** GREEDY **

1. Completion times. (4.13)

13. A small business—say, a photocopying service with a single large machine—faces the following scheduling problem. Each morning they get a set of jobs from customers. They want to do the jobs on their single machine in an order that keeps their customers happiest. Customer $i$'s job will take $t_i$ time to complete. Given a schedule (i.e., an ordering of the jobs), let $C_i$ denote the finishing time of job $i$. For example, if job $j$ is the first to be done, we would have $C_j = t_j$, and if job $j$ is done right after job $i$, we would have $C_j = C_i + t_j$. Each customer $i$ also has a given weight $w_i$ that represents his or her importance to the business. The happiness of customer $i$ is expected to be dependent on the finishing time of $i$'s job. So the company decides that they want to order the jobs to minimize the weighted sum of the completion times, $\sum_{i=1}^{n} w_i C_i$.

Design an efficient algorithm to solve this problem. That is, you are given a set of $n$ jobs with a processing time $t_i$ and a weight $w_i$ for each job. You want to order the jobs so as to minimize the weighted sum of the completion times, $\sum_{i=1}^{n} w_i C_i$.

**Example.** Suppose there are two jobs: the first takes time $t_1 = 1$ and has weight $w_1 = 10$, while the second job takes time $t_2 = 3$ and has weight $w_2 = 2$. Then doing job 1 first would yield a weighted completion time of $10 \cdot 1 + 2 \cdot 4 = 18$, while doing the second job first would yield the larger weighted completion time of $10 \cdot 4 + 2 \cdot 3 = 46$. 
2. ClubNet. (4.28)

28. Suppose you’re a consultant for the networking company CluNet, and they have the following problem. The network that they’re currently working on is modeled by a connected graph \( G = (V, E) \) with \( n \) nodes. Each edge \( e \) is a fiber-optic cable that is owned by one of two companies—creatively named \( X \) and \( Y \)—and leased to CluNet.

Their plan is to choose a spanning tree \( T \) of \( G \) and upgrade the links corresponding to the edges of \( T \). Their business relations people have already concluded an agreement with companies \( X \) and \( Y \) stipulating a number \( k \) so that in the tree \( T \) that is chosen, \( k \) of the edges will be owned by \( X \) and \( n - k - 1 \) of the edges will be owned by \( Y \).

CluNet management now faces the following problem. It is not at all clear to them whether there even exists a spanning tree \( T \) meeting these conditions, or how to find one if it exists. So this is the problem they put to you: Give a polynomial-time algorithm that takes \( G \), with each edge labeled \( X \) or \( Y \), and either (i) returns a spanning tree with exactly \( k \) edges labeled \( X \), or (ii) reports correctly that no such tree exists.

3. Kruskal’s variant. (4.31)

31. Let’s go back to the original motivation for the Minimum Spanning Tree Problem. We are given a connected, undirected graph \( G = (V, E) \) with positive edge lengths \( \{\ell_e\} \), and we want to find a spanning subgraph of it. Now suppose we are willing to settle for a subgraph \( H = (V, F) \) that is “denser” than a tree, and we are interested in guaranteeing that, for each pair of vertices \( u, v \in V \), the length of the shortest \( u-v \) path in \( H \) is not much longer than the length of the shortest \( u-v \) path in \( G \). By the length of a path \( P \) here, we mean the sum of \( \ell_e \) over all edges \( e \) in \( P \).

Here’s a variant of Kruskal’s Algorithm designed to produce such a subgraph.
• First we sort all the edges in order of increasing length. (You may assume all edge lengths are distinct.)

• We then construct a subgraph \( H = (V, E) \) by considering each edge in order.

• When we come to edge \( e = (u, v) \), we add \( e \) to the subgraph \( H \) if there is currently no \( u-v \) path in \( H \). (This is what Kruskal’s Algorithm would do as well.) On the other hand, if there is a \( u-v \) path in \( H \), we let \( d_{uv} \) denote the length of the shortest such path; again, length is with respect to the values \( \{ \ell_e \} \). We add \( e \) to \( H \) if \( 3\ell_e < d_{uv} \).

In other words, we add an edge even when \( u \) and \( v \) are already in the same connected component, provided that the addition of the edge reduces their shortest-path distance by a sufficient amount.

Let \( H = (V, E) \) be the subgraph of \( G \) returned by the algorithm.

(a) Prove that for every pair of nodes \( u, v \in V \), the length of the shortest \( u-v \) path in \( H \) is at most three times the length of the shortest \( u-v \) path in \( G \).

(b) Despite its ability to approximately preserve shortest-path distances, the subgraph \( H \) produced by the algorithm cannot be too dense.

Let \( f(n) \) denote the maximum number of edges that can possibly be produced as the output of this algorithm, over all \( n \)-node input graphs with edge lengths. Prove that

\[
\lim_{n \to \infty} \frac{f(n)}{n^2} = 0.
\]

** DYNAMIC PROGRAMMING **

4. Mobile wireless devices. (6.14)

14. A large collection of mobile wireless devices can naturally form a network in which the devices are the nodes, and two devices \( x \) and \( y \) are connected by an edge if they are able to directly communicate with each other (e.g., by a short-range radio link). Such a network of wireless devices is a highly dynamic object, in which edges can appear and disappear over time as the devices move around. For instance, an edge \((x, y)\) might disappear as \( x \) and \( y \) move far apart from each other and lose the ability to communicate directly.

In a network that changes over time, it is natural to look for efficient ways of maintaining a path between certain designated nodes. There are
two opposing concerns in maintaining such a path: we want paths that are short, but we also do not want to have to change the path frequently as the network structure changes. (That is, we’d like a single path to continue working, if possible, even as the network gains and loses edges.) Here is a way we might model this problem.

Suppose we have a set of mobile nodes \( V \), and at a particular point in time there is a set \( E_0 \) of edges among these nodes. As the nodes move, the set of edges changes from \( E_0 \) to \( E_1 \), then to \( E_2 \), then to \( E_3 \), and so on, to an edge set \( E_b \). For \( i = 0, 1, 2, \ldots, b \), let \( G_i \) denote the graph \((V, E_i)\). So if we were to watch the structure of the network on the nodes \( V \) as a “time lapse,” it would look precisely like the sequence of graphs \( G_0, G_1, G_2, \ldots, G_{b-1}, G_b \). We will assume that each of these graphs \( G_i \) is connected.

Now consider two particular nodes \( s, t \in V \). For an \( s-t \) path \( P \) in one of the graphs \( G_i \), we define the length of \( P \) to be simply the number of edges in \( P \), and we denote this \( \ell(P) \). Our goal is to produce a sequence of paths \( P_0, P_1, \ldots, P_b \) so that for each \( i, P_i \) is an \( s-t \) path in \( G_i \). We want the paths to be relatively short. We also do not want there to be too many changes—points at which the identity of the path switches. Formally, we define \( \text{changes}(P_0, P_1, \ldots, P_b) \) to be the number of indices \( i \) \((0 < i < b - 1)\) for which \( P_i \neq P_{i+1} \).

Fix a constant \( K > 0 \). We define the cost of the sequence of paths \( P_0, P_1, \ldots, P_b \) to be

\[
\text{cost}(P_0, P_1, \ldots, P_b) = \sum_{i=0}^{b-1} \ell(P_i) + K \cdot \text{changes}(P_0, P_1, \ldots, P_b).
\]

(a) Suppose it is possible to choose a single path \( P \) that is an \( s-t \) path in each of the graphs \( G_0, G_1, \ldots, G_b \). Give a polynomial-time algorithm to find the shortest such path.

(b) Give a polynomial-time algorithm to find a sequence of paths \( P_0, P_1, \ldots, P_b \) of minimum cost, where \( P_i \) is an \( s-t \) path in \( G_i \) for \( i = 0, 1, \ldots, b \).
5. **Iterated Matrix Products**

Consider the product of rectangular matrices $M_1 \cdot M_2 \cdot \ldots \cdot M_{n-1} \cdot M_n$. Each of these matrices $M_i$ has dimensions $r_i \times c_i$, $1 \leq i \leq n$, but for compatibility reasons, $c_i = r_{i+1}$, $1 \leq i \leq n-1$ (otherwise $M_i$ and $M_{i+1}$ cannot be multiplied together). The final resulting product $M$ is an $r_1 \times c_n$ matrix.

You may be surprised to learn that the total number of scalar multiplications used to compute the product actually depends on the order in which we perform it. For instance, depending on the dimensions $r_1$, $c_1=r_2$, $c_2=r_3$, $c_3$, computing $(M_1\cdot M_2)\cdot M_3$ may take more (or less) scalar products than computing $M_1\cdot(M_2\cdot M_3)$.

**A)** Argue that if we use the simple (straight forward) definition of matrix product, multiplying two compatible matrices $A$ (of size $a \times b$) and $B$ (of size $b \times c$), uses in total $a \times b \times c$ scalar multiplications.

**B)** How many scalar multiplications would be involved in the computation of the products $(M_1\cdot M_2)\cdot M_3$ and $M_1\cdot(M_2\cdot M_3)$, if the matrices have dimensions $r_1=9$, $c_1=r_2=101$, $c_2=r_3=53$, and $c_3=41$.

**C)** Given a sequence of compatible matrices $M_i : r_i \times c_i$, $1 \leq i \leq n$, find a **Dynamic Programming** algorithm to determine the minimal number of scalar multiplications used to compute

$$M = M_1 \cdot M_2 \cdot \ldots \cdot M_{n-1} \cdot M_n.$$ 

Your algorithm must figure out the cheapest way of using parentheses to compute this product. NOTE: The value of $M$ does not depend on the order in which you compute the product, but the running-time will!

**Analyze the running-time of your algorithm.**
D) Given a sequence of compatible matrices $M_i : r_i \times c_i$, $1 \leq i \leq n$, and a matrix final $M : r_1 \times c_n$, we can determine whether $M = M_1 \cdot M_2 \cdot \ldots \cdot M_{n-1} \cdot M_n$ (or not) by choosing a random $c_n \times 1$ binary vector $R$ and comparing $M \cdot R$ to $M_1 \cdot (M_2 \cdot \ldots \cdot (M_{n-1} \cdot (M_n \cdot R)) \ldots)$. 

Prove that

If $M = M_1 \cdot \ldots \cdot M_{n-1} \cdot M_n$ then for all $R$, $M \cdot R = M_1 \cdot \ldots \cdot (M_{n-1} \cdot (M_n \cdot R)) \ldots$.

However, if $M \neq M_1 \cdot \ldots \cdot M_{n-1} \cdot M_n$ then $M \cdot R \neq M_1 \cdot \ldots \cdot (M_{n-1} \cdot (M_n \cdot R)) \ldots$ for at least half the possible $R$'s. (Your proof should not make any assumption on the entries of the matrices; they could be Integers, Rationals, Reals, Complex or anything compatible with products with the bits of $R$…)

E) Analyze the running-time and success probability of the previous algorithm.

F) Compare this approach to the running-time you analyzed in C).

6. Maximum Deadline. (6.28)

28. Recall the scheduling problem from Section 4.2 in which we sought to minimize the maximum lateness. There are $n$ jobs, each with a deadline $d_i$ and a required processing time $t_i$, and all jobs are available to be scheduled starting at time $s$. For a job $i$ to be done, it needs to be assigned a period from $s_i \geq s$ to $f_i = s_i + t_i$, and different jobs should be assigned nonoverlapping intervals. As usual, such an assignment of times will be called a schedule.

In this problem, we consider the same setup, but want to optimize a different objective. In particular, we consider the case in which each job must either be done by its deadline or not at all. We'll say that a subset $J$ of the jobs is schedulable if there is a schedule for the jobs in $J$ so that each of them finishes by its deadline. Your problem is to select a schedulable subset of maximum possible size and give a schedule for this subset that allows each job to finish by its deadline.

(a) Prove that there is an optimal solution $J$ (i.e., a schedulable set of maximum size) in which the jobs in $J$ are scheduled in increasing order of their deadlines.

(b) Assume that all deadlines $d_i$ and required times $t_i$ are integers. Give an algorithm to find an optimal solution. Your algorithm should run in time polynomial in the number of jobs $n$, and the maximum deadline $D = \max_i d_i$.

†Read Section 4.2 if necessary.