Solutions to Section II

COMP3121/9101 22T1

Final Exam, 2nd May 2022

This document provides solutions for Section II of the 22T1 final exam. These solutions were prepared primarily as an aid to markers rather than to model full mark solutions for students, so they are a bit rough in places.

Question 9

You are given a positive integer n, an array A of n positive integers and a positive integer S. For each pair of indices $1 \leq i \leq j \leq n$, consider the subarray $A[i..j]$.

Design an algorithm which determines the number of these subarrays with sum strictly less than S and runs in $O(n \log n)$ time. You must provide reasoning to justify the correctness and time complexity of your algorithm.

The input consists of the positive integers n and S, as well as n positive integers $A[1], \ldots, A[n]$.

The output is the number of pairs (i, j) such that $A[i] + \ldots + A[j] < S$ where $1 \leq i \leq j \leq n$.

For example, suppose $n = 4$, $S = 8$ and the array is $\langle 2, 5, 3, 4 \rangle$. There are ten nonempty subarrays, and six of these have sum strictly less than eight, so the answer is six.

Solution

First, construct an array B of size n. Set $B[1] = A[1]$, and for all $2 \leq i \leq n$, let $B[i] = B[i-1] + A[i]$. $B[0]$ will be understood as zero. Now, each term $B[i]$ stores the sum $A[1] + \ldots + A[i]$, so we can look up the sum of the subarray $A[i..j]$ in constant time by evaluating $B[j] - B[i-1]$.

Now, we return to the original problem, which is to find the number of pairs (i, j) such that $B[j] - B[i-1] < S$. For a fixed i, this amounts to finding the number of indices j such that $B[j] < S + B[i-1]$. However, the array B is increasing, so we can find the largest such j using binary search and add $j - i + 1$ to the answer. Repeating this for all i, we find the number of pairs.

Calculating array B takes linear time, and for each index i the binary search takes $O(\log n)$ time, so the total time complexity is $O(n \log n)$.

Question 10

You are given an n digit positive integer A and a positive integer $k < n$. Let the digits of A be $a_{n-1}, a_{n-2}, \ldots, a_0$, in order from most to least significant. You can delete k digits from A to leave an $(n-k)$ digit number B, potentially with leading zeroes.

Design an algorithm which determines the smallest possible value B and runs in $O(n)$ time.

The input consists of the positive integers n, A and k.

The output is an $(n-k)$ digit positive integer, which may include leading zeroes.

For example, if $n = 5$, $A = 90834$ and $k = 2$, the correct answer is $B = 034$.

Solution

We loop from the most significant digit to the least significant digit and maintain a stack which is initially empty.

For each digit a_i, while it is smaller than the last element on the stack, then keep popping off from the top of the stack (this represents removing the digit from the final number). Once it is no longer smaller, or there are no more elements left in the stack, or we have reached the maximum k deletions, then place a_i onto the top of the stack. Finally, read off the stack from bottom to top in order to get our final number. If this number is still longer than $n-k$ digits, this would imply that it is sorted in non-decreasing order, so we can delete everything but the first $n-k$ digits.

Each digit is added onto the stack once, and can be deleted off the stack at most once. All stack operations are $O(1)$ time, so the final complexity is $O(N)$.

Proof of Correctness:

Consider the optimal solution B equal to $b_{n-k-1}b_{n-k-2}\ldots b_0$. We show that our algorithm makes the exact effort required to bring a digit equal to b_{n-k-1} to the front, then b_{n-k-2} to second place, and so on. Suppose a_i is the leftmost digit in n that is equal to b_{n-k-1}. Then there is always an optimal solution where b_{n-k-1} corresponds to a_i in the original number. If there were not, and b_{n-k-1} corresponded to some other digit $a_j = a_i$ where $j > i$, then we could simply replace a_j with a_i to get a solution just as good.

We show that our algorithm will always bring a_i to the bottom of the stack. Firstly, every element from a_{n-1} to a_{i+1} must be larger than a_i. They cannot be equal by the definition of a_i, and they cannot be smaller since if there existed an $a_j < a_i$ where $j < i$, then we could improve the optimal solution to be better (i.e. $a_j b_{n-k-2}\ldots b_0$), which is a contradiction. Now since everything before a_i is bigger than a_i, then right before a_i is placed on the stack, everything in the stack will be removed by our algorithm. Furthermore, it's guaranteed that we can do so within k deletions since if we couldn't, then it would be impossible for a digit equal to b_{n-k-1} to be at the front of the optimal solution. In other words, our algorithm always places a_i at the bottom of the stack at one point.

Next we show that our algorithm never removes a_i after it is placed there. For a_i to be removed, that implies that there is another digit a_j after a_i that is the first digit in the final number generated by our algorithm. This implies that every digit before a_j (including a_i) must have been greater than a_j, since if they weren't, we could take the smallest out all of these numbers, and see that our algorithm would never have deleted it off the stack in order to put a_j at the bottom. Therefore a_j is smaller than a_i. But this

would imply that our algorithm found a solution that is better than the optimal solution, which once again is a contradiction. Hence a_i is never removed.

But now we have shown that it is always optimal to use the leftmost occurrence of a digit a_i equal to b_{n-k-1} to bring to the front. Furthermore, we've shown that our algorithm always does so in $i-1$ deletions, and keeps it there. In other words, how we handle the first i digits in A is optimal. But now that we know this, we can solve a new subproblem: What is the minimum number we can get from digits $a_{i-1}a_{i-2}\ldots a_0$ after deleting $k-(i-1)$ digits? This is a smaller instance of the same problem, and we can simply continue on with our current algorithm to solve it. Thus our greedy algorithm will always find an optimal solution.

Question 11

You are given an integer n and a sequence of n books on a bookshelf. The ith book has p_i pages. Define the *disjointedness* of the sequence of books to be

$$D = \sum_{i=1}^{n-1} |p_{i+1} - p_i|.$$

You are also given a positive integer $k < n$. You can remove up to k books from the sequence, while keeping the rest in their original order.

Design an algorithm which finds the minimum disjointedness of the bookshelf with up to k books removed, and runs in $O(nk^2)$ time.

The input consists of the positive integers n and k where $k < n$, as well as n positive integers $p_1, \ldots, p_n$.

The output is a single non-negative integer, the minimum disjointedness.

For example, if $n = 7$ and the sequence of p_i is $\langle 6, 8, 4, 2, 7, 3, 9 \rangle$, the disjointedness is originally 19. If $k = 2$, we can remove up to two books. Removing the second and sixth books gives a disjointedness of 11, which is the smallest value that can be achieved.

Solution

Subproblems

Let $P(i, j)$ be the problem of determining $\text{opt}(i, j)$, the minimum disjointedness of a sequence of j books ending at the ith book.

Recurrence

For all $1 < j \leq n$, we have

$$\text{opt}(i, j) = \min_{j-1 \leq t < i} \left(\text{opt}(t, j-1) + |p_i - p_t| \right).$$

We iterate over all choices t for the index of the penultimate remaining book, and add the difference in pages between book t and book i.

Base cases

The base cases are when $j = 1$, i.e. when there is no penultimate book. Clearly $\text{opt}(i, 1) = 0$ for all i.

Overall solution

The solution to $P(i, j)$ depends on the solutions to various $P(t, j-1)$, so we fill the grid from left to right, i.e. we solve all $P(i, 1)$ then all $P(i, 2)$ and so on.

The overall solution is the best way to delete up to k books, i.e. leave at least $n - k$ remaining books. Thus the answer is

$$\min_{n-k \leq i \leq n, j \geq n-k} \text{opt}(i, j).$$

Time complexity

Note that in any subproblem $P(i, j)$, we must have $i - j \leq k$, as otherwise we would have already deleted more than k books. Thus for each of $O(n)$ values of i there are only $O(k)$ subproblems, and each is solved in $O(k)$, giving a total time complexity of $O(nk^2)$.

Question 12

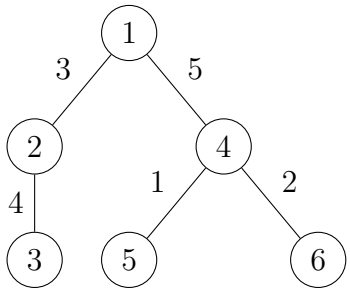
You are given a tree with n vertices, where vertex 1 is the root. Each edge e has an associated value, which is the cost to cut this edge.

Design an algorithm which runs in time polynomial in n and determines a set of edges of minimum total cost such that cutting all chosen edges would disconnect all leaf vertices from the root.

The input consists of the positive integer n, as well as $n-1$ pairs $(p_2, c_2), \ldots, (p_n, c_n)$, denoting for each vertex i other than the root its parent p_i and the cost c_i of the edge between p_i and i.

The output consists of a subset of the edges of this graph. These edges may be presented in any data structure or format.

For example, suppose the tree is as pictured below.



The best set of edges to cut is $\{(1,2),(4,5),(4,6)\}$ for a total cost of 6.

Solution

Construct a flow network with:

- source s and sink t
- a vertex v_i for each vertex of the tree
- an edge from s to v_1 with capacity ∞
- for each $i > 1$, an edge from v_{p_i} to v_i with capacity c_i
- for each leaf vertex i of the tree, an edge from v_i to t with capacity ∞.

Any finite-capacity cut in this graph will disconnect the source from the sink (since the edges from s and to t have infinite capacity). To minimise the cost, we must look for a minimum cut.

To find which edges to cut, we first run the Edmonds-Karp algorithm, and after it terminates we perform one more breadth-first search from s. The minimum cut will then be (S, T) where S contains all vertices seen in this search, and T contains all others (including t). We disconnect those wires which go from a vertex in S to a vertex in T.

Constructing the graph takes $O(V + E)$ time, and finding a minimum cut takes $O(VE^2)$. Since $V = n + 2$ and $E = 1 + (n-1) + \#(\text{leaves}) \leq 2n$, the time complexity is clearly polynomial in n.