# CS 124 Homework 3: Spring 2021

**Your name:** Sean Ty

**Collaborators:**

**No. of late days used on previous psets:** 0
**No. of late days used after including this pset:** 0

Homework is due Wednesday 2021-03-03 at 11:59pm ET. You are allowed up to **twelve** (college)/**forty** (extension school) late days through the semester, but the number of late days you take on each assignment must be a nonnegative integer at most **two** (college)/**four** (extension school).

Try to make your answers as clear and concise as possible; style will count in your grades. Be sure to read and know the collaboration policy in the course syllabus. Assignments must be submitted in pdf format on Gradescope. If you do assignments by hand, you will need to scan in your results to turn them in.

For all homework problems where you are asked to design or give an algorithm, you must prove the correctness of your algorithm and prove the best upper bound that you can give for the running time. Generally better running times will get better credit; generally exponential time algorithms (unless specifically asked for) will receive no or little credit. You should always write a clear informal description of your algorithm in English. You may also write pseudocode if you feel your informal explanation requires more precision and detail, but keep in mind pseudocode does NOT substitute for an explanation. Answers that consist solely of pseudocode will receive little or not credit. Again, try to make your answers clear and concise.

1. Solve the following two problems using heaps. (No credit if you're not using heaps!)

   (a) **(12 points)** Give an $O(n \log k)$ algorithm to merge $k$ sorted lists with $n$ total elements into one sorted list.

   First note that we can assume WLOG that $k \leq n$, because if not then at least $k - n > 0$ lists are empty and can be ignored. We perform the following algorithm, where we are given $k$ lists $L_1, L_2, \ldots L_k$: construct a minheap of size $k$. The lists $L_i$ are added into the heap, where we say that a list $L_i$ is greater than a list $L_j$ if $L_i[0] > L_j[0]$, and an empty list is considered to be "infinite" (and is thus always at the bottom of the heap). Initialize a result list that is currently empty. Now for $1 \leq i \leq n$, we pop the first element of the topmost list in the minheap and reconstruct the heap by trickling down. The popped element is then appended to the list. Then, we output the result list, which is the merge of the $k$ sorted lists.

   We prove correctness. To do this, note that in each step the minheap contains the smallest elements of each list $L_i$ (or infinity if $L_i$ is empty), since the lists are all sorted. Therefore, when we pop an element, this element must be the smallest element of all the lists. Since in each step we are taking the currently smallest element, it follows that the result list gets the next smallest number in each step. Thus, the result list is exactly the list we are looking for.

   Now we examine the running time. We know from section that constructing the heap takes $O(k \log k)$ time. Popping and heapifying takes $O(\log k)$ time and we do this $n$ times, so the complexity of the for loop is $O(n \log k)$. Appending takes $O(1)$ time and we are doing this $n$ times, equating to $O(n)$ time. So the time complexity is $O(n \log k)$, as desired. The space complexity here is $O(n)$ from the heap and the result list.

   (b) **(12 points)** Say that a list of numbers is $k$-close to sorted if each number in the list is less than $k$ positions from its actual place in the sorted order. (Hence, a list that is 1-close to sorted is actually sorted.) Give an $O(n \log k)$ algorithm for sorting a list of $n$ numbers that is $k$-close to sorted.

   Initialize a result list. We construct a minheap $H$ of size $k$ using the first $k$ elements of the list. Then for each step, we pop the top element of $H$, push it into the result list, and insert the next element of the list into $H$ (if it exists. If not, insert positive infinity). Repeat until all elements are in the result list (e.g. use a for loop). Then, return the result list. This is the desired sorted list.

   We prove correctness. We inductively prove that the $i^{\text{th}}$ element is correctly placed. For the base case $i = 1$, we know this is true since we are given that the smallest number must be at most $k$ away from its "right" position, and must therefore already be in the heap. Now suppose that this is true for $i \geq 1$. We will prove the result for $i + 1$. By the inductive hypothesis, the smallest $i$ numbers are already in their right positions. Then consider position $i + 1$. We know that the smallest element must be in $H$ because we know that the "correct" element must be within $k$ away the current position (which should be included in the $H$ since by definition $H$ contains the next $k$). Therefore, the top (smallest) element in $H$ is the "correct" entry, which is indeed appended into the result list next by the algorithm. This completes the inductive step, and proves correctness.

   Now we examine the running time. Just as before, the minheap construction takes $O(k \log k)$ time. For the for loop, each step takes $O(\log k)$ time since we are removing and adding entries (appending takes $O(1)$, so still $O(\log k)$). Since we are doing this $n$ times, the complexity becomes $O(n \log k)$. Since again we can WLOG $k \leq n$ (does not make sense for something to be more than $n - 1$ away from the correct position if the size is just $n$), it follows that the time complexity is $O(n \log k)$. Space is $O(n)$ from the heap and the result list.

2. **(10 points)** You are given a graph $G = (V, E)$ and a minimum spanning tree $T$. The weight of one of the edges in $E \setminus T$ (that is, not in $T$) is decreased. Give a linear time algorithm that determines whether the MST changes and, if so, returns a new MST.

   Suppose that $e$ is the edge whose weight is changed, and we add $e$ to $T$. This induces a cycle in $T$. We start at any vertex of a cycle and traverse through it to determine the edge with the maximum weight, and remove it. Then, this clearly creates a spanning tree (since it must be connected and there are now no cycles–we can find the cycles by DFS which is linear time). If the removed edge is one of the edges of the original tree $T$, we output true (to signify that the MST has changed) and return the resulting tree. Otherwise, $T$ must be unchanged and we output false.

   Now we prove correctness. Let $S$ be the new MST of the graph after we updated the weight of $e$, and $T'$ the tree we obtain from the algorithm. First note that $T'$ is a spanning tree because it is connected and has $|V| - 1$ edges, also it is acyclic because we broke the only cycle that was created. Now for the sake of contradiction that $T'$ disagrees with $S$ in an edge. Then we have two cases: either $e \in S$ and $e \notin T'$, or there is some other edge $e' \in S$ but $e' \notin T'$. In the first case, it follows that $e$ has weight less than some edge in the cycle it induces in $T$, which contradicts the algorithm. In the second case, we can swap $e'$ with the largest in the induced cycle in $T$ when we add it, which by definition has weight larger than $e'$ (since it is in $S$ which is an MST). Then $T$ could not have been an MST since we could have done the same thing for $T$, which is a contradiction. Therefore, $T'$ is indeed an MST, which proves correctness.

   The run time of this algorithm is $O(|V| + |E|)$ since it just goes through the edges of $T$ plus the added edge and uses DFS, so this is the desired linear time algorithm.

3. **(15 points)** Suppose you are given a weighted graph $G = (V, E)$ and an edge $e \in E$. You are asked whether there exists a minimum spanning tree of $G$ containing $e$. Give a linear time algorithm for the problem. (If it helps, first consider the case where all edge weights are distinct – this will get you most of the credit – and then try the more general case.)

   We use the following lemmas:

   **Lemma 1:** If for some cycle $C$ in $G$ the weight of an edge $e \in C$ is the unique maximum edge weight in the cycle, then $e$ cannot belong to an MST.

   To prove this lemma, we consider an edge $e$ that is the maximum edge weight in some cycle $C$. Let $T$ be an MST and suppose that $e \in T$. Deleting $e$ from $T$ partitions $G$ into 2 spanning trees, and the remainder of the cycle connects these two parts, so there is an edge that connects these two parts that is lighter than $e$. But then this forms an MST with total weight less than $T$'s, contradiction.

   **Lemma 2:** If for all cycles containing an edge $e$ has $e$ not the unique heaviest edge, then $e$ is a part of some MST.

   To prove this, consider an MST $T$. If $e \in T$, we are done. Else, $e \notin T$ and we add $e$ to $T$. This creates a cycle, and we remove the heaviest edge in this cycle to obtain a spanning tree (spanning tree because it is connected and has $|V| - 1$ edges). This has weight strictly less than the previous MST by assumption since we are adding a strictly lighter edge, which is a contradiction. (or we remove an edge that has the same weight as $e$, which makes the new tree still an MST which contains $e$, so the conclusion still follows).

   Back to the problem, let $e$ be the edge in question and suppose it connects two vertices $(u, v)$. Perform DFS on any one of the vertices (say $u$), where we only consider edges that have weights less than the weight of $e$. If this creates another path between $u$ and $v$, then we output false (i.e. $e$ cannot be in an MST). Otherwise, no other path can be constructed, and we output true.

   We prove correctness. Suppose that we find another path. Then there must exist a cycle with $e$ being the unique maximum edge weight, so by lemma 1 it follows that $e$ cannot be in an MST, just as we returned. On the other hand, if we did not find another path, then by definition for all cycles containing $e$ does not have $e$ as the unique heaviest edge, so by lemma 2 it follows that $e$ is a part of some MST, as returned. This proves correctness.

   Now note that DFS takes $O(|V| + |E|)$ time and ignoring the edges as needed takes $O(|E|)$, so the total running time is $O(|V| + |E|)$, as desired.

4. Recall that we ask you to put each problem set solution on its own page. For purposes of this problem, assume that each solution may be on at most one page. You've solved the $n$ problems of a problem set, with values $v_1$, $v_2$, ..., $v_n$, but you only have available a set of $k$ pieces of paper $p_1$, $p_2$, ..., $p_k$ of various shapes and sizes. Each page can fit only some problem solutions: for each $i \in [n]$ and $j \in [k]$, you know either that solution $i$ can fit on page $p_j$ (in which case we say $a_{i,j} = 1$) or not ($a_{i,j} = 0$). (Note that it's possible that solution $i$ fits on page $j$ but not $j'$ and solution $i'$ fits on page $j'$ but not $j$.) You want to submit the set of solutions that maximizes your score.

   (a) **(5 points)** Consider the following "greediest" algorithm: starting with the highest-value problem and working down, assign a solution to an arbitrary empty page that can fit it if there is one, and throw it out if not. Show that the greediest algorithm doesn't always maximize your score. (To show this, you should give an example instance of the problem—an $n$, $k$, set of problem values, and a description of which problems fit on which pages—and show a solution produced by the greedy algorithm and another solution that gives you a better score.)

   Consider three problems $v_1, v_2$ and three pages $p_1, p_2$. Let's say that $a_{ij} = 1$ iff $(i, j) \in \{(1, 1), (2, 1), (2, 2)\}$. Suppose that problem $v_1$ has value 69 and problem $v_2$ has value 420. Then if we perform the greedy algorithm, it is possible that we choose $p_1$ as the paper for $v_2$, so we are unable to choose a paper for $v_1$. This makes our total score 420. But we can do better by selecting $p_2$ for $v_2$ and $p_1$ for $v_1$ giving us a total score of 489, so the greedy algorithm is not optimal.

   (b) **(20 points)** To design a less greedy algorithm, we need a subroutine. Suppose we have an assignment of a set $S$ of solutions to pages, and we're given a new solution $t$. We'd like to know whether it's possible to assign all the solutions $S \cup \{t\}$ to pages (possibly rearranging which solutions are assigned to which pages) and output an assignment if so or "impossible" if not. Give an algorithm to do so. (Hint: First, given a solution, find all the other solutions whose pages it could commandeer, making a graph on the solutions. Search that graph, looking for a solution that could be assigned to a new page.)

   We perform the following algorithm: construct a directed graph $G = (V, E)$ where $V$ consists of the $v_i$'s (i.e. solutions) in $S$,[1] and there is an edge $v_i \to v_j$ iff solution $v_i$ can fit in the page $v_j$ is currently in. Create a dummy node for $t$ and connect an edge from this to all nodes whose paper for which $t$ can fit in. Now perform (slightly modified) DFS in $G$ starting at the dummy vertex until all vertices are visited ($G$ may not be connected, but we can select an arbitrary point to perform DFS on that has not been visited yet if our DFS terminates prematurely). The DFS is as follows: when we previsit an unvisited $v$, we see if we can insert $t$ there. If not, recursively DFS. If we can, we check if $v$ can be placed in any of the pages that do not yet have a solution. If it can, we place $v$ in a page and place $t$ in $v$'s previous location. Otherwise, we attempt to move $v$ to the next vertex in the DFS and recursively perform this with the next vertex but with $v$ in place of $t$, but we only do this if the next vertex is still unvisited (if it was already visited, we start backtracking). At postvisit, we mark $v$ as visited. Repeat this until all vertices are marked as visited. If at some point in the algorithm places a $v$ in $S$ into an empty page, we terminate and place $t$ according to our current DFS path.

   We prove correctness. Note that when we perform the DFS we are effectively checking if $t$ can replace any one of the solutions in a page it is currently in, and accounting for where the next solution can be placed if possible. When we mark a vertex as visited, we are doing this at postvisit, so this must mean that we cannot move the solution representing the current vertex (since by recursion, the DFS shows us where we can possibly put the next vertex, so the fact that we went back and postvisited means we cannot move it). Therefore, it does indeed suffice to only check unvisited vertices, which allows us to check each solution at most once. Finally, note that cycles that may be found in the DFS would be useless since we are then shifting the solutions in $S$ around but not including $t$. Thus, we exhaust all possibilities for placing $t$ and rearranging the other solutions, so if we find a path this will work, and if we don't then none must exist. This proves correctness.

   The construction of the graph takes $O(n + nk) = O(nk)$ time since we traverse the matrix of $a_{ij}$'s. The DFS takes $O(|V| + |E|) = O(nk)$ time since checking if we can place a vertex in a page takes $O(1)$. Thus, the running time is $O(nk)$.

   (c) **(15 points)** Consider the following "less greedy" algorithm: starting with the highest-value problem and working down, attempt to assign a solution to a page using the subroutine from the previous problem part. If there is a way, do it; if not, throw out the solution. Show that the "less greedy" algorithm finds the optimal score. (Hint: Suppose you have two assignments of solutions to pages: an "optimal" one and one found by the "less greedy" algorithm. It suffices (why?) to show that you can match each solution $s$ assigned to a page in only the "optimal" assignment with a solution $s'$ at least as valuable as $s$ that's assigned to a page in only the "less greedy" algorithm. To find such a matching, try swapping whether pages hold the solution from the "optimal" algorithm or the solution they held in the "less greedy" algorithm, starting with the page that held $s$.)

[1] Sorry a little abuse of notation.

Consider an optimal solution and suppose that it has some solution $s$ contained in page $p$ that was not included in the less greedy result. Then consider page $p$ in the less greedy result. If no solution was placed there, since $s$ was not included in the less greedy it follows that it is empty, which is a contradiction since we could still add $s$ in page $p$ in the greedy so the algorithm should have done that. Otherwise, there is some other solution $s'$ placed in $p$. If $s' < s$, we again get a contradiction because the greedy would have picked $s$ over $s'$ since it is larger. So, $s' \geq s$. We can repeat this for all other solutions $s$ in the optimal solution that was not included in the greedy. It then follows that the total sum in the less greedy algorithm is at least as big as the optimal, and so it must be optimal as well, proving correctness.

This algorithm must repeat the previous problem (the subroutine) $n$ times because this is the number of solutions we need to attempt inserting, so the overall run time is $O(n^2k)$.

5. Consider the following scheduling problem: we have two machines, and a set of jobs $j_1, j_2, j_3, \ldots, j_n$ that we have to process. To process a job, we place it on a machine; each machine can only process one job at a time. Each job $j_i$ has an associated running time $r_i$. The load on the machine is the sum of the running times of the jobs placed on it. The goal is to minimize the completion time, which is the maximum load over all machines.

   Suppose we adopt a greedy algorithm: each job $j_i$ is put on the machine with the minimum load after the first $i-1$ jobs. (Ties can be broken arbitrarily.)

   (a) **(5 points)** For all $n > 3$, give an instance of this problem for which the output of the greedy algorithm is a factor of $3/2$ away from the best possible placement of jobs.

   We split this into two cases depending on the parity of $n$. If $n$ is odd, let $n = 2k+1$ where $k \geq 1$. Then consider the jobs with running times $r_i = 1$ for $1 \leq i \leq 2k$ and $r_{2k+1} = 2k$. Then if we perform a greedy algorithm, we end up splitting the first $2k$ jobs into $k$ per machine, and add the last job into any machine, giving us a maximum load of $k + 2k = 3k$. We can do better though by putting all the 1 loads in one machine and the $2k$ load in the other, giving us a run time of $2k$. In this case, the greedy is exactly $3/2$ the optimal (this is clearly the optimal because we know that the machine that runs the job with $2k$ must run for at least $2k$).

   If $n$ is even, let $n = 2k+2$ where $k \geq 1$ and we consider the jobs with run times $r_i = 1$ if $i = 1$ or $3 \leq i \leq 2k+1$, $r_2 = 2k$, and $r_{2k+2} = 4k$. Then the greedy algorithm places the first job in the first machine, the $2k$ job in the second, the remaining 1 jobs in the first, and the last $4k$ job anywhere, giving us a maximum load of $2k + 4k = 6k$. On the other hand, the optimal is to just place all jobs but the $4k$ in one machine and the $4k$ in another, giving us a maximum load of $4k$. Again, this is optimal because we know that the machine that does the $4k$ job should take at least $4k$ to finish. This then gives us the equality case as $6k = 4k \cdot \frac{3}{2}$.

   (b) **(15 points)** Prove that the greedy algorithm always yields a completion time within a factor of 3/2 of the best possible placement of jobs. (Hint: Think of the best possible placement of jobs. Even for the best placement, the completion time is at least as big as the biggest job, and at least as big as half the sum of the jobs. You may want to use both of these facts.)

   Disclaimer: I copy pasted this from my proof in (c) and just changed some numbers, since the proof is exactly the same.

   We begin with the following observations (let $B$ be the optimal run time from now on):

   (i) $B$ is at least the sum of the run times of the jobs divided by the number of machines. This is true because by the pigeonhole principle, there must be a machine whose total run time is at least $\sum_{i=1}^{n} \frac{r_i}{2}$.

   (ii) $B$ is at least the maximum run time of the jobs $\max_{1 \leq i \leq n} r_i$. This is true because there will be a machine that will take the heaviest job, and it will then take at least that amount of time.

   Now we prove the result by induction on $n$, the number of jobs. The base case $n = 1$ is obvious.

   Now suppose it is true for all $k < n$. We will prove this for $n$ jobs. Consider the last step of the greedy algorithm where we add the last job $j_n$ to the machine with the least total run time. Suppose that this machine's run time is currently $L$. Then since $L$ is the minimum run time of a machine, it follows from the pigeonhole principle that

$$L \leq \frac{\sum_{i=1}^{n-1} r_i}{2} \leq B - \frac{r_n}{2},$$

   where the last step follows from our observation above. Then again since we know from our above observation that $r_n \leq \max_{1 \leq i \leq n} r_i \leq B$, it follows that

$$L + r_n \leq B + \left(1 - \frac{1}{2}\right) r_n \leq B + \left(1 - \frac{1}{2}\right) B = \left(2 - \frac{1}{2}\right) B.$$

   By the inductive hypothesis, it follows that for all the remaining machine, its run time is also at most $\left(2 - \frac{1}{2}\right) B'$, where $B'$ is the optimal allocation of run times for the first $n-1$ jobs. But clearly we have $B' \leq B$, and so the maximum run time of all the machines is at most $2 - \frac{1}{2} = \frac{3}{2}$ of the optimal allocation $B$, as desired.

   (c) **(10 points)** Suppose now instead of 2 machines we have $m$ machines. Prove the best upper bound you can on the ratio of the performance of the greedy solution to the optimal solution, as a function of $m$?

   We first show that if we have $m \geq 2$ machines, then the greedy algorithm's run time is at most $2 - \frac{1}{m}$ times the optimal run time. To prove this, we begin with the following observations (let $B$ be the optimal run time from now on):

(i) $B$ is at least the sum of the run times of the jobs divided by the number of machines. This is true because by the pigeonhole principle, there must be a machine whose total run time is at least $\sum_{i=1}^{n} \frac{r_i}{m}$.

(ii) $B$ is at least the maximum run time of the jobs $\max_{1 \le i \le n} r_i$. This is true because there will be a machine that will take the heaviest job, and it will then take at least that amount of time.

Now we prove the result by induction on $n$, the number of jobs. The base case $n = 1$ is obvious.

Now suppose it is true for all $k < n$. We will prove this for $n$ jobs. Consider the last step of the greedy algorithm where we add the last job $j_n$ to the machine with the least total run time. Suppose that this machine's run time is currently $L$. Then since $L$ is the minimum run time of a machine, it follows from the pigeonhole principle that

$$L \le \frac{\sum_{i=1}^{n-1} r_i}{m} \le B - \frac{r_n}{m},$$

where the last step follows from our observation above. Then again since we know from our above observation that $r_n \le \max_{1 \le i \le n} r_i \le B$, it follows that

$$L + r_n \le B + \left(1 - \frac{1}{m}\right) r_n \le B + \left(1 - \frac{1}{m}\right) B = \left(2 - \frac{1}{m}\right) B.$$

By the inductive hypothesis, it follows that for all the remaining $m - 1$ machines, their run times are also at most $\left(2 - \frac{1}{m}\right) B'$, where $B'$ is the optimal allocation of run times for the first $n - 1$ jobs. But clearly we have $B' \le B$, and so the maximum run time of all the machines is at most $\left(2 - \frac{1}{m}\right)$ of the optimal allocation $B$, as desired.

(d) **(5 points)** Give a family of examples (that is, one for each $m$ – if they are very similar, it will be easier to write down!) where the factor separating the optimal and the greedy solutions is as large as you can make it.

For $m$ machines, a construction that achieves the desired bound would be to consider $n = m^2 - m + 1$ jobs $j_1, j_2, \ldots, j_n$ with corresponding run times $r_1 = r_2 = \ldots = r_{n-1} = 1$ and $r_n = m$. Then the optimal run time is $m$ by distributing $m$ 1's to each of the first $m - 1$ machines then adding the last job to the final machine for a run time of $m$ (this is clearly the best). However, using a greedy algorithm would yield a run time of $2m - 1$ since every machine will have a run time of $m - 1$ right before we add the final job, which makes the overall run time $2m - 1 = \left(2 - \frac{1}{m}\right) m$. So this example works, and the bound is tight.

6. **(0 points, optional)**[2] Consider the following generalization of binary heaps, called $d$-heaps: instead of each vertex having up to two children, each vertex has up to $d$ children, for some integer $d \geq 2$. What's the running time of each of the following operations, in terms of $d$ and the size $n$ of the heap?

   (a) delete-max()

   (b) insert(x, value)

   (c) promote(x, newvalue)

   The last operation, promote(x, newvalue), updates the value of $x$ to $newvalue$, which is guaranteed to be greater than $x$'s old value. (Alternately, if it's less, the operation has no effect.)

[2] We won't use this question for grades. Try it if you're interested. It may be used for recommendations/TF hiring.