COSC 311: ALGORITHMS
HW3: GREEDY AND DP ALGORITHMS
Solutions

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, $t(j) = f(j) - a(j)$.

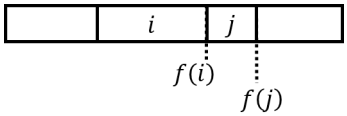
(a) Assume that all jobs arrive at time 0, so $a(j) = 0$ for all j. Give an efficient (i.e., polynomial in n) algorithm that produces a schedule (i.e., a start time $s(j)$ for each job j) that minimizes the total response time $T = \sum_{j=1}^{n} t(j)$. Prove that your algorithm correctly produces the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

Solution: The optimal algorithm is Shortest Job First (SJF). This is a greedy algorithm that at all times schedules the unfinished job j with the smallest size $x(j)$. For example, if we have three jobs with $x(1) = 4$, $x(2) = 2$, and $x(3) = 7$, then we schedule the jobs in order 2, 1, 3. Job 2 finishes at time $f(2) = 2$, job 1 finishes at time $f(1) = 6$, and job 3 finishes at time $f(3) = 13$. The total response time is 21.

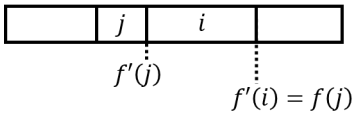
Intuitively, SJF is the right thing to do because if we choose a job j to run first, *all* jobs experience the size of job j as part of their response time. Since all jobs have to incur a cost equal to $x(j)$ for whatever j we schedule first, it makes sense to choose the smallest job possible as the first job. Consider our above example. If we scheduled job 3 first, then jobs 2 and 1, we would have $f(3) = 7$, $f(2) = 9$, and $f(1) = 13$. The total time it takes to run all jobs does not change (the last job still ends at time 13), but now the total response time is 29 because jobs 2 and 1 had to wait for the much larger job 3 to finish.

Theorem 1. *SJF minimizes the total response time.*

Proof. Our proof will be by contradiction. Suppose we have some schedule OPT that minimizes the total response time, but is not SJF. Then at some point, OPT schedules a job i before job j, where $x(i) > x(j)$. In particular, there must be a time when job i is scheduled *immediately* before job j. (Why? Because if there is ever a time when job a is scheduled before job b and $x(a) > x(b)$, we can imagine walking down the schedule from a to b. Since $x(b)$ is smaller than $x(a)$, there must be some first point along this walk when the job size decreases. This occurs at a pair of jobs i, j where i is scheduled immediately before j and $x(i) > x(j)$.) So we have the scenario in figure (a) below.



(a) Original schedule



(b) After swapping i and j

Consider what happens if we swap the order of jobs i and j in our schedule; now we have the scenario in figure (b). Under OPT, the total response time for all jobs is

$$T = \sum_{k=1}^{n} f(k) = \sum_{k \neq i,j} f(k) + f(i) + f(j).$$

After making the swap, job j finishes at some earlier time $f'(j) < f(j)$. Job i finishes later than it did before, but job i's new finish time is the same as job j's old finish time: $f'(i) = f(j)$. So our new total response time is

$$T' = \sum_{k \neq i,j} f(k) + f'(j) + f(j) \leq T.$$

This contradicts our assumption that there is some optimal schedule OPT that is not SJF. Hence the optimal schedule must be the schedule produced by SJF. □

In order to schedule the jobs in SJF order, we must first sort the jobs in increasing order of $x(j)$. This takes time $\Theta(n \lg n)$. We then make a single pass through the sorted list of jobs—imagine that at this point we relabel the jobs so that job 1 is the job with smallest $x(j)$ and job n is the job with largest $x(j)$—and assign each job a start time $s(j) = s(j-1) + x(j-1)$ (the start time of job 1 is $s(1) = 0$). This additional pass takes linear time. The overall runtime therefore is $\Theta(n \lg n)$.

(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).

Solution: SJF is no longer optimal when our jobs have positive arrival times and preemption is allowed. Here is a simple counterexample: suppose job 1 arrives at time $a(1) = 0$ and has size $x(1) = 5$, and job 2 arrives at time $a(2) = 4$ and has size $x(2) = 2$. Under SJF, we run job 1 from time 0 until time 4. At time 4, job 2 arrives and since $x(2) < x(1)$ we preempt job 1 and run job 2 instead. At time 6, job 2 finishes and we resume job 1, which finishes at time 7. Job 1 has response time $t(1) = f(1) - a(1) = 7 - 0 = 7$ and job 2 has response time $t(2) = f(2) - a(2) = 6 - 4 = 2$; the total response time is $T = 9$. A better schedule would be to run job 1 from time 0 until time 5, when it finishes, and then run job 2 from time 5 to time 7. Under this schedule, job 1 has response

time $t'(1) = f'(1) - a(1) = 5 - 0 = 5$ and job 2 has response time $t'(2) = f'(2) - a(2) = 7 - 4 = 3$; the total response time is $T' = 8$.

Intuitively, the problem with SJF is that when job 2 arrives, a lot of job 1 has already completed running, so our intuition that it's best to make everyone wait for the job with smallest size no longer makes sense. If we make job 2 wait for job 1 to finish running, job 2 only has to wait time 1, not the entire $x(1) = 5$. This suggests that a more relevant factor is the *remaining time* of a job, rather than its original size. Indeed, the optimal policy in the setting with positive arrival times and preemption is to at all times schedule the job with Shortest Remaining Processing Time (SRPT). Under SPRT, every time a job j arrives to the system we compare $x(j)$ to the remaining time of the job currently in service. If the job currently in service has a remaining time that is less than $x(j)$, we continue running that job. If $x(j)$ is smaller than the current job's remaining time, we preempt the current job and run j instead. When a job finishes, we start running the job with the smallest *remaining* time. The intuition for why this works is the same as the intuition behind SJF. At all times we want to schedule the job that causes all of the other jobs to have to wait the least amount of time. In this setting, the amount of time the other jobs will have to wait is the remaining time of the job that we schedule.

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?

Solution: When we're allowed to let jobs run past their deadlines and we have to schedule all of the jobs, we saw that Earliest Deadline First was an optimal policy in that it minimizes the maximum lateness of any job. Unfortunately, EDF doesn't work when we have hard deadlines. The problem is that by choosing the job with the earliest deadline, it's possible that all of our other jobs will get irrevocably behind and we'll have to drop them all. Indeed, any greedy algorithm we could imagine will run into this problem. Instead, we will take a dynamic programming approach: we have to decide whether or not to include each job, and we'll make this decision based on comparing the best solution we can find if we do include the job to the best solution we can find if we don't include it.

The deadlines are still relevant to us, so our algorithm will begin by sorting the jobs in order of their deadlines, so that $d(1) \leq d(2) \leq \cdots \leq d(n) = D$. Ultimately, we want to know the maximum number of jobs we can schedule among all jobs $1, \ldots, n$ such that all jobs in our schedule meet their deadlines, and all jobs are complete by time D (the latest deadline). We can either choose to include job n or not. If we don't include job n, we now want to know the maximum number

of jobs we can schedule among all jobs $1, \ldots, n-1$ such that all jobs in our schedule meet their deadlines, and all jobs are complete by time D. If we do include job n, we now want to know the maximum number of jobs we can schedule among all jobs $1, \ldots, n-1$ such that all jobs in our schedule meet their deadlines, and all jobs are complete by time $D - x(n)$. (Why $D - x(n)$? Because scheduling job n as late as possible leave the most time possible in which to complete all the other jobs). We see that in each of these cases, we are left with a subproblem to solve optimally.

In general, let $N(k, t)$ be the maximum number of jobs we can schedule among jobs $1, \ldots, k$ such that all jobs in our schedule meet their deadlines, and all jobs are complete by time t. We first observe that if we are going to finish job k by both its deadline and by time t, we must start running job k by time $t' = \min\{t, d(k)\} - x(k)$. We then can write the following recurrence for $N(k, t)$:

$$N(k,t) = \begin{cases} 0 & k = 0 \\ N(k-1,t) & k > 0, t' < 0 \\ \max\{N(k-1,t), 1 + N(k-1,t')\} & k > 0, t' \geq 0 \end{cases}$$

The first case is a base case: we have no jobs left to schedule. In the second case, we can't finish job k in time so we must choose not to include job k in our schedule. In the third case, we get to choose whichever is better: not including job k at all, in which case we need to schedule the first $k-1$ jobs by time t, or including job k, in which case we need to schedule the first $k-1$ jobs by time t'.

Our algorithm will fill in a 2-dimensional array A, where entry $A(k, t) = N(k, t)$. Our array has n rows, one for each job, and D columns, one for each time from time 0 up to the latest deadline D. We fill the rows in increasing order, and for each row we fill the columns in increasing order. We can do this because to fill entry $A(k, t)$, we only need to look up entries in row $k-1$ and in columns $< t$. Here's some pseudocode:

```
DeadlineScheduler(J)
   sort jobs by deadline, so d(1) ≤ d(2) ≤ ··· ≤ d(n)
   A[n][D] = new array
   set A[0][t] = 0 for all t
   for k = 1 to n
      for t = 1 to D
         set t' = min{t,d(k)} − x(k)
         if t' < 0
            set A[k][t] = A[k−1][t]
         else
            set A[k][t] = max{A[k−1][t], 1 + A[k−1][t']}
   return A[n][D]
```

Once we have filled the array, the number of jobs that we can schedule is contained in entry $A(n, D)$. We still need to reconstruct the optimal schedule that achieves this maximum number of jobs. Observe that we can determine whether we included job k by checking whether $A(k, t) = A(k-1, t)$. If so, we did not include job k in our schedule. If so, we did include job k. We can use this fact to write a recursive method that prints our schedule:

```
printSchedule(k,t)
   if k = 0 return
   if A(k,t) = A(k-1,t)
      printSchedule(k-1, t)
   else
      t' <- min{t, d(k)} - x(k)
      printSchedule(k-1, t')
      print "Schedule job " + k + " at time " + t'
```

An initial call to `printSchedule(n,D)` will print the optimal schedule.

Our algorithm takes time $O(nD)$. The initial sort in increasing order of deadlines takes time $O(n \lg n)$. Filling in the array takes time $O(nD)$ because to fill each entry we do a constant-time lookup to previously filled entries, and there are nD total entries in the array. Our recursive print method takes time $O(n)$. This is a total of $(n \lg n + nD)$. If D is much larger than n, the second term dominates and we can say the runtime is $O(nD)$. If D is smaller than n, then when we sort by deadline we are sorting n elements with integer values less than n, so we can do a linear-time sort. So in this case we can also say that the runtime is $O(nD)$.

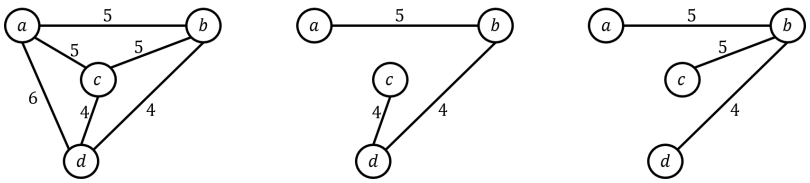
2 Graph Algorithms

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.

Solution: A minimum bottleneck spanning tree of G is not necessarily a minimum spanning tree. Here is a counterexample:



The figure on the left shows our original graph. The figure in the middle is a minimum spanning tree; its cost is 13. The figure on the right shows a minimum bottleneck spanning tree. The cost of the bottleneck edge is 5, and the total cost of the tree is 14, which is greater than the cost of our MST.

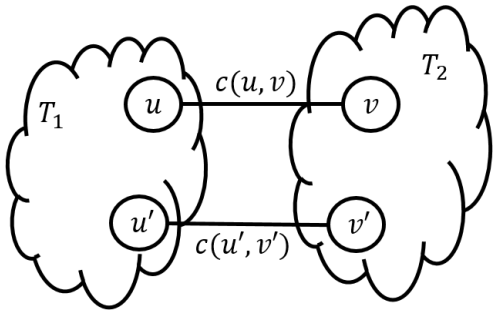
(b) Is every minimum spanning tree of G a minimum bottleneck spanning tree? If so, prove it. If not, give a counterexample.

Solution: A minimum spanning tree is guaranteed to be a bottleneck spanning tree.

Theorem 2. *Every minimum spanning tree T of G is a minimum bottleneck spanning tree.*

Proof. Our proof will be by contradiction. Suppose we have a minimum spanning tree (MST) T that is not a minimum bottleneck spanning tree (MBST). Let b be the weight of the maximum-cost edge in the MBST, and let (u, v) be the edge of maximum cost in the MST, where the cost of this edge is $c(u, v)$. Since the MST is not an MBST, we must have $c(u, v) > b$.

The edge (u, v) splits our MST into two subtrees, which we will call T_1 and T_2. This looks like:



Since $c(u, v) > b$, we know that edge (u, v) is not in any MBST. So any MBST must include some alternative edge (u', v') to connect subtrees T_1 and T_2. Since the maximum cost edge in the MBST has cost b, we know that $c(u', v') \leq b$. At the same time, we must have $c(u, v) \leq c(u', v')$, since we included edge (u, v) in our MST instead of (u', v'). Putting this together, we get $c(u, v) \leq c(u', v') \leq b$, which contradicts our assumption that our MST is not an MBST (because in that case we would have $c(u, v) > b$). Hence the MST must be an MBST. □

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?

Solution: As a start, it will help us to know the length of the shortest path from s to t for all nodes $t \in V$. We can compute these lengths by running Dijkstra's algorithm, which will find one shortest path from s to t. Let $d(t)$ denote the shortest path distance from s to t found by Dijkstra's algorithm. Clearly if there are multiple shortest paths from s to t, they must all have the same length $d(t)$.

If we think about our graph, there are some edges that could be part of some shortest path, and other edges that cannot be in any shortest path. We will say that an edge (u, v) has *no slack* if $d(v) = d(u) + c(u, v)$, where $c(u, v)$ is the cost of edge (u, v). Every edge on a shortest path must be an edge with no slack. Why? Suppose that edge (u, v) has slack, so $d(v) < d(u)+c(u, v)$. Then the shortest path from s to v cannot contain edge (u, v), which in turn means that no shortest path from s to any other node t can contain edge (u, v) (if the path from s to t involves going through v, we can do better than choosing edge (u, v) by following whatever path has lower cost $d(v)$). Conversely, every path from s to some other node t that only uses edges with no slack must be a shortest path from s to t.

Let $E' \subseteq E$ be the set of edges with no slack. Let $G' = (V, E')$ be a subgraph of G that contains all the vertices in G and only the edges in G that have no slack. The number of shortest paths from s to t in G is the total number of paths from s to t in G'. Observe that G' is a directed acyclic graph (DAG).

Let $P(t)$ be the number of paths from s to t in G'. We are interested in computing $P(t)$ for all vertices t. The key observation that will enable us to do this is that the number of paths to t is the sum of the number of paths to nodes u for which there is an edge (u, t) in G'. That is:

$$P(t) = \sum_{(u,t)\in E'} P(u).$$

So we would like to ensure that by the time we consider a node, we have already counted up the number of paths to each of its parents.

So how should we order our nodes? The order that we want is called the *topological sort* of G'. This is an ordering of the nodes that ensures that if there is a path from u to v, then u appears before v in the ordering. There typically are many possible topological sorts for a given DAG; one way to find a topological sort is to use depth-first search (DFS) as follows:

```
topologicalSort(G)
   call DFS(G) to compute finish times for each vertex v in V
   return vertices sorted in reverse order of DFS finish times
```

Now, we just need to make a single pass through our nodes, sorted in topological order. For each node t, we can set $P(t) = \sum_{(u,t)\in E'} P(u)$, and we know that $P(u)$ already has been computed because our topological sort guarantees that we only consider node t after all of its parent nodes.

Our final algorithm is:

```
countAllShortestPaths(G,s,t)
   run Dijkstra(G,s), setting d(t) = shortest path distance from s to t
   set E' = {}
   for each edge (u,v) in E
      if (u,v) has no slack, add (u,v) to E'
   set G' = (V,E')
   set A = topologicalSort(G'), where A[i] is the ith node in topological sorted order
   for i = 1 to n
      set P[A[i]] = ∑_{(A[j],A[i])∈E'} P[A[j]]
   return P[t]
```

The initial run of Dijkstra's algorithm takes time $O(m \lg n)$ if we are clever about using priority queues in our implementation (if not, Dijkstra's algorithm more naively takes time $O(n^2)$. We then take time $O(m)$ to form the subset of edges E'. The topological sort is a DFS, which takes time $O(m+n)$. And we make one final pass to set the values $P(v)$ for each vertex; this requires one lookup per edge in E', so the time required is $O(m+n)$. The run of Dijkstra's algorithm in the preprocessing step dominates the runtime, so our overall algorithm takes time $O(m \lg n)$.

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?

Solution: We have n precincts, each of which has m voters, so the total number of voters is nm. Each district will contain $\frac{nm}{2}$ voters, so in order for party A to win a district it must have at least $\frac{nm}{4} + 1$ voters in the district. Let a be the total number of party-A voters, where $a \geq \frac{nm}{2} + 2$ in order for gerrymandering to be a possibility. Party A wins both districts if there is a subset $\mathcal{S}$ of $n/2$ precincts such that the total number of party-A voters in subset $\mathcal{S}$ is $s \geq \frac{nm}{4} + 1$ and the total number of party-A voters in the remaining precincts is $a - s \geq \frac{nm}{4} + 1$. Putting these two inequalities together, we have that the set of precincts is susceptible to gerrymandering if there is a subset $\mathcal{S}$ of precincts with s total party-A voters such that

$$\frac{nm}{4} + 1 \leq s \leq a - \frac{nm}{4} - 1. \tag{1}$$

Our problem is essentially to come up with a set of precincts that works, if one exists. This does not lend itself to a greedy algorithm: it's hard to imagine building a "balanced" district by always adding the precinct with the most party-A voters, always adding the precinct the the fewest party-A voters, always adding the precinct with the most balanced party-A/party-B ratio...none of the greedy options make sense. Instead, we'll take a dynamic programming approach.

Our goal will be to build a district that consists of $n/2$ precincts and s total party-A voters, where s satisfies inequality (1). Let's first consider precinct n, and suppose precinct n has a_n party-A voters. We can choose either to include precinct n in our district or not. If we do include precinct n, our system is susceptible to gerrymandering if we can come up with a set of $n/2 - 1$ precincts from among the remaining precincts $1, \ldots, n-1$ that has $s - a_k$ total party-A voters. If we do not include precinct n, our system is susceptible to gerrymandering if we can come up with a set of $n/2$ precincts from among the remaining precincts $1, \ldots, n-1$ that has s total party-A voters. This relationship suggests the following recursive relationship, where $G(k, \ell, w)$ answers the question "is it possible to form a set of ℓ precincts from among the first k precincts that has exactly w total party-A voters?":

$$G(k,\ell,w) = \begin{cases} \texttt{true} & \text{if } G(k-1,\ell-1,w-a_k) \text{ is } \texttt{true} \\ \texttt{true} & \text{if } G(k-1,\ell,w) \text{ is } \texttt{true} \\ \texttt{true} & \text{if } \ell = 0, w = 0 \\ \texttt{true} & \text{if } k = 1, \ell = 1, w = a_1 \\ \texttt{false} & \text{otherwise} \end{cases}$$

In this recurrence, the first case corresponds to the case in which we choose to include precinct k in our district. The second case corresponds to the case in which we choose not to include precinct k in our district. The third case is a base case: no matter what k is, we can always form a subset of $\ell = 0$ precincts that contains $w = 0$ party-A voters. The fourth case is another base case; we

can only form a set of 1 precinct with exactly w party-A voters from among the first 1 precinct if precinct 1 has exactly $a_1 = w$ party-A voters. If none of these cases is satisfied, we cannot gerrymander in this particular fashion.

How many of these $G(k, \ell, w)$ questions must we answer in order to know if our set of precincts is susceptible to gerrymandering? We let k range from 1 to n, since we can choose either to include or not each possible precinct. We let ℓ range from 1 to $n/2$, since ultimately we are trying to form districts with $n/2$ precincts. And w ranges from 1 to $a - \frac{nm}{4} - 1$, since this is the largest value s can take on as per inequality (1).

Our algorithm will proceed by filling in an $n \times n/2 \times a - \frac{nm}{4} - 1$ table. Each entry in this table will store the answer to the corresponding $G(k, \ell, w)$ question. In order to fill in entry $G(k, \ell, w)$, we simply need to look up the values $G(k-1, \ell-1, w-a_k)$ and $G(k-1, \ell, w)$ (plus check if we are in a base case). We can fill this table in increasing order of k, and in any order for ℓ and w.

Once we have filled in the table, in order to determine whether the system is susceptible to gerrymandering we must look at the rows $G(n, \frac{n}{2}, s)$ for all s satisfying inequality (1). If the table stores `true` in any of these entries, the system is susceptible to gerrymandering. Here's some pseudocode:

```
Gerrymander(n)
   G[n][n/2][a−nm/4−1]: new array
   for k = 1 to n
      set G[k][0][0] = true
   for k = 1 to n
      for l = 1 to n/2
         for w = 1 to a − nm/4 − 1
            if k == 1, l == 1, w == a_1
               set G[k][l][w] = true
            else if G[k−1][l−1][w−a_k]
               set G[k][l][w] = true
            else if G[k−1][l][w]
               set G[k][l][w] = true
            else
               set G[k][l][w] = false
   for s = nm/4 + 1 to a − nm/4 − 1
      if G[n][n/2][s]
         return true
   return false
```

Running this algorithm requires one pass through the 3-dimensional table to fill in all of the entries, plus a final pass to check whether `true` appears in any of the rows where s satisfies inequality (1). Filling in the table takes time $O(n \cdot \frac{n}{2} \cdot (a - \frac{nm}{4} - 1)) = O(n^3 m)$. The final traversal of the last row takes time $O(nm)$. The first term dominates, so the algorithm takes time $O(n^3 m)$.

Here is some pseudocode for an equivalent top-down recursive version with memoization:

```
G[n][n/2][a−nm/4−1]: new (global) array, every entry initialized to NULL

Gerrymander(n)
   for s = nm/4 + 1 to a − nm/4 − 1
      if GerryHelper(n, n/2, s)
         return true
   return false

GerryHelper(k, l, w)
   if G[k][l][w] ≠ NULL
      return G[k][l][w]
   if l == 0 and w == 0
      set G[k][l][w] = true
      return true
   if k == 1 and l == 1
      if w == a_k
         set G[k][l][w] = true
      else set G[k][l][w] = false
      return G[k][l][w]
   set G[k][l][w] = GerryHelper(k−1, l, 1) OR GerryHelper(k−1, l−1, w−a_k)
   return G[k][l][w]
```