Homework 2: Greedy Algorithms and Dynamic Programming

Handed out Tue, Feb 25. **Due at the start of class (11am), Tue, Mar 11 - Updated!**. (electronic submission through Gradescope.)

Problem 1. (15 points) Recall the interval scheduling problem: Given a set of n start-finish intervals, $[s_i, f_i]$, find a maximum-sized subset of intervals that are pairwise disjoint. In class, we showed that earliest finish first (EFF) is optimal. Consider the following alternative greedy solutions:

ESF: Earliest start first: Sort the intervals by start time s_i.

SDF: Shortest duration first: Sort by the intervals by duration $f_i - s_i$.

If there are any ties, break them in favor of earliest finish time. We repeatedly schedule the first interval according to the sorted order, and then remove all intervals that overlap it. For a given set of intervals I, let $\text{Opt}(I)$ denote the maximum number of non-overlapping intervals, and let $\text{ESF}(I)$ and $\text{SDF}(I)$ denote the number of intervals scheduled by each of the above solutions.

Answer the following questions:

(a) (2 points) Show (by giving a counterexample) that ESF is not optimal. Further, explain how to extend your counterexample to an arbitrarily large interval sets I so that the performance ratio $\text{Opt}(I)/\text{ESF}(I)$ is arbitrarily large.

(b) (5 points) Consider the following variant ESF, called ESF*. For simplicity, let's assume that there are no duplicate intervals. First, go through and remove any interval that completely contains another. That is, as long as there exist itervals $[s_i, f_i]$ and $[s_j, f_j]$, such that $[s_i, f_i] \subset [s_j, f_j]$, remove $[s_j, f_j]$ from the set of requests. Repeat until there are no nested intervals. (You do not need to explain how to do this. Assume that you are given a procedure that does this.) Then run standard ESF on the remaining "nested-free" set of requests.

Prove that ESF* is optimal (for the original set of requests).

(c) (2 points) Show (by giving a counterexample) that SDF is not optimal.

(d) (6 points) Prove that for any instance I, $\text{SDF}(I) \geq \text{Opt}(I)/2$, that is, SDF schedules at least half as many as the optimum.

Problem 2. (10 points) An interesting feature of Gonzalez's algorithm is that it can be applied even to sets of infinite size. (Formally, we need the set to be closed and bounded, but let's not worry about these formalities.) In this problem, we will explore an intriguing connection between Gonzalez's algorithm and the concept of fractal dimension.

Consider the sequence of geometric sets shown in Fig. 1 below. If this sequence is carried out in the limit to infinity, the result is an a set T of infinite cardinality, called the *Sierpiński triangle*. This is a famous example of a *fractal*, that is, a self-similar shape whose Hausdorff dimension is a fraction. (The Hausdorff dimension of the Sierpiński triangle is $\log 3/\log 2 \approx 1.585$.)

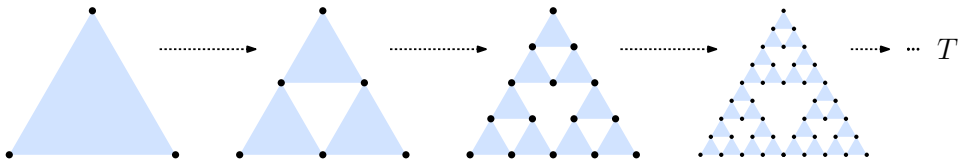


Figure 1: The limit of this process is the Sierpiński triangle.

Let's apply Gonzalez's algorithm to T. Let's assume that T has a side length of 1. Let $G = \{g_1, \ldots, g_k\}$ denote the first k Gonzalez points. Recall that for any $k \geq 0$, $\Delta(G_k)$ denotes the maximum distance of any point of T to its closest point in G_k. Let's start by placing g_1 at the lower-left vertex of T, yielding $\Delta(G_1) = 1$ (see Fig. 2(a)). The next two points will be placed at the other two vertices of the triangle, yielding $\Delta(G_3) = \frac{1}{2}$ (see Fig. 2(b)). The next three points will be placed at the midpoints of the triangle edges, yielding $\Delta(G_6) = \frac{1}{4}$ (see Fig. 2(c)). We can continue this process forever.

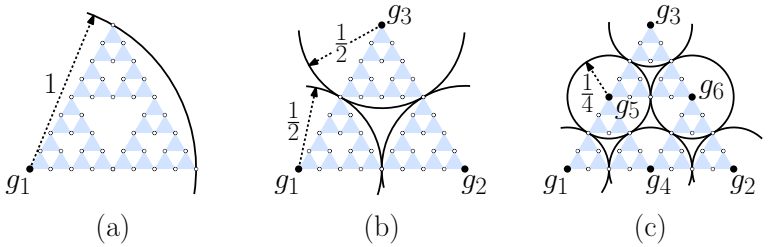


Figure 2: Running Gonzalez on the Sierpiński triangle.

(a) (6 points) From the above example, it should be clear that after adding a sufficient number of points k, $\Delta(G_k)$ decreases to a power of $\frac{1}{2}$. How many points do we need for this to happen? For any $i \geq 0$, let $k(i)$ denote the minimum number of points such that $\Delta(G_{k(i)}) \leq 1/2^i$. (The example in the figure shows that $k(0) = 1$, $k(1) = 3$, and $k(2) = 6$.)

Give a formula for $k(i)$. (For full credit, your answer should be an exact closed-form formula. For partial credit, you can express $k(i)$ as a recurrence or a closed-form formula that is within a constant factor of the exact value.)

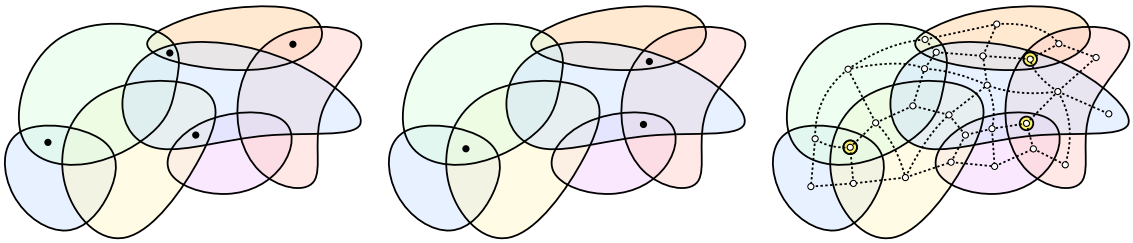
Justify your answer. (That is, explain how you derived it.)

(b) (4 points) We can use Gonzalez's algorithm to bound the Hausdorff dimension of any fractal. Let's do this for the Sierpiński triangle. Here is the definition of Hausdorff dimension.

> Given a set T and real $r > 0$, define $N_T(r)$ to be the number of balls of radius at most r required to cover T completely. The *Hausdorff dimension* of T is the unique number d such that $N_T(r)$ grows as $1/r^d$, as r approaches zero.

Prove that the Hausdorff dimension of the Sierpiński triangle is $d = \log 3 / \log 2$. (Hint: Use your result from (a). Express the radius r and covering number $N_T(r)$ in terms of i and $k(i)$. Consider the limit as i grows to infinity.)

Problem 3. (10 points) Let us return to a problem from Homework 1. We discussed the layered graph to represent a collection of overlapping regions. Imagine that the regions are sheets of paper that you want to pin to a bulletin board. You don't have many pins, so you want to use the minimum number of pins so that every piece of paper has at least one pin going through it. (For example, in Fig. 3(a) we show a possible pinning using four pins, and in Fig. 3(b) we show that there is an even better pinning using just three pins.)



(a) (b) (c)

Figure 3: Pinning papers to a bulletin board.

Observe that even though there are an infinite number of possible places pins could be placed, we can limit consideration to selecting just the vertices of the layered graph, since a pin placed anywhere within a given overlap region holds up the same subset of papers. (For example, the pinning of Fig. 3(b) is given by the three highlighted nodes of Fig. 3(c)).

(a) (4 points) Suppose that you are given a set $P = \{p_1, \ldots, p_n\}$ of sheets of papers to pin and the associated layered graph $G(P) = (V, E)$. (We don't care about edge directions). Assume that each vertex $u \in V$ is labeled with both its depth, denoted $\text{depth}(u)$, and the set of papers that overlap the region associated with u, denoted $\text{papers}(u)$. A subset of vertices $V' \subseteq V$ is called a *pinning set* if placing pins in these regions holds up all the sheets of paper, that is,

$$\bigcup_{u \in V'} \text{papers}(u) \;=\; P.$$

Present a simple *greedy algorithm* for computing a small pinning set. (Don't expect your algorithm to be optimal in terms of the size of the pinning set, since this problem can be shown to be NP-hard.) Note that you cannot move the papers, just select pin locations.

(b) (6 points) Given any set P of overlapping papers, let $\text{Opt}(P)$ denote the size of the optimum pinning set, that is, the smallest number of pins needed to hold up all the sheets of P. Let $\text{G}(P)$ the size of the pinning set generated by your greedy algorithm. Prove that $\text{G}(P) \leq (\ln n)\text{Opt}(P)$, where n is the number of papers to be pinned. (Hint: You can either prove this from first principles by modifying the proof of the greedy set-cover heuristic, or you can show how this problem can be viewed as an instance of a set-cover problem, and appeal directly to the greedy set-cover analysis.)

Problem 4. (15 points) The Parks and Recreation Department has received a budget increase, and they now have two picnic tables! Table A is near a lake and Table B is near a playground. People in the town have preferences which table they prefer by bidding different amounts for the two tables.

Formally, you are given a set of requests $R = \{r_1, \ldots, r_n\}$, where each request has three pieces of information: a start-finish interval $[s_i, f_i]$, the amount a_i they are willing to pay if the request is assigned to Table A, and the amount b_i they are willing to pay if the request is assigned to Table B. Your task is to compute an assignment of requests to tables subject to:

- A request can be assigned to at most one table, and may be assigned to none.
- If two requests are assigned to the same table, their time intervals cannot overlap.

Your objective is to find a schedule that maximizes the total payments.

An example is shown in Fig. 4. The input is shown in Fig. 4(a), where each request shows its a_i, b_i bids for the two tables. An optimal solution is shown in Fig. 4(b). Observe that one of the requests for Table B $(4, 3)$ actually preferred Table-A, but it was more profitable to assign them to Table B.

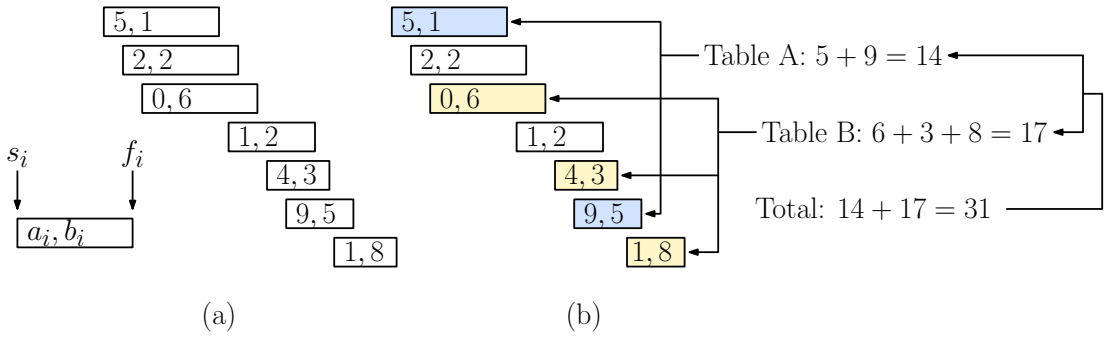


Figure 4: Weighted interval scheduling with two resources. Blue intervals are assigned to Table A and yellow to Table B.

(a) (6 points) Present a DP formulation for solving this problem. The input consists of arrays $s[1..n]$, $f[1..n]$, containing the interval start and finish times, and the arrays $a[1..n]$, and $b[1..n]$ containing the value bids for the two tables. Your DP formulation should take the form of a recursive function that computes the optimum value achievable by a valid schedule. (Don't worry about computing the schedule. This will be addressed in parts (b) and (c).)

Justify the correctness of your formulation. Also, indicate what the argument(s) to your recursive function that yields the optimum cost. (Hint: I believe that the best way to solve this is to modify the solution to the weighted interval scheduling (WIS) problem. Rather than maintaining one index $W(j)$, you will maintain two indices, one indicating which requests can be assigned to Table A and which can be assigned to Table B.)

(b) (6 points) Present an efficient algorithm implementing your recursive formulation. Derive the running time of your algorithm. (Hint: You may assume that you are given a procedure to compute the prior$[1..n]$ array from the WIS algorithm given in class. This prior array can be computed in $O(n)$ time. It would be a good idea to add whatever additional information is needed to solve part (c) below.)

(c) (3 points) Present an efficient algorithm that outputs an optimal schedule (say, by generating a list of requests that were granted and which table they were assigned to). Explain how your algorithm works and derive its running time.

Challenge Problem 1. (Note: Challenge problems count for extra credit points. These additional points are factored in only *after* the final cutoffs have been set and can only increase your final grade.)

Prof. M. has assigned each student a seat number for the final exam. There are 100 students in the class and exactly 100 seats in the room. When the first student arrives, however, they inform Prof. M. that they forgot their assigned seat. Prof. M. doesn't know the seat assignment either, and tells this student to select a random seat. After this, as each student arrives, if their assigned seat is empty, they sit there. If their assigned seat is taken, they select a random empty seat to sit in.

On the day of the final exam, your alarm clock fails to go off, and you arrive *last* to the exam. Since there are 100 seats and 99 students so far, exactly one seat is empty. What is the probability that your originally assigned seat is available? Chose one of the following options and justify your choice:

- It is extremely unlikely (probability nearly 0) that your originally assigned seat is still available.
- It is extremely likely (probability nearly 1) that your originally assigned seat is still available.
- There is some fixed probability p (which does not depend on the class size) such that your seat will be available with this probability. What is p?