1. Recursive time bounds: give a big-O bound for $T(n)$ given each of the following recursive formulas:

(a) $T(n) = 3T(n/4) + n \log n$

Proof. By the master theorem, since $n \log n = \Omega(n^{\log_4 3})$ and $3(n/4) \log(n/4) < 0.9n \log n$ for sufficiently large $n$, $T(n) = \Theta(n \log n)$.

(b) $T(n) = 2T(n/2) + \sqrt{n}$

Proof. By the master theorem, since $\frac{1}{2} < \log_2 2 = 1$, $T(n) = \Theta(n)$.

(c) $T(n) = 5T(n/4) + n$

Proof. By the master theorem, since $1 < \log_4 5$, $T(n) = \Theta(n^{\log_4 5})$.

(d) $T(n) = T(2n/3) + T(n/3) + n/6$

Proof. Construct the recursion tree. Every root-to-leaf path has length between $\log_3 n$ and $\log_3/2 n$, or $\Theta(\log n)$. Let $S_i$ be the sum of the sizes of all nodes in the recursion tree at a given level $i$. We have that $S_i = n$ for $i \leq \log_3 n$, since the recursion does not change the total number of elements until some paths end, and for all $i$ we have $S_i \leq n$. Hence the total work spent is

$$\sum_i S_i/6 \geq (\log_3 n)n/6$$

and

$$\sum_i S_i/6 \leq (\log_3/2 n)n/6$$

for $T(n) = \Theta(n \log n)$.

with the base case $T(1) = 1$.

2. In class we discussed interval packing problems. Here we explore interval cover problems.

(a) You are given a set of $n$ intervals $[s_i, f_i)$ and a range $[0, T)$. You would like to find a minimal set $I \subset [n]$ of intervals whose union covers the range. That is, we say that $I$ is a valid cover if, for all $t \in [0, T)$, there exists an $i \in I$ such that $t \in [s_i, f_i)$. Give a greedy algorithm to compute a valid cover with the smallest number of intervals, in linear time after sorting.
Proof. We sort the intervals by finish time. Starting with \( S = \{ \} \) and \( t = 0 \), we repeatedly:

i. Find the last index \( i \) with \( s_i \leq t \).

ii. Set \( S = S \cup \{ i \} \).

iii. Set \( t = f_i \).

until \( t \geq T \), and output \( S \).

To compute the first step quickly, we precompute for each \( i \) the value \( s_i = \min_{j \geq i} s_j \). This can be computed in \( O(n) \) time by iterating from the end, with \( s_i = \min(s_i, s_{i+1}) \). Then, because the indices \( i \) chosen by the algorithm will increase in each round, the first step can be done in \( O(n) \) time over all rounds (we increment \( i \) to the maximum value such that \( s_i \leq t \)). Since the other steps are simple, the running time is \( O(n) \) after sorting.

For correctness, we observe the following lemma:

**Lemma 1.** Suppose after \( k \) rounds the algorithm has reached state \((S, t)\), so \( k = |S| \). Then:

i. \( S \) is a valid cover of \([0, t)\).

ii. No set \( S' \subset [n] \) of size \( k \) is a valid cover of \([0, t')\) for any \( t' > t \).

**Proof of Lemma 1.** We prove this by induction on \( k \). In the base case, \( k = 0 \), we have \( t = 0 \) and the result is trivial.

For the inductive hypothesis, suppose the result is true for \( k \), and let this state be \((S, t)\). In the next round, the algorithm will pick an \( i \) with \( s_i \leq t \) and update the state to \((S \cup \{ i \}, f_i)\). We would like to show that this satisfies the lemma.

That \( S \cup \{ i \} \) is a valid cover of \([0, t)\) is clear, since \( S \) covers \([0, t)\) and \( i \) covers \([s_i, f_i) \supseteq [t, f_i) \). But what about the second part?

If the second part were not true, there would exist a size \( k + 1 \) set \( S' \) that covers \([0, t')\) for \( t' > f_i \). Let \( i' \) be the largest index in \( S' \), so \( f_{i'} \geq t' > f_i \). Therefore \( i' > i \).

Since \( i \) is the last index with \( s_i \leq t \), this means \( s_{i'} > t \). But since \( S' \) covers \([0, t')\), \( S' \setminus \{ i' \} \) must cover \([0, t') \setminus [s_{i'}, f_{i'}) = [0, s_{i'}) \). This means \( S' \setminus \{ i' \} \) is a size-\( k \) set covering a strict superset of \([0, t)\), which violates the second part of the inductive hypothesis, a contradiction.

Hence both parts of the lemma are true in round \( k + 1 \), so the induction holds, giving the lemma.

Therefore, if the algorithm outputs a set \( S \) of size \( k \), we have that \( S \) is a valid cover of \([0, T)\) and (from the previous round) no size-\( k - 1 \) set can cover \([0, T)\), giving correctness.

(b) Now suppose that each interval also has a cost \( c_i \), and your goal is to find a valid cover \( I \) minimizing the total cost \( \sum_{i \in I} c_i \). Give a dynamic programming solution to this problem that takes \( O(n^2) \) time. [Extra credit: \( O(n \log n) \) time.]
Proof. For the $O(n^2)$ solution: Create a start node $s$, a finish node $t$, and a node for each interval $i$. Draw an edge from $s$ to $i$ if $s_i \leq 0$, from $i$ to $t$ if $f_i \geq T$, and from $i$ to $j$ if $f_i \geq s_j$. Make the weight of an edge be the cost of its left endpoint (or zero, if that is $s$). The answer is the minimum cost $s$-$t$ path on this DAG, which we can compute in $O(\# \text{ edges}) = O(n^2)$ time. We omit the proof, in favor of the $O(n \log n)$ solution.

For the $O(n \log n)$ solution: sort the intervals by finish time. We will build two arrays, $C_i$ and $F_i$, as we scan through the intervals in order, that will be sorted in increasing order.

1: $C = [0], F = [0]$
2: for $k = 1, \ldots, n$ do
3: Binary search on $F$ to find the minimum $j$ such that $F_j \geq s_k$.
4: Define $c' = C_j + c_k$
5: While the last element of $C$ is bigger than $c'$, pop the last element of both $C$ and $F$.
6: Append $c'$ to $C$ and $f_k$ to $F$.
7: end for
8: Output $C_j$ for the minimum $j$ such that $F_j \geq T$.

The running time is dominated by $n$ binary searches, for $O(n \log n)$ time. (Any given round $k$ may spend a lot of time popping elements, but since only $n$ elements are inserted, only $n$ elements are popped total.)

We claim the following invariants:

Lemma 2. After any round $k$, the arrays $C$ and $F$ have the following properties:

- $C$ and $t$ have the same length, and are both monotonically increasing.
- $F_0 = 0$ and the last entry of $F$ is $f_k$.
- For any $t$ and $\ell$ with $F_{\ell-1} < t \leq F_\ell$, the minimum cost cover of $[0, t)$ using a subset of the first $k$ intervals is $C_\ell$.

The first two properties are simple, and correctness follows from the third property, so we focus on proving that.

Proof of third property. We prove this by induction on $k$. The base case of $k = 0$ is trivial. Now, suppose the lemma is true for $k - 1$. This means that the $j$ computed by the algorithm in round $k$ has $C_j$ being the minimum cost cover of $[0, s_k)$ using a subset of the first $k - 1$ intervals. For any $t$, the minimal cover of $[0, t)$ that uses only the first $k$ intervals and includes interval $k$ then has cost $c' = C_j + c_k$. Hence the minimal cover of $[0, t)$ using only the first $k$ intervals is the minimum of $c'$—the result of including interval $k$—and what that cost was in the previous round—the result of not using interval $k$.

Since the $C$ are sorted, popping the values larger than $c'$ makes $(C, F)$ represent this desired minimum.
3. Suppose that you have \( n \) jobs that you would like to schedule. Each job takes a different duration of time \( d_i > 0 \) to complete, and a different “urgency” \( u_i > 0 \). You can only work on one job at a time, but you can choose an arbitrary order among the jobs.

For a given order of the jobs, let \( t_i \) be the time that you finish job \( i \), which is the sum of the durations of the previous jobs and this one. Your total cost of a given order is defined as \( \sum_{i=1}^{n} u_i t_i \): the more urgent a job is, the more important it is that it be finished earlier. Your goal is to find the job order that minimizes the cost.

(a) (No response necessary) Think about this problem on your own for 10 minutes before reading the spoilers below.

(b) Suppose that \( n = 2 \). What order should you take?

Proof. The costs are either \( u_1 d_1 + u_2 (d_1 + d_2) \) or \( u_1 (d_1 + d_2) + u_2 d_2 \). Subtracting off the common terms and dividing by \( d_1 d_2 \), we find that we start with the first element if \( u_2 / d_2 < u_1 / d_1 \) and the second element if the reverse.

(c) Consider any ordering among the jobs for general \( n \), and look at any pair of adjacent jobs in that ordering. How would the total cost change if you swap the ordering?

Proof. As in the two-variable case, the total cost of moving \( i \) from right after \( j \) to right before \( j \) will change the cost by an additive

\[
u_j d_i - u_i d_j.
\]

(d) Observe that repeatedly applying the idea in the previous part would lead to an \( O(n^2) \) time bubble sort of the jobs, based on some function \( f(u_i, d_i) \) of each element.

Proof. By dividing the above by \( d_i d_j \), we see that moving \( i \) before \( j \) gives an improvement if and only if \( u_i / d_i \) is bigger than \( u_j / d_j \). Hence each step of bubble sorting by \( u_i / d_i \) decreasing will improve the total cost, leading to a local optimum. But is this a global optimum? Yes: any non-sorted order has some pair of adjacent elements that are inverted, and swapping them would give an improvement. Hence no non-sorted order is a local optimum, and the only optimum is the sorted order.

(e) Give an \( O(n \log n) \) time algorithm for the problem.

Proof. Sort by \( u_i / d_i \) decreasing, using a faster sorting algorithm (e.g., merge sort).

4. There’s a Jupyter Notebook linked from the class webpage. Run through it, then answer the questions at the end. Don’t wait till the last day to do this: setting up the required libraries may take some time.
• Why do they not appear to be straight lines the whole way?

*Answer.* In the case of the matrix algorithms, it seems that there’s an additive constant of about 30 microseconds – the startup cost of running numpy code, perhaps – that dominates when \( n \) is below about ten thousand. In the case of the iterative method, it appears to be a linear term with a large constant factor:

```
This is what one would expect, since the linear term will dominate when the answer fits in a constant number of words.
```

• What is the relation between the iterative and sliding window lines?

*Answer.* They appear to be offset by a fixed constant in the log log plot, suggesting a constant factor difference.

• What do you think the shape of the sliding window line is, and how would it extend for larger \( n \)?

*Answer.* It appears to be a mixture of constant, linear, and quadratic terms. For larger \( n \), the straight line of slope 2 should dominate.

• What is the shape of the gmp curve, and how would it extend for larger \( n \)? Why?

*Answer.* GMP switches between many algorithms, so it’s a mixture of many exponents (2, 1.585, 1.4, and 1). Eventually the FFT-based method will dominate and it will converge to a line of slope 1, but that could take a while.