

Discussion 3

1. Let's consider a long, quiet country road with houses scattered very sparsely along it. We can picture the road as a long line segment, with an eastern endpoint and a western endpoint. Further, let's suppose that, despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations. Give an efficient algorithm that achieves this goal and uses as few base stations as possible. Prove that your algorithm correctly minimizes the number of base stations.

Solution:

Find the first house from the left (western most house) and go four miles to its right and place a base station there. Eliminate all houses covered by that station and repeat the process until all houses are covered.

Proof:

The proof is similar to that for the interval scheduling solution we did in lecture. We first show (using mathematical induction) that our base stations are always to the right of (or not to the left of) the corresponding base stations in any optimal solution. Using this fact, we can then easily show that our solution is optimal (using proof by contradiction).

- 1- Our base stations are always to the right of (or not to the left of) the corresponding base stations in any optimal solution:

Base case: we place our first base station as far to the right of the leftmost house as possible. If the base station in the optimal solution is to its right then the first house on the left will not be covered.

Inductive step: Assume our k th base station is to the right of the k th base station in the optimal solution. We can now show that our $k+1$ st base station is also to the right of the $k+1$ st base station in the optimal solution. To prove this we look at the leftmost house to the right of our k th base station which is not covered by our k th base station. We call this house H . If H is not covered by our k th base station, then it cannot be covered by the k th base station in the optimal solution since our base station is to the right of it. Our $k+1$ st base station is placed 4 miles to the right of H . If the $k+1$ st base station in the optimal solution is further to the right of our $k+1$ st base station, then H is not going to be covered by neither the k th base station nor the $k+1$ st base station in the optimal solution so the $k+1$ st base station in our solution must also be to the right of the $k+1$ st base station in the optimal solution.

- 2- Now assume that our solution required n base stations and the optimal solution requires fewer base stations. We now look at our last based station. The reason we needed this base station in our solution was that there was a house to the right of our $n-1$ st base station that was not covered by it. We call this house H . If H is not covered by our $n-1$ st

base station, then H will not be covered by the $n-1^{\text{st}}$ base station in the optimal solution either (since our base stations are always to the right of the corresponding base stations in the optimal solution). So, the optimal solution will also need another base station to cover H .

Complexity of our solution will be $O(n \log n)$ since we need to sort the houses first from left to right by their x coordinates. The actual positioning of the base stations will require $O(n)$ time.

2. Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of campers. One of his plans is the following mini-triathlon exercise: each contestant must swim 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first one contestant swims the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second contestant begins swimming the 20 laps; as soon as he or she is out and starts biking, a third contestant begins swimming, and so on. Each contestant has a projected *swimming time*, a projected *biking time*, and a projected *running time*. Your friend wants to decide on a *schedule* for the triathlon: an order in which to sequence the starts of the contestants. Let's say that the *completion time* of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathlon, assuming the time projections are accurate.

What is the best order for sending people out, if one wants the whole competition to be over as soon as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible. Prove that your algorithm achieves this.

Solution:

Sort the athletes by decreasing biking + running time.

Proof:

The proof is similar to the proof we did for the scheduling problem to minimize maximum lateness. We first define an inversion as an athlete i with higher $(b_i + r_i)$ being scheduled after athlete j with lower $(b_j + r_j)$. We can then show that inversions can be removed without increasing the competition time. We then show that given an optimal solution with inversions, we can remove inversions one by one without affecting the optimality of the solution until the solution turns into our solution.

1- Inversions can be removed without increasing the competition time.

Remember that if there is an inversion between two items a and b , we can always find two adjacent items somewhere between a and b so that they have an inversion between them. Now we focus on two adjacent athletes (scheduled one after the other) who have an inversion between them, e.g. athlete i with higher $(b_i + r_i)$ is scheduled after athlete j with lower $(b_j + r_j)$. Now we show that scheduling athlete i before athlete j is not going to push out the completion time of the two athletes i and j . We do this one athlete at a time:

- By moving athlete i to the left (starting earlier) we cannot increase the completion time of athlete i
- By moving athlete j to the right (starting after athlete i) we will push out the completion time of athlete j but since the swimming portion is sequential and athlete j gets out of the pool at the same time that athlete i was getting out of the pool before removing the inversion, and since athlete j is faster than athlete i in the biking and

running sections, then the completion time for athlete j will not be worse than the completion time for athlete i prior to removing the inversion.

- 2- Since we know that removing inversions will not affect the completion time negatively, if we are given an optimal solution that has any inversions in it, we can remove these inversions one by one without affecting the optimality of the solution. When there are no more inversions, this solution will be the same as ours, i.e. athletes sorted in descending order of biking+running time. So our solution is also optimal.

3. The values 1, 2, 3, . . . , 63 are all inserted (in any order) into an initially empty min-heap. What is the smallest number that could be a leaf node?

Solution: since there are $2^6 - 1$ elements in the heap, the heap will consist of a full binary tree with 6 levels. So, the smallest possible value at a leaf node would be 6.

4. Given an unsorted array of size n . Devise a heap-based algorithm that finds the k -th largest element in the array. What is its runtime complexity?

Solution: build a max heap of size n in $O(n)$ time. Do k extract_max operations to find the k th largest element. The overall complexity will be $O(n + k \log n)$

5. Suppose you have two min-heaps, A and B, with a total of n elements between them. You want to discover if A and B have a key in common. Give a solution to this problem that takes time $O(n \log n)$ and explain why it is correct. Give a brief explanation for why your algorithm has the required running time. For this problem, do not use the fact that heaps are implemented as arrays; treat them as abstract data types.

Solution:

We can compare the element at the top of A (T_A) with the element at the top of B (T_B).

| | |
|--------------------|--|
| If $T_A < T_B$ | Extract_Min (A) |
| Elseif $T_A > T_B$ | Extract_Min (B) |
| Else | A must be equal to B. We have found a common key |

We repeat the above until we either find a common key or one of the heaps is empty.

The complexity of the solution is $O(n \log n)$ since at each iteration we do two extract_min operations at a cost of $\log n$ and there could be at most $O(n)$ iterations.

6. You have a stack data type, and you need to implement a FIFO queue. The stack has the usual POP and PUSH operations, and the cost of each operation is 1. The FIFO has two operations: ENQUEUE and DEQUEUE. We can implement a FIFO queue using two stacks. What is the amortized cost of ENQUEUE and DEQUEUE operations.

Implementation of the queue using two stacks:

Enqueue – Push element enqueued onto stack A

Dequeue – If stack B is not empty, Pop top element from B, otherwise, Pop all elements from A and Push them onto B one at a time. Then Pop the top element from B.

Solution 1 : Aggregate Method

A worst-case sequence of operations occurs when we do n Enqueues and one Dequeue. So there will be a total of $n+1$ operations involved.

$T(n+1)$ = total time to Push n elements onto A + total time to Pop all elements from A + total time to Push all elements onto B + time to Pop top element from B

$$T(n+1) = n + n + n + 1 = 3n + 1$$

$$\text{Average cost of the operations} = T(n+1) / (n+1) = (3n+1) / (n+1)$$

$$\text{When } n \rightarrow \infty, (3n+1) / (n+1) = 3 = O(1)$$

$O(1)$ is the amortized cost of both Enqueue and Dequeue operations

Solution 1 : Accounting Method

Charge Enqueue more than its actual cost so we can save credit for the expensive Dequeue operation. Since each element Pushed onto A needs to be Popped from A and then Pushed onto B, we can charge Enqueue 3 units instead of its actual cost (which is 1 unit). We can charge Dequeue 1 unit.

Check to see if there will be enough credit to pay for a worst case sequence of operations:
 n Enqueue results in a total charge of $3n$. When we subtract the actual cost of $1n$, we are left with $2n$ units of credit. Dequeue is also charged 1 unit so we have a total to $2n+1$ to pay for the Dequeue operation. Let's see if that is enough. The actual cost of Dequeue will be $n + n + 1$ which is exactly $2n+1$. The charges are therefore sufficient.

$$\text{Amortized cost of Enqueue} = 3 = O(1)$$

$$\text{Amortized cost of Dequeue} = 1 = O(1)$$