# Section 6: Solutions

## 1. Short answer

The exam will include some questions that are multiple choice, followed by a space to briefly explain your answer. This station gives you some practice, but certainly not comprehensively—review the lectures before the exam!

Some problems, marked with (KT), are from your textbook, *Algorithm Design* by Kleinberg and Tardos.

(a) (KT 1.2) Consider an instance of the Stable Matching Problem in which there exists a proposer $p$ and a receiver $r$ such that $p$ is ranked first on the preference list of $r$ and $r$ is ranked first on the preference list of $p$. Then in every stable matching $S$ for this instance, the pair $(p, r)$ belongs to $S$.

◯ True

◯ False

Briefly justify your answer.

**Solution:**

True. If $p$ and $r$ were not matched, then they prefer each other over their current matches, so this is an instability.

(b) Select all that apply. Running DFS on a directed acyclic graph may produce:

◯ Tree edges

◯ Back edges

◯ Forward edges

◯ Cross edges

Briefly justify your answer.

**Solution:**

All except back edges. A back edge would create a cycle. Students should provide an example of a graph and execution of DFS that produces tree edges, forward edges, and cross edges.

(c) A recursive algorithm for a problem that reduces the problem on inputs of size $n$ to 2 sub-problems of the same type of size $\frac{n}{3}$ plus $\Theta(n^2)$ extra work has running time:

◯ $\Theta(n^2)$

◯ $\Theta(n^2 \log n)$

◯ $\Theta(n^{\log_2 3})$

◯ $\Theta(n^{\log_3 2})$

Briefly justify your answer.

**Solution:**

$\Theta(n^2)$. The recurrence is $T(n) = 2T(\frac{n}{3}) + \Theta(n^2)$. By the master theorem, since $2 < 3^2$, the running time is $T(n) = \Theta(n^2)$.

(d) (KT 4.2(a)) Suppose we are given an instance of the Minimum Spanning Tree Problem on a graph $G$, with edge costs that are all positive and distinct. Let $T$ be a minimum spanning tree for this instance. Now suppose we replace each edge cost $c_e$ by its square, $c_e^2$ , thereby creating a new instance of the problem with the same graph but different costs. Then, $T$ must still be a minimum spanning tree for this new instance.

◯ True

◯ False

Briefly justify your answer.

**Solution:**

True. Kruskal's algorithnm (or Prim's algorithm) only depends on the relative order of edge costs. Furthermore, because costs are distinct, there is a unique MST, so Kruskal's algorithm found $T$ with the original edge weights, and will still find $T$ after updating. Thus by correctness of Kruskal's algorithm, $T$ is the MST of the new graph.

(e) (KT 4) Let $G = (V, E)$ be any weighted, undirected graph. Let $S \subseteq V$ be any cut, and let $e$ be an edge of minimum weight across the cut $S$. Then every minimum spanning tree contains the edge $e$.

◯ True

◯ False

Briefly justify your answer.

**Solution:**

False. The theorem requires edge weights be distinct. For a counterexample, consider the square with edge weight 1 on every edge. Deleting any edge creates an MST, so there is no edge contained in every minimum spanning tree.

## 2. Stable matching reduction: Horseback riding

*A version of this problem appeared on the Section 1 handout.*

Suppose there is a set of $R$ riders and $H$ horses with more riders than horses, in particular, $2H < R < 3H$. You wish to set up a set of $3$ rounds of rides which will give each rider exactly one chance to ride a horse. To keep things fair among the horses, you wish for each to have exactly $2$ or $3$ rides.

Because it's winter, by the time the third ride starts it will be very dark, so every rider would prefer *any* horse on the first two rides over being on the third ride. Between the first two rides, each rider doesn't have a preference over time of day, and have the same preference over horses. If a rider must be on the third ride, it has the same preference list for that ride as well.

Each horse has a single list over riders, which doesn't change by ride. Since horses love their jobs, they prefer to being one of the horses on the third ride to one of the ones left home.

Design an algorithm which calls the following library *exactly once* and ensures there are no pairs $(r, h)$ which would both prefer to change the matching and get a better result for themselves.

`BasicStableMatching`
**Input:** A set of $2k$ agents in two groups of $k$ agents each with preference lists over the other group.
**Output:** A stable matching among the $2k$ agents.

**Solution:**

**Algorithm**:

For each horse $h$ in the original instance, create three agents $h_1$, $h_2$, and $h_3$ representing three potential rides with horse $h$. To make the total number of riders equal to $3H$, add "dummy" riders $d$ until the number of riders and horses is equal. The update preference lists shall be:

- For each real rider $r$: from $r$'s original list, replace each $h$ with $h_1$ and $h_2$ (either order). Then at the end, add another copy of the original list with each $hi$ replaced by $h_3$.
- For each dummy rider $d$: all of the $h_3$ (in any order), followed by everything else (in any order).
- For each horse-in-round $h_k$: identical to $h$'s original list, then listing the dummy riders in any order.

Now, run the `BasicStableMatching` algorithm, then delete the dummy riders, and leave any horse whose partner was deleted unmatched.

**Correctness/Ride Quantities**: First, each (real) rider is matched exactly once because `BasicStableMatching` returns a perfect matching. To show that no horse is free on the first two rides: suppose for contradiction a dummy rider $d$ matched with a horse $h$ on the first two rides. This creates an unstable pair, since $h$ and any rider assigned to the third round prefer each other, which contradicts the correctness of `BasicStableMatching`.

**Correctness/Stable**: Suppose for contradiction there is a pair $(r, h)$ where $r$ and $h$ would both prefer to be paired on ride $k$ (over their current state). Then, by carefully considering the construction, $(r, h_k)$ is an unstable pair for the `BasicStableMatching` instance, contradiction

**Running time**: $\Theta(h^2)$. We have $3h$ agents on each side, so the guarantee on `BasicStableMatching` gives a $\Theta(h^2)$ guarantee for that call. All the other operations (copying lists, creating agents, etc.) can be done in time linear in the size of the final instance, which is also $\Theta(h^2)$ ($\Theta(h)$ agents, each with lists of length $\Theta(h)$).

## 3. Graph modeling: Moving apartments

In a certain apartment building, people are moving around. There are some people moving between units in the same building (for example, they want a bigger or smaller space), some new arrivals, and some people moving out entirely. Your job is to determine in what order everyone should move, so that for every unit, the person inside moves out before the next person moves in. Call this a "valid moving order".

In particular, you are given a list of pairs $P = [(a_1, b_1), (a_2, b_2), ..., (a_n, b_n)]$ where each pair is the starting and ending unit of a particular person. (Apartment unit numbers are integers.) People who are moving in from somewhere else have $a_i =$ null and people who are moving out entirely have $b_i =$ null.

You may assume that in total, at most 1 person is leaving each unit, and at most 1 person is moving into each apartment. Give an efficient algorithm that returns:

- All of the pairs in a valid moving order, or
- "Not possible" and a minimal list of pairs preventing such an ordering from being possible.

**Examples**

- **Input**: $(1, 2)$, $(2, 3)$, $(\text{null}, 1)$, $(4, \text{null})$
  **Output**: $(2, 3)$, $(1, 2)$, $(\text{null}, 1)$, $(4, \text{null})$ (other solutions may exist)
- **Input**: $(1, 2)$, $(2, 3)$, $(3, 1)$, $(4, 1)$
  **Output**: "Not possible" because $(1, 2)$, $(2, 3)$, $(3, 1)$. (A valid moving ordering is not possible in this example because no one can be the first to move – they each need one of the others to vacate their unit first.)

(a) Describe an algorithm to solve this problem.

(b) Briefly justify the correctness of the algorithm.

(c) What the running time of your algorithm?

**Solution:**

(a) The algorithm is as follows:

1. Let each pair be a vertex. Create a directed edge from $(a_i, b_i)$ to $(a_j, b_j)$ if and only if $a_i = b_j$ (and neither are null).
2. Run B/DFS to find a cycle in the graph, if a cycle exists.
   a. If there is a cycle, return "Not possible" and the pairs in the cycle.
   b. Otherwise, topologically sort the graph and return all vertices in topological order.

(b) If the constructed graph has a cycle, then no one in the cycle can be the first to move, for the same reason as the example above. If there is no cycle, then by definition of topological sort, all edges point forward. That is, $(a_i, b_i)$ will be before $(a_j, b_j)$ whenever $a_i = b_j$. This means unit $a_i = b_j$ will be vacant when $j$ move in, because $i$ moved out first.

(c) Since every vertex $(a_i, b_i)$ has at most one in-edge (requiring $b_i$'s current tenant to move out first) and one out-edge (so that $a_i$'s future tenant can move in), the number of edges is at most $n$. Thus, B/DFS and topological sort run in $O(n)$ time. However, a naive construction of the graph would require $O(n^2)$ time, since as defined, we are comparing every pair of vertices. (Note: Any running times that include $m$ are incorrect, since $m$ is not part of the problem statement.)

As an aside, this can be improved. Since there is at most one out-edge per vertex, we can first create all the vertices, then use binary search trees to find the right vertex to point to, using $O(n \log n)$ time total. This was not required, as we only asked for an "efficient" (polynomial-time) algorithm.

# 4. Greedy: Minimizing covers again

*A version of this problem appeared on the Section 3 handout.*

You have a set $\mathcal{X}$ of (possibly overlapping) integer intervals $[a, b] = \{x \in \mathbb{Z} \mid a \leq x \leq b\}$. You wish to choose a subset $\mathcal{Y}$ of the intervals to cover the full set. Here, cover means the union of all intervals in $\mathcal{X}$ is equal to the union of all intervals in $\mathcal{Y}$. Describe (and prove correct) an algorithm which gives you a cover with the fewest intervals.

**Solution:**

**Key idea**: Consider all intervals that cover the next point, and among all such intervals take one that goes the farthest right.

**Algorithm**:

**Input**: A set $\mathcal{X}$ of intervals $[a, b] \subseteq \mathbb{Z}$.
**Expected output**: A smallest set $\mathcal{Y} \subseteq \mathcal{X}$ such that $\bigcup_{X \in \mathcal{X}} X = \bigcup_{Y \in \mathcal{Y}} Y$.

```
1:  Place X in a deque and sort it increasing by start point.
2:  i ← 1
3:  Y ← ∅
4:  while X ≠ ∅ do
5:      [s, t] ← X.peek_front()
6:      y_i ← max(b_{i−1} + 1, s)                    ▷ with b_0 = −∞
7:      [a_i, b_i] ← [s, t]
8:      while X ≠ ∅ do
9:          [a, b] ← X.peek_front()
10:         if y_i ∈ [a, b] then
11:             if b > b_i then
12:                 [a_i, b_i] ← [a, b]
13:             X.delete_front()
14:         else
15:             break
16:     Add [a_i, b_i] to Y.
17:     i ← i + 1
18: output Y
```

First, we make a few observations (loop invariants) about what this code is doing in high-level terms.

(a) At the beginning of every iteration of the outer loop, $\mathcal{X}$ consists of the set of intervals not containing $y_1, \ldots, y_{i-1}$.

(b) On line 6, $y_i = \max(b_{i-1}, s)$ is the smallest point in $\mathcal{X}$ not covered by $\mathcal{Y}$. This is because $b_{i-1}$ is the largest covered point. If $b_{i-1} + 1$ is in $\mathcal{X}$, then it is the smallest uncovered point, but if there is a gap, then $s$ is the smallest uncovered point.

(c) Lines 7 through 15 find the interval $[a_i, b_i]$ with largest endpoint that contains $y_i$, and deletes all intervals containing $y_i$.

**Each line is valid**: The only lines that require justification are $\mathcal{X}$.`peek_front`(), and in each case we have checked that $\mathcal{X} \neq \varnothing$ immediately before, so we are good.

**Termination**: For the inner loop, we either reduce the size of $\mathcal{X}$ or break, so this terminates. For the outer loop, we always at least delete $[s, t]$, so it terminates too.

**Correctness/Covering**: In iteration $i$, we only delete the intervals that are covered by $[a_1, b_1], \ldots, [a_i, b_i]$. (In particular, if $[a, b]$ is deleted, then by observation (b), everything before $y_i$ is covered by $[a_1, b_1], \ldots, [a_{i-1}, b_{i-1}]$, and by observation (c), everything between $y_i$ and $b$ is covered by $[a_i, b_i]$.)

**Correctness/Optimality**: Let $\mathsf{ALG} = [a_1, b_1], [a_2, b_2], \ldots, [a_k, b_k]$ be the list of intervals found by the algorithm, and let $\mathsf{OTH} = [o_1, p_1], \ldots, [o_j, p_j]$ be the list of intervals in any other cover. Assume that the $o_i$ are increasing.

**Claim 1.** *For all $i$, we have $b_i \geq p_i$.*

Once we have this, by noting that for every $i < k$, there are uncovered points larger than $b_i$ after iteration $i$ (hence also larger than $p_i$), we conclude that OTH must also have at least $k$ intervals, as was to be shown.

*Proof of Claim 1.*

**BC**: Both ALG and OTH must cover $y_1$. Furthermore, $[o_1, p_1]$ must cover $y_1$, because the $o_i$ are increasing, and $y_1$ is the smallest start point of any interval $\mathcal{X}$. Then, $b_1$ was chosen to be the largest end point of the all intervals covering $y_1$, so $b_1 \geq p_1$.

**IH**: Suppose $b_j \geq p_j$.

**IS**: The point $y_{j+1}$ must be covered. It is not covered by $[o_1, p_1], \ldots, [o_j, p_j]$ by induction. If $[o_{j+1}, p_{j+1}]$ does not cover $y_{j+1}$, then by sortedness, OTH is invalid. But if $[o_{j+1}, p_{j+1}]$ covers $y_{j+1}$, we selected $b_{i+1}$ such that $b_{i+1} \geq p_{j+1}$, as was to be shown. □

**Running Time**: Sorting take $O(n \log n)$ time, and for the rest, note that a constant amount of work is done in between each time we call $\mathcal{X}$.`delete_front`(). Thus, the rest takes $O(n)$ time, where $n = |\mathcal{X}|$.

# 5. Divide and conquer: Peak performance

*A version of this problem appeared on the Section 4 handout.*

Let $A[1\,..\,n]$ be an array of integers. Call $A$ a *mountain* if there exists an index $i$ called the "peak", such that $A[1] < \cdots < A[i-1] < A[i]$ and $A[i] > A[i+1] > \cdots > A[n]$. We allow the peak to be at index 1 or $n$ (that is, a strictly increasing or strictly decreasing array is still a mountain). For simplicity, we are not allowing $A[j] = A[j+1]$ for any $j$. More formally, mountains satisfy:

- For all $1 \le j < i$, we have $A[j] < A[j+1]$, and
- For all $i \le j < n$, we have $A[j] > A[j+1]$.

(a) Given a mountain $A$, describe an algorithm to find the index of the peak in $O(\log n)$ time.

(b) (Extra credit) Prove that there is no deterministic algorithm for deciding in the same $O(\log n)$ running time: Given an array, whether or not it is a mountain.

(Deterministic means that we don't use randomness, like almost all algorithm we study in this class. In other words, the algorithm always has the same output when viewing the same input.)

**Solution:**

(a) **Key idea**: We adapt binary search — by looking at consecutive elements, we can see if we're on the "upward" or "downward" slope and find the peak.

```
1:  global A

    Input: Indices i and j such that A[i..j] contains the peak of A.
    Output: The peak of A
2:  function PEAKFINDER(i, j)
3:      if i = j then
4:          return i
5:      m ← ⌊(i+j)/2⌋
6:      if A[m+1] exists and A[m] < A[m+1] and m+1 ≤ j then
7:          return PEAKFINDER(m+1, j)
8:      else if A[m-1] exists and A[m-1] > A[m] and i ≤ m-1 then
9:          return PEAKFINDER(i, m-1)
10:     else
11:         return m
12: output PEAKFINDER(1, n).
```

The running time is the same as binary search, which is $O(\log n)$.

We will show by strong induction on $k$ that for all $k$, given indices $i$ and $j$ such that $k = |\{i, \ldots, j\}|$ and $A[i\,..\,j]$ contains the peak of $A$, the function PEAKFINDER$(i, j)$ returns the peak of $A$.

**BC**: When $k = 1$, the only way for $A[i\,..\,j]$ to contain the peak of $A$ is for $i = j$ to be the peak. In this case, we return $i$ as intended.

**IH**: Suppose that the claim above is true for all $k \le \ell$.

**IS**: We will show the claim for $k = \ell + 1$. Take the following three cases, corresponding to the three cases in the code.

*Case 1*: The peak is in $A[m+1\,..\,j]$. Then $a_{m+1}$ certainly exists, and $a_m$ is in the increasing part of the mountain, and certainly $m + 1 \le j$. Thus, the code will enter the first "if" branch on line 6. The case hypothesis allows us to apply the IH, which tells us that PEAKFINDER$(m+1, j)$ returns the peak, as desired.

*Case 2*: The peak is in $A[i\,..\,m-1]$. Similar to Case 1.

*Case 3*: The peak is $a_m$. Then $a_m > a_{m+1}$ or $a_{m+1}$ doesn't exist, and $a_{m-1} < a_m$ or $a_{m-1}$ doesn't exist.

Thus, the code will hit the "else" case and return $m$, as desired.

(b) The main idea is that you need to examine *every* element of the array to know if it's a mountain, which requires at least $O(n)$ time.

To see why, suppose for contradiction that there exists an algorithm for this problem that looks at only $\leq n-1$ entries. Let A be a mountain. By correctness of the algorithm, it outputs true on input A. Let $a_k$ be an array element that the algorithm did not look at. Let $A'$ be an array identical to $A$ on all elements except $a_k$, which we set to make $A'$ not a mountain (e.g. by lowering $a_k$ below both of its neighbors, or above its neighbor if $k$ is an endpoint). Since the algorithm did not look at $a_k$, its execution on $A$ must be identical to its execution on $A'$. So the algorithm will return true on $A'$, contradicting its correctness.

**Solution:**

**Remark**: It is very easy to make a mistake in the IH here. The following are not correct IH's:

- For all arrays of length $\leq k$, the function PEAKFINDER$(i, j)$ returns the peak.

  (Many issues, not least of which is that PEAKFINDER doesn't even take an array as input.)

- For all arrays of length $\leq k$, the outer function (the one which returns PEAKFINDER$(1, n)$) returns the peak of $A$.

  (Although the claim is true, this is not a useful IH. You are never calling this outer function, so you have no opportunity to use the IH in your proof.)

- For all indices $i, j$, the function PEAKFINDER$(i, j)$ returns the peak.

  (Not specific enough—the peak of what? If you wrote this, we would assume you meant the peak of $A$. But that's wrong. If you call PEAKFINDER$(i, j)$ when $A[i\,..\,j]$ does not contain the peak of $A$, it will not return the peak of $A$.)

Another correct IH would be:

"For all indices $i, j$, the function PEAKFINDER$(i, j)$ returns the peak of $A[i\,..\,j]$."

The proof would be slightly more complicated, but also correct. You would need to notice (and maybe prove) that every subarray of a mountain is a mountain.

## 6. Dynamic programming: Driving in Manhattan

Imagine this problem taking place in a city with a grid of one-way streets like Manhattan, but where each street only goes East or North (all routes lead to the Upper East Side). As usual, the city is under construction, so some intersections are blocked and impassible. At other intersections, as you pass through the intersection you may be able to collect a reward, or be asked to pay a toll. Turning at an intersection counts as passing through. You want to get the largest net gain possible while taking a route following the one-way streets from an intersection designated $(0, 0)$ to an intersection designated $(m, n)$ that is $m$ blocks North and $n$ blocks East (if such a route even exists). (Your net gain is the sum of the rewards minus the sum of the tolls you need to pay.)

You are given this information in an 2D array of real numbers $R[1\,..\,m, 1\,..\,n]$. If $R[i, j] > 0$, then this is the value of the reward for going through this intersection. If $R[i, j] < 0$, you must pay a toll of $|R[i, j]|$ to go through the intersection. If $R[i, j] = 0$, there is no reward or toll. If $R[i, j] = -\infty$, then this intersection is impassible. You should return the maximum net gain, or if there is no valid path at all, your algorithm should return $-\infty$. Note that you do collect the rewards/pay the tolls at $(1, 1)$ and $(m, n)$.

Design a dynamic programming solution to this problem.

(a) Define, in English, the quantities that you will use for your recursive solution.

**Solution:**

Let $\mathsf{OPT}(i, j)$ be the largest net gain possible in taking a route from $(1, 1)$ to $(i, j)$ in the grid if one exists and $-\infty$ otherwise. The parameter $i$ ranges from 1 (or 0) to $m$, and the parameter $j$ ranges from 1 (or 0) to $n$.

(b) Given a recurrence relation for the quantities you have defined.

**Solution:**

The recursive case is, for $i, j > 2$:

$$\mathsf{OPT}(i, j) = R[i, j] + \max(\mathsf{OPT}(i - 1, j), \mathsf{OPT}(i, j - 1)),$$

where $-\infty + v = -\infty$ and $\max(-\infty, v) = v$ for all $v \in \mathbb{R}$. The base cases are

$$\begin{aligned} \mathsf{OPT}(1, 1) &= R[1, 1] \\ \mathsf{OPT}(1, j) &= R[1, j] + \mathsf{OPT}(1, j - 1) && (j > 2) \\ \mathsf{OPT}(i, 1) &= R[i, 1] + \mathsf{OPT}(i - 1, 1). && (i > 2) \end{aligned}$$

(c) Argue for the correctness of your recurrence relation.

**Solution:**

For the recursive case, if at $(i, j)$, we must take the reward or pay the toll $R[i, j]$, and we must have come from either $(i - 1, j)$ or $(i, j - 1)$. Thus, we take the maximum net gain possible from either case.

For the base cases, this expresses that the only valid way to reach $(1, j)$ is to go up from $(1, 1)$ until hitting $(1, j)$, and similarly for $(i, 1)$. Note that although technically still correct, it would have be suboptimal to write $\mathsf{OPT}(1, j) = R[1, 1] + \cdots + R[1, j]$ and similarly for $(i, 1)$. The reason is because naively implementing such base cases as written would result in $O(\max(m, n)^2)$ running time, which is worse than what we actually get.

(d) Describe the parameters for the subproblems in your recursion and how you will store their solutions.

**Solution:**

There are subproblems for $i = 1, \ldots, m$ and $j = 1, \ldots n$ which could be stored in a (2D) array of size $m \times n$.

(e) In what order can you evaluate them iteratively?

**Solution:**

There are many possibilities. Some options include:

- All base cases, then outer loop on $i$ from 2 to $m$, then inner loop on $j$ from 2 to $n$.
- All base cases, then outer loop on $j$ from 2 to $n$, then inner loop on $i$ from 2 to $m$.

(f) Write the pseudocode for your iterative algorithm.

**Solution:**

```
OPT[1, 1] ← R[1, 1]
for j ← 2 to n do
    OPT[1, j] ← R[1, j] + OPT[1, j − 1]
for i ← 2 to m do
    OPT[i, 1] ← R[i, 1] + OPT[i − 1, 1]
for i ← 2 to m do
    for j ← 2 to n do
        OPT[j, K] ← min(OPT[j − 1, K], OPT[s, K − 1] + (j − s))
output OPT[n, k]
```

(g) Analyze the running time of your algorithm in terms of $n$ and $k$.

**Solution:**

The base cases take $O(m + n)$ time. Each inner iteration takes $O(1)$ time, so in total, this is $O(mn)$.