Problem 1

Given a road with houses along it, n cell phone towers need to be placed at certain points along the road such that each house is within 4 miles (in range) of a cell phone tower. Say that a house is covered if it is within 4 miles of a cell phone tower. Produce an efficient algorithm that minimizes n, the number of cell phone towers needed to cover all houses.

Solution

Let the road be represented as the x axis, and choose one end to be the origin. Place the first tower 4 miles past the house closest to the origin (at position x_1) on the road at position $x_1 + 4$. Find the next house that is not in range of the first tower (at position $x_i > x_1 + 8$), and place the second tower 4 miles past that house at $x_i + 4$. Do this iteratively for each out of range house until the no such houses remain, taking $\mathcal{O}(m)$ time, where m is the number of houses that need to be covered.

Proof

We can use a 'stay ahead' argument to show that the solution above is optimal: Suppose there exists some optimal solution that produces a set of cell phone towers S. Let G be the set of cell phone towers produced by the greedy solution. We can show by induction that $\|S\| = \|G\|$ for any optimal solution set S, and that the towers in G cover at least as many houses as the towers in S. Let n be the number of cell phone towers placed by a solution.

Base case (n = 1):

G consists of a single tower that was placed past the first house encountered at x_1 at position $G_1 = x_1 + 4$, covering h houses between x_1 and $x_1 + 8$. S consists of a single tower covering i houses. We now examine the possible placement positions for S_1.

Case 1, $S_1 < G_1$:

G_1 covers h houses between x_1 and $x_1 + 8 = G_1 + 4$, and S_1 covers i houses between $x_a < x_1$ and $x_a + 8 < x_1 + 8$. Thus, S covers at most h houses, and G is an optimal solution.

Case 2, $S_1 > G_1$:

In this case, S fails to cover the first house at x_1.

So S cannot be the optimal solution (a contradiction).

So, for n = 1, G is an optimal solution, with $G_1 \geq S_1$. Assume that G is optimal up to and including placement of the n^{th} tower, and that $G_n \geq S_n$.

Case (n=n+1):

G consists of n towers that cover h houses between $x_1 and G_n + 4$, and S covers at most h houses between x_a and $S_n + 4$, with $S_n < G_n$. The $(n+1)^{th}$ tower in G is placed past the first out of range house encountered at x_{n+1} so that $G_{n+1} > G_n + 4$. We now examine the possible placement positions for S_{n+1}.

Case 1, $S_{n+1} < G_{n+1}$:

G_{n+1} covers k houses between x_{n+1} and $x_{n+1} + 8$, and S_{n+1} covers i houses between $x_b < x_{n+1}$ and $x_b + 8 < x_{n+1} + 8$. Thus, S covers at most k houses, and G remains an optimal solution.

Case 2, $S_{n+1} > G_{n+1}$:

In this case, S fails to cover the house at x_{n+1}, and is not optimal.

Thus, we see that for every tower in an optimal solution S, there is a tower in G that is farther along, so that towers in G always stay ahead of those in S. Since the greedy algorithm only places towers if it encounters an out of range house, the only way that $\|S\| < \|G\|$ for a set of n houses is if the towers in S cover fewer than n houses, which contradicts our initial assumption that S was an optimal solution set. Thus the greedy solution is an optimal solution.

Problem 2

The exercise describes a set of n jobs, where each job J_i requires both preprocessing time, p_i and finishing time, f_i. Jobs can only be preprocessed one at a time using a supercomputer, but all n jobs can be finished in parallel. A *schedule* is an ordering of jobs for the supercomputer to preprocess and the *completion time* of the schedule is the earliest time at which all jobs will have finished processing. The exercise asks for a polynomial time algorithm that, given n jobs, produces a schedule with minimal completion time.

Solution

This is similar to problem [KT] 4.6, where we were asked to do the same thing for triathlon times. Since the jobs can be finished in parallel, the algorithm must maximize overlap between p_i's and f_i's. The fastest way to do this is to sort each finishing time in non-increasing order and then pick the job with the longest finishing time to go first. If two jobs have the same finishing time, schedule the one with the shortest preprocessing time first. The final job in the schedule is the one with the shortest finishing time.

Proof

Imagine that the solution above is not optimal. Then, there exists some other optimal solution in which the jobs are not scheduled in non-increasing order of finishing times. Suppose that in the optimal solution, some job k is scheduled before some other job k+1, with finishing time $f_k < f_{k+1}$. Then, job k will leaves the supercomputer after p_k seconds, and k+1 leaves the supercomputer after $p_k + p_{k+1}$ seconds have passed, taking $p_k + p_{k+1} + f_{k+1}$ seconds to complete. The total time taken for both jobs to complete is $p_k + p_{k+1} + f_{k+1}$ seconds. In the greedy solution, the order of k and k+1 is swapped, so that job k+1 is scheduled first, taking p_{k+1} seconds to leave the supercomputer. Job k then leaves the supercomputer after $p_k + p_{k+1}$ seconds have passed, taking $p_{k+1} + p_k + f_k$ seconds to complete. Since job k+1 now completes earlier, the total completion time for both jobs is less than $p_{k+1} + p_k + f_{k+1}$ seconds. Also, notice that swapping the order of k and k+1 doesn't change the total preprocessing time needed by k and k+1, thus such a swap doesn't change the time at which adjacent jobs enter the supercomputer for preprocessing. Thus, the swap results in a lower completion time for jobs k and k+1 and doesn't increase completion time for any adjacent jobs, illustrating that the greedy solution is an optimal solution.

Time Complexity

Sorting the n jobs by finishing time takes $\mathcal{O}(nlog(n))$ time.

Problem 3

The exercise proposes a scenario in which we must send n video streams over a communication link. The i^{th} stream consists of b_i bits that take t_i seconds to transmit, so that $r_i = \frac{b_i}{t_i}$ is the rate at which the i^{th} stream transmits. We must decide on the order of transmission of the streams subject to the following constraint: $b_i \leq r \sum_{k=0}^{i} t_k$, where r is the constant bandwidth available (r is the transmission rate of the communication link). In other words, for each $t \in \mathbb{N}$, the total number of bits sent over the time interval from 0 to t cannot exceed rt. If a proposed schedule satisfies the above constraint, we say that it is *valid*. The problem asks us to verify or falsify a claim and then produce an algorithm that, given a set of n streams as defined above, *determines* whether or not a valid schedule exists in polynomial time. The problem *does not* ask for an algorithm that produces a valid schedule (yay).

Claim

There exists a valid schedule if and only if each stream i satisfies $b_i \leq rt_i$.

The constraint, as defined, states that each $b_i \leq r\sum_{k=0}^{i} t_k$, not that each $b_i \leq rt_i$. It's possible for $rt_i < b_i \leq r\sum_{k=0}^{i} t_k$ such that the constraint is still satisfied. For example, let $b_i = 100, t_i = 1, r = 50, \sum_{k=0}^{i} t_k = 5$. Then, $50 < 100 < 250$. Thus, the forward direction of this claim is clearly false, and the claim is false.

Algorithm

If the total number of bits we desire to transmit over a given time interval exceeds the total bandwidth available to us, then no permutation of streams will produce a valid schedule. This is because, eventually, some stream i will try to send b_i bits, such that $\sum_{k=0}^{i} b_i > r\sum_{k=0}^{i} t_k$ (by the pigeon-hole principle), at which point the schedule fails to be valid. Since the upper bound on the total number of bits sent by the n streams is $\sum_{i=0}^{n} b_i \leq r\sum_{i=0}^{n} t_i$, all our algorithm has to do is check whether the constraint is satisfied for all n streams. That is, it must sum the bit lengths of all n streams and check whether or not the sum is less than or equal to the bandwidth, r, times the time needed to send all n streams. This can be done in linear time with respect to n.

Problem 4

The exercise asks us to partition a set of n sensitive processes, where each process has a start time s_k and a finishing time f_k, using as few partitions as possible. We put a process i into the same partition as some other process j if their running times overlap, i.e., if $s_i < f_j$ or $s_j > f_i$. All processes in the k^{th} partition must be equivalent with respect to overlap, that is, each process must intersect with every other process in the partition at some point in time, t_k. We want to insert a status_check at each point t_k, so that status_check is invoked at least once during each sensitive process.

Solution

The given input is a set of n start times and n finish times for n sensitive processes. Clearly, the algorithm's first step is to sort the processes in non-decreasing order of finish and start times (in other words, establish a timeline). If every start time precedes every finish time, then we're done, and only one invocation of status_check between the final start time and the first finish time is necessary. Otherwise, the algorithm inserts a status_check in front of the first finish time, f_i, and moves all j processes with start times s_j such that $s_j < f_i$ to partition i. The algorithm creates k partitions and inserts k status_checks in this manner, iterating through the list of processes until no unpartitioned processes remain. When the algorithm finishes running, it returns the set of k inserted status_checks. The algorithm sorts the list of n processes in $\mathcal{O}(nlogn)$ time and partitions/iterates through the list in $\mathcal{O}(n)$ time. Thus, the algorithm runs in $\mathcal{O}(nlogn)$ time.

Proof

Suppose that the greedy algorithm does not return a minimal set of status_checks, and that there is some other optimal solution that returns a smaller set of status_checks. Let A be the set returned by the algorithm above and B be the set returned by the optimal algorithm. We can show by using a 'stay-ahead' argument that the cardinality of A is never greater than that of B. Let n be the number of status_checks in A.

Base Case (n=1):

Since the greedy algorithm always places its first status_check, a_1 at the last instant possible, the first status_check in B, b_1, must precede a_1, thus B has at least as many status_checks as A. Now, inductively assume that B has at least as many status_checks as A for n = 1, 2, ..., n-1.

Case (n=n):
The greedy algorithm only inserts a status_check if unpartitioned processes remain in the list. Thus, there is at least one sensitive process, p_n that needs a status_check invoked during its runtime. Since the greedy algorithm places a_n right before the finish time of p_n, B is forced to place a status_check before or at a_n in order to fulfill the status_check requirement. Now, B also contains n status_checks, and the proof of optimality is complete.

Problem 5

Description

Consider a circularly sorted list $X = x_1...xn$ such that $x_j < x_n < x_1... < x_{n-1}$ for some minimal value x_j. Design an algorithm with efficient time complexity that finds the value of j. Describe your algorithm by providing pseudo-code for it and analyze the time complexity of your algorithm.

Solution

The minimum element j is the element at which $x_n < ... < x_{j-1} > x_j < x_{j+1}...x_{n-1}$. The solution consists of performing a modified binary search through the list: the algorithm checks the endpoints of each sub-list to determined which half to search, until the minimum element is found:

```
If X has only one element, return x_1.
If X has only two elements, return min(x_1, x_2).
Find the midpoint of X, say x_mid.
If the x_1 < x_mid, then the left half is sorted.
        If x_k < x_n, then the right half is also sorted.
                Return min(right minimum, left minimum).
        Else, find right minimum by performing a recursive call with X = (x_mid...x_n).
                Return min(right minimum, left minimum).
Else, find left minimum by performing a recursive call with X = (x_1...x_mid).
Return min(right minimum, left minimum).
```

Correctness and Time Complexity

Since either the left half or the right half of the list must be sorted, we can eliminate half of the list each time we recursively call the algorithm. Thus, a binary search through the circularly sorted list takes $\mathcal{O}(logn)$ time.