Problem Set 5 qyj4rf

data structures and algorithms 2 - spring 2024 due march 13, 2024 at 11:59p

---

Collaboration Policy: You are encouraged to collaborate with up to 4 other students, but all work submitted must be your own *independently* written solution. List the computing ids of all of your collaborators in the `collabs` command at the top of the tex file. Do not share written notes, documents (including Google docs, Overleaf docs, discussion notes, PDFs), or code. Do not seek published or online solutions for any assignments. If you use any published or online resources (which may not include solutions) when completing this assignment, be sure to cite them. Do not submit a solution that you are unable to explain orally to a member of the course staff. Any solutions that share similar text/code will be considered in breach of this policy. Please refer to the syllabus for a complete description of the collaboration policy.

Collaborators:

Sources: https://cs.stackexchange.com/questions/42224/people-crossing-a-bridge-a-proof-for-a-greedy-algorithm

---

problem 1 *Goldilocks and the n Bears*

There's been a crime is BookWorld!!

Fictional Detective Thursday Next is investigating the case of the mixed up porridge bowls. Mama and Papa Bear have called her in to help "sort out" the mix-up caused by Goldilocks, who mixed up their n childrens' bowls of porridge (there are n children total and n bowls of porridge total). Each child likes his/her porridge at a specific temperature, and thermometers haven't been invented in BookWorld at the time of this case. Since temperature is subjective (without thermometers), we can't ask the bears to compare themselves, and since porridge can't talk, we can't ask the porridge to compare themselves. Therefore, to match up each bear with it's preferred bowl, Thursday Next must ask the bears to check a specific bowl of porridge. After tasting a bowl of porridge, the bear will say one of "this porridge is too hot", "this porridge is too cold", or "this porridge is just right".

1. Give an algorithm for matching up bears with their preferred bowls of porridge which performs $O(n^2)$ total "tastes". Prove that your algorithm is $O(n^2)$.

   Solution:

   A solution that runs in $O(n^2)$ time could take the form of the following:

   Line the bears up in a line n units long and then have each bear try each of the n bowls of porridge, such that every single bear in the line tries every bowl of porridge exactly one time. Then, each bear will get a bowl of porridge they deemed "just right".

   A proof for this algorithm's runtime can be done rather elementary. There are a total of n^2 operations being done that is n number of bears are checking n number of bowls (or $n \cdot n = n^2$). The time it takes for bears to declare which bowl is "just right" and be assigned that bowl is negligible when compared to the n^2 term. As such, since the maximum amount of work that can be done within this algorithm is n^2, a $O(n^2)$ would accurately work as a time complexity.

2. Give an algorithm which matches bears with their preferred bowls of porridge which performs expected $O(n \log n)$ total "tastes". Intuitively, but precisely, describe how you know the algorithm is $O(n \log n)$.

   Solution:

   This algorithm will draw heavy inspiration from the quicksort algorithm which is on average $O(n \log n)$.

   To begin, we will once again line up the bears in a line of n units long (the order of bears is irrelevant). Then we let 1 bear go through the entire list of n bowls and try each bowl. Once every bowl has been tasted, the bear's "just right" bowl will serve as a pivot such that every bowl the bear deemed "too cold" will be placed to the left of the pivot bowl and every bowl the bear deemed "too hot" will be placed to the right of the pivot bowl (in this manner the bowls are also placed within a line). This process will repeat until a pivot bowl that has $> 30\%$ of the total bowls on either side of it is produced. This is done so that a decent pivot is generated.

   At this point, we have a pivot bowl and we have guaranteed that all the bowls to its left are colder and all the bowls to the right are hotter. From this point of the crux of the algorithm will begin.

   The next bear in the line will then first try the pivot bowl. If this bear deems the pivot bowl "too cold", they will only ever try bowls to the right of the pivot bowl as all of these bowls are hotter; conversely, if the bear deems the pivot bowl "too hot", they will only ever try bowls to the left of the pivot bowl as all of these bowls are colder.

   Then a similar process to the initial pivot creation will occur. This bear will try every bowl within the sub-list of bowls. Once the bear tries every bowl they will arrange the bowls such that every bowl they deem "too cold" will be placed to the left of their bowl and every bowl they deem "too hot" will be placed to the right of their bowl (Note that during this placing to the left or right, no sorting will be done bowls will be essentially shifted down still retaining their relative index order. So like if bowl 4 was objectively colder than bowl 2, but both bowls 4 and 2 were considered "too cold" by the bear bowl 2 will still appear first in index order, until it is stored later on in the algorithm). This bowl will then serve as a secondary pivot bowl (with the first pivot bowl being the primary pivot bowl). This secondary pivot bowl will then be added to one of two lists: one list that keeps track of all the secondary pivots to the left of the primary pivot or one list that keeps track of all the secondary pivots to the right of the primary pivot.

   From here each bear remaining in the line will follow a similar process. First, they will try the primary pivot bowl and then move accordingly (too cold –¿ right side, too hot –¿ left side). After trying the primary pivot the bears will then begin to try each of the secondary pivots such that a pivot is only tried if it falls within the sub-list partitioned by the previous secondary pivot and move accordingly. In this way, a pivot will only be tried if it has the potential to make the sub-list of bowls smaller (continuously narrowing down the list). Consider the example:

   $2, 1, 3, 4, 6, 5, 7, 8, 9, 11, 10, 12, 18, 17, 13, 14, 16, 15, 19, 20$

In this example let 8 be the primary pivot and $[9, 12, 19]$ be the secondary pivots. If we are to test a bear who has a preference for a bowl of temperature 14, they would first start testing on the primary pivot 8 and thus would limit their next bowl search to:

$9, 11, 10, 12, 18, 17, 13, 14, 16, 15, 19, 20$

Then using up to two secondary pivots we would see that the search would have to be limited to a bowl that is hotter than 12 and colder than 19 narrowing the bowl search to:

$18, 17, 13, 14, 16, 15.$

In terms of validating the $n \log n$ runtime, however, we can very easily intuitively prove the n portion of the runtime. The absolute minimum time it would take each bear to taste just one bowl is $O(n-1) = O(n)$. However, since there are more than just one bowl for the bears to taste, we have to multiply this by how many bowls the bears are tasting. As shown above, as more and more bears try bowls, we progressively sort the bowls by temperature and optimize the taste protocol as we avoid having bears taste bowls we can guarantee to not be "just right". Due to this optimization, we lower our overall from $O(n^2)$ to $O(n \log n)$ as increasing the number of bowls will also increase the number of tastes required, but not to the extent of n^2. Further, since this is an algorithm modeled heavily around quicksort, which is an algorithm known to have an average time complexity of $O(n \log n)$, we can generally infer that a time complexity of $O(n \log n)$ for this algorithm is reasonable.

PROBLEM 2 *Load Balancing*

You work for a print shop with 4 printers. Each printer i has a queue with n jobs: $j_{i,1}, \ldots, j_{i,n}$. Each job has a number of pages, $p(j_{i,m})$. A printer's workload $W_i = \sum_\ell p(j_{i,\ell})$ is the sum of all pages across jobs for for that printer. Your goal is to *equalize* the workload across all 4 printers so that they all print the same number of total pages. You may only remove jobs from the end of their queues, i.e., job $j_{i,n}$ must be removed before job $j_{i,n-1}$, and you are allowed to remove a different number of jobs from each printer. Note that jobs can only be removed, not added. Give a **greedy algorithm** to determine the maximum equalized workload (possibly 0 pages) across all printers. Be sure to state your greedy choice property.

Solution:
To begin, we determine the amount of pages each printer has to print. If at this point every printer has the same amount of pages to print our algorithm is done and we end!

Otherwise we take note of the maximum number of pages that are being printed and the printer that is associated with that number (if more than one printer is printing a page number equal to the maximum, then take the printer with the lower number. For example, if printers 2 and 4 are both printing 20 pages, prioritize printer 2). Then remove a single job from that printer and call this algorithm once again.

Through this manner the printer with the highest page number will recursively be reduced until all printers have the same page number (whether this number is a zero or non-zero number).

The greedy choice element of this algorithm occurs when choosing which printer to remove a job from. The algorithm specifically chooses the printer with the highest (maximum) number of pages to print. Since we cannot add jobs to any of the printers, the only way we can get all the printers to have the same number of pages printed is to whittled down the pages until they are equal. Thus starting with the printer that has the most pages to whittle away is an attractive choice.

PROBLEM 3 *Crossing the Bridge*

There are n people that need to cross a narrow rope bridge as quickly as possible, and each respective person crosses at speeds $s_1, s_2, ..., s_n$ *(note: you can assume these are integers and are sorted in descending order)*. You must also follow these additional constraints:

1. It is nighttime and you only have a single flashlight. One requires the flashlight to cross the bridge.
2. A max of two people can cross the bridge together at one time (and they must have the flashlight).
3. The flashlight must be walked back and forth, it cannot be thrown, mailed, raven'd, etc.
4. A pair walking across together crosses at the speed of the slowest individual. They must stay together!

Describe a greedy algorithm that solves this problem optimally. State the runtime of your algorithm and prove your algorithm always returns the optimal solution. *Note: The obvious greedy algorithm does NOT work here. Be careful! This is more complicated than it appears.*

Solution:
This algorithm has two cases: $n \leq 3$, Case 1, and $n \geq 4$, Case 2. Let us first start with Case 1.

Case 1 is more simple. This case will actually use the "obvious" solution mentioned in the problem description where the fastest person will walk every other person across the bridge. For example, a walking speed list of $[3, 2, 1]$ would result in a total time of $3 + 1 + 2 = 6$.

Case 2, however, uses the fact that a pair of people will always walk at the speed of the slowest person no matter what. We can use this fact to essentially "smuggle" across another person such that the remaining two slowest people will go together (so we essentially get them both across for the price of one).

To get this to work, we will first second the fastest two people, leave the fastest on the other side of the bridge and bring the second fastest back with the flashlight. Then the second fastest will give the flashlight to the slowest person and they will walk together with the second slowest person, giving the flashlight to the fastest person at the end of the bridge who will then come back and this process will repeat for as long as there are a distinct (and non-overlapping) two slowest and fastest people. Once this criterion is no longer satisfied, the list of people needing to walk across must be < 3, at which point the algorithm from Case 1 can be used. For example, a walking speed list of $[3, 4, 14, 20]$ would result in a total time of $4 + 4 + 20 + 3 + 4 = 35$, whereas using the Case 1 algorithm for everything would result in $20 + 3 + 14 + 3 + 4 = 44$.

The greedy choice within this algorithm is to (for as long as possible) send the two slowest remaining members together. This works, once again, because it essentially "smuggles" across the second slowest remaining person along with the slowest remaining person and avoids having to add two (comparatively) large numbers to the time it takes for everyone to cross the bridge. Allowing faster times to be achieved when compared to the "obvious" method.

Now let us prove that this algorithm will always return the optimal solution. Consider the following two lists of walking speeds: $[4, 5, 6, 7]$ and $[4, 5, 7, 8]$. Looking at their completion times we see $5 + 5 + 7 + 4 + 5 = 26$ (with this algorithm) compared to $7 + 4 + 6 + 4 + 5 = 26$ (with the obvious algorithm) and $5 + 5 + 8 + 4 + 5 = 27$ compared to $8 + 4 + 7 + 4 + 5 = 28$. We see that this algorithm is always at least on par with the obvious algorithm, and the larger the second smallest number is the more time it saves. To demonstrate this point even further, consider the example list $[1, 2, 3, 45, 60, 75, 90]$; within this list the 90 will smuggle the 75 and the 60 will smuggle the 45 leading to a completion time of $2 + 2 + 90 + 1 + 2 + 2 + 60 + 1 + 3 + 1 + 2 = 166$ compared to $90 + 1 + 75 + 1 + 60 + 1 + 45 + 1 + 3 + 1 + 2 = 280$. It is with this example that we see exactly how beneficial this smuggling property is, so much so that it will always result in at least a result on par with the obvious solution.

An underlying assumption of this proof is that the obvious solution is a good comparison algorithm. Let's attempt to validate this assumption. To begin, when using an algorithm that does not involve "smuggling", it would be fruitless to use a person other than the fastest person to run the flashlight back and forth as the fastest person will always carry the flashlight faster than anyone else (this is trivially true) and thus result in a faster completion time. As such, we can be sure that the obvious solution is the fastest solution that could result from having one person carry the flashlight back and forth, bringing along someone else every time. Thus, we can be certain that the obvious solution is the best of the solutions that do not involve smuggling.

Further, the proposed algorithm is the most optimal form of an algorithm that involves smuggling as it smuggles the second slowest person with the slowest. This ensures that the next slowest person's (after the slowest person's) travel time is essentially ignored and since they are the next slowest person, their travel time is guaranteed to be slower than everything that comes after them (since the list is sorted).

Because we have shown that the proposed algorithm is at least on par with the obvious solution, that the obvious solution is the best solution that does not use smuggling, and that the proposed solution is the most optimal solution that uses smuggling we can say that it will always return the optimal solution.

Finally, the runtime of this algorithm is $O(n \log n)$. As it will run in at least linear time (since it needs to go through every person in the list of size n); however, in addition to this there are intermediate steps where the fastest and second fastest persons will run back and forth, thus a runtime of $O(n)$ would be insufficient. These intermediate steps are not so significant that the algorithm would become $O(n^2)$, because although they increase as the size of the list n increases, the only increase at a fraction of this rate. Thus, a runtime of $O(n \log n)$ would be sufficient.