Midterm 1

Write in the following boxes clearly and then double check.

Name :	
SID :	

- **The exam will last 110 minutes.**
- The exam has 4 questions with a total of 100 points. You may be eligible to receive partial credit for your proof even if your algorithm is only partially correct or inefficient.
- Only your writings inside the answer boxes will be graded. **Anything outside the boxes will not be graded.** The last page is provided to you as a blank scratch page.
- Answer all questions. Read them carefully first. Not all parts of a problem are weighted equally.
- Be precise and concise.
- The problems may **not** necessarily follow the order of increasing difficulty.
- The points assigned to each problem are by no means an indication of the problem's difficulty.
- Unless the problem states otherwise, you should assume constant time arithmetic on real numbers.
- If you use any algorithm from lecture and textbook as a blackbox, you can rely on the correctness of the quoted algorithm. If you modify an algorithm from textbook or lecture, you must explain the modifications precisely and clearly, and if asked for a proof of correctness, give one from scratch or give a modified version of the textbook proof of correctness.
- Good luck!

1 Antipasti Short Answers

(a) (5 pts) Let $f(n) = \log(n!)$. Prove $f(n) = \Theta(n \log n)$.

(b) (3 pts * 4) Fill out the following four blanks in terms of n, then give a construction of such a graph.

i.e. if your answer for a blank is $f(n)$, then describe describe how, given n, you can construct a graph on n vertices with $f(n)$ edges that satisfy the conditions. You are allowed to fill the blank in form of sums. **All graphs are simple** (no self-loops, at most one edge between two vertices).

In the first two parts below, we say a directed graph G is *weakly connected* if the undirected graph obtained by removing all edge directions from G is connected.

(i) There are at least ______ edges in a weakly connected directed acyclic graph on n vertices.

(ii) There are at most ______ edges in a weakly connected directed acyclic graph on n vertices.

(iii) There are at least ______ edges in a strongly connected directed graph on n vertices.

(iv) There are at most ______ edges in a strongly connected directed graph on n vertices.

(c) (7 pts) To implement `Dijkstra`'s algorithm, is it faster to use an adjacency list to represent the graph or is it faster to use an adjacency matrix? Explain why. Assume $|E| = o(|V|^2)$ (informally, assume the graph is not dense).
Hint: Recall from lecture that an adjacency matrix is stored as a 2D array (hence the word "matrix") and an adjacency list is stored as a 1D array of linked lists.

(d) (4 pts) How would you speed up computing the `FFT` of $(a_0, a_0, a_1, a_1, a_2, a_2, \dots)$ by a factor of roughly 2 (*i.e.* halve the runtime)?

(e) (8 pts) Is the following statement true or false? Justify your answer.

"Let an M-path between two vertices s and t (in an undirected, connected graph G) be a path such that the weight of every edge is at most M. If G has an M-path between s and t, then there is a minimum spanning tree of G with an M-path between s and t."

(a)

$$\begin{aligned}\log(n!) &\le \log(n^n) \\ &= n \log n\end{aligned}$$

Therefore $\log(n!) = O(n \log n)$

$$\begin{aligned}\log(n!) &\ge \log((\frac{n}{2})(\frac{n}{2}+1)\cdots(n)) \\ &\ge \log((\frac{n}{2})^{\frac{n}{2}}) \\ &= \frac{n}{2}\log\frac{n}{2} = \frac{n}{2}(\log n - \log 2)\end{aligned}$$

Therefore $\log(n!) = \Omega(n \log n)$

(b) (i) $n-1$. A tree of n vertices.

(ii) $\binom{n}{2}$. $E = \{(v_i, v_j) \mid i < j\}$.

(iii) n. A simple cycle of n vertices.

(iv)
- Assume there is at most one edge between two vertices regardless of directions: $\binom{n}{2}$. Same construction as (ii) except we replace (v_1, v_n) with (v_n, v_1).
- Assume there is at most one edge between two vertices for each direction: $2\binom{n}{2}$. Any complete directed graph with vertices in both directions will do.

(c) Adjacency list is better. In `Dijkstra`, we need to iterate through all edges coming out of u whenever we pop a vertex u out of the heap. It takes $O(n)$ to iterate through all edges starting from u in adjacency matrix representation, and it takes $O(\deg u)$ in adjacency list representation.

(d) Make only one call of the first recursive call $\texttt{FFT}(a_0, a_1, a_2, \dots)$ and then use that result as the result for both recursive calls: *i.e.* the `FFT` evaluation at odd indices and even indices.

(e) The statement is true. In fact, **every** MST must contain an M-path from s to t.

To see this, take some MST $\mathcal{T}$. Since $\mathcal{T}$ is connected, it must have *some* path P from s to t; in fact, it has a unique path from s to t, since $\mathcal{T}$ is acyclic. Suppose for the sake of contradiction that P is not an M-path. Then there exists some edge e on P with $w(e) > M$. Removing e from $\mathcal{T}$ partitions the vertices into two sets S, T with $s \in S$ and $t \in T$. Now, we were promised that there exists an M-path from s to t in G; call that M-path P'. Then since P' starts in S and ends in T, it must have some edge e' crossing the S, T cut. But then removing e from $\mathcal{T}$ and replacing it with e' gives a cheaper spanning tree, contradicting $\mathcal{T}$ being of minimum weight.

Alternative Proof (this only proves there *exists* an M-path from s to t): Run Kruskal or Prim to generate a MST. During the MST's generation, s and t must be connected via a path in the generated MST before the MST adds a edge heavier than M. This is because

- Kruskal sorts the edges so every edge with weight at most M must be already considered.
- Prim always adds the lightest edge attached to the current MST. Since there exists an M-path from s to t in G, every edge in this path must be already considered before a edge heavier than M is added.

Therefore the MST generated by Kruskal or Prim satisfies the requirement.

2 Whole Lotta Love for Chicken Wings

(20 pts) Kevin is placing L-shaped chicken wings on a n by n oven tray, where n **is a power of 2**. Each chicken wing can be viewed as a 2 by 2 square with one of its 1 by 1 squares missing. One of the oven tray's squares was severely burnt and must be left empty.

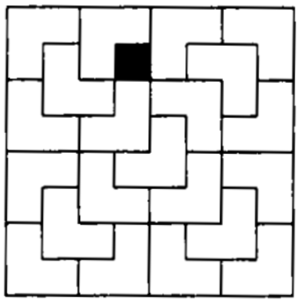
Given n, x, y (n is a power of 2, $n \geq 2$, $x, y \in \{1 \ldots n\}$), where (x, y) is the position of the burnt square that he wants to leave empty, how do we fill the entire n by n tray excluding square (x, y) with chicken wings? The chicken wings cannot overlap each other, nor can they extend out of the tray (or they'll get burnt!).

Describe an efficient algorithm that outputs a possible placement of the chicken wings, given n and (x, y). Then prove the algorithm's correctness and analyze its runtime.

Feel free to draw figures in your answer if it helps you clarify your description. Marking and deleting the placement of one chicken wing takes constant time.
Example:

$$n = 8; (x, y) = (4, 2)$$

One possible output. We denote the burnt square by shading it black.



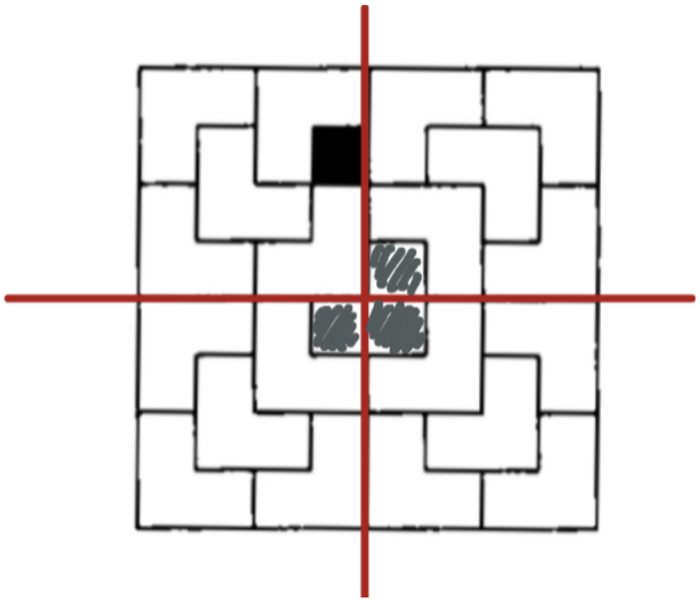
Hint: Divide and conquer. Draw a 8x8 or 16x16 grid to try something out first.

Method 1

Main Idea: First, note that the base case when $n = 2$ is trivial: just place a chicken wing onto the available three squares.

Divide into four quadrants. We will use Divide & Conquer to solve the problem for each quadrant and then merge the solutions of each subproblem to solve the initial problem. In order to call our solution recursively for each of the four quadrants, we need to identify what will solve as a burnt square for each of the quadrant since our algorithm accepts as input a grid with a burnt square.

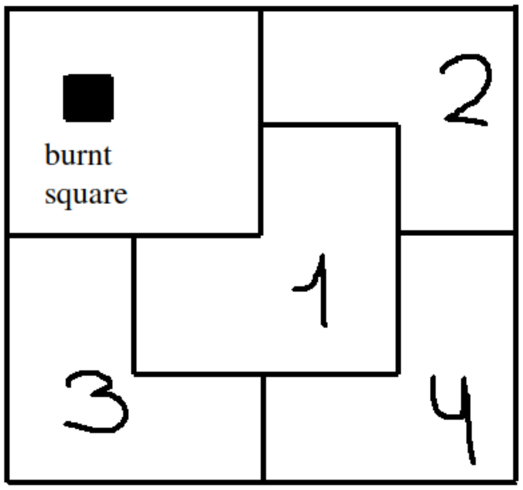
Take the quadrant that contains (x, y) and let the same burnt square be the burnt square for that quadrant. For the other 3 quadrants, mark the corner that was at the middle of the original tray as the burnt square for that sub problem. Then, call the algorithm recursively for each of the quadrants. Using the "recursive leap of faith", we can now say that each quadrant has a valid tiling with chicken wings. Note that for the 3 quadrants, the burnt squares make place for exactly one chicken wing. We will place one chicken wing there and get a valid tiling for the initial problem. See below for visualization:



Proof of correctness: Proof by induction.
Base case: For n = 1, where the tray is 2 by 2, place a wing to avoid the burnt square.
Inductive step:
Assuming the algorithm holds for n = k, we want to prove that it holds for n = k + 1. Given a 2^{k+1} by 2^{k+1} tray, split it into 4 trays each of size 2^k by 2^k. Each problem can be solved by induction hypothesis. Piecing together the 4 sub trays, there are 3 unburnt squares in the center to place a wing.

Time complexity: $T(n) = 4T(\frac{n}{2}) + O(1)$
Using master's theorem, this solves to $O(n^2)$.

Method 2



Main Idea: Split the grid into two shapes: a $\frac{n}{2}$ by $\frac{n}{2}$ subsquare with the burnt tile and a remaining L-shape with no burnt squares. Now, we can treat this as two separate problems: tile the L-shape and tile the subsquare with the burnt square.

To tile the subsquare, we can call the same algorithm recursively for the small subsquare. Base case is a 2 by 2 square where we just put a chicken wing onto the three free slots.

To tile the L-shape, note that we can break it up into four L-shapes with the length of the side of each L-shape equal to one half of the length of the side of the original L-shape (see the image). Then, we call the algorithm recursively on each L-shape. The base case is an L-shape with the side length equal to 1 which is basically one chicken wing.

Note that here we are solving the initial problem by solving **two separate problems**: the initial problem for a smaller subsquare and a new problem of tiling an L-shape, and the problem of tiling an L-shape requires a separate D&C reasoning.

Many students did not specify that we can divide the L shaped area into equal sized L shaped areas, so they did not full credit for their algorithm.

Proof of correctness: Proof by induction.
First, we will prove by induction tiling of the L-shape region. *Base case*: For n = 1, where the L-shape region has side length 1, it is just one chicken wing.
Inductive step: Say that the algorithm holds for $n = k$. Then, for $n = 2k$, we can split up the L-shape as shown on the image and use the tiling for smaller L-shapes, using the induction hypothesis $n = k$. Then, collect those tiling to get the tiling for the bigger L-shape.

Now, we will prove by induction that the whole algorithm works.
Base case: For n = 2, just fill the three empty squares with one chicken wing.
Inductive step: Assume that there is a valid tiling for $n = k$. Now, for $n = 2k$, we can break up the square into a subsquare with the burnt tile of size k and into an L-shape. We already showed that there is a valid tiling for an L-shape, and we will solve the problem for the subsquare using the induction hypothesis.

Time complexity: There are two recurrence relations associated with this problem: $T(n)$ defines the runtime of tiling the entire board, while $S(n)$ is the runtime of the tiling the L-shaped portions.

$$T(n) = T(\frac{n}{2}) + S(n)$$
$$S(n) = 4S(\frac{n}{2})$$

Note that by master's theorem, $S(n)$ resolves to $O(n^2)$. Then,

$$T(n) = T(\frac{n}{2}) + O(n^2).$$

By master's theorem, $T(n)$ is equal to $O(n^2)$.

Comment:
Many students argued in the following fashion: "We will split up the initial square into four subsquares, then continue this splitting until we reach squares of length 2 by 2. Then, we will merge these 2 by 2 squares in this way: ...., and then we will merge the 4 by 4 squares so and so...". Oftentimes, such solutions had the right idea at heart: to introduce those empty squares in the middle to place a wing there (method 1) or to split it into a subsquare with a burnt square and into an L-shape (method 2). However, for such arguments to be formal, they must be phrased in a D&C fashion, like they are phrased in this solution. They must include a "recursive leap of faith" e.g. we split the initial problem into four subproblems, **introduce a "fake" burnt square for three of the subproblems** and then recurse, and also they must include a base case: tiling of a 2 by 2 square. To understand why we insist on a D&C reasoning, try coding up your solution. Most likely, you will quickly realize, that the only adequate way to do it would be to use recursion, which is at heart the D&C approach.

3 When the Levee Breaks

(18 pts) A levee broke and the water is flooding a group of villages[1]. There are n villages and m roads between the villages. The water floods one road at a time. For each road e_i, you're given the unique integer timestep t_i of when this road is flooded. $(t_i)_{i=1}^{m}$ is a permutation of $\{1 \dots m\}$. Give an efficient algorithm that computes the array A, where A_i denotes the number of connected components of villages[2] at timestep $i \in \{1 \dots m\}$ and then analyze its runtime. **Proof of correctness is not needed for this question.**
Example:

$$n = 4, m = 4.$$
$$\text{The roads } (e_i)_{i=1}^{4} = (1,2);\ (2,3);\ (3,4);\ (4,1)$$
$$\text{Timesteps } (t_i)_{i=1}^{4} = 3;\ 2;\ 4;\ 1$$

At timestep 1, $e_4 = (1,4)$ is flooded, there's 1 connected component: $\{1,2,3,4\}$.
At timestep 2, $e_4 = (1,4)$ and $e_2 = (2,3)$ are flooded, there are 2 connected components: $\{1,2\},\{3,4\}$.
At timestep 3, e_4, e_2, and e_1 are flooded, there are 3 connected components: $\{1\},\{2\},\{3,4\}$.
At timestep 4, all roads are flooded, there are 4 connected components: $\{1\},\{2\},\{3\},\{4\}$.
Therefore $A = (1,2,3,4)$

Solution using Union find:

First, we sort the edges by the timestamp they're flooded. This can be done by building a list T where T_k represents the edge e_i flooded at time $t_i = k$. To build this list, process through all the edges once and assign e_i to T_{t_i}.

Now, consider flowing time backwards: we start with all roads/edges flooded and iterate through T backwards. Reconstruct an edge e_i at time t_i. At any timestamp, if the two vertices e_i connects are already in the same connected component, it means that reconstructing that road does not decrease the number of connected components (in our time machine mode). Equivalently, this means that in the usual time flow, this road being destroyed does not create new connected components. Otherwise, if the two vertices e_i connects are not yet in the same connected component, call `union` on the vertices to connect them, and decrease the number of connected components by 1 in our time machine mode. Equivalently, this means that in the usual time flow, this road being destroyed creates one extra component. Do this for every T_i in reverse order until all roads have been reconstructed.

```
for all $i \in \{1 \dots m\}$ do
    $T_{t_i} \leftarrow e_i$
end for
$C \leftarrow n$
for $i \leftarrow m$ to 1 do
    $(u, v) \leftarrow T_i$
    $A_i \leftarrow C$
    if find$(u) \neq$ find$(v)$ then
        union$(u, v)$
        $C \leftarrow C - 1$
    end if
end for
return $A$
```

Solution using Kruskal's:

Note that the above algorithm looks like Kruskal's algorithm run with the edge weights decreasing in t_i (e.g. $w(e_i) = -t_i$ or $1/t_i$), but instead of adding edges to an MST when they wouldn't create a cycle we decrement a counter. This means we could equivalently solve the MST problem with these edge weights, iterate through them in reverse order of flooding, and decrement the counter when we find encounter an edge in the MST. Another way to understand this algorithm is using the cut property. A graph will be split

[1] All villagers have evacuated so there are no casualties.
[2] A group of villages in which any two villages are connected to each other by unflooded roads, and which is connected to no additional villages in the rest of the n total villages.

along a cut when the highest-flooding-time edge across that cut floods. So we can find out when all cuts happen by finding the highest-flooding-time edges across each cut in the graph, which corresponds to the elements of an MST with edge weights decreasing in t_i.

Time complexity: Sorting is linear here because (t_i) is a permutation of $\{1, \cdots, m\}$. For Kruskal's, the runtime is $O(m \log^* n)$. For the union find solution, we do $O(m)$ path-compressed union find operations, which together cost $O(m \log^* n)$ as well.

4 String matching with wildcards

We are given two strings P, T over the alphabet $\Sigma = \{0, 1, \dots, s-1\}$. P has length m and T has length n, where $m \le n$. We would like to find all occurrences of the pattern P in T. For example, the pattern $P = 101$ appears twice in the string $T = 10101$: namely at positions 0 and 2 in T (using 0-based indexing). In what follows, you only need to provide a short algorithm without proof.
Note: The last two parts of this problem are difficult. We assigned fewer points to the last two parts to encourage you to check your answers for other problems instead if you finished everything else on the exam but stuck on (d) and (e).

(a) (6 pts) When $s = 2$, give an $O(n \log n)$ time algorithm for this problem. **Hint:** what if you replace each 1 in an input string with 01 and replace 0 with 10?

(b) (5 pts) What if we only want to find positions in T that patch P up to at most 10% error?

(c) (7 pts) Show how to improve the runtime of (a) to $O(n \log m)$. **Hint:** solve $O(n/m)$ instances of a problem solved in class.

(d) (4 pts) Give an $O(n \log n)$ time (no dependence on s) algorithm for exact matching over general alphabets, i.e. where s is not necessarily 2. **Hint:** $a = b$ iff $(a-b)^2 = a^2 + b^2 - 2ab = 0$.
Note: If your answer for this part is entirely correct you automatically get full points for part (a). If you wrote something like "use my algorithm described in (d)" in (a), you'll get 0 points for (a) unless you obtain full score in (d).

(e) (4 pts) Give an $O(n \log n)$ time algorithm for exact matching (i.e. not 10% error) even if the strings are allowed to have wildcard characters (a wildcard matches any single character).

(a) Replace each 1 in an input string with two elements $(0, 1)$ and replace 0 with $(1, 0)$; then do cross-correlation between the resulting vectors; we have a match at position i whenever the cross-correlation output at $2i$ is equal to m.

(b) Same as the previous part, except having at most 10% error means the cross-correlation is at least $9m/10$ (instead of being exactly m).

(c) Create the strings $T_0 = T[0..(2m-1)]$, $T_1 = T[m..(3m-1)]$, $T[2m..(4m-1)]$, etc. (overlapping parts of length $2m$). Now run part (a) between P and T_j for each j; each such cross-correlation takes $O(m \log m)$ time and gives us potential matches in m locations.

(d) String P matches $T[j..(j+m-1)]$ iff $\sum_{i=0}^{m-1} (P_i - T_{j+i})^2 = 0$. We expand the square so that this equals $\sum_{i=0}^{m-1} P_i^2 + \sum_{i=j}^{j+m-1} T_j^2 + \sum_{i=0}^{m-1} P_i T_{j+i}$. The first sum is a constant we can calculate in $O(m)$ time. If we define the prefix sum σ_k as $\sum_{i=0}^{k-1} T_i^2$ then we can obtain all prefix sums and store them in an array in $O(n)$ time, then we can calculate the second sum as $\sigma_{j+m} - \sigma_j$. Lastly, obtaining all the third category of sums is simply cross-correlation, which takes $O(n \log n)$ time.

Alternative Solution

Similar to part (a), we will replace a single character in an input string with a tuple in the vector to be cross-correlated; however, we will use a different mapping for the two different strings. Whenever the first string has a at position i, the first vector should have $(a^2, 1, 2a)$ at position $3i, 3i+1, 3i+2$, whenever the second string has b at position i, the second vector should have $(1, b^2, -b)$ at position $3i, 3i+1, 3i+2$. This way, element $3k$ of the output will be $\sum_{i=0}^{m-1} (P_i^2 + T_{i+k}^2 - 2P_i T_{i+k})$, which is zero if and only if $P_i = T_{i+k}$ for all i. These can be read off as every third element of the output vector.

(e) The solution is similar to the previous part, but where we map symbols to the alphabet to be one more (i.e. map 0 to 1, 1 to 2, ..., and finally $s-1$ to s). We then use 0 to denote the wildcard character. Then character a matches b iff $ab(a-b)^2 = 0$. We can then similarly as the previous part expand the square and obtain matches using prefix sums and cross-correlation.

Alternative Solution

If we use the alternative solution from part (d), then instead of adding another number to denote a wildcard, we can simply set the tuple in the output to $(0,0,0)$ whenever the input is a wildcard. This will contribute zero to the output sum regardless of what the corresponding tuple in the other vector is.