Instructions. You make work in groups, but you must write solutions yourself. List collaborators on your submission.

If you are asked to design an algorithm, please provide: (a) the pseudocode or precise description in words of the algorithm, (b) an explanation of the intuition for the algorithm, (c) a proof of correctness, (d) the running time of your algorithm and (e) justification for your running time analysis.

Submissions. Please submit a PDF file. You may submit a scanned handwritten document, but a typed submission is preferred. Please assign pages to questions in Gradescope.

1. (15 points) Stable Matching Running Time. In class, we saw that the Propose-and-reject algorithm terminates in at most $n^2$ iterations, when there are $n$ students and $n$ colleges.

(a) Construct an instance so that the algorithm requires only $O(n)$ iterations, and prove this fact. Your construction should be possible for all values of $n$.

(b) Construct an instance so that the algorithm requires $\Omega(n^2)$ iterations (that is, it requires at least $cn^2$ iterations for some constant $0 < c \leq 1$), and prove this fact. Your construction should be possible for all values of $n$.

Solution:

(a) Construct any instance where all colleges have different top-ranked students. In the first $n$ iterations, each college will propose to an unmatched student, who will accept the offer from the college. Then, after $n$ iterations, no colleges will remain free and the algorithm will terminate. A concrete construction for any $n$ is as follows: college $c_i$ ranks student $s_i$ first, and then ranks the remaining students in any order.

(b) Construct an instance of size $n$ with “universal” preferences—all colleges have the same preference list, say $s_1 > s_2 > ... > s_n$. Regardless of student preferences, after running the algorithm each college will be matched to some student. Because of the property that colleges propose to students in order until, whichever college is matched to student $s_j$ in the end will have proposed to the first $j$ students. This is true for all $j = 1, \ldots, n$. Thus, the total number of proposals is $T(n) = 1 + 2 + ... + n = n(n + 1)/2 \geq n^2/2 = \Omega(n^2)$, as we argued in class.

2. (20 points) Stable Matchings: K&T Ch 1, Ex 5. Consider a version of the stable matching problem where there are $n$ students and $n$ colleges as before. Assume each student ranks the colleges (and vice versa), but now we allow ties in the ranking. In other words, we could have a school that is indifferent two students $s_1$ and $s_2$, but prefers either of them over some other student $s_3$ (and vice versa). We say a student $s$ prefers college $c_1$ to $c_2$ if $c_1$ is ranked higher on the $s$’s preference list and $c_1$ and $c_2$ are not tied.

(a) Strong Instability. A strong instability in a matching is a student-college pair, each of which prefer each other to their current pairing. In other words, neither is indifferent about the switch. Does there always exist a matching with no strong instability? Either give an example instance for which all matchings have a strong instability (and prove it), or give and analyze an algorithm that is guaranteed to find a matching with no strong instabilities.

(b) Weak Instability. A weak instability in a matching is a student-college pair where one party prefers the other, and the other may be indifferent. Formally, a student $s$ and a college $c$ with pairs $c'$ and $s'$ form a weak instability if either

- $s$ prefers $c$ to $c'$ and $c$ either prefers $s$ to $s'$ or is indifferent between $s$ and $s'$.
- $c$ prefers $s$ to $s'$ and $s$ either prefers $c$ to $c'$ or is indifferent between $c$ and $c'$.
Does there always exist a perfect matching with no weak instability? Either give an instance with a weak instability or an algorithm that is guaranteed to find one.

Solution:

(a) Yes, there is always a matching with no strong instability. A simple way to find this matching is to design a way to break ties, then run the stable matching algorithm. Consider breaking ties lexicographically: if a college is indifferent to students $s_i$ and $s_j$ and $i < j$, then $s_j$ is ranked higher; otherwise $s_j$ is ranked higher. Ties in students’ preference lists are broken similarly. This establishes an ordered preference list for every college and student. Running the stable matching algorithm will produce a stable matching $M$, which by definition has no instabilities with respect to the fully ordered preference lists. Then, note that a strong instability with respect to the weak preferences would imply an instability with respect to the fully-ordered preferences, so a strong stability cannot exist in $M$.

(b) There doesn’t always exist a perfect matching with no weak instability. Consider the case where $n = 2$ and $c_1$ is indifferent between $s_1$ and $s_2$ while both students prefer $c_1$ to $c_2$. No matter which student matches with $c_1$, there will always be a weak instability since the student not matched with $c_1$ prefers $c_2$ and $c_1$ is indifferent between both students.

3. (15 points) Big-O. For each function $f(n)$ below, find (1) the smallest integer constant $H$ such that $f(n) = O(n^H)$, and (2) the largest positive real constant $L$ such that $f(n) = \Omega(n^L)$. Otherwise, indicate that $H$ or $L$ do not exist. All logarithms are with base 2. Your answer should consist of: (1) the correct value of $H$, (2) a proof that $f(n)$ is $O(n^H)$, (3) the correct value of $L$, (4) a proof that $f(n)$ is $\Omega(n^L)$.

(a) $f(n) = \frac{1}{2}n^2$.
(b) $f(n) = n(\log n)^3$.
(c) $f(n) = \sum_{i=0}^{\lfloor \log n \rfloor} \frac{n}{2^i}$.
(d) $f(n) = \sum_{i=1}^{n} i^3$.
(e) $f(n) = 2^{(\log n)^2}$.

Solution:

(a) $H = 2$. It is immediate from the definition that $f(n)$ is $O(n^2)$, since $f(n) \leq \frac{1}{2}n^2$ for all $n \geq 0$. But $f(n)$ cannot be $O(n)$. This would imply the existence of constants $c$ and $n_0$ such that $\frac{1}{2}n^2 \leq cn$ for all $n \geq 0$. Dividing both sides by $n$ and rearranging, this would imply that $n \leq 2c$ for all $n \geq n_0$, which is an obvious contradiction.

(b) $H = 2$. We can observe that $f(n) \geq \frac{1}{2}n^2$ which would imply that $f(n) = \Omega(n^2)$. To show that there is no positive real $c > 2$ such that $f(n) = \Omega(n^c)$ we use proof by contradiction. Assume that for some positive real $c > 2$, $f(n) \geq n^c \Rightarrow \frac{1}{2}n^2 \geq n^c \Rightarrow \frac{1}{2} \geq n^{c-2}$ for all $n \geq 0$ which is a contradiction.

(b) $H = 2$. To show that $n(\log n)^3$ is not $O(n)$, divide by $n$ and notice that $f(n)/n = \log n$ is not $O(1)$ since $\log n$ is an increasing function of $n$. To show that $n(\log n)^3$ is $O(n^2)$, divide by $n$ and observe that $f(n)/n = \log n$ is $O(n)$ — this follows from the relationship between logarithms and polynomials from class and the book.

(c) $H = L = 1$. An exact expression for $f(n)$ is $f(n) = n[1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{\log n + 1}]$, which is always bigger than $n$. However, the infinite sum $1 + \frac{1}{2} + \frac{1}{3} + \ldots$ is equal to 2, so we have $f(n) \leq 2n$. Also, $f(n) \geq n$ from the definition of $f(n)$. These facts imply that $f(n)$ is $\Theta(n)$. 


(d) $H = L = 4$. Assume $n$ is even. One easy observation is that $f(n) = 1^3 + 2^3 + \ldots + n^3 \leq n^3 + n^3 + \ldots + n^3 = n^3$ by using $n$ as an upper bound for each term in the sum. So $f(n)$ is $O(n^4)$. Similarly, $f(n) = 1^3 + 2^3 + \ldots + n^3 \geq 0 + \ldots + 0 + (n/2)^3 + \ldots + (n/2)^3 = \frac{4}{2}(\frac{n}{2})^3 = \frac{n^4}{16}$ by using 0 as a lower bound for the first $n/2$ terms of the sum and $n/2$ as a lower bound for the last $n/2$ terms. Therefore, $f(n)$ is $\Omega(n^4)$. These facts together imply that $f(n)$ is $\Theta(n^4)$.

(c) $2^{(\log n)^2} = n^{(\log n)$. This is not a polynomial function of $n$. Hence no such $c$ value exists for which $f(n) = O(n^c)$. It can proved formally by contradiction.

We can prove that for every real value $c$, there exists an integer $n_0$, such that $f(n) \geq n^c$ for all $n \geq n_0$. This implies that no positive real constant largest value for $L$ can exist.

4. (20 points) Asymptotics. K&T Ch 2, Ex 6. Given an array $A$ of $n$ integers, you’d like to output a two-dimensional $n \times n$ array $B$ in which $B[i, j] = \max \{A[i], A[i+1], \ldots, A[j]\}$ for each $i < j$.

For $i \geq j$ the value of $B[i, j]$ can be left as is.

for $i = 1, 2, \ldots, n$
for $j = i + 1, \ldots, n$
    Compute the maximum of the entries $A[i], A[i+1], \ldots, A[j]$.
    Store the maximum value in $B[i, j]$.

(a) Find a function $f$ such that the running time of the algorithm is $O(f(n))$, and clearly explain why.

(b) For the same function $f$ argue that the running time of the algorithm is also $\Omega(f(n))$. (This establishes an asymptotically tight bound $\Theta(f(n))$.)

(c) Design and analyze a faster algorithm for this problem. You should give an algorithm with running $O(g(n))$, where $\lim_{n \to \infty} g(n)/f(n) = 0$.

Solution:

(a) The running time of the algorithm is $O(n^3)$. The outer for loop executes exactly $n$ times and the inner for loop executes at most $n$ times for each value of $i$, so, the line of code that computes the maximum executes $O(n^2)$ times. Computing the maximum requires $j - i - 1$ steps, which is again $O(n)$. Hence the total running time for this line of code is $O(n^3)$. All other lines of the algorithm are dominated by this one, so the overall running time is $O(n^3)$.

(b) The running time is also $\Omega(n^3)$. To show this, we must prove the running time is at least $cn^3$ for some value $c$. To do this, it is OK to ignore some of the steps of the algorithm, and show that the remaining steps take time at least $cn^3$ for some $c$. We will look at executions of the body of the loop only for the cases where $i \leq n/4$ and $j \geq 3n/4$. In this case, we have $j - i - 1 \geq n/2$, so the line of work that does the addition takes at least $n/2$ time steps. There are $(n/4)^2$ pairs of $i$ and $j$ that match our criteria, so the total running time is at least $(n/4)^2(n/2) = n^3/32$. Hence, it is $\Omega(n^3)$, and, since we already showed it is $O(n^3)$, it is $\Theta(n^3)$.

(c) Here is an algorithm with running time that is $O(n^2)$:

for $i = 1, 2, \ldots, n$
    let running_max = $A[i]$
    for $j = i + 1, \ldots, n$
        let running_max = $\max(\text{running_max}, A[j])$
    $B[i,j] = \text{running_max}$

5. (10 points) DFS and BFS. K&T Ch 3, Ex 5. Suppose we have a connected graph $G = (V, E)$ and a vertex $u \in V$. If we run DFS from $u$, we obtain a tree $T$. Suppose that if we run BFS from $u$ we obtain exactly the same tree $T$. Prove that $G = T$. (Hint: see Facts (3.4) and (3.7) from the book.)

Solution: Let $T$ be the tree returned by both BFS and DFS. We will show that $G$ is equal to $T$. Suppose for the sake of contradiction that $G$ has an edge $(x, y)$ that does not belong to $T$. By the property we proved about DFS trees (Fact (3.7) in the book), either $x$ is an ancestor of $y$ or vice versa. Assume without loss of generality that $x$ is an ancestor of $y$. By the property we proved about BFS
trees (Fact (3.4) in the book), the layers of $x$ and $y$ differ by at most one. Since $x$ is an ancestor of $y$, this implies that $x$ must be the immediate parent of $y$ in $T$, or, in other words $(x, y)$ belongs to $T$. This is a contradiction.

6. **(20 points) Butterfly ID. K&T Ch 3 Ex 4.** Some of your friends have become amateur lepidopterists (they study butterflies). Often when they return from a trip with specimens of butterflies, it is very difficult for them to tell how many distinct species they’ve caught—thanks to the fact that many species look very similar to one another.

One day they return with $n$ butterflies, and they believe that each belongs to one of two different species, which we’ll call $A$ and $B$ for purposes of this discussion. They’d like to divide the $n$ specimens into two groups—those that belong to $A$ and those that belong to $B$—but it’s very hard for them to directly label any one specimen. So they decide to adopt the following approach.

For each pair of specimens $i$ and $j$, they study them carefully side by side. If they’re confident enough in their judgment, then they label the pair $(i, j)$ either “same” (meaning they believe them both to come from the same species) or “different” (meaning they believe them to come from different species). They also have the option of rendering no judgment on a given pair, in which case we’ll call the pair ambiguous.

So now they have the collection of $n$ specimens, as well as a collection of $m$ judgments (either “same” or “different”) for the pairs that were not declared to be ambiguous. They’d like to know if this data is consistent with the idea that each butterfly is from one of species $A$ or $B$. So more concretely, we’ll declare the $m$ judgments to be consistent if it is possible to label each specimen either $A$ or $B$ in such a way that for each pair $(i, j)$ labeled “same,” it is the case that $i$ and $j$ have the same label; and for each pair $(i, j)$ labeled “different,” it is the case that $i$ and $j$ have different labels. They’re in the middle of tediously working out whether their judgments are consistent, when one of them realizes that you probably have an algorithm that would answer this question right away. Give an algorithm with running time $O(m + n)$ that determines whether the $m$ judgments are consistent.

**Solution:** We construct a graph $G$ with $n$ nodes, and the edges labelled with the $m$ judgments collected. The idea is to use any graph traversal algorithm (such as BFS/DFS). Suppose we are using BFS to traverse $G$. Assume the graph is connected. If not, the same algorithm can be run on each connected component.

Pick a starting node $s$ and an arbitrary label, say $A$, for the starting node. In traversing the graph, when a new node $v$ is explored via edge $(u, v)$, use the label of $u$ and the judgment on edge $(u, v)$ to label $v$. This process uniquely labels all nodes based on tree edges and the arbitrary choice of label $A$ for $s$. When an already visited node is reached via a (non-tree) edge $(u, v)$, check if the endpoints $(u, v)$ are consistent—either same or different—with the judgment on the edge. If not, abort the traversal and report “inconsistent.” If the traversal ends normally, report “consistent.”

To argue correctness, we argue two cases. First, if the algorithm reports “consistent”, then the judgments are consistent because the algorithm has labeled all nodes using the labels $A$ or $B$ and (1) the labeling process guarantees that the labels are consistent with tree edges, and (2) the algorithm checks that the labels are consistent with non-tree edges—all edges have been traversed, so all judgments are covered. Second, if the algorithm reports “inconsistent”, the the judgments are inconsistent. This is true because each nodes was labeled in the unique way possible when discovered by a tree edge (up to switching the labels $A$ and $B$)—and the resulting labeling is inconsistent on some non-tree edge.

There is no way to change the labels to make all edges consistent.

The algorithm takes $O(m + n)$ time because it is a modification of BFS, with only constant-time operations (for labeling and checking) added for each edge within the execution of BFS.

7. **(0 points).** How long did it take you to complete this assignment?