Tutorial -- Greedy Algorithms

1. You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed limit **W** on the maximum amount of weight they are allowed to carry. Boxes arrive at the New York station one by one, and each package **i** has a weight $\mathbf{w_i}$. The trucking station is quite small, so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive. At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.
But they wonder if they might be using too many trucks, and they want your opinion on whether the situation can be improved. Here is how they are thinking. Maybe one could decrease the number of trucks needed by sometimes sending off a truck that was less full, and in this way allow the next few trucks to be better packed. Prove that, for a given set of boxes with specified weights, the greedy algorithm currently in use actually minimizes the number of trucks that are needed.

2. Some of your friends have gotten into the burgeoning field of time-series data mining, in which one looks for patterns in sequences of events that occur over time. Purchases at stock exchanges—what's being bought— are one source of data with a natural ordering in time. Given a long sequence S of such events, your friends want an efficient way to detect certain "patterns" in them—for example, they may want to know if the four events.

buy Yahoo, buy eBay, buy Yahoo, buy Oracle

occur in this sequence S, in order but not necessarily consecutively.

They begin with a collection of possible events (e.g., the possible transactions) and a sequence S of n of these events. A given event may occur multiple times in S (e.g., Yahoo stock may be bought many times in a single sequence S). We will say that a sequence S is a subsequence of S if there is a way to delete certain of the events from S so that the remaining events, in order, are equal to the sequence S . So, for example, the sequence of four events above is a subsequence of the sequence

buy Amazon, buy Yahoo, buy eBay, buy Yahoo, buy Yahoo, buy Oracle

Their goal is to be able to dream up short sequences and quickly detect whether they are subsequences of S. So this is the problem they pose to you: Give an algorithm that takes two sequences of events—S of length m and S of length n, each possibly containing an event more than once—and decides in time O(m + n) whether S is a subsequence of S.

3. Let's consider a long, quiet country road with houses scattered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.

Give an efficient algorithm that achieves this goal, using as few base stations as possible.

4. 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_i weight w_i for each job. You want to order the jobs so as to minimize the weighted sum of the completion times, $\sum_{i=1}^{n} w_i C_i$

5. Consider the following variation on the Interval Scheduling Problem. You have a processor that can operate 24 hours a day, every day. People submit requests to run daily jobs on the processor. Each such job comes with a start time and an end time; if the job is accepted to run on the processor, it must run continuously, every day, for the period between its start and end times. (Note that certain jobs can begin before midnight and end after midnight; this makes for a type of situation different from what we saw in the Interval Scheduling Problem.)
Given a list of n such jobs, your goal is to accept as many jobs as possible (regardless of their length), subject to the constraint that the processor can run at most one job at any given point in time. Provide an algorithm to do this with a running time that is polynomial in n. You may assume for simplicity that no two jobs have the same start or end times.

(6 P.M. , 6 A.M.), (9 P.M. , 4 A.M.), (3 A.M. , 2 P.M.), (1 P.M. , 7 P.M.)

The optimal solution would be to pick the two jobs (9 P.M. , 4 A.M.) and (1 P.M., 7 P.M.), which can be scheduled without overlapping.

6. The wildly popular Spanish-language search engine El Goog needs to do a serious amount of computation every time it recompiles its index. Fortunately, the company has at its disposal a single large supercomputer, together with an essentially unlimited supply of high-end PCs.

They've broken the overall computation into n distinct jobs, labeled $J_1, J_2, \ldots, J_n$, which can be performed completely independently of one another. Each job consists of two stages: first it needs to be preprocessed on the supercomputer, and then it needs to be finished on one of the PCs. Let's say that job J_i needs p_i seconds of time on the supercomputer, followed by f_i seconds of time on a PC.
Since there are at least n PCs available on the premises, the finishing of the jobs can be performed fully in parallel—all the jobs can be processed at the same time. However, the supercomputer can only work on a single job at a time, so the system managers need to work out an order in which to feed the jobs to the supercomputer. As soon as the first job in order is done on the supercomputer, it can be handed off to a PC for finishing; at that point in time a second job can be fed to the supercomputer; when the second job is done on the supercomputer, it can proceed to a PC regardless of whether or not the first job is done (since the PCs work in parallel); and so on.
Let's say that a schedule is an ordering of the jobs for the super-computer, and the completion time of the schedule is the earliest time at which all jobs will have finished processing on the PCs. This is an important quantity to minimize, since it determines how rapidly El Goog can generate a new index.
Give a polynomial-time algorithm that finds a schedule with as small a completion time as possible.

Q1. Say n boxes arrive in the order $b_1, \ldots, b_n$. Say each box b_i has a positive weight w_i, and the maximum weight each truck can carry is W. To pack the boxes into N trucks *preserving the order* is to assign each box to one of the trucks $1, \ldots, N$ so that:

- No truck is overloaded: the total weight of all boxes in each truck is less or equal to W.
- The order of arrivals is preserved: if the box b_i is sent before the box b_j (i.e. box b_i is assigned to truck x, box b_j is assigned to truck y, and $x < y$) then it must be the case that b_i has arrived to the company earlier than b_j (i.e. $i < j$).

We prove that the greedy algorithm uses the fewest possible trucks by showing that it "stays ahead" of any other solution. Specifically, we consider any other solution and show the following. If the greedy algorithm fits boxes $b_1, b_2, \ldots, b_j$ into the first k trucks, and the other solution fits $b_1, \ldots, b_i$ into the first k trucks, then $i \leq j$. Note that this implies the optimality of the greedy algorithm, by setting k to be the number of trucks used by the greedy algorithm.

We will prove this claim by induction on k. The case $k = 1$ is clear; the greedy algorithm fits as many boxes as possible into the first truck. Now, assuming it holds for $k - 1$: the greedy algorithm fits j' boxes into the first $k - 1$, and the other solution fits $i' \leq j'$. Now, for the k^{th} truck, the alternate solution packs in $b_{i'+1}, \ldots, b_i$. Thus, since $j' \geq i'$, the greedy algorithm is able at least to fit all the boxes $b_{j'+1}, \ldots, b_i$ into the k^{th} truck, and it can potentially fit more. This completes the induction step, the proof of the claim, and hence the proof of optimality of the greedy algorithm.

[1] ex73.193.591

Q2. Let the sequence S consist of $s_1, \ldots, s_n$ and the sequence S' consist of $s'_1, \ldots, s'_m$. We give a greedy algorithm that finds the first event in S that is the same as s'_1, matches these two events, then finds the first event after this that is the same as s'_2, and so on. We will use $k_1, k_2, \ldots$ to denote the match have we found so far, i to denote the current position in S, and j the current position in S'.

```
Initially i = j = 1
While i ≤ n and j ≤ m
    If s_i is the same as s'_j, then
        let k_j = i
        let i = i + 1 and j = j + 1
    otherwise let i = i + 1
EndWhile
If j = m + 1 return the subsequence found:  k_1, ..., k_m
Else return that "S' is not a subsequence of S"
```

The running time is $O(n)$: one iteration through the while look takes $O(1)$ time, and each iteration increments i, so there can be at most n iterations.

It is also clear that the algorithm finds a correct match if it finds anything. It is harder to show that if the algorithm fails to find a match, then no match exists. Assume that S' is the same as the subsequence $s_{l_1}, \ldots, s_{l_m}$ of S. We prove by induction that the algorithm will succeed in finding a match and will have $k_j \leq l_j$ for all $j = 1, \ldots, m$. This is analogous to the proof in class that the greedy algorithm finds the optimal solution for the interval scheduling problem: we prove that the greedy algorithm is always ahead.

- *For each $j = 1, \ldots, m$ the algorithm finds a match k_j and has $k_j \leq l_j$.*

Proof. The proof is by induction on j. First consider $j = 1$. The algorithm lets k_1 be the first event that is the same as s'_1, so we must have that $k_1 \leq l_1$.

Now consider a case when $j > 1$. Assume that $j - 1 < m$ and assume by the induction hypothesis that the algorithm found the match k_{j-1} and has $k_{j-1} \leq l_{j-1}$. The algorithm lets k_j be the first event after k_{j-1} that is the same as s'_j if such an event exists. We know that l_j is such an event and $l_j > l_{j-1} \geq k_{j-1}$. So $s_{l_j} = s'_j$, and $l_j > k_{j-1}$. The algorithm finds the first such index, so we get that $k_j \leq l_j$. ■

[1] ex876.936.4

Q3. Here is a greedy algorithm for this problem. Start at the western end of the road and begin moving east until the first moment when there's a house h exactly four miles to the west. We place a base station at this point (if we went any farther east without placing a base station, we wouldn't cover h). We then delete all the houses covered by this base station, and iterate this process on the remaining houses.

Here's another way to view this algorithm. For any point on the road, define its *position* to be the number of miles it is from the western end. We place the first base station at the easternmost (i.e. largest) position s_1 with the property that all houses between 0 and s_1 will be covered by s_1. In general, having placed $\{s_1, \ldots, s_i\}$, we place base station $i+1$ at the largest position s_{i+1} with the property that all houses between s_i and s_{i+1} will be covered by s_i and s_{i+1}.

Let $S = \{s_1, \ldots, s_k\}$ denote the full set of base station positions that our greedy algorithm places, and let $T = \{t_1, \ldots, t_m\}$ denote the set of base station positions in an optimal solution, sorted in increasing order (i.e. from west to east). We must show that $k = m$.

We do this by showing a sense in which our greedy solution S "stays ahead" of the optimal solution T. Specifically, we claim that $s_i \geq t_i$ for each i, and prove this by induction. The claim is true for $i = 1$, since we go as far as possible to the east before placing the first base station. Assume now it is true for some value $i \geq 1$; this means that our algorithm's first i centers $\{s_1, \ldots, s_i\}$ cover all the houses covered by the first i centers $\{t_1, \ldots, t_i\}$. As a result, if we add t_{i+1} to $\{s_1, \ldots, s_i\}$, we will not leave any house between s_i and t_{i+1} uncovered. But the $(i+1)^{\text{st}}$ step of the greedy algorithm chooses s_{i+1} to be *as large as possible* subject to the condition of covering all houses between s_i and s_{i+1}; so we have $s_{i+1} \geq t_{i+1}$. This proves the claim by induction.

Finally, if $k > m$, then $\{s_1, \ldots, s_m\}$ fails to cover all houses. But $s_m \geq t_m$, and so $\{t_1, \ldots, t_m\} = T$ also fails to cover all houses, a contradiction.

[1] ex198.453.676

Q4. An optimal algorithm is to schedule the jobs in decreasing order of w_i/t_i. We prove the optimality of this algorithm by an exchange argument.

Thus, consider any other schedule. As is standard in exchange arguments, we observe that this schedule must contain an *inversion* — a pair of jobs i, j for which i comes before j in the alternate solution, and j comes before i in the greedy solution. Further, in fact, there must be an adjacent such pair i, j. Note that for this pair, we have $w_j/t_j \geq w_i/t_i$, by the definition of the greedy schedule. If we can show that swapping this pair i, j does not increase the weighted sum of completion times, then we can iteratively do this until there are no more inversions, arriving at the greedy schedule without having increased the function we're trying to minimize. It will then follow that the greedy algorithm is optimal.

So consider the effect of swapping i and j. The completion times of all other jobs remain the same. Suppose the completion time of the job before i and j is C. Then before the swap, the contribution of i and j to the total sum was $w_i(C + t_i) + w_j(C + t_i + t_j)$, while after the sum it is $w_j(C + t_j) + w_i(C + t_i + t_j)$. The difference between the value after the swap, compared to the value before the swap is (canceling terms in common between the two expressions) $w_i t_j - w_j t_i$. Since $w_j/t_j \geq w_i/t_i$, this difference is bounded above by 0, and so the total weighted sum of completion times does not increase due to the swap, as desired.

[1] ex948.540.252

Q5. Let $I_1, \ldots, I_n$ denote the n intervals. We say that an *I_j-restricted solution* is one that contains the interval I_j.

Here is an algorithm, for fixed j, to compute an I_j-restricted solution of maximum size. Let x be a point contained in I_j. First delete I_j and all intervals that overlap it. The remaining intervals do not contain the point x, so we can "cut" the time-line at x and produce an instance of the Interval Scheduling Problem from class. We solve this in $O(n)$ time, assuming that the intervals are ordered by ending time.

Now, the algorithm for the full problem is to compute an I_j-restricted solution of maximum size for each $j = 1, \ldots, n$. This takes a total time of $O(n^2)$. We then pick the largest of these solutions, and claim that it is an optimal solution. To see this, consider the optimal solution to the full problem, consisting of a set of intervals S. Since $n > 0$, there is some interval $I_j \in S$; but then S is an optimal I_j-restricted solution, and so our algorithm will produce a solution at least as large as S.

[1] ex434.357.684

Q6. It is clear that the working time *for the super-computer* does not depend on the ordering of jobs. Thus we can not change the time when the last job hands off to a PC. It is intuitively clear that the last job in our schedule should have the shortest *finishing* time.

This informal reasoning suggests that the following greedy schedule should be the optimal one.

```
Schedule G:
  Run jobs in the  order of decreasing finishing time f_i.
```

Now we show that G is actually the optimal schedule, using an exchange argument. We will show that for any given schedule $S \neq G$, we can repeatedly swap adjacent jobs so as to convert S into G without increasing the completion time.

Consider any schedule S, and suppose it does not use the order of G. Then this schedule must contain two jobs J_k and J_l so that J_l runs directly after J_k, but the finishing time for the first job is less than the finishing time for the second one, i.e. $f_k < f_l$. We can optimize this schedule by swapping the order of these two jobs. Let S' be the schedule S where we swap only the order of J_k and J_l. It is clear that the finishing times for all jobs except J_k and J_l does not change. The job J_l now schedules earlier, thus this job will finish earlier than in the original schedule. The job J_k schedules later, but the super-computer hands off J_k to a PC in the new schedule S' at the same time as it would handed off J_l in the original schedule S. Since the finishing time for J_k is less than the finishing time for J_l, the job J_k will finish earlier in the new schedule than J_l would finish in the original one. Hence our swapped schedule does not have a greater completion time.

If we define an inversion, as in the text, to be a pair of jobs whose order in the schedule does not agree with the order of their finishing times, then such a swap decreases the number of inversions in S while not increasing the completion time. Using a sequence of such swaps, we can therefore convert S to G without increasing the completion time. Therefore the completion time for G is not greater than the completion time for any arbitrary schedule S. Thus G is optimal.

Notes. As with all exchange arguments, there are some common kinds of mistakes to watch out for. We summarize some of these here; they illustrate principles that apply to others of the problems as well.

- The exchange argument should start with an arbitrary schedule S (which, in particular, could be an optimal one), and use exchanges to show that this schedule S can be turned into the schedule the algorithm produces without making the overall completion time worse. It does not work to start with the algorithm's schedule G and simply argue that G cannot be improved by swapping two jobs. This argument would show only that a schedule obtained from G be a *single swap* is not better; it would not rule out the possibility of other schedules, obtainable by multiple swaps, that are better.

[1]ex172.268.910

- To make the argument work smoothly, one should to swap neighboring jobs. If you swap two jobs J_l and J_k that are not neighboring, then all the jobs between the two also change their finishing times.

- In general, it does not work to phrase the above exchange argument as a proof by contradiction — that is, considering an optimal schedule $\mathcal{O}$, assuming it is not equal to G, and getting a contradiction. The problem is that there could many optimal schedules, so there is no contradiction in the existence of one that is not G. Note that when we swap adjacent, inverted jobs above, it does not necessarily make the schedule better; we only argue that such swaps do not make it worse.

Finally, it's worth noting the following alternate proof of the optimality of the schedule G, not directly using an exchange argument. Let job J_j be the job that finishes last on the PC in the greedy algorithm's schedule G, and let S_j be the time this job finishes on the supercomputer. So the overall finish time is $S_j + f_j$. In any other schedule, one of the first j jobs, in the other specified by G, must finish on the supercomputer at some time $T \geq S_j$ (as the first j jobs give exactly S_j work to the supercomputer). Let that be job J_i. Now job J_i needs PC time at least as much as job j (due to the ordering of G), and so it finishes at time $T + f_i \geq S_j + f_j$. So this other schedule is no better.