Assignment 2 Solutions

COMP3121/9101 21T3

Released September 29, due October 13

This document gives model solutions to the assignment problems. Note that alternative solutions may exist.

1. (20 points) You are given n stacks of blocks. The ith stack contains $h_i > 0$ identical blocks. You are also able to move any number of blocks from the ith stack to the $(i + 1)$th stack. You want to know if the sizes of the stacks can be made *strictly* increasing. For example $\langle 1, 3, 6, 8\rangle$ is acceptable, but $\langle 1, 4, 4, 7\rangle$ is not.

   Design an $O(n)$ algorithm that determines whether it is possible to make the sizes of the stacks strictly increasing.

   In a sequence of stacks with strictly increasing height, the kth stack have have height at least k blocks, and therefore the first k stacks must have at least $k(k+1)/2$ blocks.

   Claim: a sequence can be made strictly increasing if and only if

$$\sum_{i=1}^{k} h_i \geq \frac{k(k+1)}{2}$$

   for all $1 \leq k \leq n$.

   Proof:

   If this inequality fails for some k, then the first k stacks cannot be made strictly increasing in height. Since blocks can be moved to later stacks but not earlier stacks, this ensures that the n stacks cannot be made strictly increasing.

   However, if this inequality is true for all k, then we can make the blocks strictly increasing using the following algorithm. Traverse the stacks from first to last, at each stage keeping exactly k blocks in the kth stack and moving the rest to stack $k + 1$. The assumed inequality ensures that when we get to stack k, it will always have at least

$$\frac{k(k+1)}{2} - \frac{(k-1)k}{2} = k$$

   blocks in it. At the last stack, no moves are available, so leave the stack as is.

   Thus, we can make the sequence strictly increasing if and only if the sum of the first k initial heights is at least $k(k + 1)/2$ for all $1 \leq k \leq n$. We can check all of these inequalities in one pass of the sequence, by maintaining the sum of the first k values h_i. Each new stack adds one term to the sum, taking $O(1)$ to update the sum and $O(1)$ to check it. The total complexity is $O(n)$.

2. (20 points) Alice has n tasks to do, the ith of which is due by the day d_i. She can work on one task each day, and will complete each task in one day. Morever, Alice is a severe procrastinator and wants to accomplish every task as close as possible to its due date. If Alice finishes the ith task on day j, her rage will increase by $d_i - j$.

   Design an $O(n \log n)$ algorithm that determines whether all tasks can be completed by their deadlines, and if so, outputs the minimum total rage that Alice can accumulate.

   Firstly, sort the tasks by their due date in descending order. Then, for each task, (greedily) choose the latest day available that isn't after the due date. This can be done by maintaining a variable for the earliest day that has a task, then assigning the task to the day before that, if it occurs before the due date, or the due date otherwise. If we attempt to assign to day 0 or earlier[1], then report that Alice cannot complete all tasks on time.

   The total rage can be calculated either incrementally while we determine which day each task is done, or afterwards if the date each task was completed was recorded in an array.

   The initial sort takes $O(n \log n)$, and the subsequent greedy assignment takes $O(n)$ (with each task taking exactly $O(1)$ time), so the overall time complexity is $O(n \log n)$.

   <ins>Proof of correctness:</ins>

   There are two aspects of our algorithm that we need to prove to show that it is correct:

   1. it correctly determines whether all tasks can be completed before their due date, and
   2. if so, it produces the minimum possible total rage.

   If there no valid scheduling of tasks exists, then we obviously won't be able to find one. Otherwise, a valid scheduling must exist. We'll show that we can transform any valid scheduling into the one produced by our algorithm without increasing total rage.

   Firstly, we can permute which task is done on which day so that if task i is scheduled before task j, then $d_i \leq d_j$. This is true because if $d_i > d_j$ for any two tasks but i is scheduled before j, then swapping those two will still give a valid schedule.

   Additionally, we can further permute it so that the relative order of tasks is identical, since tasks with the same due date can be swapped without changing the total rage or validity of a schedule.

   Now, our algorithm will always schedule tasks on the latest day possible under this ordering. We can then repeatedly move individual tasks back a day until we obtain the schedule produced by our algorithm. Doing so can only decrease total rage.

   This means that our algorithm will always find a schedule if one exists, and the one it finds will always have minimum total rage.

[1]The problem didn't specify whether the first day a task can be done is day 0 or day 1. The intended interpretation was the latter (see this forum post), but solutions using either assumption should be accepted.

Notes from the problem setters:

The above proof doesn't rigorously show that we can obtain always the schedule that the algorithm produces – we technically assume its existence when we say that the last step of moving days back produces the schedule that our algorithm finds. A proper proof of this would be quite involved and is beyond what's required for this course.

Iterating over days, from the largest due date down to zero, would exceed the required time complexity. This alternative approach would take $O(n \log n + \max d_i)$, which isn't $O(n \log n)$ as d_i is unbounded.

3. (20 points) Define the *separation* of an array of integers to be the smallest difference between any two integers in the array.

   You are given an array A of n distinct positive integers, each no larger than m. For a given positive integer k satisfying $2 \leq k \leq n$, you wish to select a length k subarray of A with the largest possible separation. This subarray need not be contiguous.

   Design an $O(n \log m)$ algorithm to select such a subarray.

   First, we determine how many integers can be chosen to form a subarray with separation at least s, for some fixed s. This is a much easier problem than the original; first sort the integers and choose greedily. We always choose the first number, and a subsequent number is chosen if and only if it is at least k larger than the most recently chosen number.

   Proof of correctness (sketch): Suppose an optimal solution first disagrees with our greedy algorithm by choosing a when our algorithm chose b. Replacing a with b creates a new solution with the same number of chosen elements and maintains a separation of at least s. Continuing in this way, we can transform an optimal sequence into the one chosen by our greedy algorithm, so the greedy algorithm is optimal.

   There is a clear relationship between the original problem and this one; the bigger S is, the shorter the subarray we can construct. For example, if $s = 1$, we can select every entry of A (since all entries are distinct, the separation of A itself is at least 1) so we can construct a subarray of length n. However, if s is $\max(A) - \min(A)$, then we can only choose the maximum and the minimum to form a subarray of length 2.

   In general, define $f(s)$ to be the number of elements that can be chosen in a subarray of separation at least s.

   Claim: f is decreasing.

   Proof: An array of separation at least $s + 1$ also has separation at least s. Therefore the longest subarray with separation at least s is no shorter than the longest subarray with separation at least $s + 1$, i.e. $f(s) \leq f(s + 1)$. This completes the proof.

   Since f is decreasing, we can use binary search to find the largest s such that $f(s) \geq k$. We output the first k entries of subarray found by our greedy algorithm above.

   For the time complexity, we need only sort A once, so this contributes $O(n \log n)$. The range of the binary search is from 1 to m, so it takes $O(\log m)$ many iterations. Each of these involves finding $f(s)$ using the above mentioned greedy algorithm in linear time, for a total of $O(n \log m)$. Since the n positive integers are distinct and each no larger than m, we have $m \geq n$. So the whole time complexity is $O(n \log n + n \log m) = O(n \log m)$.

   Note that we could have instead binary searched over $g(s) := (f(s) \geq k)$, which takes values in $\{0, 1\}$ and is also decreasing.

4. (20 points) You are given a set of real numbers $S = \{t_1, t_2, \ldots, t_n\}$, where $n = |S|$ is a positive integer. Your task is to construct a polynomial P of degree n and leading coefficient 1, i.e.

$$P(x) = x^n + a_{n-1}x^{n-1} + a_{n-2}\,x^{n-2} + \ldots + a_2\,x^2 + a_1\,x + a_0,$$

such that $P(t_1) = P(t_2) = \ldots = P(t_n)$.

Design an $O(n \log^2 n)$ algorithm to construct such a polynomial and evaluate its coefficients.

We choose the common value $P(t_1) = \ldots = P(t_n)$ to be zero. Note that changing this common value corresponds to a change in the constant term of P.

Now that we know the roots of P, we can write it as a product of linear factors:

$$P(x) = \prod_{i=1}^{n} (x - t_i).$$

This allows us to rewrite P, a polynomial of degree n, as the product of two polynomials P_1 and P_2 of degree[2] approximately $n/2$ as follows:

$$P(x) = \underbrace{\left(\prod_{i=1}^{\lfloor n/2 \rfloor} (x - t_i)\right)}_{P_1(x)} \underbrace{\left(\prod_{i=\lfloor n/2 \rfloor + 1}^{n} (x - t_i)\right)}_{P_2(x)}.$$

We will compute P by applying divide and conquer. Suppose the time taken to solve an instance of size n is $T(n)$. Then to compute a polynomial P with n specified roots, we must:

- first compute P_1 and P_2 recursively, each taking $2T(n/2)$ time;
- then multiply P_1 and P_2 using the FFT-based polynomial multiplication algorithm presented in lectures, taking $\Theta(n \log n)$ time.

The base case is an instance of size 1, in which we produce the polynomial $P(x) = x - t_1$ with coefficients 1 and $-t_1$.

Clearly, this algorithm solves the $n = 1$ case, as the statement $P(t_1) = \ldots = P(t_n)$ is vacuously true. Furthermore, if polynomials P_1 and P_2 have roots at the elements of the disjoint sets S_1 and S_2 respectively, then the product of these two polynomials has roots at the elements of $S_1 \cup S_2$. Therefore $P(t_1) = \ldots = P(t_n)$, and P has degree n and leading coefficient 1, so the algorithm is correct.

Therefore, the recurrence for $T(n)$ is

$$T(n) = 2T(n/2) + \Theta(n \log n).$$

If we attempt to use the Master Theorem, we find that $a = b = 2$, so the critical exponent is $c^* = \log_2 2 = 1$ and the critical polynomial is n. However, $f(n) = \Theta(n \log n)$, so we have

[2]To be precise, the degrees are $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, but it is stated in lectures that these floors and ceilings can be ignored for complexity analysis.

1. $f(n) \neq O\left(n^{1-\epsilon}\right)$, so Case 1 does not apply;
2. $f(n) \neq \Theta(n)$, so Case 2 does not apply;
3. $f(n) \neq \Omega\left(n^{1+\epsilon}\right)$, so Case 3 does not apply.

Instead, we must unravel the recurrence as follows:

$$\begin{aligned}
T(n) &= 2\,T\left(\frac{n}{2}\right) + f(n) \\
&= 2\left[2\,T\left(\frac{n}{4}\right) + f\left(\frac{n}{2}\right)\right] + f(n) \\
&= 4\,T\left(\frac{n}{4}\right) + 2\,f\left(\frac{n}{2}\right) + f(n) \\
&= \dots \\
&= 2^k\,T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} f\left(\frac{n}{2^i}\right) \\
&\approx nT(1) + \underbrace{\sum_{i=0}^{\lfloor \log_2 n \rfloor - 1} 2^i\,f\left(\frac{n}{2^i}\right)}_{S},
\end{aligned}$$

where the last line is achieved by letting $k = \lfloor \log_2 n \rfloor$.

Now, we simplify:

$$\begin{aligned}
S &= \sum_{i=0}^{\lfloor \log_2 n \rfloor - 1} 2^i\,\Theta\left(\frac{n}{2^i}\log\frac{n}{2^i}\right) \\
&= \Theta\left(\sum_{i=0}^{\lfloor \log_2 n \rfloor - 1} 2^i\left(\frac{n}{2^i}\log\frac{n}{2^i}\right)\right) \\
&= \Theta\left(\sum_{i=0}^{\lfloor \log_2 n \rfloor - 1} n\log\frac{n}{2^i}\right) \\
&= \Theta\left(\sum_{i=0}^{\lfloor \log_2 n \rfloor - 1} n\left(\log n - i\log 2\right)\right) \\
&= \Theta\left(n\log n\,\lfloor \log_2 n \rfloor - n\log 2\sum_{i=0}^{\lfloor \log_2 n \rfloor - 1} i\right) \\
&= \Theta\left(n\log n\,\lfloor \log_2 n \rfloor - \frac{1}{2}n\log 2\,\lfloor \log_2 n \rfloor(\lfloor \log_2 n \rfloor - 1)\right) \\
&= \Theta\left(n\log^2 n\right),
\end{aligned}$$

since logarithms of n in any base differ only by a constant factor.

Finally, substituting back into $T(n) \approx nT(1) + S$, we conclude that the time complexity of our algorithm is $\Theta(n\log^2 n)$ as required.

5. (20 points) Aleks received an offer from UNSW and he wants to graduate as soon as possible. His program requires him to complete n courses in an order of his choice. The courses are labelled $1, 2, \ldots, n$, where course i takes t_i weeks to complete. Aleks gives you these values in an array A.

   However, some pairs of courses *overlap*. If courses i and j overlap, then a student who has already completed either course can complete the other in a number of weeks less than both t_i and t_j.

   Using the UNSW handbook, Aleks has produced another array B with m entries. Each entry consists of an *unordered* pair of *distinct* courses which overlap (say $p = \{i, j\}$), as well as the number of weeks t_p required to complete either course if the other has already been completed. For each such pair, you are guaranteed that $t_p < \min(t_i, t_j)$.

   Design an $O((n + m) \log(n + m))$ time algorithm that finds the minimum number of weeks required to complete all n courses.

   For each i, the number of weeks taken to complete course i will either depend on one other course j (in which case $t_{\{i,j\}}$ weeks are required), or no course (t_i weeks). We can simplify this by introducing a new course numbered 0 that takes zero time and represents "no course", as well as n extra overlaps $t_{\{0,k\}} = t_k$ for $k = 1, \ldots, n$. This allows us to begin with course 0 and henceforth only consider times arising from an overlap, rather than the direct cost of doing any course.

   Construct a graph G with $n + 1$ vertices and $m + n$ undirected edges. The vertices are labelled from 0 to n, with each corresponding to a course. Each edge $e = \{i, j\}$ corresponds to a pair of overlapping courses and has weight $t_{\{i,j\}}$.

   Claim: For any given order of courses, the number of weeks taken to do all courses is the total weight of some spanning tree of G.

   Proof: If the number of weeks taken to complete course i depends on its overlap with course j (potentially 0), then select the edge between i and j. Let the subgraph formed by these selected edges be H. Now H has n edges, as courses 1 to n were completed. But H is also acyclic as there cannot be a cycle of dependencies; later courses depend on earlier courses. Therefore H is a spanning tree.

   Clearly, the minimum number of weeks is achieved by the minimum spanning tree, which can be found using Kruskal's algorithm, implemented with the Union-Find data structure.[3]

   We can also recover a valid ordering of courses from the minimum spanning tree by doing a DFS or BFS starting at node 0, though this isn't necessary to obtain the minimum time needed.

   Our adjusted graph will have $m + n$ edges and $n + 1$ vertices, so it takes $O(n + m)$ time to construct an adjacency list representation of the graph. Kruskal's algorithm then finds the minimum spanning tree in $O(|E| \log |E|) = O((n + m) \log(n + m))$ time, so the overall time complexity of our algorithm is also $O((n + m) \log(n + m))$.

[3] Prim's algorithm is also acceptable, using an augmented heap.

We also include an alternate, more complex solution.

Instead of directly finding the total number of week, we'll find the amount of time Aleks can save by doing overlapping courses, compared to taking the full t_i weeks for each one.

We will represent the overlapping course information as a weighted directed graph, where an edge with weight w from node i to j means that doing j before i will save us w weeks, with $w = p_i - p_{\{i,j\}}$. For example, if $t_1 = 3, t_2 = 5, t_{\{1,2\}} = 2$, then the edge $1 \to 2$ will have weight $3 - 2 = 1$, and $2 \to 1$ will have weight $5 - 2 = 3$.

Now, we need to pick a subset of these edges to specify which time savings we want – picking an edge $i \to j$ means that we will do i after j. There are a few restrictions:

1. We can only pick at most one out-edge per node. This corresponds to the fact that Aleks cannot combine the benefits of multiple overlaps.
2. Our selected edges cannot form a cycle. This represents the fact that Aleks cannot time travel.

To find the best subset of edges, sort the list of edges by weight then iterate in descending order, greedily taking edges if they won't violate the above restrictions. We also accumulate the time saved from picking these edges.

1. To check condition 1, maintains a boolean array of length n indicating whether we've picked an out-edge for each node, and
2. To check condition 2, we can treat our edges as undirected and use Kruskal's Algorithm to detect whether our edges will form an *undirected* cycle. Why this is valid will be demonstrated later.

Finally, we can subtract the maximum time saved from the sum of t_i to get the minimum number of weeks required to complete all the courses.

Complexity-wise,

- constructing the list of edges takes $O(m)$,
- sorting it takes $O(m \log m)$,
- checking and maintaining the array for condition 1 takes $O(n)$ overall, and
- using Kruskal's Algorithm for condition 2 takes $O(m \log m)$ overall.

The overall time complexity is the sum of these, which comes out to $O(m \log m + n)$. This satisfies the $O((m + n) \log(m + n))$ requirement of the problem.

<u>Proving that our graph representation is valid</u>

We will prove that any ordering of courses corresponds to a subgraph that satisfies our conditions, and every valid subgraph also corresponds to an ordering of courses.

Suppose we have an arbitrary ordering of courses, and we want to find the subgraph that corresponds to it. For every course i whose time is reduced to $t_{\{i,j\}}$ by doing j before, include the edge $i \to j$ in our subgraph. Since at most once course can reduce each course's time, we will not break requirement 1. Since j comes before i in our ordering of courses, we cannot have cycles and so won't break requirement 2. Finally, each edge corresponds to exactly one time reduction, so the total time, minus the total weight of edges, gives us the minimum time of the ordering. Thus, any ordering of courses corresponds to a valid subgraph.

Now, suppose we have an arbitrary subgraph satisfying our requirements. By performing a topological sort on the nodes (which is possible since we don't have cycles), we obtain an ordering of courses that can take (total time – sum of edge weights).

Proving that our solution is optimal

If the no-cycle condition didn't exist, we can obtain the subgraph with highest total edge weight by simply picking the greatest out-edge for each node. This subgraph could have cycles, but it turns out that there is exactly one (the proof of this is left up to the student). Hence, the optimal valid subgraph is one with the smallest edge in the cycle removed. This also explains why using Kruskal's Algorithm is valid – by greedily taking edges in descending weight order, we'll automatically omit the last edge as it will form a cycle.