MID-TERM

Remarks: In all algorithm, always explain how and why they work. ALWAYS, analyze the complexity of your algorithms. In all algorithms, always try to get the fastest possible. A correct algorithm with slow running time may not get full credit.

1. Question 1: In this question we deal with a lower bound for the number of comparison needed for an algorithm that finds both the maximum and the minimum. For any algorithm, define:

   • $a$: the number of elements that did not participate in any comparison.
   • $b$: the number of elements that only won so far (and won at least once).
   • $c$: the number of elements that only lost so far (and lost at least once).
   • $d$: the number of elements that lost at least once, and won at least once.

(a) Show that (for any algorithm that finds both the maximum and the minimum) at first $(a, b, c, d) = (n, 0, 0, 0)$ and at the end of the algorithm $(a, b, c, d) = (0, 1, 1, n - 2)$.

Answer: At start no element was compared. So $a = n$. Clearly, $b, c, d, f$ are 0.
At the end, I claim that $a = 0$. Otherwise, there is at least one element that was not compared at all. But then this element may be the maximum or the minimum. So the algorithm can’t really know the right solution yet (so how come it stopped?).
Also, if $b \geq 2$ at the end, this means that at least two elements only won so far. This can’t be. If this is the case, the algorithm can’t know which of the two is maximum. Similarly, $c \geq 2$ can’t be. So $c = b = 1$ at the end (as the maximum will always win and the minimum always will lose, we have both $1 \leq b, c < 2$, and so $c = b = 1$ at the end).

(b) Define $f = b + c + 3 \cdot d$. Show that at first $f = 0$ and at the end $f = 3 \cdot n - 4$.

Answer: Follows by plugging the above values.

(c) Show that for any algorithm one can find an input so that at each stage $f$ increases by at most 2.

Answer: 10 types of comparison can happen:

   i. The algorithm chooses to compare two $a$ type (didn’t play yet) elements.
      After this comparison, $a$ drops by 2, $b$ increases by 1 and $c$ increase by 1.
      It does not matter what the values are. In total (see the definition of $f$) $f$
      increases by 2.

   ii. The algorithm chooses to compare two $b$ elements: one of this type $b$ elements
      is turned to a type $d$ element (never minding the values they have). The
      other (the winning one) remains a $b$ element. So, $b$ drops by 1. In addition,
      $d$ increases by 1. The change in $f$ is a +2 increment.
iii. Two \( c \) elements: the same explanation as Item 1(c)ii.

iv. Two \( d \) elements: It does not matter. All will stay the same anyway, so no increase for \( f \) (if you think of it, such a comparison will not be made by an algorithm that tries to find the min and max, unless the algorithm had a lot to drink first.)

v. An \( a \) versus a \( b \) element. In the worse case, the \( b \) element will win. We can make sure by deciding on a large enough value of the \( b \) element as input.

**Remark:** given some algorithm, the goal is to show that for some input the increment in \( f \) for this algorithm on this specific input is by at most 2 for any comparison done by the algorithm. This means that you can decide upon the input, so it will make life “bad” for the algorithm.

Note that \( c \) increases by 1. The increase is only by 1 here.

vi. \( a \) versus \( c \). Similar to Item 1(c)v.

vii. \( a \) versus \( d \). Does not matter. \( c \) or \( b \) increase by 1. \( d \) unchanged. So \( f \) increases by 1.

viii. \( b \) versus \( c \). We can always set an input so that the \( b \) element wins. Nothing will change and \( f \) remains the same.

ix. \( b \) versus \( d \). Let the \( b \) element win. Nothing changed. So \( f \) stays the same.

x. \( c \) versus \( d \). The same as the previous item.

(d) Show that any algorithm requires at least \( 3 \cdot n/2 - 2 \) comparisons to find both the maximum and the minimum.

**Answer:** For any algorithm, \( f \) starts from 0 and has to get to \( 3n - 4 \). For any algorithm we can find a bad input that will make \( f \) increase by steps of at most 2. The number of comparison, must “bring” \( f \) to \( 3n - 4 \). Since we do this at “jumps” of 2, in the worst case, \( 3n/2 - 2 \) comparisons are required for any algorithm.

2. **Question 2:** Assume we are given \( k \) sorted arrays each with \( m \) elements. The total number of elements is thus \( n = m \cdot k \). Give an algorithm that produces one sorted array out of the \( k \) sorted arrays.

**Answer:** Let the sorted arrays be \( A_1, A_2, \ldots, A_k \). Pair the arrays into \( A_1, A_2 \), and \( A_3, A_4 \), and \( A_5, A_6 \) and so on. Since each pair has 2 arrays, both of which are sorted, we can use them using the procedure Merge shown in class. Call these \( k/2 \) merging, the first level merging.

After you merge the \( k/2 \) pairs, you remain with \( k/2 \) arrays. One for each pair. Now you do the same: pair them, and merge every 2 using mergesort. Lets call these merges the second level.

And so on. At the end we merge two arrays. This is the last level.

At each level, all the elements participate in the merge. They are all scanned, but only once. So, each level has \( O(n) \) basic operations. The number of levels is \( \lceil \log_2 k \rceil \): first you have \( k \) arrays; then \( k/2 \). Then \( k/4 \). After \( \lceil \log_2 k \rceil \) levels, you get \( k/2^{\lfloor \log_2 k \rfloor} \leq 1 \) arrays. Namely, one sorted array.
3. **Question 3:** A $k$ coloring of a collection of intervals is an assignment of one of the numbers $\{1, \ldots, k\}$ to every interval so that intersecting intervals have different colors (intervals colored $i$ are an independent set for every $i$). Give an algorithm that accepts as input a collection of intervals and finds the maximum size subset of intervals that is 2-colorable.

**Answer:** The algorithm should generalize the algorithm given in class for a maximum independent set: Order the intervals by increasing finish times. Let $\mathcal{L}$ be the list.

(a) Start with a set $S \leftarrow \emptyset$ of intervals.
(b) If $\mathcal{L} \neq \emptyset$ do
   i. Let $I$ be the next interval in $\mathcal{L}$.
   ii. If the collection of intervals $S \cup \{I\}$ is 2-colorable, add $I$ to $S$
   iii. Else, delete $I$ from $\mathcal{L}$ and Go-to 3b.
(c) Output $S$.

To check if the collection of intervals is two colorable, use the following.

We can create a graph with the vertices being the intervals, and two vertices sharing an edge if the two respective intervals intersect. Observe that the collection of intervals can be 2-colored, if and only if the graph resulting can be 2-colored so that no two adjacent vertices receive the same color.

Now, use the following lemma.

**Lemma 0.1** Let $v$ be an arbitrary vertex in $G$. A graph $G$ is 2-colorable, if and only if two vertices at the same distance from $v$ in $G$ are not neighbors

**Proof:** Say without loss of generality that the colors are 1 and 2 and $v$ is colored by 1. Then, $N(v)$ must be colored 2. So, the vertices at distance 2 from $v$ MUST be colored by 1, since they have an incoming edge from an $N(V)$ vertex, which is colored by 2. The vertices of distance 3 from $v$ must be colored 3, and so on. Thus, two vertices at the same distance from $v$ must be colored by the same color, and so can not share an edge.

In the second direction, if vertices at the same distance from $v$ share no edges, give vertices at even distance from $v$ color 1. Give vertices of odd distance from $v$ color 2. Two things remain for you. Show that this is a legal coloring and show that the running time of checking if a graph is 2-colorable is $O(E) = O(V^2)$. 

Thus, the total running time for the 2-coloring algorithm is $O(n^3)$ (n applications of the 2-colorability check).

4. **Question 4:** Assume that in the $\{0, 1\}$ Knapsack problem, we are given that $p_1 \leq p_2 \leq \ldots \leq p_n$ and $w_1 \geq w_2 \geq \ldots \geq w_n$. Give an algorithm for the problem in this special case.
**Answer:** We prove by induction that the partial solution produced by greedy is “extendible”, namely, there is some optimum containing the elements added so far by greedy (a witness).

The base of the induction is the empty set which is contained in any optimum.

Now, say that a collection $S = \{1, \ldots, i - 1\}$ of elements is selected. Therefore $i$ is the next element added. By the induction hypothesis, there is some optimum $OPT_1$ so that $S \subseteq OPT_1$.

If $i \in OPT_1$, then $OPT_1$ is still a good witness, even after adding $i$. Otherwise, $i \notin OPT_1$. Thus, there must be an element $j$ that belongs to $OPT_1$ but not to greedy. Otherwise, greedy has larger weight than $OPT_1$. Since the first $i$ items belong to greedy, $j > i$ and so $j$ has no smaller weight than $i$. So, define: $OPT_2 = (OPT_1 \cup \{i\}) \setminus \{j\}$.

As $j$ has no smaller weight than $i$, $OPT_2$ is a legal solution. In addition, as $j > i$, $j$ has no larger price. This means that their price is equal. Thus, $OPT_2$ is also an optimum, and is a witness for $\{1, \ldots, i\}$.

5. **Question 5:** Design an algorithm that solves correctly the coin changing problem for any collection of coins. Assume that the largest coin has value $k$, and 1 is one of the coins.

**Answer:** We use dynamic programming. Use an array $A$. The value of $A[i]$ is the number of coins in the optimum breaking of $i$. Clearly, $A[0] = 0$. Also $A[1] = 1$. We now explain how to fill $A[i]$ given that $A[0], A[1], \ldots, A[i - 1]$ are filled.

Let $S = \{1, \ldots, k\}$ be the coins. Let

$$\text{val} = \min_{j \in S} \{A[i - j]\},$$

and let $q$ be the index that has $A[q] = \text{val}$. This means that the best way to break $i$ is to first select $q$ and add to this coin, the optimal breaking of $i - j$. Thus, $A[i] \leftarrow 1 + A[i - q]$ (the extra of 1 is due to the coin $q$).

At the end, the optimum will be in $A[n]$. If the number of coins is $p$, the running time is $O(np)$.

6. **Question 6:** In the weighted maximum independent set of intervals problem, design an algorithm that finds the actual subset (not just its value).

Recall the algorithm shown in class. We assume that the activities are sorted in non-decreasing starting time $(s_i \leq s_{i+1}$ for all $i$). Let $P_i$ be the subproblem defined by the activities $\{i, \ldots, n\}$. Namely, $P_i$ is the problem of finding a maximum weight independent set in the sub-instance $\{i, \ldots, n\}$. Define a vector $M$. We wish to compute in $M(j)$ the value of the optimum solution $P_j$. We know that $M(n) = w_n$ (the only nonempty subset of $\{M(n)\}$ is $\{M(n)\}$ itself.)

how to compute $M(i)$. Let $\text{comp}(i)$ be the minimum $j > i$ so that $s_j \geq f_i$. Namely, $j$ is the smallest index of an activity, that starts after $i$ ends and thus is compatible with $i$ (so, if $i$ is chosen, all activities between $i + 1$ and $j - 1$, intersect activity $i$, and hence are not legal for choice). Note that in time $O(n^2)$ it is easy to find $\text{comp}(i)$ for every $i$. 

4
Now, we have
\[ M(i) = \max\{M(i + 1), w_i + M(\text{comp}(i))\}. \]

This is explained as follows. If \( i \) does not belong to the optimal solution, then \( M(i) = M(i + 1) \). Otherwise, \( i \) belongs to the optimum solution, and therefore, all the activities \( i + 1, \ldots, \text{comp}(i) - 1 \) cannot belong to the solution. Thus, we can add to \( i \) the best solution in \( \{\text{comp}(i), \ldots, n\} \), to get the best solution in \( \{i, \ldots, n\} \).

The question if \( i \) belongs to the solution or not depends on which maximum do we get. If \( M(i + 1) \) is the maximum, put “–” in that entry, and otherwise, put a plus.

Now, you scan the array from \( i = 1 \) onward. If the entry has a minus, this means that \( i \) is not in the solution. Thus, go to the next entry \( i + 1 \). Otherwise, \( i \) is in the solution and go to \( \text{comp}(i) \). The correctness is proved by induction and is left to you. The complexity is \( O(n^2) \).