# CS 621 - Take home final exam

Ali Hassani

December 7, 2022

## 1 Chapter 4, Exercise 4 (pp 190)

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 $\mathcal{O}(m+n)$ whether $S'$ is a subsequence of $S$.

## Answer

```
Inputs: S, S'

int i=0, j=0;
LinkedList<int> result;

while (i < n && j < m) {
    if (S[i] == S'[j]) {
        result.push(i);
        j++;
    }
    i++;
}

return (j == m), result;
```

This algorithm matches the earliest occurrence of an event in both $S$ and $S'$, and runs in $\mathcal{O}(n)$ time, because the loop runs at most $n$ times given that $m \leq n$.

**Correctness:** Assume the greedy method above finds a solution (`result` is length $m$). Because of the formulation, it is clear that `result` is sorted in ascending order, and contains only indices of $S$. Therefore, we can construct sequence $T$ based on indices in `result`, and it will be a subsequence of $S$. We also know that for every $i$ in `result`, there exists a unique $j$ that satisfies `S[i] == S'[j]`. Those $j$s are also in ascending order. Therefore, $T = S'$ by definition. We already know that $T$ is a subsequence of $S$, therefore $S'$ is a subsequence of $S$, and the greedy solution is correct.

Now assume the greedy method returns a `false`, with `result` of length less than $m$. If there is truly no solution ($S'$ is not a subsequence of $S$), then the greedy solution is correct. Assuming there is a solution ($S'$ is a subsequence of $S$), then there exists a list `truth` of length exactly $m$, which contains indices corresponding to events in $S$ with ascending order. Let's assume $p$ is the first index of `result` that does not match `truth` (`result[p] != truth[p]` and
`for (int l=0; l < p; l++) result[l] == truth[l]`
or `result` is length $p-1$.) Then by definition `S[truth[p]] == S'[p]`. Because everything preceding the $p$-th element (up to $p-1$) in `truth` and `result`

match, then by the loop definition the condition `S[truth[p]] == S'[p]` was checked in the for loop, and because we know it is true, then result would have its $p$-th element set to `truth[p]`, which is a contradiction. Therefore no such $p$ exists, and as a result our assumption that there is a solution and `result` was of length smaller than $m$ is incorrect. Therefore the greedy method returns a `result` of length less than $m$ only when $S'$ is not a subsequence of $S$.

## 2 Chapter 4, Exercise 6 (pp 191)

Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of junior-high-school-age campers. One of his plans is the following mini-triathalon exercise: each contestant must swim 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first one contestant swims the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second contestant begins swimming the 20 laps; as soon as he or she is out and starts biking, a third contestant begins swimming . . . and so on.)

Each contestant has a projected *swimming time* (the expected time it will take him or her to complete the 20 laps), a projected biking time (the expected time it will take him or her to complete the 10 miles of bicycling), and a projected running time (the time it will take him or her to complete the 3 miles of running). Your friend wants to decide on a schedule for the triathalon: an order in which to sequence the starts of the contestants. Let's say that the completion time of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathalon, assuming they each spend exactly their projected swimming, biking, and running times on the three parts. (Again, note that participants can bike and run simultaneously, but at most one person can be in the pool at any time.) What's the best order for sending people out, if one wants the whole competition to be over as early as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible.

### Answer

Assuming there are $n$ contestants (1 through $n$), and $s_i$, $b_i$, and $r_i$ represent projected swimming, biking, and running times for contestant $i$ respectively:
**Greedy choice:** Sort contestants by $b_i + r_i$ in descending order, and pick the one with the longest projected bike and run time to swim first. (This is similar to the supercomputer problem, where jobs were first preprocessed on a supercomptuer that handles one job at a time, and then set off to a PC, and there are infinitely many PCs available.)
**Correctness:** For simplicity in notation, let $t_i = b_i + r_i$.
For a given solution set $A$ that is not equal to the greedy solution $G$, and

cannot be produced by the greedy algorithm: Firstly both will be of length $n$, since they are both sortings of the $n$ contestants. Therefore, if they are not equivalent, there exists at least one instance of two back-to-back contestants $C_b$ and $C_a$ where $t_a > t_b$. In other words:

- $C_b$ finishes at time $d_b + s_b + t_b$, where $d_b$ is the sum of $s_i$s before $C_b$.
- $C_a$ finishes at time $d_b + s_b + s_a + t_a$.

Now if we were to swap $C_a$ and $C_b$'s position in schedule $A$, leading to a new schedule $A'$:

- $C_a$ finishes at time $d_b + s_a + t_a$, which is less than $d_b + s_b + s_a + t_a$, so $C_a$ now finishes sooner.
- $C_b$ finishes at time $d_b + s_a + s_b + t_b$, which is slower than $d_b + p_b + f_b$.

For simplicity, we'll exclude $d_b$ from the notations, because it is a constant in both times.
In the original $A$, finish times for $C_a$ and $C_b$ were $s_b + s_a + t_a$ and $s_b + t_b$. Of the two, we know $C_a$ finishes later because:

$$\begin{aligned} t_a > t_b \quad &\implies \\ s_b + t_a > s_b + t_b \quad &\implies \\ s_b + s_a + t_a > s_b + t_a > s_b + t_b&. \end{aligned}$$

Therefore, the maximum of the two times is the time $C_a$ takes, or $s_b + s_a + t_a$. We take the maximum, because the maximum of these two contributes to the global maximum of all jobs, therefore to the efficiency of the solution.
In the new $A'$, the time the two jobs $C_a$ and $C_b$ took is $s_a + t_a$ and $s_a + s_b + t_b$. We already know that:

$$s_a + t_a \le s_b + s_a + t_a.$$

And we can also see:

$$\begin{aligned} t_b \le t_a \quad &\implies \\ s_a + s_b + t_b \le s_a + s_b + t_a&, \end{aligned}$$

which means the maximum of the times in $A'$ is less than or equal to the maximum in $A$, therefore this swap either does not affect $A$ or improves it. Now if we consider $A'$ to be our initial set, and find another pair of back-to-back jobs with a similar property and swap their order, we would find again that it either does not affect the solution's overall score or improves it. Repeating this will act as a bubble sort and we will eventually end up with a solution that the greedy method could have produced.

## 3 Chapter 6, Exercise 8 (pp 319)

The residents of the underground city of Zion defend themselves through a combination of kung fu, heavy artillery, and efficient algorithms. Recently they have become interested in automated methods that can help fend off attacks by swarms of robots.

Here's what one of these robot attacks looks like.

- A swarm of robots arrives over the course of $n$ seconds; in the $i^{\text{th}}$ second, $x_i$ robots arrive. Based on remote sensing data, you know this sequence $x_1, x_2, ..., x_n$ in advance.
- You have at your disposal an electromagnetic pulse (EMP), which can destroy some of the robots as they arrive; the EMP's power depends on how long it's been allowed to charge up. To make this precise, there is a function $f()$ so that if $j$ seconds have passed since the EMP was last used, then it is capable of destroying up to $f(j)$ robots.
- So specifically, if it is used in the $k^{\text{th}}$ second, and it has been $j$ seconds since it was previously used, then it will destroy $min(x_k, f(j))$ robots. (After this use, it will be completely drained.)
- We will also assume that the EMP starts off completely drained, so if it is used for the first time in the $j^{\text{th}}$ second, then it is capable of destroying up to $f(j)$ robots.

**The problem.** Given the data on robot arrivals $x_1, x_2, ..., x_n$, and given the recharging function $f()$, choose the points in time at which you're going to activate the EMP so as to destroy as many robots as possible.

### Answer

**Part (a)** Changing $x_1$ from 1 to 2 will get this algorithm to only use the EMP in the last second, whereas the optimal solution remains unchanged (use EMP at seconds 3 and 4). More generally, if $f(i) = 1$ for every $i$, while $x_i > 1$ for every $i$, the optimal solution is to use the EMP every second, but this algorithm will return just the one time in the end.

**Subproblem:** Let $R(i)$ be the maximum robots that can be killed at time step $i$.

**Recurrence:**

$$R(i) = \begin{cases} 0 & \text{if } i < 1 \\ min(f(1), x_1) & \text{if } i = 1 \\ \max\limits_{j=0}^{i-1} \big(R(j) + min(f(i-j), x_i)\big) & \text{otherwise} \end{cases}$$

**Desired output:** $R(n)$.

**Space and time:** $\mathcal{O}(n)$ space required to store the $n$ recurrence values and $\mathcal{O}(n^2)$ time, because the maximum is a second $i$-length loop.

## 4 Chapter 6, Exercise 9 (pp 320)

You're helping to run a high-performance computing system capable of processing several terabytes of data per day. For each of $n$ days, you're presented with a quantity of data; on day $i$, you're presented with $x_i$ terabytes. For each terabyte you process, you receive a fixed revenue, but any unprocessed data becomes unavailable at the end of the day (i.e., you can't work on it in any future day).

You can't always process everything each day because you're con- strained by the capabilities of your computing system, which can only process a fixed number of terabytes in a given day. In fact, it's running some one-of-a-kind software that, while very sophisticated, is not totally reliable, and so the amount of data you can process goes down with each day that passes since the most recent reboot of the system. On the first day after a reboot, you can process $s_1$ terabytes, on the second day after a reboot, you can process $s_2$ terabytes, and so on, up to $s_n$; we assume $s_1 > s_2 > s_3 > ... > s_n > 0$. (Of course, on day i you can only process up to $x_i$ terabytes, regardless of how fast your system is.) To get the system back to peak performance, you can choose to reboot it; but on any day you choose to reboot the system, you can't process any data at all.

**The problem.** Given the amounts of available data $x_1, x_2, ..., x_n$ for the next $n$ days, and given the profile of your system as expressed by $s_1, s_2, ..., s_n$ (and starting from a freshly rebooted system on day 1), choose the days on which you're going to reboot so as to maximize the total amount of data you process.

### Answer

**Part (a)** Optimal solution: reboot on day 2 and day 4 in order to process

| | **Day 1** | **Day 2** | **Day 3** | **Day 4** | **Day 5** |
|---|---|---|---|---|---|
| $x$ | 20 | 20 | 20 | 20 | 20 |
| $s$ | 15 | 5 | 4 | 3 | 2 |

15 terabytes on days 1, 3, and 5, which sum up to 45 of 100 terabytes.

**Subproblem:** Let $R(i, j)$ be the maximum number of bytes that can be processes between time steps 1 and $i$, given that the system has been up for $j$ time steps.

**Recurrence:**

$$R(i,j) = \begin{cases} 0 & \text{if } i < 1 \text{ OR } j < 1 \\ 0 & \text{if } i < j \\ \min(s_1, x_1) & \text{if } i = 1 \text{ AND } j = 1 \\ \max_{k=1}^{i-1}(R(i-1,k)) & \text{if } i > 0 \text{ AND } j = 0 \\ \max_{k=1}^{i-1}\big(R(i-1,k-1) + \min(s_k, x_i)\big) & \text{otherwise} \end{cases}$$

Invalid spaces are those where the machine has been up for longer than there's been events ($i < j$), which we set to 0 (for example, 5th day with the machine up for 7 days). First day and first event ($R(1,1)$) is always at peak processing power ($min(s_1, x_1)$). If there's ever a reboot ($j = 0$), that day is excluded and the best possible throughput is that of the previous day's, which depends on when the system was rebooted last.

**Desired output:** $\max_{k=1}^{n} R(n,k)$.

**Space and time:** $\mathcal{O}(n^2)$ space required to store the $n$ recurrence values and $\mathcal{O}(n^3)$ time, because the maximum is a second $i$-length loop.

## 5 Chapter 8, Exercise 3 (pp 505)

Suppose you're helping to organize a summer sports camp, and the following problem comes up. The camp is supposed to have at least one counselor who's skilled at each of the $n$ sports covered by the camp (baseball, volleyball, and so on). They have received job applications from m potential counselors. For each of the $n$ sports, there is some subset of the $m$ applicants qualified in that sport. The question is: For a given number $k < m$, is it possible to hire at most $k$ of the counselors and have at least one counselor qualified in each of the $n$ sports? We'll call this the *Efficient Recruiting Problem.*

Show that Efficient Recruiting is NP-complete.

### Answer

Let's denote the set of sports as $S$, and the set of applicants $C = \{c_1, c_2, ..., c_m\}$, where for every $1 \leq i \leq m$, $c_i \subset S$. The assumption is that $\cup_{c \in C} = S$. The goal is to find whether or not given a number $k < m$, $C_k \subset C$ exists so that $|C_k| = k$ and $\cup_{c \in C_k} = S$. Let's denote an instance of $ER$ as $R = (S, C)$

$ER \in NP$ because given any $X \subset C$, $|X| = k$, $\cup_{c \in C_k} = S$ can be checked in polynomial time (or if guaranteed that no sports other than the $n$ exists in the sets of skills, simply checking the length of the union.)

To show that it is $NP$-complete, we show that vertex cover poly time many-one reduces to $ER$ ($VC \leq_m^P ER$); because vertex cover is $NP$-complete, and $ER \in NP$, therefore if $VC \leq_m^P ER$, then $ER$ is $NP$-complete.

Given any graph $G = (V, E)$, let every edge be represented as a sport, and let every vertex represent an applicant, and therefore every sport has exactly two qualified applicants. In other words, let $f(V) = \{\{e \in E | v \in e\} | \forall v \in V\}$. Therefore, any graph $G = (V, E)$ can be mapped to $R = (E, f(V))$.

If $G = (V, E)$ has a vertex cover of size $k$ ($G \in VC_k$), there exists a set $K \subset V$ where $|K| = k$ and for every edge $e \in E$ (consider edges to be sets of length 2), there exists some $v \in K$ so that $v \in e$. Now consider the $ER$ instance $(E, f(V))$, and $f(K)$. By definition, for every sport $s \in E$, there exists some applicant in $f(V)$ corresponding to some $v \in K$ so that $v \in s$. Therefore every sport has some applicant skilled in it.

On the other hand, assume some $(E, f(K)) \in ER_k$, which means there exists some subset of $f(K)$, $C_k$, with length $k$, so that $\cup_{c \in C_k} = E$. Therefore $K$ itself is by definition a vertex cover for $G = (V, E)$, and is of length $k$.

As a result, given any graph $G = (V, E)$, it has a vertex cover of length $k \iff (E, f(V))$ has an efficient recruiting of size $k$.

## 6 Chapter 8, Exercise 8 (pp 507)

Your friends' preschool-age daughter Madison has recently learned to spell some simple words. To help encourage this, her parents got her a colorful set of refrigerator magnets featuring the letters of the alphabet (some number of copies of the letter A, some number of copies of the letter B, and so on), and the last time you saw her the two of you spent a while arranging the magnets to spell out words that she knows.

Somehow with you and Madison, things always end up getting more elaborate than originally planned, and soon the two of you were trying to spell out words so as to use up all the magnets in the full set—that is, picking words that she knows how to spell, so that once they were all spelled out, each magnet was participating in the spelling of exactly one of the words. (Multiple copies of words are okay here; so for example, if the set of refrigerator magnets includes two copies each of C, A, and T, it would be okay to spell out CAT twice.)

This turned out to be pretty difficult, and it was only later that you realized a plausible reason for this. Suppose we consider a general version of the problem of *Using Up All the Refrigerator Magnets*, where we replace the English alphabet by an arbitrary collection of symbols, and we model Madison's vocabulary as an arbitrary set of strings over this collection of symbols. The goal is the same as in the previous paragraph.

Prove that the problem of Using Up All the Refrigerator Magnets is NP-complete.

### Answer

Given any instance of the problem $UAM$ (short for using all magnets), it is easy to verify in polynomial time whether or not it is a solution. Simply constructing a key-value pair and counting the number of times each letter is used (given all the words), and comparing to the information on magnets available and their counts to see if they match exactly, is poly time with respect to the total number of magnets available. Therefore $UAM \in NP$.

To show that it is $NP$-complete, we show that 3D matching poly time many-one reduces to $UAM$ ($3DM \leq_m^P UAM$).

Consider any instance of 3D matching, disjoint sets $X$, $Y$, and $Z$, and $T \subseteq X \times Y \times Z$ or simply $(X, Y, Z, T)$, where $|X| = |Y| = |Z| = n$, and $|T| > n$. Considering the general problem above, let our magnets to be elements in $X \cup Y \cup Z$, and $T$ be Madison's vocabulary. In that case: the existence of $n$ triples in $T$ so that each element of $X \cup Y \cup Z$ is contained in exactly one of these triples (solution to $3DM$) is the same as the existence of $n$ words in $T$ so that each magnet is participating in the spelling of exactly one of the words (solution to $UAM$).

Madison making $n$ words, each with 3 letters uniquely drawn from $X$, $Y$, and $Z$ guarantees all magnets are used.

Therefore, given any $I = (X, Y, Z, T)$, $I \in 3DM \iff f(I) \in UAM$ where $f(I)$ is an instance of $UAM$ with $X \cup Y \cup Z$ as our magnets, and $T \subset X \times Y \times Z$ as Madison's vocabulary.