Analysis of Algorithms - Home Work 3

CSE 548 Fall '19

Prof. Jie Gao

Submission By:

Narayan Acharya

112734365

Contents

1 Question 1

Textbook [Kleinberg & Tardos] Chapter 4, page 190, problem #6.
Solution:

Let us say that the i^{th} contestant has a swimming, biking and running time denoted by s_i, b_i and r_i respectively. We are to order the contestants such that the last contestant to finish has the earliest possible finishing time. We also have a constraint that no 2 contestants can have overlapping swimming times. Although, there is NO such constraint on their biking and running times. Let us consider the biking and running time of i^{th} contestant to be p_i where $p_i = b_i + r_i$.

Consider the following way of ordering the contestants to achieve our goal: We order them in decreasing order of p_i. That is, for contestants i and j, if i has a higher p_i then he/she goes before p_j. We can argue that this is the most optimal way of ordering contestants by considering existence of another solution that does not follow above rule and is optimal.

Consider that there exists an optimal solution different from the one proposed above such that for contestants i and j, where i starts before j and $p_i < p_j$. Given that i start before j, i will leave the pool earlier and. Time to complete the race from when i started:

$$t_{i1} = s_i + p_i$$

$$t_{j1} = s_i + s_j + p_j$$

Comparing the two, we can easily see that i will finish earlier (strike our s_i from both, $p_i < p_j$ and $s_j > 0$). Thus, $t_{i1} < t_{j1}$

Let us swap i and j now and compare the times to complete from when j started:

$$t_{j2} = s_j + p_j$$

$$t_{i2} = s_j + s_i + p_i$$

Comparing these with t_{i1} and t_{j1}, we can easily see that:

$$t_{i1} < t_{j2} < t_{j1}$$

$$t_{i1} < t_{i2} < t_{j1}$$

In all cases t_{j1} is highest. That means our approach with the first option is not as good compared to our second options where we switched contestants and brought them in accordance of our proposed solution. Hence, it is better if we swap i and j. We can continue this argument for all such i and j to reach the same ordering as proposed by our algorithm.

2 Question 2

Textbook [Kleinberg & Tardos] Chapter 4, page 190, problem #12.
Solution:

(a) Given constraint is such that any time t, the total number of bits we've sent through from time 0 to t cannot exceed rt. The constraint does NOT bound number of bits we can send at any specific time t. So, at any time we can send more bits than the rate specified, if we've sent lesser bits in the previous stream.

So the constraint, $b_i < rt_i$ only holds for the first stream of bits. Subsequent streams may or may not hold to this inequality. In either case they will still be valid. So the claim is false.

(b) We can use the 'Greedy Algorithm Stays Ahead' argument to prove our case. In order to find a valid and feasible schedule we sort by bits we need to transmit in each stream i.e. We sort by b_i/t_i. In such an ordering we always transmit the least number of bits we can possibly send at that time. If at any time, our constraint does not hold true then there is no way we can send another stream of bits ahead of this as it will at least transmit as many bits as the current stream because we have sorted them in increasing order of bit-rate. The running time of this algorithm will be bound by $O(nlogn)$, the amount of time we need to sort the streams in increasing order of bit-rate.

To only find whether there exists a schedule where we can order all streams without breaking the constraint we simply keep of track of number of bits 'saved' at each transmission. Bits saved can be used to transmit another stream having a higher bit-rate than the allowed maximum rate of r. In these cases, we can consider the bits saved to be negative. If the total sum of all such bits saved is greater than 0, then we can claim that we can order streams in some order without breaking the constraint. The running time for this will be $O(n)$ as we only need to calculate bits saved at each bit-stream and keep track of total bits saved.

Algorithm 1 Algorithm to find if valid schedule exists $O(n)$

```
1: $saved \leftarrow 0$
2: for $i \leftarrow 1, 2, 3 \ldots n$ do
3:     $saved \leftarrow saved + rt_i - b_i$
4: end for
5: if $saved \geq 0$ then
6:     return True
7: else
8:     return False
9: end if
```

3 Question 3

Textbook [Kleinberg & Tardos] Chapter 4, page 190, problem #14.
Solution:

(a) We wish to minimize the number of times we invoke the script to check our critical processes. The greedy approach to this would be holding off on the check as long as possible and doing one if absolutely necessary so that we don't miss a process. One approach would be to sort the processes according to their finish times and to invoke the status check right before the process finishes if the process has not been checked at least once prior. Let us consider the correctness and optimality of our approach.

   (1) Correctness: The approach ensures that all the processes are checked because we invoke the status check right before the end time of every process if it has not yet been checked. Thus it computes a valid set of status check invocation timings.

   (2) Optimality: To argue that this approach computes the minimum number of status checks we use 'Greedy Algorithm Stays Ahead' argument. Consider, that our algorithm produces k status checks up to a certain process evaluation, and the set is denoted by S. Also, there exists another set of optimal status checks, S'. We would like to show that S' too has at least k status checks up until the same process is evaluated. We prove this by induction:

      i. For $n = 1$, both S and S' will have to have at least 1 status check.
      ii. For some $n = m$, say S and S' have k status checks.
      iii. For $n = m+1$, we have two options, $(m+1)^{th}$ overlaps or does not overlap with our current process. In case it does, our k^{th} status check in S will take care of the status check for $(m+1)^{th}$ process. In case it does not, we will need to add our $(k+1)^{th}$ status check at the finish time of the $(m+1)^{th}$ process. In both cases, S' too will need to add at least one more check to cover the $(m+1)^{th}$ process. Thus we have $(k+1)$ checks for in both S and S'.

   Thus our algorithm will perform at least as well any other optimal algorithm. The algorithm is bound by $O(nlogn)$ - the amount it would required to sort the processes by their finishing times and $O(n)$ - the amount of time to iterate over the sorted processes to insert the process checks. Thus, running time is bound by $O(nlogn)$.

(b) Our solution from above will produce k status checks as each one will be covering for at most one process from the k processes given that none of them overlap with another. Consider the following two arguments:

   (a) If we can find the maximum number of non-overlapping processes, k^*, then we need at least k^* checks. Reusing variable from the first part, we can say that $|S| \geq k^*$.

   (b) Our algorithm from above will at max produce k^* status checks for these processes Hence, we can say that $|S| \leq k^*$.

   Thus we can claim that $|S| = k^*$. Hence, the claim of there being a stronger rule between number of status checks and max number of non-overlapping processes is true.

4 Question 4

Textbook [Kleinberg & Tardos] Chapter 4, page 190, problem #16.
Solution:

Consider the timestamps and error margins of for the i^{th} suspicious event be denoted by t_i and e_i respectively. The following Greedy Algorithm should provide us with a perfect matching between the recent events and the suspicious events, if one exists:

Sort the recent events, x, in ascending order of timestamp and the suspicious events, t, in ascending order of finish times i.e $t_i + e_i$ for i^th suspicious event. Attempt to match each event in x with the suspicious event in t that finishes earliest. Once matched, remove the suspicious event matched from further consideration. Continue until all recent events are matched. If at any point, no recent event can be matched to a suspicious event then end the search as no perfect matching exists.

```
for $i \leftarrow 1, 2, 3 \ldots n$ do
    $M[i] \leftarrow None$
    $j \leftarrow 0$
    while $j < size(t)$ do
        if $x[i] > t[j] - e[j]$ and $x[i] < t[j] + e[j]$ then
            $M[i] \leftarrow (x[i], t[j])$.
            Remove $t[j]$
            break
        end if
        $j \leftarrow j + 1$
    end while
    if then$M[i] = None$
        return No perfect match exists!
    end if
end for
```

In order to prove correctness, we show that if a perfect matching exists then the above algorithm is bound to find it. We prove this using a combination of exchange argument and contradiction. Consider, the matches, M, produced by our algorithm is incorrect there exists another set of pairs between the suspicious and recent events, say M' which is correct. Let us assume that the first i matches in M and M' are identical. M' matched element $i+1$ to another element, say t_k but M matches element x_i to t_j. Given the behavior of our algorithm, we can claim that $t_j + e_j < t_k + e_k$. Suppose t_j is paired with some other element $x_{>i+1}$. Drawing up the timeline would look like:

$$t_j - e_j < x_i < x_{>i+1} < t_j + e_j < t_k + e_k$$

But from the above timeline we can easily see that matching x_i to t_j and $x_{>i+1}$ to t_k will also be a valid pairing. This pairing agrees with our algorithm M. Thus our choice of i was incorrect and should have been $i+1$.

The running time of the above algorithm is bound by $O(n^2)$ due to the two nested loops that iterate over n. We have some sorting as well to do initially, but that is $O(nlogn)$ which is upper bounded by $O(n^2)$.

5 Question 5

Textbook [Kleinberg & Tardos] Chapter 5, page 246, problem #1.
Solution:

The median elements is the average of $(2n/2)^{th}$ and $(2n/2+1)^{th}$ elements. There are $n-1$ elements smaller and $n-1$ larger than these two numbers. Another thing to note is that if we remove the same number of elements from the beginning and the end of such an array, the median will still remain the same. Also, we can consider the two databases, A and B, be equivalent to two sorted arrays where accessing the k^{th} element gives the k^{th} smallest elements in that array. With these things in mind, we can suggest a recursive solution to find the median of two arrays(read databases) with minimum number of array accesses(read database queries).

Consider, accessing the $k = \lceil n/2 \rceil$ element in both arrays, A and B. If $A[k] > B[k]$, then we can infer the following things:

1. $B[k]$ is greater than at least the first k items in A and is greater than $k-1$ items in B. Thus $B[k]$ is greater than $2k-1$ items in the combined arrays. Given our choice of k, all elements greater than $B[k]$ are greater than the median.
2. For $A[k]$, the first k elements are smaller than $B[k]$ and thus are also smaller than all elements greater than $B[k]$. They are also smaller than the second half of the elements in A. Again, given our choice of k we can easily see that these all numbers less than $A[k]$ are less than the median.

We can now safely eliminate the first half of A and the second half of B from our consideration of median. Given we have removed equal number of elements from the beginning and the end, we can still claim that the median lies in the remaining set of elements and the median of this set will continue to remain the median of the whole set.

We can continue to do this recursively by considering the new reduced arrays to find the median. Also, for $A[k] < B[k]$ we can simply reverse the roles of A and B and arguments will still hold.

Important thing to note here is that given these are actually databases and not arrays, we cannot simply drop parts of them from consideration. We achieve this by carefully updating the index to use for querying after each recursion. For eg. In our case, for A we drop the its first half. In the next recursion we simply add $\lfloor n/2 \rfloor$ to adjust for the indices we do query for.

We have accessed our database only twice and reduced each array by half in the process. Essentially, our algorithm should be able to find the median in $2 \times logn$ steps or simplified $O(logn)$.

6 Question 6

Textbook [Kleinberg & Tardos] Chapter 5, page 246, problem #7.
Solution:

The essence of the question is to find a local minima element in a 2D-Matrix.

Consider the following algorithm:

1. Probe all the elements on the boundary, the middle row and middle column of our graph G. (Let us call them periphery elements). [6n-4 or O(n) probes].
2. Find the minimum element, say e, in these probed nodes.
3. If e is a local minima stop the search and declare e as desired output.
4. If e is not a local minima then we select node, say f, neighbor of e with least value. [1 or 2 probes]
5. We continue our search with the quadrant of the graph that contains f. [Important thing to note here is that the reduced search space will still contain our periphery elements to keep our search bounded.]

Correctness of above algorithm: We know that f does not lie on the border of our grid. And also f is smaller than all the elements that surround it **and** lie on the periphery of our initial search space, i.e the boundary, middle row and column. Thus f has to be an internal minimum of the graph G.

Running Time: We do work of order $O(n)$ in each recursion and reduce the search from $n \times n$ to $\frac{n}{2} \times \frac{n}{2}$. We can right down the recurrence relation as follows:

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

Using Master's Theorem (or See Kleinberg & Tardos, proof 5.5, page 219) this reduces to $T(n) = O(n)$