Solutions to Homework 2: Greedy Algorithms and Dynamic Programming

Solution 1:

(a) To create a counterexample, we create one very long request with the earliest start time, and a set of $n-1$ pairwise disjoint requests that overlap the first request (see Fig. 1(a)). The ESF strategy will succeed in scheduling only the first request, while the optimum schedules the remaining $n-1$. Thus, the performance ratio (that is, the ratio between the optimum and greedy) is $n - 1/1 = n - 1$. This can be made arbitrarily large by increasing n.

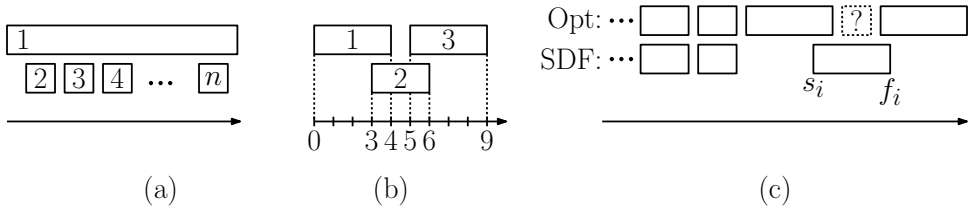

Figure 1: Interval scheduling.

(b) First observe that there is no loss of optimality by eliminating nesting. If $[s_i, f_i] \subset [s_j, f_j]$, then any schedule that uses $[s_j, f_j]$ cannot also use $[s_i, f_i]$, because they overlap. This means that we can replace $[s_j, f_j]$ with $[s_i, f_i]$ in any schedule without inducing any other overlaps. By repeating this, we convert any schedule to a nesting-free schedule without decreasing the number of scheduled requests. Therefore, we may assume that optimum schedule is nesting-free.

We assert that once nested intervals have been removed, sorting by start time is equivalent to sorting by finish time. To see why, suppose that $s_j \leq s_i$. By nesting-freeness, $f_j \leq f_i$, since otherwise we would have $s_j \leq s_i < f_i < f_j$, implying that $[s_i, f_i] \subset [s_j, f_j]$, a contradiction. Since EFF (earliest finish first) is known optimal for *any* set of requests, it follows that ESF is optimal on any set of nested-free intervals. Therefore ESF* is optimal.

(c) As a counterexample, consider three requests $I = \{[0, 4], [3, 6], [5, 9]\}$ (see Fig. 1(b)). Requests 1 and 3 have duration 4 are non-overlapping, while request 2 has duration 3. Thus, SDF schedules only request 2, while the optimum schedules both 1 and 3. Therefore, $\text{Opt}(I) = 2$, but SDF(I) = 1.

(d) As with the proof that EFF is optimal, we will employ an induction proof where we repeatedly modify any Opt schedule to match SDF. In that proof, we swapped one-for-one, implying that both solutions had the same size. Here, we will swap at most two-for-one, implying that the size of SDF is at least half as large as Opt.

Given any instance I, consider the optimum and SDF sizes, denoted $\text{Opt}(I)$ and SDF(I), respectively. If they are the same, then $\text{Opt}(I) = \text{SDF(I)}$, and hence the both have the same size. Otherwise, find the first interval (in start-time order) such that $[s_i, f_i]$ is in the SDF

solution but not in the Opt solution. We assert that at most two intervals from the Opt solution can overlap this interval. To see why, suppose to the contrary that three or more intervals of Opt were to overlap $[s_i, f_i]$. Then any interval other than the first and last would be completely contained within $[s_i, f_i]$, implying that this middle interval has a strictly smaller duration than $[s_i, f_i]$ (see Fig. 1(c)). Thus, SDF would have chosen this interval, rather than $[s_i, f_i]$, a contradiction.

Given the first difference $[s_i, f_i]$, we modify Opt by removing the (at most two) overlapping intervals from Opt and add $[s_i, f_i]$. The resulting schedule is clearly valid, and it has suffered a net decrease in size by at most one interval for each greedy interval on which we differ. By repeating this, we arrive at a schedule, denoted $\text{Opt}'(I)$ that is identical to greedy. The number of intervals that have been removed from the original optimum is not greater than the number of greedy intervals. Therefore, we have

$$\text{SDF}(I) \;=\; \text{Opt}'(I) \;\geq\; \text{Opt}(I) - \text{SDF}(I).$$

This implies that $2 \cdot \text{SDF}(I) \geq \text{Opt}(I)$, or equivalently, SDF(I) $\geq \text{Opt}(I)/2$, as desired.

Here is (arguably simpler) proof, which is based on a charging argument. Assign each interval of the optimum a *token*. The total number of tokens t is equal to $\text{Opt}(I)$. Whenever an interval of the optimum overlaps an interval of the SDF solution, transfer its token to the SDF interval. (If there are multiple such intervals in SDF, transfer the token to any one.) By the above observation, each interval of SDF receives at most two tokens. By adding up all the tokens in the SDF solution, we conclude that

$$\text{Opt}(I) \;=\; t \;\leq\; 2 \cdot \text{SDF}(I),$$

which implies that SDF(I) $\geq \text{Opt}(I)/2$, as desired.

Solution 2:

(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.

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.

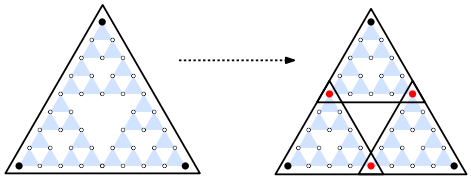

Figure 2: Interval scheduling.

(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 3:

(a) The greedy algorithm works analogously to the set-cover algorithm from class. Initially, none of the sheets are pinned. We select the node of the layered graph that has the highest depth and place a pin there. We then mark all of the sheets overlapping this region as "pinned." Next, we remove the newly pinned sheets from each of the paper sets associated with each region, and we update the depth counts for each node accordingly. (If a region is overlapped by k of the newly pinned sheets, we decrement its depth count by k.) We repeat this until all the sheets have been pinned.

(b) We could prove this by modifying the set-cover proof given in class, but we can do it in another way, by showing that the pinning problem is equivalent to the set-cover problem, and the above algorithm is doing exactly what the greedy set cover algorithm would do.

The problem of computing a pinning set is an instance of a famous optimization problem called the *hitting set* problem. We are given a set of items, $V = \{v_1, \ldots, v_m\}$ and a collection of sets $P = \{p_1, \ldots, p_n\}$. The problem is to compute a subset $V' \subset V$ of minimum size such that every set in P contains at least one element of V' (or is "hit" by at least one element of V').

In our case, V is the set of vertices in the layer graph, and for each vertex $u \in V$, we associate a set $p_u = \text{papers}(u)$ of the papers that overlap this region. Let $P = \{p_u\}_{u \in V}$. Finding a pinning set is equivalent to identifying a subset of vertices V' such that every set p_u is hit by at least element of $u \in V'$. Thus, this is an instance of hitting set.

It turns out that covering and hitting are equivalent problems. To see this, consider an instance (X, S) of the set cover problem (see Fig. 3(a)). We can express the element-set relations as a bipartite graph, where there is an edge (x_i, s_j) if $x_i \in s_j$ (see Fig. 3(b)). Now, let's swap the rolls of X and S, so that the s_j's are the elements, and the x_i's are the sets (see Fig. 3(c)). This is called the *dual set system*. We just reinterpret each edge (x_i, s_j) as meaning that item s_j lies in set x_i.

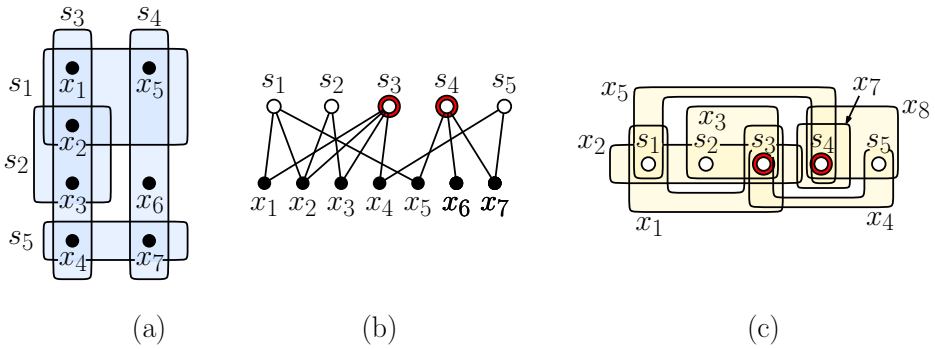

Figure 3: Equivalence between set cover and hitting (pinning) set.

We claim that a set cover in the set system (X, S) is a hitting set in the dual set system (S, X), and vice versa. Suppose that a subset $S' \subset S$ covers all the elements of X (as s_3 and s_4 do in Fig. 3(a)). Then the neighborhoods of these two vertices in the bipartite graph include all the elements of X (see Fig. 3(b)). But this is equivalent to saying that all the sets of X contain at least one of the elements of S'. (In Fig. 3(c), every set contains either s_3 or s_4.)

Having established the equivalence of set cover and hitting set, all that remains is showing that the greedy set cover heuristic (that is, repeatedly finding the set s_j that covers the greatest number of elements of X) is equivalent to the greedy hitting-set heuristic (that is, finding the element s_j that hits the greatest number of sets in X). In each case, exactly the same subset of elements will be chosen by both algorithms. Thus, the $\ln n$ approximation ratio proved in class applies to both.

Solution 4: In the standard DP for weighted interval scheduling, we needed to keep track of the latest (with respect to finish time) of the intervals under consideration. In this case, we will need to keep two such indices, one for the lake table and one for the tennis-court table.

(a) Let's assume that the requests have been sorted by finish times. In the standard version of the WIS problem, for $0 \leq j \leq n$, we defined $W(j)$ to be the maximum possible value achievable if we consider just requests $\{1, \ldots, j\}$. The trick here will be to maintain two indices, one for the requests that we can schedule for Table A and the other for requests we can schedule for Table B.

For $0 \leq i, j \leq n$, define $W(i, j)$ to be the maximum possible value achievable if we consider just requests $\{1, \ldots, i\}$ for Table A and $\{1, \ldots, j\}$ for Table B. (If i or j is zero, then we will assign no more requests to the associated table.) The final answer to our problem will be $W(n, n)$. We will make use of the function prior(j) introduced in the lecture. Recall that this is defined to be the largest integer such that $f_{\text{prior}(j)} < s_j$,

Computing the function $W(i, j)$ correctly requires care. The issue is that in the course of the recursive construction, the values of i and j may become equal. Suppose that you make a decision regarding one index (e.g., assign request $i = 7$ to Table A) and at some later time, when the value of the other index is equal, you make an inconsistent assignment (e.g., assign request $j = 7$ to Table B).

Can this be avoided by designing a data structure that keeps track of which requests have been assigned to which tables? This will not work. The reason that DP is efficient is that the state of the optimization problem is determined by just two numbers, i and j. Once you start allowing for additional constraints, the number of possible constraints quickly grows to quadratic size.

The trick is to make sure that whenever a decision is made to assign a request to a table, we can never assign the same request to the other table. We will exploit two properties to guarantee this: (1) whenever recursive calls are made, the indices can never increase, and (2) we always process the request for the larger of the two indices.

Let's derive our recursive formulation. For the basis cases, we define $W(0,0) = 0$. Otherwise, the larger of the two indices is at least 1.

- If $i > j$:
  - (Request i is not in the optimal schedule:) Since i is not in the schedule, optimality demands that we do the best possible with the remaining $i-1$ requests. Therefore, $W(i,j) = W(i-1,j)$.
  - (Request i is assigned to Table A:) We gain a profit of a_i, but we cannot assign any request after prior(i) to Table A. Since $j < i$, and indices cannot increase, this decision does not limit future assignments to Table B. Therefore, $W(i,j) = a_i + W(\text{prior}(i), j)$.
- If $i < j$: (This is symmetrical to the previous case.)
  - (Request j is not in the optimal schedule:) Since j is not in the schedule, optimality demands that we do the best possible with the remaining $j-1$ requests. Therefore, $W(i,j) = W(i,j-1)$.
  - (Request j is assigned to Table B:) We gain a profit of b_j, but we cannot assign any request after prior(j) to Table B. Since $i < j$, and indices cannot increase, this decision does not limit future assignments to Table A. Therefore, $W(i,j) = b_j + W(i, \text{prior}(j))$.
- If $i = j$:
  - (Request i is not in the optimal schedule for either table:) We should do the best we can with the remaining $i-1$ requests. Therefore, $W(i,i) = W(i-1,i-1)$.
  - (Request i is assigned to Table A:) We gain a profit of a_i, but we cannot assign any request to Table A after index prior(i). Also, we cannot assign this same request to Table B. There are no other constraints imposed by this decision, and therefore optimality demands that we do the best possible with the first prior(i) requests for Table A and the first $j-1$ requests for Table B. Therefore, $W(i,i) = a_i + W(\text{prior}(i), i-1)$.
  - (Request i is assigned to Table B:) We gain a profit of b_i, but we cannot assign any request to Table B after index prior(i). Also, we cannot assign this same request to Table A. There are no other constraints imposed by this decision, and therefore optimality demands that we do the best possible with the first $i-1$ requests for Table A and the first prior(i) requests for Table B. Therefore, $W(i,i) = b_i + W(i-1, \text{prior}(i))$.

Among each of the options available in each case, we take the best. This suggests the following recursive rule for $0 \le i, j \le n$:

$$W(i,j) \;=\; \begin{cases} 0 & \text{if } i = j = 0 \\ \max\left(\begin{array}{l} W(i-1,j), \\ a_i + W(\text{prior}(i), j) \end{array}\right) & \text{if } i > j \\ \max\left(\begin{array}{l} W(i,j-1), \\ b_j + W(i, \text{prior}(j)) \end{array}\right) & \text{if } j > i \\ \max\left(\begin{array}{l} W(i-1,i-1), \\ a_i + W(\text{prior}(i), i-1), \\ b_i + W(i-1, \text{prior}(i)) \end{array}\right) & \text{if } i = j \end{cases}$$

(b) To implement this efficiently, we will employ memoization. Let $W[0..n, 0..n]$ denote the final matrix. We assume that the value arrays $a[1..n]$ and $b[1..n]$ are global. For reconstructing the solution, we will use an parallel array $H[1..n, 1..n]$ where $H[i,j] \in \{A, B, R\}$, depending whether we chose to accept the request for Table-A, Table-B, or we reject it. Initially, the requests are sorted by finish times and the prior$[1..n]$ array is computed. Also, the W array is initialized to -1, meaning "undefined". The recursive function is presented in the code block below.

The final optimum value is given by $W[n,n]$. The algorithm's correctness follows from the explanations given in the DP formulation.

The algorithm fills a single entry of an $(n+1) \times (n+1)$ array with each recursive call. Due to memoization, the same recursive call is never made twice. Since the body of each recursive call takes $O(1)$ time (there are no loops), it follows that the total time $O(n^2)$. (The sorting of the requests can be done in $O(n \log n)$ time, and the computation of the prior array can be done in $O(n)$ time, thus, these do not affect the asymptotic running time.)

(c) We use the H array to retrace the algorithms decision and construct the optimum schedule. We start with $W[n,n]$ and work backwards. We know that each value of $W[i,j]$ arose from either two or three possibilities. If we accepted request i for Table A, we add i to Table A's schedule. If we accepted request j for Table B, we add j to Table B's schedule. Finally, if we rejected either, we do not add anything to either schedule. We then continue with the same table entry based on the recursive formulation. The algorithm for generating the schedule is given in the code block, below.

The correctness follows from the fact that we are just backtracing the recursive calls from `WIS-AB` that led to the best values. The running time is $O(n)$. The reason is that each time through the loop we strictly decrease the value of i or the value of j, and we never increase either value. Since both are initialized to n, after at most $2n$ iterations, both must be 0, and the algorithm terminates.

Solution to the Challenge Problem: The answer is that the probability of your seat being free on your arrival is exactly 1/2. The easy way to see this is to focus on just two seats. Let seat A be your assigned seat, and let seat B be the seat that was supposed to be occupied by the first student. The algorithm makes many random choices, but almost all of these are "red herrings,"

Two-Table WIS Value

```
WIS-AB(i, j) {                                      // memoized WIS for two tables
    if ( W[i, j] == -1 ) {                          // W[i, j] undefined?
        if ( i == j == 0 ) {                        // basis case
            W[i, j] = 0
        } else if ( i > j ) {                       // process index i
            rejVal = WIS-AB(i-1, j)                 // reject value
            accVal = a[i] + WIS-AB(prior[i], j)     // accept value
            if (rejVal > accVal)                    // better to reject?
                W[i, j] = rejVal; H[i, j] = 'R'
            else                                    // accept for table A
                W[i, j] = accVal; H[i, j] = 'A'
        } else if ( j > i ) {                       // process index j
            rejVal = WIS-AB(i, j-1)                 // reject value
            accVal = b[j] + WIS-AB(i, prior[j])     // accept value
            if (rejVal > accVal)                    // better to reject?
                W[i, j] = rejVal; H[i, j] = 'R'
            else                                    // accept for table B
                W[i, j] = accVal; H[i, j] = 'B'
        } else { // (i == j)
            rejVal = memo-WIS(i-1, i-1)             // reject value
            accAVal = a[i] + WIS-AB(prior[i], i-1) // value for Table A
            accBVal = b[i] + WIS-AB(i-1, prior[i]) // value for Table B
            if ( rejVal > max(accAVal, accBVal) )  // better to reject
                W[i, i] = rejVal; H[i, i] = 'R'
            else if ( accAVal > accBVal )          // better to accept for A
                W[i, i] = accAVal; H[i, i] = 'A'
            else                                    // better to accept for B
                W[i, i] = accBVal; H[i, i] = 'B'
        }
    }
    return W[j]                                     // return value
}
```

and don't affect the final answer. The key event that completely determines the final result is the first time any student selects either seat A or seat B. The result of this choice determines the final outcome.

To see why, let's consider both cases. Clearly, if A (your assigned seat) is chosen by any student before you, they will be sitting in that seat when you arrive and so your seat is gone. The other case is a bit trickier to see. Suppose that before anyone has taken seat A, some student selects seat B (the first student's seat). We claim that every student that follows will sit in their assigned seat, leaving your seat available. To see why, observe that the misplaced students who select random seats form a chain (linking the occupied seat from which they were misplaced to the random seat where they finally sat). Once someone fills seat B, the chain closes into a loop, and none of the remaining seats will ever be taken, except by the student assigned to them.

Since the misplaced students select seats at random, the probability that B is taken before A is $1/2$. Therefore, the probability that your seat is still available is exactly $1/2$, irrespective of the number of students.

```
get-schedule() {                                    // get the WIS schedule
    i = j = n                                       // start with the last request
    schedA = schedB = empty
    while (max(i, j) > 0) {
        if ( i > j ) {
            if ( H[i, j] == 'A' ) {                 // accept request i for table A?
                prepend i to schedA
                i = prior[i]
            } else
                i = i-1
        } else if ( j > i ) {
            if ( H[i, j] == 'B' ) {                 // accept request j for table B?
                prepend j to schedB
                j = prior[j]
            } else
                j = j-1
        } else // ( i == j )
            if ( H[i, i] == 'A' ) {                 // accept request i for table A?
                prepend i to schedA
                i = prior[i]; j = i-1
            } else if ( H[i, i] = 'B' ) {           // accept request i for table B?
                prepend i to schedB
                i = i-1; j = prior[j]
            } else
                i = j = i-1
        }
    }
    return (schedA, schedB)                         // return the final schedules
}
```