For each of the algorithm problems, design as asymptotically efficient an algorithm as possible. Give a correctness argument (explanation, if it is relatively simple, or proof if not) and time analysis. You may use any well-known algorithm or data structure, or algorithm from the text or from class, as a sub-routine without needing to provide details.

**Merging lists** You want to collect sorted lists from different databases and merge them into a single sorted list. The cost to merge a list of size $l_1$ with one of size $l_2$ is $l_1 + l_2$, and creates a new list of size $l_1 + l_2$ replacing the old ones. You are given as input the sizes of $n$ lists $l_1, \ldots, l_n$ and need to schedule merges in order to unite them into a single list. You wish to minimize the total cost for all merges. Give an efficient algorithm that, given $l_1, \ldots, l_n$, finds the lowest cost schedule to merge lists of those sizes. (12 pts. correct poly-time alg. and correctness proof, 8 pts. efficiency)

Our greedy strategy will be, until only one list remains, to find the two smallest lists, record a merge of the two in the solution, and replace the two items with an item of the sum of the sizes representing the merger of the lists.

Using a min-heap, we can implement this strategy in $O(n \log n)$ time, by placing all items in the heap, then finding and deleting the minimum twice, storing the answers, and inserting a new item whose key is the sum. Since we repeat $n - 1$ times, each time reducing the number of lists by 1, and the three heap operations all take time $O(\log n)$, this gives total time $O(n \log n)$.

For any schedule of list mergings, we can represent the schedule as a binary tree, with the parent of a list (either original or created by merges) being the list we obtain the first time it is merged. The original lists will be leaves of this tree, and the size of an interior node will be the sum of the sizes of the leaf nodes in the sub-tree rooted at that node. Since the cost to create a list is its size, the total cost will be the sum of the sizes of all the interior nodes. Rearranging terms, each leaf will contribute its depth to the total cost.

We prove the greedy strategy is optimal by induction on $n$ the number of input lists, using a change method proof. For $n = 1$, there are no merges to be done, so the null schedule is optimal. Assume the greedy strategy is optimal for any $n - 1$ lists, where $n - 1 \geq 1$. Assume without loss of generality that list sizes are sorted from smallest to largest. The greedy strategy merges $l_1$ and $l_2$ as its first step. Let $T$ be the tree representing
any other schedule of merges for these lists. We claim that there is a tree
$T'$ of cost at most that of $T$ that merges $l_1$ and $l_2$. If $T$ already does
so, we let $T' = T$. Otherwise, let $u$ be a leaf in $T$ of maximal depth $d$.
Then its sibling $v$ is also a leaf, since otherwise any leaf in its sibling has
greater depth. So $u$ and $v$ are original lists. Let $T'$ be the tree with
the same structure as $T$, but where we swap $l_1$ with the smaller of $u, v$ and
$l_2$ with the larger of $u, v$. Let $l_1$ have depth $d_1$ in $T$, and $l_2$ depth $d_2$. We
know $d_1, d_2 \leq d$, and we know $l_1 \leq \min(u, v)$ and $l_2 \leq \max(u, v)$. (Note
that either $u$ or $v$ could be equal to $l_1$ or $l_2$, but this just means the swap
might leave them in place). Then the change in cost between $T$ and $T'$ is
$$(d - d_1)(l_1) + (d - d_2)(l_2) + (d_1 - d)(\min(u, v)) + (d_2 - d)(\max(u, v)) = (d - d_1)(l_1 - \min(u, v)) + (d - d_2)(l_2 - \max(u, v)) \leq 0.$$ So $T'$ is no more
expensive in total cost than $T$.

Now, we can list the merges in $T'$ as merge $l_1, l_2$, then perform some
schedule of merges on $l_3, \ldots, l_n$, $l_1 + l_2$. The greedy schedule is of the same
form, merge $l_1$ and $l_2$ and then perform the greedy algorithm on $l_3, \ldots, l_n$, $l_1 + l_2$. Since the greedy algorithm is optimal for this set of $n - 1$ lists (from the
induction hypothesis), the total cost of the remaining schedule is no more
than that of $T'$, and so the total cost for the complete greedy schedule is
no more than the total cost of $T'$ (since both add $l_1 + l_2$ to the remaining
costs.) Thus, the total costs for the greedy algorithm are no more than
$T'$, and that is no more than that of $T$. Since $T$ was an arbitrary schedule,
that means the greedy algorithm’s total costs are the minimum possible
for any schedule.

**Homework grade maximization** In a class, there are $n$ assignments. You
have $H$ hours to spend on all assignments, and you cannot divide an
hour between assignments, but must spend each hour entirely on a single
assignment. The $I$’th hour you spend on assignment $J$ will improve your
grade on assignment $J$ by $B[I, J]$, where for each $J$, $B[1, J] \geq B[2, J] \geq
\ldots \geq B[H, J] \geq 0$. In other words, if you spend $h$ hours on assignment
$J$, your grade will be $\sum_{i=h}^{\infty} B[i, J]$ and time spent on each project has
diminishing returns, the next hour being worth less than the previous one.
You want to divide your $H$ hours between the assignments to maximize
your total grade on all the assignments.

Give an efficient algorithm for this problem. (6 pts. correct poly-time
algorithm, 6 points correctness proof, 8 points efficiency. My best time is
$O(n + H \log n)$.)

A greedy algorithm for this problem is to initialize time for each assignment
to 0. For each of $H$ hours, we do the following: Say that we are currently
reserving at least $T_I$ hours for project $I$. Then we find the largest $B[I, T_I +
1]$, the project where spending an additional hour increases our grade the
most. We increment $T_I$, reserving one more hour for that assignment.
We prove the correctness of the strategy by the following “Change” argument. We’ll show that, after we’ve allocated any number \( h \) hours, there is an optimal distribution of time \( t_I \) with \( \sum t_I = H \) where each \( t_I \geq T_I \).

Initially, all \( T_I \) are 0, so any optimal distribution can be used for \( t_i \).

Say that we have such a distribution \( t'_I \) before we allocate the \( h + 1 \)st hour. Let \( J \) be the project we allocate that hour to. \( t'_J \geq T_J \) before we increment it. If \( t'_J \geq T_J + 1 \), the incremented value, we let \( t = t' \). otherwise, \( t'_J = T_J \), and we obtain \( t \) by incrementing \( t'_J \) to \( T_J + 1 \). Then we pick some project \( K \) where \( t'_K > T_K \) which must exist since the sum of the \( t'_I \) is \( H \) and the sum of the \( T_I \) is \( h < H \). We let \( t_K = t'_K - 1 \), so that we still total \( H \) hours. Between \( t \) and \( t' \), the grade on assignment \( K \) has dropped by \( B[K, t'_K] \) and the grade on assignment \( J \) has increased by \( B[J, T_J + 1] \). Now, by monotonicity, and the fact that \( t'_K \geq T_K + 1 \), we know \( B[K, t'_K] \leq B[K, T_K + 1] \leq B[J, T_J + 1] \) because \( J \) was the assignment with maximum immediate gain by incrementing \( T \).

Thus, our total grades in \( t \) are at least those in \( t' \), and since \( t' \) achieved maximum total grades, so does \( t \).

By induction, we then get such \( t \) after any number of hours has been allocated. After we allocate all \( H \) hours, since each \( t_I \geq T_I \), and \( \sum t_I = \sum T_I = H \), we must have each \( t_I = T_I \), so at the end \( t = T \). So the final output of the greedy algorithm is an optimal solution.

To implement the algorithm efficiently, we use a max heap of pairs \( I, B[I, T_I + 1] \) keyed by the second value. We also keep the \( T_I \) in an array indexed by \( I \). We initialize the array to all 0’s, and build a heap with all the values \((I, B[I, 1])\). We can use the divide and conquer Heapify algorithm from e.g., CLRS Section 6.3, to do this in linear time, \( O(n) \). We then, for \( H \) iterations, find and delete the largest pair \((I, B[I, T_I + 1])\), increment \( T_I \), and insert the replacement \((I, B[I, T_I + 1])\) (for the new value of \( T_I \)).

Thus, we need to find and delete the minimum and insert a new element \( H \) times, for total time \( O(H \log n) \) (The heap stays size \( n \), since we delete and insert one element each iteration).

This gives total time \( O(n + H \log n) \).

**Preemptive scheduling** Consider the following preemptive scheduling problem. You are trying to schedule jobs on a machine that are arriving at different times, and require different numbers of steps to finish. Your schedule can be preemptive, in that you can start one job, then switch to another, then finish the first job. You are trying to minimize the sum over all jobs of the time they finish.

More precisely, the input is a sequence of \( n \) jobs, \( Job_i = (a_i, d_i) \), where \( a_i \) is an integer giving the arrival time of the job (first time step when we could start the job), and \( d_i \) is a positive integer giving the duration of the
job, the number of steps required to finish the job. A schedule specifies for each time step, which job we are working on. At time step, \( t \), we can only work on \( Job_j \) if \( a_j \leq t \); and there must be at least \( d_j \) steps where we are working on \( Job_j \). The finish time for \( Job_j \) is the last time when \( Job_j \) is scheduled. The objective is to find a schedule that minimizes the sum of all the finish times.

Example: Job 1: Arrives at 8 AM: Practice piano. Duration: 3 hours.
Job 2: Arrives at 9 AM. Answer morning email. Duration 1 hour.
Job 3: Arrives at 11 AM. Do CSE homework. Duration 4 hours.
Finish times: email: 10; piano: 12; homework: 16. Total: 38.

Give an efficient greedy algorithm for this problem. (4 points correct algorithm, 10 points correctness proof, 6 points efficiency)

Our algorithm will use the following strategy: At each time slot, we will schedule the available, incomplete job with the smallest remaining duration (if any such jobs exist).

We will use the exchange method to prove this strategy correct. We will show by induction on \( t \) that there is an optimal schedule \( S_t \) that schedules the first \( t \) time units the same as the schedule above.

The base case, \( t = 0 \) is trivial. To prove the inductive step, say we have an optimal schedule \( S_t \) that agrees with the first \( t \) time units with the greedy schedule. If there are no jobs available at time \( t + 1 \) in the greedy schedule, then the same is true for \( S_t \), so both \( S_t \) and the greedy schedule leave step \( t + 1 \) idle, and we let \( S_{t+1} = S_t \). Similarly if both the greedy schedule and \( S_t \) schedule the same job on \( S_t \). If \( S_t \) is idle at time \( t + 1 \), and the greedy schedule schedules job \( J \), job \( J \) is available and not completed at time \( t + 1 \) in \( S_t \). So we can replace the idle slot with performing \( J \), and the last time \( J \) is performed in \( S_t \), we can let the schedule idle. Then all jobs but \( J \) finish at the same time, and \( J \) finishes earlier, so we have a better schedule \( S_{t+1} \) (a contradiction). If \( S_t \) schedules \( K \) and the greedy algorithm \( J \), the remaining duration of \( J \), \( rd_J \), is less than or equal to the remaining duration of \( K \), \( rd_K \) at time \( t + 1 \). We let \( S_{t+1} \) schedule \( J \) in the first \( rd_J \) time slots that \( S_t \) uses for either \( J \) or \( K \), and \( K \) in the remaining \( rd_K \) such slots. Since neither \( K \) nor \( J \) can finish before the \( rd_J \)th such slot, and in the new schedule \( J \) does finish then, the first of the two jobs to finish is only earlier in \( S_{t+1} \) than \( S_t \). The last job of the two to finish does so at the same time. All other jobs finish at the same time in the two schedules. Thus, the total finish times in \( S_{t+1} \) are at most those in \( S_t \). Since \( S_t \) had minimum possible total finish times, so does \( S_{t+1} \).
Applying this to the final step the greedy algorithm finishes, there is a schedule that is optimal and identical to the greedy strategy. Thus, the greedy strategy is optimal.

To implement this strategy quickly, observe that we only switch from one job to another if our current job finishes or another job arrives (of smaller duration). Thus, there will be at most 2n switches from one job to another. We sort the jobs by arrival time, first to last. Then we maintain a min-heap of jobs that have arrived but not yet completed, and a time counter T, and a pointer to the first job to arrive after T. We look at the minimum remaining duration $rd$ of a job on the heap. If $T + rd$ is less than or equal to the arrival time of the next job, we will complete that job (since its remaining duration only gets smaller, and no other job’s changes). We schedule that job from $T$ to $T + rd$, updating $T$, and delete it from the heap. If the next job arrives first, at some time $T' < T + rd$, we schedule the current minimum from $T$ to $T'$, update $T$ to $T'$, and we subtract $T' - T$ from the remaining duration of the minimum element on the heap, and add in all jobs arriving at $T'$ with their durations.

As mentioned above, there are at most 2n times we update. Over all, each job gets inserted into the heap once, and each insert into a heap takes $O(\log n)$ time, so inserts take $O(n \log n)$ time total. For each of $O(n)$ steps, we might additionally have to delete the minimum of the heap or decrement the minimum, and perform constantly many compares and arithmetic operations. The most expensive of these is to delete the minimum, $O(\log n)$ time, so the total time for all steps is $O(n \log n)$, which is the same as the insert time and the time to sort by arrival time. So the total time is $O(n \log n)$.

Approximation for bin filling In the bin filling problem, you have $n$ items of positive integral sizes $a_1, .., a_n$ and $m < n$ bins, where bin $j$ is of size $B_j$. You need to assign each item $i$ to a bin $A[i]$, in a way to fill the maximum number of bins, where a bin $j$ is full if $\sum_{i|A[i]=j} a_i \geq B_j$.

Give an efficient approximation algorithm for this problem. Most of the points will be based on your approximation ratio and the proof that it achieves this ratio. The best possible ratio is to fill at least 1/2 as many bins as the optimal solution.

We claim the following simple algorithm has an approximation ratio of 2, i.e., fills up at least 1/2 the bins as the optimal allotment:

Sort the bins from smallest to largest.

In any order, put each item in the smallest unfilled bin.

If there are $m$ bins and $n$ items, this takes time $O(m \log m + n)$.

To prove that it is within a factor of 2, we will use the obstacle method.
One obvious obstacle to any allotment is that the total size of filled bins must be larger than the total sizes of the items. More precisely, say the bins have sizes \( B_1, \ldots, B_k, \ldots, B_m \) and that they are sorted from smallest to largest. For a set of items \( A \), let \( TS(A) = \sum_{a \in A} a.\text{size} \) be the total of all sizes of items in \( A \). Then we’ll use \( SB(A) \) to denote the maximum number of bins that total to at most \( TS(A) \), i.e., \( SB(A) = k \), where \( B_1 + \ldots B_k \leq TS(A) < B_1 + \ldots B_k + B_{k+1} \). (If \( TS(A) \geq B_1 + \ldots + B_m \), we set \( SB(A) = m \).) Then the items in \( A \) cannot fill more than \( SB(A) \) bins no matter how they are arranged.

Now, the size bound might not be tight, especially if there are very large items that overfill bins. Since we cannot split items, we can fill at most one bin per item. As usual, we let \( |B| \) represent the cardinality of set \( B \), i.e., number of items regardless of the sizes of the items.

Putting these thoughts together, we get the following lemma: Lemma 1: If the set of items is partitioned into two sets \( A \) and \( B \), the items cannot fill more than \( SB(A) + |B| \) bins in any allotment.

Proof: There are at most \( |B| \) bins with any item from \( B \) in them, since each element of \( B \) is in only one bin.

The remaining filled bins have only elements of \( A \) in them. Thus, the total sizes of all elements in those bins is at most the total size of all elements of \( A \). Thus, the total sizes of all of these bins is at most \( TS(A) \). But then there can be at most \( SB(A) \) of them, since the smallest \( SB(A) + 1 \) bins total to strictly more than \( TS(A) \).

Now we can use our algorithm to find such a partition. Say that our algorithm fills \( k \) bins. Let \( B \) be the set of last items put into each filled bin, and \( A \) be the remaining items. Since when we remove the items from \( B \), we don’t fill any of the first \( k \) bins with items from \( A \), and we didn’t fill the \( k + 1 \)’st bin, \( TS(A) \) is strictly less than \( B_1 + \ldots B_{k+1} \), so \( SB(A) \leq k \). Also, by construction, \( |B| = k \). So the lemma says that any allotment can fill at most \( SB(A) + |B| \leq k + k = 2k \) bins, so no allotment can fill more than twice as many bins as the greedy algorithm fills.

Implementation of Independent Set Heuristic The independent set problem is to find as large as possible a set of nodes of an undirected graph so that the set does not contain both endpoints of any edge. Consider a greedy heuristic for independent set that selects the lowest degree node, puts it in the set, and deletes it and its neighbors and repeats. Implement this greedy heuristic, and test it on random graphs where every pair of nodes has an edge between them independently with probability \( 1/2 \). Try it on graphs with as wide a range of number of nodes as you can get reasonable times for. Plot the size of the independent set found as a function of the graph size. What do you conjecture about how the average size of the independent set grows as a function of \( n = |V| \)?
As \( n \) grows, even the smallest degree node will have about \( n/2 \) neighbors, and similarly recursively. So the independent set found will scale as \( \log_2 n(1 + o(1)) \).

When describing an algorithm, don’t write out an entire pseudo-code; just describe it at a high level. Be sure to specify completely all data structures used in the algorithm. Include correctness proofs and time analysis for all algorithms, except for the implementation problem.