Homework 5 Solution

Due: November 26, in class.

Instruction. There are 7 problems in this homework, worth 100 points in total. For all algorithm design problems, you need to provide proofs on the correctness and running time of the algorithms unless otherwise instructed.

Problem 1 [15 pts]

You're trying to run a large computing job in which you need to simulate a physical system for as many discrete *steps* as you can. The lab you're working in has two large supercomputers (which we'll call A and B) which are capable of processing this job. However, you're not one of the high-priority users of these supercomputers, so at any given point in time, you're only able to use as many spare cycles as these machines have available. Here's the problem you face. Your job can only run on one of the machines in any given minute. Over each of the next n minutes, you have a "profile" of how much processing power is available on each machine. In minute i, you would be able to run $a_i > 0$ steps of the simulation if your job is on machine A, and $b_i > 0$ steps of the simulation if your job is on machine B. You also have the ability to move your job from one machine to the other; but doing this costs you a minute of time in which no processing is done on your job.

So, given a sequence of n minutes, a *plan* is specified by a choice of A, B, or "*move*" for each minute, with the property that choices A and B cannot appear in consecutive minutes. For example, if your job is on machine A in minute i, and you want to switch to machine B, then your choice for minute $i+1$ must be *move*, and then your choice for minute $i+2$ can be B. The *value* of a plan is the total number of steps that you manage to execute over the n minutes: so its the sum of a_i over all minutes in which the job is on A, plus the sum of b_i over all minutes in which the job is on B.

The problem. Given values $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$, find a plan of maximum value. (Such a strategy will be called *optimal.*) Note that your plan can start with either of the machines A or B in minute 1.

Example. Suppose $n = 4$, and the values of a_i and b_i are given by the following table.

	Minute 1	Minute 2	Minute 3	Minute 4
A	10	1	1	10
B	5	1	20	20

Then the plan of maximum value would be to choose A for minute 1, then *move* for minute 2, and then B for minutes 3 and 4. The value of this plan would be $10 + 0 + 20 + 20 = 50$.

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
In minute 1, choose the machine achieving the larger of $a_1$, $b_1$
Set $i = 2$
while $i \leq n$ do
    What was the choice in minute $i - 1$?
    If $A$:
        if $b_{i+1} > a_i + a_{i+1}$ then
            Choose move in minute $i$ and $B$ in minute $i + 1$
            Proceed to iteration $i + 2$
        else
            Choose $A$ in minute $i$
            Proceed to iteration $i + 1$
        end if
    If $B$: behave as above with roles of $A$ and $B$ reversed
end while
```

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an efficient algorithm that takes values for $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$ and returns the *value* of an optimal plan.

Solution:

(a) Here are two examples:

	Minute 1	Minute 2
A	1	10
B	1	20

The greedy algorithm would choose A for both steps, while the optimal solution would be to choose B for both steps.

	Minute 1	Minute 2	Minute 3	Minute 4
A	2	1	1	200
B	1	1	20	100

The greedy algorithm would choose A, then move, then choose B for final two steps. The optimal solution would be to choose A for all four steps.

(b) Let $Opt(i, A)$ denote the maximum value of a plan in minutes 1 through i that ends on machine A, and define $Opt(i, B)$ analogously for B.

Now, if you are machine A in minute i, where were you in minute $i - 1$? Either on machine A, or in the process of moving from machine B. In the first case, we have $Opt(i, A) = a_i + Opt(i - 1, A)$. In the second case, since you were last at B in minute $i - 2$, we have $Opt(i, A) = a_i + Opt(i - 2, B)$. Thus, overall, we have

$$Opt(i, A) = a_i + \max(Opt(i - 1, A), Opt(i - 2, B))$$

A symmetric formula holds for $Opt(i, B)$. The full algorithm initializes $Opt(1, A) = a_1$ and $Opt(1, B) = b_1$. Then, for $i = 2, 3, \cdots, n$, it computes $Opt(i, A)$ and $Opt(i, B)$ using the recurrence. This takes constant time for each of $n - 1$ iterations, and so that total time is $O(n)$.

This is an alternate solution. Let $Opt(i)$ be the maximum value of a plan in minutes 1 through i. Also, initialize $Opt(-1) = Opt(0) = 0$. Now, in minute i, we ask: when was the most recent minute in which we moved? If this was minute $k - 1$, then $Opt(i)$ would be equal to the best we could do on a single machine from minutes k through i. Thus, we have

$$Opt(i) = \max_{1 \le k \le i} Opt(k-2) + \max\left[\sum_{l=k}^{i} a_l, \sum_{l=k}^{i} b_l\right]$$

The full algorithm then builds up these values for $i = 2, \cdots, n$. Each iteration takes $O(n)$ time to compute the maximum, so the total running time is $O(n^2)$.

Problem 2 [15 pts]

You have just opened a new bank account, where you will receive amounts in euro currency. When you opened the account, you had no euros in it and no lei in your possession (lei is the currency used in Romania). During each of the next N days, you will receive various amounts of euros (a_i on the i-th day); the amounts may be negative, in which case the total amount of euros in your account will decrease. It is possible for you to have a negative balance of euros in your account at times. At the end of each day, the bank allows you to convert the whole amount of euro in your account into lei. The daily exchange rate varies according to the following rule: during the K-th day, an euro can be exchanged for K lei. The bank also charges you T lei for each conversion. Therefore, if at the end of day K you have S euros in your account, and you decide to convert them, you will receive $(S \cdot K - T)$ lei (of course, S can be negative). At the end of day N, you must convert your remaining euros into lei, even if the amount is non-positive.

Input. N, T, and $a_1, a_2, \ldots, a_N$, where a_i is the amount of euros you will receive at the beginning of the i-th day.

Output. Your objective is to maximize the final amount of lei you have at the end of the N-day period.

Example. When $N = 7$, $T = 1$, and the a_i sequence is $-10, 3, -2, 4, -6, 2, 3$. The optimal solution is 17 lei via the following conversion choices. At the end of the first day, you exchange the -10 euros you have received for $-11 = -10 \cdot 1 - 1$ lei. The second conversion takes place at the end of the 5-th day, when you have -1 euros in your account $(3 - 2 + 4 - 6)$. The amount of lei you gain is $-6 = (3 - 2 + 4 - 6) \cdot 5 - 1$. You make the last conversion at the end of the 7th day, using the last 5 euros in your account $(2 + 3)$. The amount of lei you gain is $34 = (2 + 3) \cdot 7 - 1$. The final amount of lei is $17 = -11 - 6 + 34$.

Task. Please design an algorithm for this problem that runs in time $O(N^2)$.

Solution: Let $OPT(j)$ be the maximum amount of lei one could have by the end of the j-th day if he has to convert all the euros received on and before that day. Clearly $OPT(N)$ is the final solution to the problem. As the boundary condition, we have $OPT(0) = 0$.

For each $j \in \{1, 2, 3, \ldots, N\}$ we enumerate the last time before the j-th day on which one makes a conversion, and we have

$$\begin{aligned} OPT(j) &= \max_{i \in \{0,1,2,\ldots,j-1\}} \{OPT(i) + (a_{i+1} + a_{i+2} + \cdots + a_j) \cdot j - T\} \\ &= \max_{i \in \{0,1,2,\ldots,j-1\}} \{OPT(i) + (S_j - S_i) \cdot j - T\}, \end{aligned}$$

where $S_j = a_1 + a_2 + \cdots + a_j$ is the subsum array and can be computed in $O(N)$ time. Since it takes $O(j)$ time to evaluate the Bellman Equation for $OPT(j)$, the overall time complexity for computing $OPT(N)$ is $O(N^2)$.

Problem 3 [15 pts]

Consider the sequence alignment problem over a four-letter alphabet $\{z_1, z_2, z_3, z_4\}$, with a given gap cost and given mismatch costs. Assume that each of these parameters is a positive integer.

Suppose you are given two strings $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_n$ and a proposed alignment between them. Give an $O(mn)$ algorithm to decide whether this alignment is the *unique* minimum-cost alignment between A and B.

Solution:

As discussed in section 6.6 and 6.7 of the book, consider the directed acyclic graph $G = (V, E)$ which say vertex s in the upper left corner and t in the lower right corner, whose s-t paths correspond to global alignments between A and B. For a set of edges $F \subset E$, let $c(F)$ denote the total cost of edges in F. If P is a path in G, let $\Delta(P)$ denote the set of diagonal edges in P (i.e. the matches in the alignment).

Let Q denote the s-t path corresponding to the given alignment. Let E_1 denote the horizontal or vertical edges in G, E_2 represent the diagonal edges in G that do not belong to $\Delta(Q)$, and $E_3 = \Delta(Q)$. Note that $E = E_1 \cup E_2 \cup E_3$.

Let $\epsilon = 1/2n$ and $\epsilon' = 1/4n^2$. We form a graph G' by subtracting ϵ from the cost of every edge in E_3. Thus, G' has the same structure as G, but new cost function c'.

Now we claim that path Q is a minimum cost s-t path in G' if and only if it is the unique minimum cost s-t path in G. To prove this, we first observe that

$$c'(Q) = c(Q) + \epsilon' |\Delta(Q)| \leq c(Q) + 1/4$$

and if $P \neq Q$ then,

$$c'(P) = c(P) + \epsilon' |\Delta(P \cap Q)| - \epsilon |\Delta(P - Q)| \geq c(P) - 1/2$$

Now, if Q was the unique minimum-cost path in G, the $c(Q) \leq c(P) + 1$ for every other path P, so $c'(Q) < c'(P)$ by the above inequalities, and hence Q is a minimum-cost s-t path in G'. To prove the converse, we observe from the above inequalities that $c'(Q) - c(Q) > c'(P) - c(P)$ for every other path P; thus if Q is a minimum-cost path in G', it is the unique minimum-cost path in G.

Thus, the algorithm is to find the minimum cost of s-t path in G', $O(mn)$ time and $O(m+n)$ space. Q is the minimum cost A-B alignment if and only if this cost matches $c'(Q)$.

Problem 4 [15 pts]

You're consulting for a group of people (who would prefer not to be mentioned here by name) whose jobs consist of monitoring and analyzing electronic signals coming from ships in coastal Atlantic waters. They want a fast algorithm for a basic primitive that arises frequently: "untangling" a superposition of two known signals. Specifically, they're picturing a situation in which each of two ships is emitting a short sequence of 0s and 1s over and over, and they want to make sure that the signal they're hearing is simply an *interleaving* of these two emissions, with nothing extra added in.

This describes the whole problem; we can make it a little more explicit as follows. Given a string x consisting of 0s and 1s, we write x^k to denote k copies of x concatenated together. We say that a string x' is a *repetition* of x if it is a prefix of x^k for some number k. So $x' = 10110110110$ is a repetition of $x = 101$.

We say that a string s is an *interleaving* of x and y if its symbols can be partitioned into two (not necessarily contiguous) subsequences s' and s'', so that s' is a repetition of x and s'' is a repetition of y. (So each symbol in s must belong to exactly one of s' or s''.) For example, if $x = 101$ and $y = 00$, then $s = 100010101$ is an interleaving of x and y, since characters $1, 2, 5, 7, 8, 9$ form 101101—a repetition of x— and the remaining characters $3, 4, 6$ form 000—a repetition of y.

In terms of our application, x and y are the repeating sequences from the two ships, and s is the signal we're listening to: We want to make sure s "unravels" into simple repetitions of x and y. Give an efficient algorithm that takes strings s, x, and y and decides if s is an interleaving of x and y.

Solution: Let's suppose that s has n characters total. To make things easier to think about, let's consider the repetition x' of x consisting of exactly n characters, and the repetition y' of y consisting of exactly n characters. Our problem can be phrased as : is s an interleaving of x' and y'? The advantage of working with these elongated strings is that we don't need to "wrap around" and consider multiple periods of x' and y' — each is already as long as s.

Let $s[j]$ denote the j^{th} character of s, and let $s[1:j]$ denote the first j characters of s. We define the analogous notation for x' and y'. We know that if s is an interleaving of x' and y', then its last character comes from either x' or y'. Removing this character (whatever it is), we get a smaller recursive problem on $s[1:n-1]$ and prefixes of x' and y'.

Thus, we consider sub-problems defined by prefixes of x' and y'. Let $M[i,j] = yes$ if $s[1:i+j]$ is an interleaving of $x'[1:i]$ and $y'[1:j]$. If there is such an interleaving, then the final character is either $x'[i]$ or $y'[j]$, and so we have the following basic recurrence:

$$M[i,j] = yes \text{ if and only if } M[i-1,j] = yes \text{ and } s[i+j] = x'[i], \text{ or } M[i,j-1] = yes \text{ and } s[i+j] = y'[j].$$

We can build these up via the following loop.

Running time analysis: There are $O(n^2)$ $M[i,j]$'s to build up, and each takes constant time to fill in from the results on previous sub-problems; thus the total running time is $O(n^2)$.

```
    $M[0,0] = yes$
    for $k = 1, 2, \ldots, n$ do
        for all pairs $(i, j)$ so that $i + j = k$ do
            if $M[i-1, j] = yes$ and $s[i+j] = x'[i]$ then
                $M[i, j] = yes$
            else if $M[i, j-1] = yes$ and $s[i+j] = y'[j]$ then
                $M[i, j] = yes$
            else
                $M[i, j] = no$
            end if
        end for
    end for
    Return "yes" if and only if there is some pair $(i, j)$ with $i + j = n$ so that $M[i, j] = yes$
```

Problem 5 [15 pts]

Suppose it's nearing the end of the semester and you're taking n courses, each with a final project that still has to be done. Each project will be graded on the following scale: It will be assigned an integer number on a scale of 1 to $g > 1$, higher numbers being better grades. Your goal, of course, is to maximize your average grade on the n projects.

You have a total of $H > n$ hours in which to work on the n projects cumulatively, and you want to decide how to divide up this time. For simplicity, assume H is a positive integer, and you'll spend an integer number of hours on each project. To figure out how best to divide up your time, you've come up with a set of functions $\{f_i : i = 1, 2, \ldots, n\}$(rough estimates, of course) for each of your n courses; if you spend $h \leq H$ hours on the project for course i, you'll get a grade of $f_i(h)$. (You may assume that the functions f_i are *nondecreasing*: if $h < h'$, then $f_i(h) \leq f_i(h')$.)

So the problem is: Given these functions $\{f_i\}$, decide how many hours to spend on each project (in integer values only) so that your average grade, as computed according to the f_i, is as large as possible. In order to be efficient, the running time of your algorithm should be polynomial in n, g, and H; none of these quantities should appear as an exponent in your running time.

Solution: First note that it is enough to maximize one's *total* grade over the n courses, since this differs from the average grade by the fixed factor of n. Let the (i, h)-*subproblem* be the problem in which one wants to maximize one's grade on the first i courses, using at most h hours.

Let $A[i, h]$ be the maximum total grade that can be achieved for this subproblem. Then $A[0, h] = 0$ for all h, and $A[i, 0] = \sum_{j=1}^{i} f_j(0)$. Now, in the optimal solution to the (i, h)-subproblem, one spends k hours on course i for some value of $k \in [0, h]$; thus

$$A[i, h] = \max_{0 \leq k \leq h} f_i(k) + A[i-1, h-k].$$

We also record the value of k that produces this maximum. Finally, we output $A[n, H]$, and can trace-back through the entries using the recorded values to produce the optimal distribution of time.

Running time analysis: The total time to fill in each entry $A[i, h]$ is $O(H)$, and there are nH entries, for a total time of $O(nH^2)$. The running time for trace-back is equal to $O(n)$. Therefore, total running time is $O(nH^2)$.

Problem 6 [15 pts]

To assess how "well-connected" two nodes in a directed graph are, one can not only look at the length of the shortest path between them, but can also count the *number* of shortest paths.

This turns out to be a problem that can be solved efficiently, subject to some restrictions on the edge costs. Suppose we are given a directed graph $G = (V, E)$, with costs on the edges; the costs may be positive or negative, but every cycle in the graph has strictly positive cost. We are also given two nodes $v, w \in V$. Give an efficient algorithm that computes the number of shortest $v - w$ paths in G. (The algorithm should not list all the paths; just the number suffices.)

Solution: Let c_e denote the cost of the edge e and we will overload the notation and write c_{st} to denote the cost of the edge between the nodes s and t.

This problem is by its nature quite similar to the shortest path problem. Let us consider a two-parameter function $Opt(i, s)$ denoting the optimal cost of shortest path from v to s using *exactly* i edges, and let $N(i, s)$ denote the number of such paths.

We start by setting $Opt(i, v) = 0$ and $Opt(i, v') = \infty$ for all $v' \neq v$. Also set $N(i, v) = 1$ and $N(i, v') = 0$ for all $v' \neq v$. Intuitively this means that the source v is reachable with cost 0 and there is currently one path to achieve this.

Then we compute the following recurrence:

$$Opt(i, s) = \min_{(t,s)\in E} \{Opt(i - 1, t) + c_{ts}\}. \tag{1}$$

The above recurrence means that in order to travel to node s using exactly i edges, we must travel a predecessor node t using exactly $i - 1$ edges and then take the edge connecting t to s. Once of course the optimal cost value has been computed, the number of paths that achieve this optimum would be computed by the following recurrence:

$$N(i, s) = \sum_{(t,s)\in E \text{ and } Opt(i,s)=Opt(i-1,t)+c_{ts}} N(i - 1, t). \tag{2}$$

Running time analysis: The above recurrence can be calculated by a double loop, where the outside loops over i and the inside loops over all the possible nodes s. Note that it is enough to search i up to $|V| - 1$. Here is the reason. If a path's length is at least $|V|$, there will be a cycle in this path. After removing this cycle, we will get a path with shorter length, which means the original path isn't shortest. For a given pair of (i, s), computing $Opt(i, s)$, and $N(i, s)$ need $O(d(s))$ time, where $d(s)$ is the degree of s. Therefore, total time for this procedure should be $O(\sum_i \sum_s d(s)) = O(|V||E|)$. Once the recurrence have been solved, our target optimal path to w is obtained by taking the minimum of all the paths of different lengths to w - this is :

$$Opt(w) = \min_i \{Opt(i, w)\}. \tag{3}$$

And the number of such paths can be computed by adding up the counters of all the paths which achieve the minimal cost.

$$N(w) = \sum_{Opt(i,w)=Opt(w)} N(i, w). \tag{4}$$

To get $Opt(w)$, and $N(w)$, only $O(|V|)$ time is required.

Overall, the running time should be equal to $O(|V| + |V||E|) = O(|V||E|)$.

Problem 7 [10 pts]

Let $G = (V, E)$ be a graph with n nodes in which each pair of nodes is joined by an edge. There is a positive weight w_{ij} on each edge (i, j); and we will assume these weights satisfy the *triangle inequality* $w_{ik} \leq w_{ij} + w_{jk}$. For a subset $V' \subset V$, we will use $G[V']$ to denote the subgraph (with edge weights) induced on the nodes in V'.

We are given a set $X \subset V$ of k *terminals* that must be connected by edges. We say that a *Steiner tree* on X is a set Z so that $X \subset Z \subset V$, together with a spanning subtree T of $G[Z]$. The *weight* of the Steiner tree is the weight of the tree T.

Show that there is function $f(\cdot)$ and a *polynomial function* $p(\cdot)$ so that the problem of finding a minimum-weight Steiner tree on X can be solved in time $O(f(k) \cdot p(n))$.

Solution: We will say that an *enriched* subset of V is one that contains at most one node not in X. There are $O(2^k n)$ enriched subsets. The overall approach will be based on *dynamic programming*: for each enriched subset Y, we will compute the following information, building it up in order of increasing $|Y|$.

- The cost $f(Y)$ of the minimum spanning tree on Y.
- The cost $g(Y)$ of the minimum Steiner tree on Y.

Consider a given Y, and suppose it has the form $X' \cup \{i\}$ where $X' \subset X$ and $i \notin X$. (The case in which $Y \subset X$ is easier.) For such a Y, one can compute $f(Y)$ in time $O(|E| + |V| \log |V|) = O(n^2)$ by Prim's algorithm based on Fibonacci heap.

Now, the minimum Steiner tree T on Y either has no extra nodes, in which case $g(Y) = f(Y)$, or else it has an extra node j of degree at least 3. Here is the reason. If it has degree 1, j will be the leaf node. Then after removing this node, we can get another valid Steriner tree with smaller weight, which is a contradiction. If it has degree 2, removing j and adding one edge connecting two nodes j is connected to will also produce another valid Steriner tree with smaller weight by triangle inequality. Let $T_1, \ldots, T_r$ be the subtrees obtained by deleting j, with $i \in T_1$. Let p be the node in T_1 with an edge to j. Let T' be the subtree induced by adding j to T_2, and let T'' be the subtree induced by adding j to $T_3 \cup \cdots \cup T_r$. Let Y_1 be the nodes of Y in T_1, Y' those in T', and Y'' those in T''. Each of these is an enriched set of size less than $|Y|$, and T_1, T', and T'' are the minimum Steiner trees on these sets. Moreover, the cost of T is simply

$$g(Y_1) + g(Y') + g(Y'') + w_{jp}.$$

Thus we can compute $g(Y)$ as follows, using the values of $g(\cdot)$ already computed for smaller enriched sets. We enumerate all partitions of Y into Y_1, Y_2, Y_3 (with $i \in Y_1$), all $p \in Y_1$, and all $j \in V$, and we determine the value of

$$g(Y_1) + g(Y_2 \cup \{j\}) + g(Y_3 \cup \{j\}) + w_{jp}.$$

This can be done by looking up values we have already computed, since each of Y_1, Y', Y'' is a smaller enriched set. If any of these sums is less than $f(Y)$, we return the corresponding tree as the minimum Steiner tree; otherwise we return the minimum spanning tree on Y.

Running time analysis: Computing $f(Y)$ and $g(Y)$ take time $O(\max\{3^k \cdot kn + n^2\}) = O(3^k \cdot kn^2)$ for each enriched set Y. Since there are $O(2^k n)$ enriched subsets, overall running time of this algorithm should be equal to $O(6^k \cdot n^4)$ noticing that $k \leq n$.