COSC 311: Algorithms
HW3: Greedy and DP Algorithms

Due Monday, October 23, 12pm

Please type up your solutions. If you discuss any of the problems with your classmates, please note at the top of your submission with whom you consulted.

1 Fun with Scheduling

1) Scheduling to minimize response time.
In many computer systems settings, all of the jobs that are submitted to a server are allowed to run, and the goal is to ensure that all jobs complete running as quickly as possible. Suppose we are given a set J consisting of $|J| = n$ jobs. Each job j arrives to the system at time $a(j)$, and has size $x(j)$ (the job must occupy the server for a contiguous block of time of length $x(j)$). Our server can only run one job at a time. A job j's *response time* $t(j)$ is defined as the time that passes between its arrival time, $a(j)$, and its finish time, $f(j)$; that is, $t(j) = f(j) - a(j)$.

(a) Assume that all jobs arrive at time 0, so $a(j) = 0$ for all j. Give an efficient (i.e., polynomial in n) algorithm that produces a schedule (i.e., a start time $s(j)$ for each job j) that minimizes the total response time $T = \sum_{j=1}^{n} t(j)$. Prove that your algorithm correctly produces the optimal schedule. What is the runtime of your algorithm?

(b) Now let's relax the assumption that all jobs arrive at time 0, and say that each job j arrives at some time $a(j) \geq 0$. We only become aware of job j at its arrival time, so at all times we can only make our scheduling decisions based on our knowledge of the jobs that already have arrived. We will make one other change, which is that we are no longer required to run a job for a contiguous block of time. Instead, we are allowed to *preempt* jobs: we can run part of the job, pause it, and then run the rest of it later on. Our goal is still to minimize the total response time. Does your algorithm from part (a) still give the optimal solution? If so, explain why (you do not need to give a formal proof). If not, give a counterexample and an optimal algorithm, and explain intuitively why your algorithm is optimal (you do not need to give a formal proof).

2) Scheduling with hard deadlines.
Suppose we have a set of n jobs J, where each job $j \in J$ has a size $x(j)$ and a deadline $d(j)$. The deadlines are *hard* deadlines, meaning that if we cannot complete a job by its deadline we must drop the job. A schedule $\mathcal{S}$ consists of some subset of our jobs, where we specify the start time $s(j)$ for each job included in the schedule. A job j's finish time is $f(j) = s(j) + x(j)$; that is, we are required to schedule each job for a contiguous block of time. A schedule is feasible if all jobs included in the schedule have $f(j) \leq d(j)$. Our goal is to come up with a feasible schedule $\mathcal{S}$ that maximizes $|\mathcal{S}|$; that is, we include as many jobs as possible in our schedule. Give an algorithm that is polynomial in n and D (where D is the largest deadline) that accomplishes this. Explain why your algorithm works (you do not need to give a formal proof). What is the running time for your algorithm?

2 Graph Algorithms

3) Minimum bottleneck spanning trees.
The motivation behind the Minimum Spanning Tree problem is to find a tree that connects all nodes in a network and has minimum *total* cost. An alternative objective is to instead find a spanning tree for which the *most expensive edge* has as low a cost as possible.

Suppose we are given a connected graph $G = (V, E)$ with $|V| = n$ vertices, $|E| = m$ edges, and positive edge weights (you may assume all edge weights are distinct). Let $T = (V, E')$ be a spanning tree of G. We define the *bottleneck edge* of T to be the edge in T with the highest weight. A spanning tree T of G is a *minimum bottleneck spanning tree* if there is no spanning tree T' of G with a lower-weight bottleneck edge.

(a) Is every minimum bottleneck spanning tree of G a minimum spanning tree? If so, prove it. If not, give a counterexample.

(b) Is every minimum spanning tree of G a minimum bottleneck spanning tree? If so, prove it. If not, give a counterexample.

4) Finding all shortest paths.
We've seen multiple algorithms for finding shortest paths in graphs. However, algorithms like Dijkstra's algorithm and the Bellman-Ford algorithm only find *one* shortest path between a pair of nodes. It's possible that there are multiple different shortest paths of the same length between a pair of nodes s and t. Suppose we are given a directed graph $G = (V, E)$, where there are $|V| = n$ nodes and $|E| = m$ edges, and each edge has a positive weight. We are also given two nodes $s, t \in V$. Give an efficient (i.e., polynomial in n and m) algorithm that computes the number of shortest paths from s to t (you do not need to list all the paths, just state how many there are). Explain why your algorithm works (you do not need to give a formal proof). What is the runtime of your algorithm?

3 Miscellaneous

5) Gerrymandering.
Gerrymandering is the practice of cutting up electoral districts in ways that tend to favor one political party over the other. It is highly controversial whether gerrymandering should be legal, and the Supreme Court just heard arguments in the case Gill v. Whitford, which addresses the specific case of gerrymandering in Wisconsin's redistricting following the 2010 census.

Apart from being a controversial political topic, gerrymandering is also an interesting computational problem. There are underlying questions about how to group voters into districts so that the districts are balanced in terms of population, and balanced (or not) in terms of political leanings.

Suppose we have a set of n precincts $P_1, P_2, \ldots, P_n$, each containing exactly m registered voters. We need to divide these precincts into two districts, each consisting of $n/2$ of the precincts (you can assume that n is even). For each precinct, we have information on how many of the m voters in that precinct are registered to each of the two political parties (assume that every registered voter is a member of one of the two major parties). We say that the set of precincts is *susceptible* to gerrymandering if it is possible to divide the precincts into two districts such that the same party holds a majority in both districts. For example, suppose we have $n = 4$ precincts, each with $m = 100$ registered voters, and the following information about the registered voters:

Precinct	**1**	**2**	**3**	**4**
Number registered in party A	55	43	60	47
Number registered in party B	45	57	40	53

This set of precincts is susceptible to gerrymandering because if we group precincts 1 and 4 into one district and precincts 2 and 3 into the other, then party A has a majority in both districts. (This demonstrates why many people consider gerrymandering unfair: party A only has a small majority overall (205 to 195), but ends up with a majority in both districts.)

Your job is to give an efficient (i.e., polynomial in n and m) algorithm that determines whether a given set of precincts is susceptible to gerrymandering. Explain why your algorithm works (you do not need to give a formal proof). What is the running time of your algorithm?

4 Submission

This assignment is due on Monday, October 23, at 12pm. Please type up your solutions and bring a hard copy of your typed responses to submit in class.