Homework #2
Introduction to Algorithms/Algorithms 1
600.363/463
Spring 2017

Due on: Tuesday, February 14th, 5pm
Late submissions: will NOT be accepted
Format: Please start each problem on a new page.
Where to submit: On Gradescope, a single PDF file.
Please type your answers; handwritten assignments will not be accepted.
To get full credit, your answers must be explained clearly,
with enough details and rigorous proofs.

March 6, 2017

1 Problem 1 (20 points)

1.1 (10 points)

Give tight asymptotic bounds (Θ) for $T(n)$ in each of the following recurrences. If you cannot provide tight bounds, provide upper and lower bounds, making them as tight as possible. Assume that $T(n)$ is a constant for $n \leq 8$ or other appropriately chosen small constant. Provide a short proof or justification of your answer. (Applying the master theorem is a proof.)

- $T(n) = T(3n/4) + 2n \log n - 4$
  Answer: Here $a = 1$, $b = 4/3$, $f(n) = 2n \log n - 4$. We know $\log_b a = \log_{4/3} 1 < 1$. Since $f(n) = \Omega(n^c)$ where $c = 1$. Thus the case 3 of the master theorem applies here. $T(n) = \Theta(n \log n)$.

- $T(n) = 4T(n/2) + n^2 \log_{10} n + 10n \log n$
  Answer: Using the Master Theorem with $a = 4$, $b = 2$, and we have $n^{\log_b a} = n^2$. Since $f(n) = \Theta(n^2 \log n)$, we fall in the case 2 extension with $k = 1$, and we conclude that $T(n) = \Theta(n^2 \log^2 n)$.

- $T(n) = 4T(n/3) + n\log^2 n$
  Answer: Using the Master Theorem with $a = 4$ and $b = 3$, we have $n^{\log_b a} = n^{\log_3 4} \approx n^{1.26}$. We have $f(n) = n\log^2 n = O(n^{1.26-\epsilon})$. Thus the case 1 applies here and $T(n) = \Theta(n^{\log_3 4})$.

1.2 (10 points)

Given a set A of n integers and an integer T, design an algorithm to test whether k of the integers in A add up to T. Prove the correctness of your algorithm and analyze the running time. (Note: full credit will be given to an $O(n^{k-1}\log n)$ algorithm).
Answer:
To test whether k of the integers in A add up to T, we need to find all possible tuples of k integers in A. So a simple solution will be find all $\binom{n}{k}$ tuples and test if there exists a tuple that has a sum T. But this is a $O(n^k)$ solution assuming that read integers and take the sum are constant time operations.

Now let's consider about a better solution:

- Convert the given set into an array A and sort the array by merge sort.
- Let's find all possible tuples with $k-1$ integers by testing $\binom{n}{k-1}$ combinations and calculate the sums of all the tuples.
- For each tuple, if its sum s is larger than or equal to T, calculate the difference $d = T - s$ and find if there exists an integer that is equal to d in the remaining array, by binary search.
- Return true if binary search finds an exact integer; otherwise false;

Algorithm's correctness:
By the correctness of the merge sort, A is sorted. By basic combinatorics, $\binom{n}{k-1}$ will find all the possible combinations of $k-1$ numbers in A — let's say find $k-1$ indices in A. When calculating the sum of each $k-1$ combination of integers, if the current sum of a $k-1$ combination is larger than T already, we don't need to consider this combination. Instead, the sum of $k-1$ integers should be less than or equal to T. We need to find one more integer in the remaining array of $n-k+1$ length to make up the difference (d) from T. Since the array is sorted, the correctness of the binary search will find whether there is an integer d in the remaining array.

Running Time:
First of all, the merge sort step uses $O(n\log n)$ time. When testing all $\binom{n}{k-1}$ tuples,

there will be in total $O(n^{k-1})$ such combinations. In the worst case, for each combination, we need to read the integers, calculate the sum, and find the difference d in the remaining array by binary search. So each combination's operations use $O(1 + \log n)$. In total the algorithm runs in $O(n^{k-1} \log n)$.

2 Problem 2 (20 points)

Given an array (length $> k$) with positive and negative numbers, find the maximum average subarray whose length should be greater or equal to given length k. For example, given an array = $\{2, 11, -7, -6, 51, 3\}$, and $k = 3$, the maximum average subarray of length 3 begins at item -6, and the maximum average is $(-6 + 51 + 3)/3 = 16$. Please justify the correctness of your algorithm and analyze the running time.

Answer:

Solution 1:

Let's first consider the maximum average subarray has a length k. Denote the size of the array is n. We can come up with a solution to find such a maximum average subarray with $O(n)$, and we can use this method as a subroutine.

Here is one solution:

- Step 1: Given the array $A[1 \dots n]$, use one extra array $B[1 \dots n]$ to store cumulative sums of elements in this array in one pass. $B[1]$ stores the sum from $B[1 \dots 1]$, and $B[i]$ stores the sum from $B[1 \dots i]$, where $i = 1 \dots n$. Once $B[]$ is stored and defined, we can calculate the sum from any index range within n in $O(1)$ time (the sum from index j to k is $B[k] - B[j-1]$ and set $B[0] = 0$).

- Step 2: Start with index 1, calculate the sum from the subarray from index 1 to k. Repeat this until the subarray from $n - k + 1$ to n. Then return the subarray with maximum sum.

The algorithm is correct since it finds all possible k subarrays and calculates each of the sums in $O(1)$ time by using an extra array to store the cumulative sums. Clearly the above solution runs in $O(n)$ time since it need one pass scan over the array. However, here is another better solution without using extra array of size n:

- Step 1: Given the array $A[1 \dots n]$, lets first calculate the sum of $A[1 \dots k]$. Denote this sum as s. Initialize a $sum_{max} = s$.

- Step 2: For $i = k + 1$ to n, add $A[i]$ to the subarray $A[1 \dots k]$ and remove $A[i-k]$; calculate the sum of the new subarray, if the sum of the new subarray is larger than sum_{max}, set new sum_{max} and store the indices.

- return sum_{max} and the associated subarray.

The algorithm is correct since it covers all possible k-subarrays by acting as a "sliding window" of size k to test all k subarrays' sums, i.e. adding one rightmost element and removing one leftmost element each time. Clearly this algorithm also runs in $O(n)$ since it scans one pass over the array.

Since the maximum average array's length can be larger than k, we need to consider the cases when the length is larger than k. However, we don't need to consider all possible subarrays of length $k+1$ to n. Instead, we only need to test the subarrays of length $k+1$ to $2k-1$.

Remark 2.1. *If there is a maximum average subarray of length larger than or equal to $2k$, there always exists a subarray of length between k and $2k-1$ that has a larger or equal average.*

Proof. Without loss of generality, let's first assume there is a maximum subarray of length $2k$, say A_{2k} of average V. Define the first half of A_{2k} as $A_{\{1\ldots k\}}$, which has average V_1; second half as $A_{\{k+1\ldots 2k\}}$, which has average V_2. Since V is the maximum average, $V_1 < V$ and $V_2 < V$. But there is a contradiction here that V_1 and V_2 can not be both less than V at the same time. Let's extend to the case when there is a maximum average subarray of length $> 2k$, denoting as $A_{>2k}$ with average V, you can always split the $A_{>2k}$ into subarays of length $< 2k$ ($A_{<2k}$). At least of the $A_{<2k}$ will have an average that is larger than or equal to V. □

Thus, we can use the above $O(n)$ algorithm as a subroutine and test the possible maximum average subarrays of length k to $2k-1$. So in total we need $O(nk)$ time to find such a maximum average subarray.

Also, a $O(n \log(\text{max-min}))$ solution (similiar to binary search) also be possible.

Solution 2: Now let's consider a solution with even better time complexity. Assume that all arrays start with index 1.

- First take the sum of first k elements. Record this k subarray's average $avgMax$, starting element, and ending element as current maximum avg subarray.
- Starting from the $(k+1)$-th element, we want to find the maximum avg subarray that is ending up to the $(k+1)$-th element. You need to consider two possible subarrays and record the one with larger average as the following: (1) a k-subarray ending at (k+1)-th element. (2) a longer subarray (adding $(k+1)$-th element) by extending previously recorded larger subarray with

average L (For $(k+1)$-th element, here L==$avgMax$). So you compare the averages of the two subarrays (1) and (2), and get the one with larger average, denoting as L. If $L > avgMax$, update the maximum avg subarray with L.

- Repeat the same step until n-th element.
- Return the recorded maximum average subarray.

  In this algorithm, we keep checking the possible maximum average subarrays that ending at index $i = k + 1$ to n to find a maximum subarray of length $>= k$. I omit the formal proof here. You can use "loop invariant" or induction proof here.

  From index $i = k+1$ to n, there are in total $n - k$ checking steps, each step needs $O(1)$ operations: check k-subarray ending at i, check the longer subarray from current recorded maximum avg subarary, and update the maximum when necessary. So in total this algorithm requires $O(k + n - k) = O(n)$ time.