

(a) Sort A in $O(n \log n)$ time using MERGE SORT. Let $a'_1 \leq a'_2 \leq \cdots \leq a'_n$ be the elements of A in the sorted order. For each $i = 1, \ldots, n$, check whether $\sum_{j=1}^{i} a'_j \geq i^2$. By maintaining a running sum, checking this sum for each value of i requires only one addition and one comparison operation. Thus, the total running time is $O(n \log n) + O(n) = O(n \log n)$.

(b) Find all the k^{th} smallest elements of A in $O(n)$ time (HW4 P4). We only need to check that the sum of these k elements is at least k^2. The total run time is $O(n) + O(k) = O(n)$. (Note that $k \leq n$)

Problem 6. *Given an array A of n integers, develop an $O(n \log n)$ algorithm to check whether the elements of A are all distinct. Why does your algorithm run in $O(n \log n)$ time?*

Idea: Sort the array A. Compare consecutive elements to see if any element is repeated. If so, the elements are not distinct.
Algorithm:

```
MergeSort(A[1...n])
i = 1
While i < n
    If A[i] ≠ A[i+1]    i = i + 1
    Else   Return FALSE
Return TRUE
```

This algorithm takes $O(n \log n)$ time in total: The initial sorting takes $O(n \log n)$ time. The **While** loop is executed at most $n - 1$ times and each iteration takes $O(1)$ time. All other steps take $O(1)$ time.

Problem 7. *Let $A_1, A_2, \ldots, A_k$ be k sorted arrays each with n elements. Develop an $\Theta(nk \log k)$ algorithm to combine them into a single sorted array of kn elements. (Assume k is a power of 2)*

Merge them pairwise: A_i with A_{i+1} for $i = 1, 3, \ldots, k-1$. Assuming that merging two arrays of size n takes $\Theta(n)$ time, the $k/2$ merges take $\Theta(nk)$ time.
In the next step, merge pairwise the resulting $k/2$ arrays each with $2n$ elements. These $k/4$ merges take $\Theta(2n \cdot k/4) = \Theta(nk)$ time.
Repeat this process until there is only one array of kn elements.
There are $\log_2 k$ steps and each step takes $\Theta(nk)$ time, giving the total running time of $\Theta(nk \log k)$.

Problem 8. *Given an array $A[1 \ldots n]$ of n distinct positive integers and another integer t, develop an $O(n \log n)$ algorithm that determines whether there exist two elements in A such that their sum is exactly t. Justify the running time of your algorithm.*

One naive solution is to try all possible pairs of elements of A and check if their sum equals t. This requires $O(n^2)$ time.
As a faster algorithm, first sort the array A and then search for the desired pair by comparing t with the sum of the minimum and the maximum elements of A, and discarding either the minimum element or the maximum element depending on the result of the comparison. Here is the algorithm:

```
FindSum(A[1...n], t)
    MergeSort(A[1...n])
    i = 1,  j = n
    While j > i
        If A[i] + A[j] = t    Return TRUE
        If A[i] + A[j] < t    i = i + 1
        If A[i] + A[j] > t    j = j - 1
    Return FALSE
```

To analyze the running time, note that the **While** loop is iterated at most n times since at each iteration, either the algorithm stops or the difference $j-i$ decreases by 1. Each iteration of the **While** loop takes $O(1)$ time, so the total running time of the **While** loop is $O(n)$. Therefore, the total running time of this algorithm is $\Theta(n \log n)$ since the sorting step with MERGESORT takes $\Theta(n \log n)$, which dominates the $O(n)$ running time of the **While** loop.

Problem 9. *We want to make change for n cents using the least number of coins among 1, 10, 25 cents. Develop an $O(n)$-time dynamic programming algorithm to find the least number of coins needed. Compute the total running time of your algorithm.*

For $i = 0, \dots, n$, let LEASTCOINS(i) denote the least number of coins required to make change for i cents. We have [Why?]

$$LeastCoins(i) = \begin{cases} 0 & \text{if } i = 0 \\ LeastCoins(i-1)+1 & \text{if } 1 \le i \le 9 \\ \min(LeastCoins(i-1)+1,\ LeastCoins(i-10)+1) & \text{if } 10 \le i \le 24 \\ \min(LeastCoins(i-1)+1,\ LeastCoins(i-10)+1,\ LeastCoins(i-25)+1) & \text{if } i \ge 25 \end{cases}$$

We have $n+1$ subproblems to solve (i.e., LEASTCOINS$(0), \dots,$ LEASTCOINS(n)), and each takes a constant time to be solved. So the total running time is $\Theta(n)$.
Exercise: Try to simplify the recursion above.
Exercise: Write the pseudo-code for the bottom-up DP approach.
Note: We will develop a simpler greedy algorithm in HW7.

Problem 10. *Given an array $A[1 \dots n]$ of n positive integers and a positive integer t, develop a dynamic programming algorithm to determine if there is a subsequence of A with sum equal to t.*

For $i = 1, \dots, n$, and $s = 1, \dots, t$, define the boolean value $r(i, s)$ as true if there is a subsequence of the first i elements of A, i.e., $A[1 \dots i]$, with sum equal to s, and define it false otherwise. Thus, the value of $r(i, s)$ is either true or false.
To obtain a recursion for $r(i, s)$, note that we have two options: the i^{th} element of A, i.e. $A[i]$, can be included in our subsequence or it cannot be included.
In the former case, the recursion we get is $r(i, s) = r(i-1, s-A[i])$ if $s > A[i]$ (Why?), and in the latter case, the recursion we get is $r(i, s) = r(i-1, s)$ (Why?). Thus, $r(i, s)$ is true if at least one of $r(i-1, s-A[i])$ or $r(i-1, s)$ is true.
Exercise: Write the pseudo-code for the bottom-up DP approach. Note that this is a two-dimensional recursion.

Problem 11. *Given two strings $X[1 \dots m]$ and $Y[1 \dots n]$, find the length of the shortest string that has both of them as subsequences (it is called a shortest superstring of X and Y).*

The idea of the solution is similar to *the longest common subsequence problem*. Let $r(i, j)$ denote the length of the shortest superstring of the first i characters of X and the first j characters of Y, i.e., $X[1 \dots i]$ and $Y[1 \dots j]$. We have [Why?]

$$r(i,j) = \begin{cases} j & \text{if } i = 0 \\ i & \text{if } j = 0 \\ r(i-1, j-1)+1 & \text{if } i, j > 0 \ \&\ X[i] = Y[j] \\ \min(\ r(i, j-1)+1, \quad r(i-1, j)+1\ ) & \text{otherwise} \end{cases}$$

Exercise: Write the pseudo-code for the bottom-up DP approach.