Solutions to Practice Problems 4

Solution 1:

(a) For the unweighted take-off problem, we sort the planes in increasing order of take-off time t_i and schedule them in this order. Clearly, the time needed to do this is dominated by the sorting time, which is $O(n \log n)$. Since the take-off order can be anything, this is clearly feasible.

To see that this is optimal, we will show that any different take-off order cannot be better than the greedy order. It will simplify the proof to assume that the times t_i are distinct. If a take-off order does not match the greedy order, there must be two *consecutive* planes j and i where $t_j > t_i$ (see Fig. 1(a)).

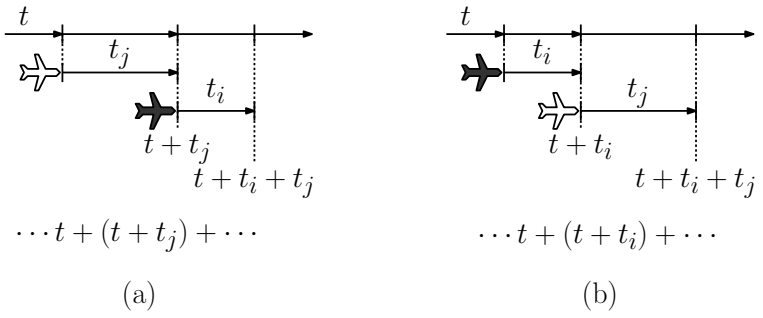

Figure 1: Proof of correctness unweighted take-off problem.

Let's assume that t time units have already transpired. If we form a new take-off order by swapping these planes, the waiting time for plane j changes from t to $t + t_i$ for a change of $(t + t_i) - t = t_i$. The weighting time for plane i decreases from $t + t_j$ to t, for a change of $t - (t + t_j) = -t_j$. No other take-off times change, so the overall change in the total wait is the sum of differences of these two changes

$$\text{change} \;=\; t_i - t_j,$$

which is negative since $t_j > t_i$. Thus, the total wait time decreases after the swap, which implies that the improperly ordered take-off sequence is *not* optimal, and therefore the greedy order is optimal.

(b) Sort the planes in increasing order of the ratio t_i/p_i and schedule them in this order. (Think of this as the time per passenger.) The running time is $O(n \log n)$.

The optimality argument is the same, but now we seek two consecutive planes j and i where $t_j/p_j > t_i/p_i$. In other words, $p_j t_i < p_i t_j$. By swapping them, plane-j's p_j passengers suffer an additional delay of t_i, for a change in the weighted wait time of $p_j t_i$, and plane-i's p_i passengers have their delays decrease by t_j for a change in the weighted wait time of $-p_i t_j$. All the other wait times are unaffected. Thus, the change in the total wait time is

$$\text{change} \;=\; p_j t_i - p_i t_j.$$

Since $p_j t_i < p_i t_j$, this is negative. Thus, the total weighted wait time decreases after the swap, which implies that the improperly ordered take-off sequence is *not* optimal, and therefore the greedy order is optimal.

Solution 2: We are given sequences of nonnegative integers $R = \langle r_1, \ldots, r_n \rangle$ and $C = \langle c_1, \ldots, c_n \rangle$, such that $0 \leq r_i, c_i \leq n$, and $\sum_i r_i = \sum_j c_j$. We will determine where to place the pawns row by row. Whenever we place a pawn in some column j, we will decrement c_j. This way, the column counts store the *remaining* number of pawns to be placed in each column.

For i running from 1 up to n, we first check whether the sequence $c[1..n]$ has at least r_i nonzero entries. If not, we terminate and declare that no layout is possible. Select the r_i columns of the c-array with the largest counts. (This can be done efficiently using counting sort. We will leave this as an exercise.) We place a pawn in each of these columns in row i. We then decrement the corresponding entries of the c-array. The pseudo-code is shown in the code block below.

Pawn Placement

```
place-pawns(r, c)                    // place pawns row/column counts r[1..n] and c[1..n]
    board[1..n][1..n] = all false
    for (i = 1 to n)
        identify the indices J of the r[i] largest values in c
        for each (j in J)
            if (c[j] == 0) return FAIL!
            else
                board[i][j] = true   // place a pawn on row i column j
                c[j]--               // decrement the column count
    return board
```

The algorithm can be implemented to run in $O(n^2)$ time. The only non-trivial part involves computing the r_i largest enties of the current c-array. This can be done by using counting sort to sort the c-array in decreasing order. Rather than permuting the entries of the array (which would wreck the column indices), we use a permutation array to keep track of the sorted order. Counting sort and all the other steps can be performed in $O(n)$ time per row, for a total time of $O(n^2)$.

In order to prove correctness, we need to establish feasibility and optimality. To show feasibility, we need to argue that, if the algorithm terminates, then each row and column has the desired number of pawns. This follows easily from the algorithm, since we places r_i pawns on row i, and since we decrement the c-array with each placed pawn, we never place too many pawns in any column. (Since the row and column sums are equal initially, it follows that the c-array will be zero on termination.)

We establish optimality by an exchange argument. Let O denote any valid layout. Let us consider the layout G generated by the greedy algorithm (or, if the algorithm fails, let G be the initial portion until it fails). If $O = G$, we are done. If $O \neq G$, we will show that it is possible to replace O with a valid layout O' that is a little more similar to G.

Let i denote the first row where O and G differ. Since both layouts have been identical up to now, and O is valid, we know that there are at least r_i nonzero values in the current c-vector. Let j be any column where G places a pawn on row i but O does not. Since G selects the largest c-values first, O must place a pawn on some other column j' that greedy does not use, where $c_{j'} \leq c_j$ (see Fig. 2(a) and (b)).

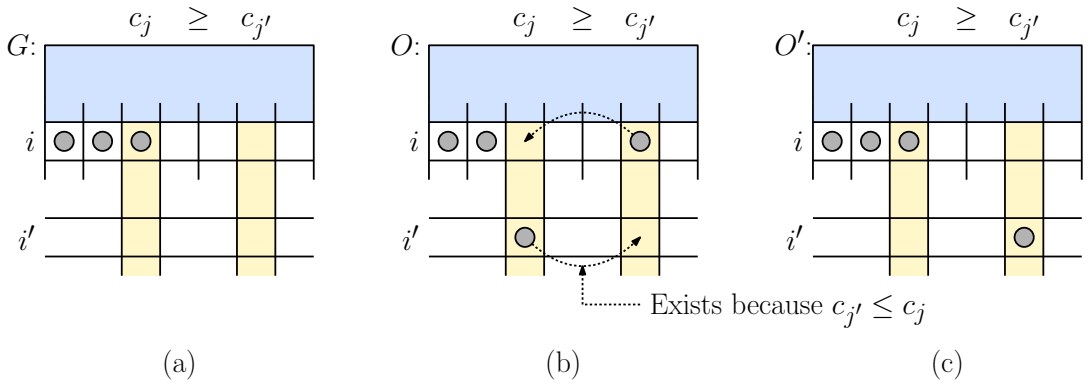

Figure 2: Proof of correctness for Solution 2. Two layouts G and L and the associated sorted c-sequences at each stage.

We modify O to form a new layout as follows. First, we move the pawn at row/column (i, j') to (i, j) (thus resulting in a layout that is a little closer to G). Now column j has one pawn too many and j' has one too few. Since $c_{j'} \leq c_j$, after moving the pawn on row i from column j' to column j, column j has at least one pawn strictly fewer than column j among rows $i+1$ through n. Therefore, by a pigeonhole argument, there exists a row $i' > i$ such that O places a pawn at (i', j) but not on (i', j'). We move this pawn from (i', j) to (i', j').

Observe that we have not altered the number of pawns on each row. By exchanging pawns between columns j and j', we have the proper number of pawns in these two columns. Since nothing else has changed, this new layout, which we call O' is valid. We have resolved one difference between O and G, as desired. By iterating this process (that is, by an induction argument), we eventually convert O into G through a sequence of valid layouts. This implies that G itself is valid, and hence the greedy algorithm runs to completion, as desired.

Solution 3: All four statements hold: Because Gonzalez only adds (never removes) centers, as more centers are added, the Δ_i and Γ_i values can only get smaller (or stay the same). When a new center is added in some iteration, it is placed at the point that has the maximum distance to its closest center. This implies that $\Gamma_{i+1} \leq \Delta_i$, and hence $\Gamma_4 \leq \Delta_3$. In the lecture on Gonzalez's algorithm (Claim 2), it was proved $\Gamma_i \geq \Delta_{i-1}$, and hence $\Gamma_4 \geq \Delta_3$.

Solution 4:

(a) We will show that, for any $i \geq 0$, the smallest k such that $\Delta(G_k) \leq 1/2^i$ satisfies the following recurrence, which we'll call $k(i)$.

$$k(i) = \begin{cases} 1 & \text{if } i = 0, \\ 3 & \text{if } i = 1, \\ 3k(i-1) - 3 & \text{otherwise.} \end{cases}$$

Thus, for example, $k(2) = 3 \cdot 3 - 3 = 6$, $k(3) = 3 \cdot 6 - 3 = 15$, and $k(4) = 3 \cdot 15 - 3 = 42$.

To see this, let's start with $k(1) = 3$. In general, assuming we know $k(i-1)$, to form the next level of the Sierpiński triangle, we make three copies at half the scale. This reduces the Δ value by exactly $1/2$. It increases the number of center points by the three copies, but three

of the points are replicated. Thus, we have $k(i) = 3k(i-1) - 3 = 3(k(i-1) - 1)$, as desired (see Fig. 3).

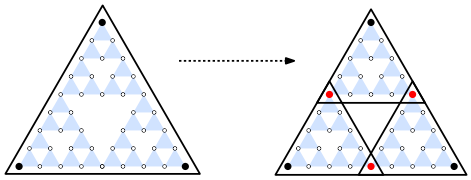

Figure 3: Fractal dimension of the Sierpiński triangle .

We claim that this recurrence solves to $k(i) = 1$ if $i = 0$, and $k(i) = (3^i + 3)/2$. It is easy to verify that the formula gives the correct in the basis cases ($i = 0$ and $i = 1$). We'll prove this works in general by induction. Suppose that $i \geq 2$. By applying the induction hypothesis and straightforward manipulations, we have

$$k(i) \;=\; 3k(i-1) - 3 \;=\; 3\frac{3^{(i-1)} + 3}{2} - 3 \;=\; \frac{3^i + 9}{2} - 3 \;=\; \frac{3^i + 3}{2} + \frac{6}{2} - 3 \;=\; \frac{3^i + 3}{2},$$

as desired.

(b) For $r = 1/2^i$, part (a) tells us that we can cover T using $k(i) = (3^i + 3)/2$ disks of radius $1/2^i$. Thus $N_T(1/2^i) = (3^i + 3)/2$. As i tends to infinity the "$+3$" term is negligible, and hence this is roughly $3^i/2$. The Hausdorff dimension of T is the value of d such that $N_T(r) = 1/r^d$, or equivalently $N_T(1/2^i) = (2^i)^d = 2^{id}$. Equating these yields,

$$2^{id} \;=\; \frac{3^i}{2} \iff \log 2^{id} \;=\; \log\frac{3^i}{2} \iff id\log 2 \;=\; i\log 3 - \log 2 \iff d \;=\; \frac{\log 3}{\log 2} - \frac{1}{i}.$$

In the limit, as i grows to $+\infty$, the $1/i$ term vanishes, and we have $d = \log 3/\log 2$, as desired.

Solution 5:

(a) A counterexample is shown in Fig. 4(a). The optimal solution consists of two pins (see Fig. 4(b)). The depth-based algorithm places a pin in the center, which has the highest depth of four. It still needs to place two more pins to cover the leftmost and rightmost intervals (see Fig. 4(c)).

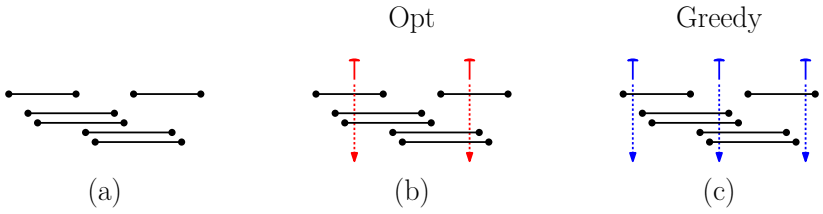

Figure 4: Counterexample to the depth-based heuristic.

(b) $\text{depth}(I)/\text{opt}(I) \leq \ln n$: This is essentially equivalent to the greedy set cover algorithm. We can restrict placement of pins to the n right endpoints of the intervals. (Any valid pin placement can be slid to the right until it hits a right endpoint.) Let B denote the set of right interval endpoints. Consider a set system (X, Σ) where the elements are intervals ($X = I$), and for each right endpoint $b \in B$, we define set $S(b)$ to be the intervals that are stabbed by b. Let Σ denote this collection of sets.

A stabbing set is equivalent to computing a minimum-sized set of right interval endpoints $b \in B$ (pins) that stab all the intervals of I. From the perspective of our set system, this is equivalent to computing a minimum-sized collection of sets $S(b) \in \Sigma$ that cover all elements $X = I$. These two problems are equivalent, and depth-based pinning heuristic is clearly equivalent to the greedy set cover heuristic. We know that the approximation ratio of the greedy heuristic for set cover is $\ln n$, where $n = |X| = |I|$.

(c) The greedy solution is based on the idea of "delaying" the placement of each pin as far as possible to the right. To begin, sort the intervals by their right endpoints $b_1 < \cdots < b_n$. We visit the intervals in this order. Place a pin at the first endpoint in the list (initially at b_1). Since this interval must contain a pin, this is the farthest right we can place this first pin. Now, remove all the intervals that are hit by this pin, or equivalently, all those whose left endpoint appears on or before b_i (see Fig. 5(b)).

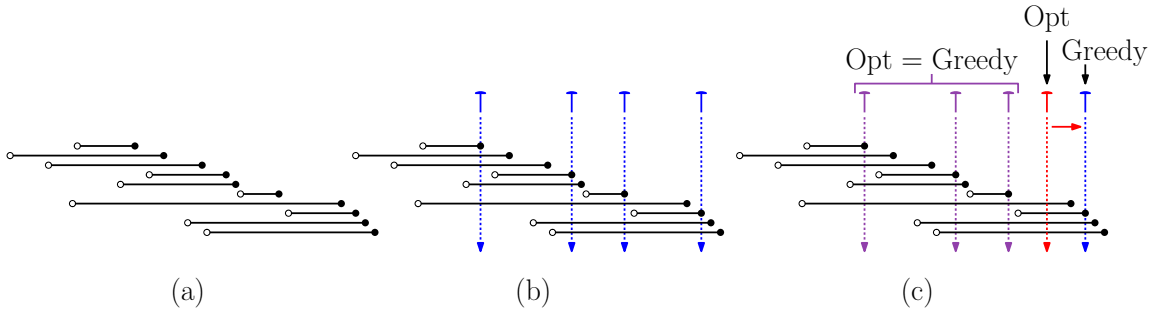

Figure 5: Greedy algorithm for 1-dimensional pinning.

It is interesting to note that this is essentially the same algorithm as the interval scheduling problem, just applied in a different context. The running time is $O(n \log n)$, dominated by the time it takes to sort the sequences by the right endpoints.

To see that this algorithm is correct, observe that each time we place a pin, we eliminate exactly those intervals that are stabbed by this pin, therefore every interval will be stabbed when the algorithm terminates. To see that this is optimal, consider list in pins in order from left to right. By its greedy nature, the greedy solution puts the each pin in the farthest right position possible. Thus, if greedy differs from opt, in their first difference, opt's pin will lie to the left of greedy's pin (see Fig. 5(c)). We can slide this pin of opt to the right as far as possible, until it hits the greedy pin. Since greedy is valid, the result of this sliding will also be a valid solution. By repeating this, eventually opt will be turned into greedy, without adding any more pins, and therefore greedy and opt use the same number of pins.