Greedy Method

1 Introduction

Context In general, when you try to solve a problem, you are trying to find a solution from among a large space of possibilities. You usually do this by making a series of decisions, what move to make at each step (for example: send 2 cannibals across first, or 1 cannibal and 1 missionary, etc.).

If you have no information or no way to tell which choice is best, then you may have to do an exhaustive search over all the possibilities using backtracking. Basically this means you pick some choice and go down that path until you either reach a solution or hit a dead end, in which case you go back (undo) and try something else. While this will work in principle, in practice it's often unusable for realistic problems because there are just too many possibilities to explore. Even relatively simple-seeming problems may have billions upon billions of possible solutions, and it would take infeasibly long time to check them all.

Greedy Algorithm In some cases, the problem has some underlying structure that lets you be more intelligent, and you can figure out the right choice at each step, without ever needing to undo or backtrack a decision. In the happiest cases, when you're especially lucky, there is sufficient structure that you can quickly reach a solution by just picking the straightforward "best" choice at each step. This is called greedy method.

The bad news is that greedy method doesn't always work! When it does work it's great, but for many problems, when you pick what looks like the best choice for part of the solution, you can later find that it actually leads you down the wrong path and forces bad choices for other parts of the solution. So before you apply greedy method, it's important to prove that it actually will find the best solution for the problem you're trying to solve.

Another point to mention here is that even for a single problem, there may be more than one potential greedy strategy, more than one way to determine what looks like the "best" choice at each step. And it could be that greedy method works using one strategy but not using another strategy. So you need to prove that it works for the particular strategy you're using or disprove the strategy by using a counter-example.

Proving and Disproving strategies Often, a strategy sounds like it has a good chance of working. How do you know for sure? If you have some doubt, you can look for a counter-example. This involves creating an input that has a better solution than the greedy strategy produces.

Well then, how do you know for sure that a greedy strategy works, i.e., that it always finds the best solution? There are many proof techniques that can assure you without any doubt that your greedy strategy works!!! To name a few, there is proof by exchange argument, proof by "greedy stays ahead," and by showing the greedy solution achieves a mathematical bound.

Other Approaches In cases where greedy method doesn't work, you still may be able to use other approaches that are much better than doing an exhaustive search, as long as there is some structure to the problem. One such approach is dynamic programming, which will be covered in the coming weeks. Incidentally, in a way you can think of greedy method as a simple special case of Dynamic Programming.

2 Fixed Priority Ordering

2.1 Subset Problems

Problem 1: Maximizing the number of tasks

You have the following list of tasks to complete:

- Walk the dog (15 minutes)
- Mow the lawn (60 minutes)
- Shovel the snow (45 minutes)
- Take out the trash (2 minutes)
- Clean the pool (45 minutes)
- Wash the windows (75 minutes)
- Wash the car (30 minutes)
- Cook dinner (20 minutes)

You would like to complete as many jobs as you can in a certain amount of time. How would you select the jobs such that you will accomplish the greatest number of jobs in any amount of time?

Problem 2: Maximizing the profit

You have the following list of tasks to complete:

- Walk the dog (15 minutes, \$5)
- Mow the lawn (60 minutes, \$50)
- Shovel the snow (45 minutes, \$20)
- Take out the trash (2 minutes, \$2)
- Clean the pool (45 minutes, \$15)
- Wash the windows (75 minutes, \$60)
- Wash the car (30 minutes, \$15)
- Cook dinner (20 minutes, \$5)

If you have 2 hours, what is the maximum profit you can make? How did you select which tasks to complete?

Problem 3: Interval Scheduling

You have n friends, and all of them would like to meet up with you in one day. Each friend can only meet between a start time t_s and an end time t_e. Unfortunately, some of the time intervals your friends would like to meet with you overlap with each other. How would you select which friends you will meet with so that you will be meeting with the greatest number of friends possible?

Problem 4: Gas stops (CLRS)

Professor Midas drives an automobile from Newark to Reno along Interstate 80. His car's gas tank, when full, holds enough gas to travel n miles, and his map gives the distances between gas stations on his route. The professor wishes to make as few gas stops as possible along the way. Give an efficient method by which Professor Midas can determine at which gas stations he should stop, and prove that your strategy yields an optimal solution.

Problem 5: 0-1 knapsack special case (CLRS)

Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value. Give an efficient algorithm to find an optimal solution to this variant of the knapsack problem, and argue that your algorithm is correct.

2.2 Covering Problems

Problem 6: Unit intervals (CLRS)

Describe an efficient algorithm that, given a set $\{x_1, x_2, \ldots, x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.

Problem 7: Covering a spectrum

You want to create a scientific laboratory capable of monitoring any frequency in the electromagnetic spectrum between L and H. You have a list of possible monitoring technologies, T_i, $i = 1, ..n$, each with an interval $[l_i, h_i]$ of frequencies that it can be used to monitor. You want to pick as few as possible technologies that together cover the interval $[L, H]$. Give an efficient algorithm for finding such a set of technologies.

Problem 8: Filing clerk

There is a long row of pigeonholes labelled as integers from 1 to P. A filing clerk will need to access the pigeonholes $p_1, ..p_n$ in that order. Note that the pigeonholes she needs to access are not necessarily consecutive, sorted, or distinct. (See example.) Standing in front of hole j, the clerk can reach any hole between $j - r$ and $j + r$. Give an efficient algorithm to determine the minimum number of times the clerk needs to move in order to access the sequence of holes. (The clerk can start standing in front of any pigeonhole the algorithm chooses, and every time the clerk moves, it is to a new location the algorithm chooses. We count a move of any distance as a single move.)

2.3 Coloring Problems

Problem 9: Scheduling to minimize the number of rooms

Let $E = \{(s_i, f_i)|1 \leq i \leq n\}$ be a set of events to be scheduled in as few rooms as possible without conflicts. Find a schedule that minimizes the number of rooms used in the schedule.

Problem 10: Interval-graph coloring problem (CLRS)

Suppose that we have a set of activities to schedule among a large number of lecture halls. We wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall.

(This is also known as the *interval-graph coloring problem.* We can create an interval graph whose vertices are the given activities and whose edges connect incompatible activities. The smallest number of colors required to color every vertex so that no two adjacent vertices are given the same color corresponds to finding the fewest lecture halls needed to schedule all of the given activities.)

2.4 Sequencing Problems

Problem 11: Job ordering

You are given a list of n jobs $j_1, ..j_n$, each with a time $t(j)$ to perform the job, and a weight $w(j)$. You wish to order the jobs as $j_{\sigma(1)}, , ...j_{\sigma(n)}$ in such a way as to minimize: $\sum_i w(j_{\sigma(i)})(\sum_{1 \leq k < i} t(j_{\sigma(k)}))$; the weighted sum of the time each job has to wait before being performed. Give an efficient algorithm for this problem. (Hint: the algorithm is trivial, but you must prove that it is correct.)

Problem 12: Scheduling to minimize average completion time (CLRS)

Suppose you are given a set $S = \{a_1, a_2, \dots, a_n\}$ of tasks, where task a_i requires p_i units of processing time to complete, once it has started. You have one computer on which to run these tasks, and the computer can run only one task at a time. Let c_i be the *completion time* of task a_i, that is, the time at which task a_i completes processing. Your goal is to minimize the average completion time, that is, to minimize $(1/n)\sum_{i=1}^{n} c_i$. For example, suppose there are two tasks, a_1 and a_2, with $p_1 = 3$ and $p_2 = 5$, and consider the schedule in which a_2 runs first, followed by a_1. Then $c_2 = 5$, $c_1 = 8$, and the average completion time is $(5 + 8)/2 = 6.5$.

1. Give an algorithm that schedules the tasks so as to minimize the average completion time. Each task must run non-preemtively, that is, once task a_i is started, it must run continuously for p_i units of time. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

2. Suppose now that the tasks are not all available at once. That is, each task has a *release time* r_i before which it is not available to be processed. Suppose also that we allow *preemption*, so that a task can be suspended and restarted at a later time. For example, a task a_i with processing time $p_i = 6$ may start running at time 1 and be preempted at time 4. It can then resume at time 10 but be preempted at time 11 and finally resume at time 13 and complete at time 15. Task a_i has run for a total of 6 time units, but its running time has been divided into three pieces. We say that the completion time of a_i is 15. Give an algorithm that schedules the tasks so as to minimize the average completion time in this new scenario. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

Problem 13: Magnetic tape

A magnetic tape contains n programs of length $l_1, l_2, \ldots, l_n$. We know how often each program is used: a fraction p_i of requests to load a program concern program i, $1 \leq i \leq n$. This of course implies $\sum_{i=1}^{n} p_i = 1$. Information is recorded along the tape at constant density, and the speed of the tape drive is assumed to be a unit constant. Each time a program has been loaded, the tape is rewound to the beginning. If the programs are held in the order $\sigma_1, \sigma_2, \ldots \sigma_n$, the average time required to load program σ_j is $T_{\sigma_j} = p_{\sigma_j} \sum_{k=1}^{j} l_{\sigma_k}$. The average time $\bar{T}$ over all programs is $\bar{T} = \sum_{i=1}^{n} T_i$. Design an efficient algorithm to determine a placement order $(\sigma_1, \sigma_2, \ldots, \sigma_n)$ to minimize the average access time $\bar{T}$. Prove its correctness and analyze its time complexity.

Problem 14: Total waiting time (DPV)

A server has n customers waiting to be served. The service time required by each customer is know in advance: it is t_i minutes for customer i. So if, for example, the customers are served in order of increasing i then the ith customer has to wait $\sum_{j=1}^{i} t_j$ minutes.

We wish to minimize the total waiting time

$$T = \sum_{i=1}^{n} (\text{time spent waiting by customer } i)$$

Give an efficient algorithm for computing the optimal order in which to process the customers. Prove the correctness of your algorithm and determine its complexity.

2.5 Multiset Problems

Problem 15: Coin Change

Suppose you are working at a bank, and a client asks you for 89¢ in quarters, dimes, nickels, and pennies. The clients wallet is small, so you should exchange as few coins as possible. What is the fewest number of coins you can exchange? If another client comes in asking for n cents in coins, how would you pick which coins to exchange with the client?

Problem 16: Coin change (CLRS)

Consider the problem of making change for n cents using the fewest number of coins. Assume that each coin's value is an integer.

1. Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

2. Suppose that the available coins are in the denominations that are powers of c, i.e., the denominations are $c^0, c^1, \dots, c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.

3. Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n.

4. Give an $O(nk)$-time algorithm that makes change for any set of k different coin denominations, assuming that one of the coins is a penny.

Problem 17: Coin change

In the United States, coins are minted with denominations of 1, 5, 10, 25, and 50 cents. Now consider a country whose coins are minted with denominations $\{d_1, d_2, \dots, d_k\}$ of units. They seek an algorithm that will enable them to make change of n units using the minimum number of coins.

- The greedy algorithm for making change repeatedly uses the biggest coin smaller than the amount to be changed until it is zero. Provide a greedy algorithm for making change of n units using US denominations. Prove its correctness and analyze its time complexity.

- Show that the greedy algorithm does not always give the minimum number of coins in a country whose denominations are $\{1, 6, 10\}$.

- Give an efficient algorithm that correctly determines the minimum number of coins needed to make change of n units using denominations $\{d_1, d_2, \dots, d_k\}$. Analyze its running time.

3 Adaptive Priority Order

Problem 18: Business plan

Consider the following problem. You are designing the business plan for a start-up company. You have identified n possible projects for your company, and for, $1 \leq i \leq n$, let $c_i > 0$ be the minimum capital required to start the project i and $p_i > 0$ be the profit after the project is completed. You also know your initial capital $C_0 > 0$. You want to perform at most k, $1 \leq k \leq n$, projects before the IPO and want to maximize your total capital at the IPO. Your company cannot perform the same project twice.

In other words, you want to pick a list of up to k distinct projects, $i_1, \dots, i_{k'}$ with $k' \leq k$. Your *accumulated capital* after completing the project i_j will be $C_j = C_0 + \sum_{h=1}^{h=j} p_{i_h}$. The sequence must satisfy the constraint that you have sufficient capital to start the project i_{j+1} after completing the first j projects, i.e., $C_j \geq c_{i_{j+1}}$ for each $j = 0, \dots, k' - 1$. You want to maximize the final amount of capital, $C_{k'}$.

Problem 19: Party planning (DPV)

Alice wants to throw a party and is deciding whom to call. She has n people to choose from, and she has made up a list of which pairs of these people know each other. She wants to pick as many people as possible, subject to two constraints: at the party, each person should have at least five other people whom they know and five other people whom they don't know.

Give an efficient algorithm that takes as input the list of n people and the list of pairs who know each other and outputs the best choice of party invitees. Give the running time in terms of n.

4 Huffman Coding

Problem 20: Minimum Cost Sum

You are given a sequence $a_1, a_2, \dots, a_n$ of nonnegative integers, where $n \geq 1$. You are allowed to take any two numbers and add them to produce their sum. However, each such addition has a cost which is equal to the sum. The goal is to the find the sum of all the numbers in the sequence with minimum total cost. Describe an algorithm for finding the sum of the numbers in the sequence with minimum total cost. Argue the correctness of your algorithm.

Problem 21: Legal Huffman codes (DPV)

We use Huffman's algorithm to obtain an encoding of alphabet $\{a, b, c\}$ with frequencies f_a, f_b, f_c. In each of the following cases, either give an example of frequencies (f_a, f_b, f_c) that would yield the specified code, or explain why the code cannot possible be obtained (no matter what the frequencies are).

1. Code: $(0, 10, 11)$
2. Code: $(0, 1, 00)$
3. Code: $(10, 01, 00)$

Problem 22: Huffman codes text (DPV)

Suppose the symbols a, b, c, d, e occur with frequencies 1/2, 1/4, 1/8, 1/16, 1/16 respectively. $1/3, 1/5, 1/7, 1/7$, and 19/105 respectively.

1. What is the Huffman encoding of the alphabet¿

2. if this encoding is applied to a file consisting of 1,000,000 characters with the given frequencies, what is the length of the encoded file in bits?

Problem 23: Huffman codes DNA (DPV)

A long string consists of the four characters A,C,G,T; they appear with frequency 31%, 20%, 9% and 40%, respectively. What is the Huffman encoding of these four characters?

Problem 24: Compressing English (DPV)

The following table gives the frequencies of the letters of the English language (including the blank for separating words) in a particular corpus.

blank	18.3%	r	4.8%	y	1.6%
e	10.2%	d	3.5%	p	1.6%
t	7.7%	l	3.4%	b	1.3%
a	6.8%	c	2.6%	v	0.9%
o	5.9%	u	2.4%	k	0.6%
i	5.8%	m	2.1%	j	0.2%
n	5.5%	w	1.9%	x	0.2%
s	5.1%	f	1.8%	q	0.1%
h	4.9%	g	1.7%	z	0.1%

1. What is the optimum Huffman encoding of this alphabet?
2. What is the expected number of bits per letter?
3. Suppose now that we calculate the entropy of these frequencies

$$H = \sum_{i=0}^{26} p_i \log \frac{1}{p_i}$$

Would you expect it to be larger or smaller than your answer above? Explain.

4. Do you think that this is the limit of how much English text can be compressed? What features of the English language, besides letters and their frequencies, should a better compression scheme take into account?

5 Matching Problems

Problem 25: Payoff maximization (CLRS)

Suppose you are given two sets A and B, each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the ith element of set A, and let b_i be the ith element of B. You will receive a payoff of $\prod_{i=1}^{n} a_i^{b_i}$. Give an algorithm that will maximize your payoff. Prove that your algorithm maximizes the payoff, and state its running time.

Problem 26: Classes and rooms

You are given a list of classes C and a list of classrooms R. Each class c has a positive enrollment $E(c)$ and each room r has a positive integer capacity $S(r)$. You want to assign each class a room in a way that minimizes the total sizes (capacities) of rooms used. However, the capacity of the room assigned to a class must be at least the enrollment of the class. You cannot assign two classes to the same room. Design an efficient algorithm for assigning classes to rooms and prove the correctness of your algorithm.

Problem 27: Cookies

Consider the following problem:

You are baby-sitting n children and have $m \geq n$ cookies to divide among them. You must give each child exactly one cookie (of course, you cannot give the same cookie to two different children). For $1 \leq i \leq n$, child i has a greed factor g_i, which is the minimum size of a cookie that the child will be content with; and for $1 \leq j \leq m$ cookie j has a size s_j. Your goal is to maximize the number of content children. Child i is content if it is assigned cookie j such that $c_i \leq s_j$.

Problem 28: Dance partners

You are pairing couples for a very conservative formal ball. There are n men and m women, and you know the height and gender of each person there. Each dancing couple must be a man and a woman, and the man must be at least as tall as, but no more than 3 inches taller than, his partner. You wish to maximize the number of dancing couples given this constraint.

Problem 29: Segment matching

You are given a set of n horizontal segments of lengths $a_1, \ldots, a_n$ and a set of vertical segments of length $b_1, \ldots, b_n$. Each horizontal segment a_i can be combined with a vertical segment b_j to form a rectangle of area $a_i \cdot b_j$. A matching between the lengths assigns each horizontal length a_i a distinct vertical length b_{j_i}. Give as efficient an algorithm as possible that on input $a_1, \ldots, a_n, b_1, \ldots, b_n$, finds a matching between the horizontal and vertical segments that maximizes the total area of all resulting rectangles, $\sum_i a_i b_{j_i}$.

Problem 30: Oxen pairing

Consider the following problem: We have n oxen, $O_1, ..O_n$, each with a strength rating S_i. We need to pair the oxen up into teams to pull a plow; if O_i and O_j are in a team, we must have $S_i + S_j \geq P$, where P is the weight of a plow. Each ox can only be in at most one team. Each team has exactly two oxen. We want to maximize the number of teams.

6 Trees

Problem 31: Largest independent set for a tree

The problem is to find the largest independent set for the special case when the input graph is a tree. Remember, an independent set S of a graph is a set of nodes that does not contain both of the endpoints of any edge, i.e. for any edge $\{x, y\}$ either $x \notin S$ or $y \notin S$. So here, we must have a set of nodes of the tree S so that we cannot have both a node and its parent in the set.

Problem 32: Perfect matching in a tree

Give a linear-time algorithm that takes as input a tree and determines whether it has a perfect matching: a set of edges that touches each node exactly once.

Problem 33: Vertex cover in a tree

Give an efficient greedy algorithm that finds an minimum size vertex cover for a tree in linear time.

Let $G = (V, E)$ be a graph. We say that a subset of vertices $C \subseteq V$ is a vertex cover if for every edge in $e \in E$, one of the endpoints of e is in C.

You can assume that G is a rooted tree and each node comes with a parent pointer and a list of child pointers. The list could be an empty list.

Problem 34: Timing circuits (KT 4.24)

Timing circuits are a crucial component of VLSI chips. Here's a simple model of such a timing circuit. Consider a complete balanced binary tree with n leaves, where n is a power of two. Each edge e of the tree has an associated length ℓ_e, which is a positive number. The distance from the root to a given leaf is the sum of the lengths of all the edges on the path from the root to the leaf.

The root generates a *clock signal* which is propagated along the edges to the leaves. We'll assume that the time it takes for the signal to reach a given leaf is proportional to the distance from the root to the leaf.

Now, if all leaves do not have the same distance from the root, then the signal will not reach the leaves at the same time, and this is a big problem. We want the leaves to be completely synchronized, and all to receive the signal at the same time. To make this happen, we will have to *increase* the lengths of certain edges, so that all root-to-leaf paths have the same length (we're not able to shrink edge lengths). If we achieve this, then the tree (with its new edge lengths) will be said to have *zero skew*. Our goal is to achieve zero skew in a way that keeps the sum of all the edge lengths as small as possible.

Give an algorithm that increases the lengths of certain edges so that the resulting tree has zero skew and the total edge length is as small as possible.

Example. Consider the tree in Figure 1, in which letters name the nodes and numbers indicate the edge lengths.

The unique optimal solution for this instance would be to take the three length-1 edges and increase each of their lengths to 2. The resulting tree has zero skew, and the total edge length is 12, the smallest possible.

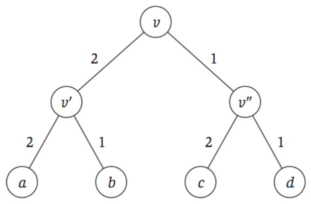

Figure 1: An instance of the zero-skew problem, described in Exercise 23.

7 Graphs

Problem 35: Coupons

Consider the following "coupon collector" problem. There are different varieties of cereal, and each comes with a single coupon for a discount on another box of cereal, perhaps of another variety. You can use multiple coupons when purchasing a new box, up to getting the new box free, but you never get money back. You want to buy one box of each variety, for as little money as possible. Describe an efficient algorithm, which, given as input, for each variety, its price , the value of the enclosed coupon, and the brand for which the coupon gives a discount, computes an optimal order in which to buy the cereal.

Problem 36: Unique minimum spanning tree (DPV)

Let $G = (V, E)$ be an undirected graph. Prove that if all its edge weights are distinct, then it has a unique minimum spanning tree.

Problem 37: Maximum spanning tree (DPV)

Show how to find the *maximum* spanning tree of a graph, that is, the spanning tree of largest total weight.

Problem 38: Feedback edge set (DPV)

A *feedback edge set* of an undirected graph $G = (V, E)$ is a subset of edges $E' \subseteq E$ that intersects every cycle of the graph. thus, removing the edges E' will render the graph acyclic.

Give an efficient algorithm for the following problem:

Input: Undirected graph $G = (V, E)$ with positive edge weights w_e.

Output: A feedback edge set $E' \subseteq E$ of minimum total weight $\sum_{e \in E'} w_e$.

Problem 39: Time-varying Minimum Spanning Tree (KT 4.26)

One of the first things you learn in calculus is how to minimize a differentiable function like $y = ax^2 + bx + c$, where $a > 0$. The Minimum Spanning Tree Problem, on the other hand, is a minimization problem of a very different flavor: there are now just a finite number of possibilities for how the minimum might be achieved–rather than a continuum of possibilities–and we are interested in how to perform the computation without having to exhaust this (huge) finite number of possibilities.

One can ask what happens when these two minimization issues are brought together, and the following question is an example of this. Suppose we have a connected graph $G = (V, E)$. Each edge e now has a *time-varying edge cost* given by a function $f_e : \mathbb{R} \to \mathbb{R}$. Thus, at time t, it has cost $f_e(t)$. We'll assume that all these functions are positive over their entire range. Observe that the set of edges constituting the minimum spanning tree of G may change over time. Also, of course, the cost of the minimum spanning tree of G becomes a function of the time t; we'll denote this function $c_G(t)$. A natural problem then becomes: Find a value of t at which $c_G(t)$ is minimized.

Suppose each function f_e is a polynomial of degree 2: $f_e(t) = a_e t^2 + b_e t + c_e$; where $a_e > 0$. Give an algorithm that takes the graph G and the values $\{(a_e, b_e, c_e) : e \in E\}$ and returns a value of the time t at which the minimum spanning tree has minimum cost. Your algorithm should run in time polynomial in the number of nodes and edges of the graph G. You may assume that arithmetic operations on the numbers $\{(a_e, b_e, c_e)\}$ can be done in constant time per operation.

8 Other

Problem 40: Averaging Down

There are 10 identical vessels, one of them with 100 pints of water and the others empty. You are allowed to perform the following operation: take two of the vessels and split the total amount of water in them equally between them. The object is to achieve a minimum amount of water in the original vessel (the one containing all the water in the initial setup) by a sequence of such operations. What is the best way to do this? You can do as many operations as you want — the goal is just to minimize the amount of water in the original vessel.

Problem 41: Off-line caching

In the off-line caching problem, you are given a sequence of *memory locations* to be read in order, $m_1, \ldots, m_t$ and an initial *cache* of k memory locations, $C_0 = \{c_1, \ldots, c_k\}$. The solution must specify a sequence $C_1, \ldots, C_t$ of cache contents where each C_i is a set of k memory locations, so that, if $m_i \in C_{i-1}$, $C_i = C_{i-1}$ (called a *hit*); and if $m_i \notin C_{i-1}, C_i = C_{i-1} \cup \{m_i\} - \{m'\}$ for some $m' \in C_{i-1}$ (called a *cache miss*). The objective is to minimize the number of cache misses. (A solution can be described more succinctly by giving the sequence of memory locations evicted at each time step, rather than the entire cache contents at each time.)

A greedy strategy for this problem is $LastFutureUse(LFU)$. When a cache miss occurs, look at the first future occurence for each location currently in the cache. Delete the location whose future use is the latest.

Give a proof that this strategy is optimal, and then give an efficient algorithm for this problem using this strategy.

Problem 42: Scheduling variations

Consider the following problem of scheduling unit-time tasks with deadlines and penalties on a single processor. There are n jobs, each of which takes a unit time to complete. If job i is completed before time d_i, then it incurs no penalty. Otherwise, it incurs a penalty p_i. Design an efficient algorithm to schedule all the jobs on a single processor so as to minimize the total penalty. Assume that the processor is available at time 0. Also assume that all deadlines are non-negative.

Design an efficient algorithm. There is an $O(n \log n)$ algorithm for this problem. Less efficient algorithms will get partial credit.

Provide a clear high-level description of your algorithm, argue its correctness and analyze its time complexity.

Problem 43: IPO

Consider the following problem. You are designing the business plan for a start-up company. You have identified n possible projects for your company, and for each project, i, you have calculated the minimum capital you need to start the project, $R_i > 0$, and the profit you will get after you complete the project, $P_i > 0$. Profit is the net revenue obtained after subtracting the capital used for carrying out the project. You also know your initial capital $C_0 > 0$. You want to perform at most k, $1 \leq k \leq n$, projects before the IPO and want to maximize your total capital at the IPO. Your company cannot perform the same project twice.

In other words, you want to pick a list of up to k distinct projects, $i_1, \cdots, i_{k'}$ with $k' \leq k$. Your *accumulated capital* after project i_j will be $C_j = C_0 + \sum_{h=1}^{h=j} P_{i_h}$. The sequence must satisfy the constraint that you have sufficient capital to start i_{j+1} after completing the first j jobs, i.e., $C_j \geq R_{i_{j+1}}$ for each $j = 0, \cdots, k'-1$. You want to maximize the final amount of capital, $C_{k'}$.

Design an efficient algorithm. There is an $O(n \log n)$ algorithm for this problem. Less efficient algorithms will get partial credit.

Provide a clear high-level description of your algorithm, write pseudo-code, argue correctness and analyze time complexity.

9 Solved Problems

Problem 44: Covering a spectrum

You want to create a scientific laboratory capable of monitoring any frequency in the electromagnetic spectrum between L and H. You have a list of possible monitoring technologies, T_i, $i = 1, ..n$, each with an interval $[l_i, h_i]$ of frequencies that it can be used to monitor. You want to pick as few as possible technologies that together cover the interval $[L, H]$. Give an efficient algorithm for finding such a set of technologies.

Solution: Covering a spectrum

Notation: l-value refers to any l_i and h-value to any h_i for $1 \leq i \leq n$.

Algorithm:

1. Sort the technologies so that the l-values are in nondecreasing order. Break any ties so that the h-values are nondecreasing.

2. We will process the technologies in the sorted order and select the next technology that extends the coverage as much as possible. More precisely, assuming that we have covered the interval $[L, m']$, we select the next technology to extend the coverage to the interval $[L, m]$ where $m > m'$ is as large as possible. Initially, we set $m = L$ and repeat the following steps while $m < H$.

   (a) A technology i is valid for the position m if $l_i \leq m \leq h_i$. Among all the technologies valid for m, select the technology with the largest h-value and let k be the index of the selected technology.

   (b) Update m to be h_k.

3. Output the selected technologies.

Time complexity. It takes $O(n \log n)$ time for sorting the technologies.

We can compute the best technology among all valid technologies by iterating through the l-values in sorted order and tracking the the maximum h-value for technologies where the l-value is less than or equal to the current value of m. Once a technology is selected, we continue from the next technology with l-value greater than m and update the value of m. It only takes $O(n)$ time to traverse the technologies while keeping tracking of the desired quantities.

The overall time complexity is $O(n \log n)$.

Correctness.

Let I be an instance of the problem with n technologies. We are given $L \leq H$, l_i and h_i for $1 \leq i \leq n$. Let $S = t_1, t_2, \ldots, t_p$ be the sequence of (indices of) technologies output by the greedy algorithm in the order they were selected during the execution.

Claim 9.1. *The greedy algorithms selects technologies in increasing order of l-values, that is, we have $l_{t_1} \leq l_{t_2} \leq \cdots \leq l_{t_p}$.*

Proof. Consider the i-th iteration of the algorithm. Let m_i be the value of m at the beginning of the i-th iteration. The algorithm considers all valid technologies $[l_i, h_i]$ such that $l_i \leq m_i \leq h_i$ and selects the technology with the largest h-value and updates m to this value. More precisely, let $h_{\max}$ be the largest h-value among the valid technologies. Since we set m to $h_{\max}$ at the end of the iteration, $m_{i+1} = h_{\max}$ at the beginning of the $(i+1)$-st iteration. However, any valid technology during the iteration $i+1$ will have its l-value strictly larger than m_i since all the intervals with l-value smaller or equal to m_i have been considered and m_{i+1} is set to their largest h-value. $\square$

For $1 \leq i \leq p$, define $m_i = h_{t_i}$ and $m_0 = L$. Since the technologies in S cover the interval $[L, H]$ and the sequence S is listed in nondecreasing order of l-values, we conclude that for $1 \leq i \leq p$, the sequence $t_1, \ldots, t_i$ covers the interval $[L, m_i]$ and that $m_p \geq H$. Furthermore, we have that for $1 \leq i \leq p-1$, $l_{t_i} \leq m_i$.

Claim 9.2. *The greedy algorithm outputs an optimal solution for every instance of the problem.*

Proof. Let I be an arbitrary instance with n projects. We assume the notation developed in the earlier paragraphs.

We will show that S is an optimal solution. For the sake of contradiction, assume there is a sequence S' of technologies such that the technologies in S' cover

$[L, H]$ and $|S'| < |S|$, where $|.|$ refers to the length of the sequence. We will prove by contradiction that such an S' cannot exist. Let $S' = t'_1, t'_2, \ldots, t'_q$ where $q < p$. Without loss of generality, assume that the l-values of the technologies in the sequence S' are nondecreasing. For $1 \leq i \leq q$, define $m'_i = \max(L, \max_{1 \leq j \leq i} h_{t'_j})$. and $m'_0 = L$. Since the technologies in S' cover the interval $[L, H]$ and the l-values of the sequence S' are nondecreasing, we conclude that for $1 \leq i \leq q$, the sequence $t'_1, \ldots, t'_i$ covers the interval $[L, m'_i]$ and that $m'_p \geq H$. Furthermore, we have $l_{t'_i} \leq m'_i$ for $1 \leq i \leq q - 1$

Let $i \geq 1$ be the smallest index in which the sequences S and S' differ. If the first q indices are the same, we define i to be $q + 1$. We obtain a contradiction by reverse induction on i.

If $i = q+1$, the technologies $t_1, \ldots, t_q$ cover the interval $[L, H]$, which leads to a contradiction since the greedy algorithm goes on to select another technology t_{q+1}.

Let $1 \leq i \leq q$. Let S'' be the sequence of technologies obtained from S' by exchanging the technology t'_i with t_i and deleting any succeeding technologies in S' whose l-value is lesser or equal to l_{t_i}. We argue that S'' covers $[L, H]$ and that S'' is ordered according to the l-value. Since the first $i - 1$ technologies in S and S' are the same, we have

- $m_{i-1} = m'_{i-1}$,
- $l_{t_i}, l_{t'_i} \leq m_{i-1}$, and
- $h_{t_i} \geq h_{t'_i}$.

The last inequality follows from the condition t_i is the interval with the largest h-value among all technologies j such that $l_j \leq m_{i-1}$. As a consequence, exchanging t'_i with t_i does not affect the coverage. By a similar reasoning, any technology t'_j for $j > i$ and $l_{t'_j} \leq l_{t_i} \leq m_i$ can be removed from S' without affecting the coverage. Hence, S'' covers $[L, H]$ since S' covers $[L, H]$ and S'' is ordered by nondecreasing l-values.

□

The following is an alternative proof.

Proof. If the input instance is not feasible, greedy algorithm clearly fails to output a solution. We will argue that the greedy algorithm produces an optimal solution for every feasible instance. For the sake of contradiction, assume that the claim is false. Let I be a feasible instance with n technologies for which the greedy algorithm fails to produce an optimal solution. Furthermore, let n be the smallest such integer. In other words, greedy algorithm outputs an optimal solution for all instances with fewer than n technologies.

Observe that any optimal solution for I must contain at least two technologies. Otherwise, if an optimal solution for I has only one technology, then the greedy algorithm would indeed output an optimal solution since it selects the technology with the largest h-value from among those technologies whose l-value is less or equal to L. One cannot select a better technology to cover $[L, H]$ than the technology selected by the greedy algorithm.

As before, let $S = t_1, t_2, \ldots, t_p$ be the sequence of (indices of) technologies output by the greedy algorithm in the order they were selected during the execution. Let $S' = t'_1, t'_2, \ldots, t'_{p'}$ be an optimal solution for I in the nondecreasing order of l-values. By assumption we have $p' < p$. We also have $l_{t'_1} \leq \cdots \leq l_{t'_{p'}}$. We divide the proof into two cases and reach a contradiction in each case thereby proving the claim.

Case I — $t_1 = t'_1$: We now define an instance I_1 based on I in the following fashion. Delete the technology t_1 from I and any other technologies whose l-value is less or equal to L. I_1 is required to cover the interval $[L_1, H_1]$ where $L_1 = h_{t_1}$ and $H_1 = H$. We also know that $L_1 \leq H_1$ since $|S| \geq 2$.

Let $S_1 = t_2, \ldots, t_p$, that is, S_1 is the sequence of technologies obtained from S by dropping the technology t_1. Let $S' = t'_2, \ldots, t'_{p'}$ be the sequence of technologies obtained from S' by dropping the technology t'_1. Observe that all the technologies in S'_1 are part of the instance I_1 since S' does not contain any technology which begins before L other than t'_1. Otherwise, S' would not be optimal since no such interval can extend past $h_{t'_1}$.

We claim that that S_1 is the solution output by the greedy algorithm for the instance I_1 since the set of technologies considered by the greedy algorithm after selecting t_1 on input I is exactly the set of technologies in I_1.

We also claim that S'_1 is a feasible solution for I_1. S'_1 does indeed cover the range $[L_1, H_1]$ since $L_1 = h_{t'_1} = h_{t_1}$ and S' covers $[L, H]$.

Since S_1 is optimal we get $|S_1| = p - 1 \leq p' - 1 = |S'_1|$ which leads to a contradiction.

Case II — $t_1 \neq t'_1$: We define $S'' = t_1, t'_2, \ldots, t'_{p'}$. We claim that S'' is an optimal solution for I. We then use the same argument as in Case I for S and S'' to obtain a contradiction.

We know that $l_{t'_1} \leq L$ since the technologies in S' are in nondecreasing order of their l-values and since S' is a feasible solution. However since t_1 is the technology with the largest h-value among such technologies we conclude that S'' is a feasible solution. Since S'' and S' have the same length, we conclude that S'' is optimal.

□

Problem 45: Coupons

Consider the following "coupon collector" problem. There are different varieties of cereal, and each comes with a single coupon for a discount on another box of cereal, perhaps of another variety. You can use multiple coupons when purchasing a new box, up to getting the new box free, but you never get money back. You want to buy one box of each variety, for as little money as possible. Describe an efficient algorithm, which, given as input, for each variety, its price

, the value of the enclosed coupon, and the brand for which the coupon gives a discount, computes an optimal order in which to buy the cereal.

Solution: Coupons

Algorithm.

1. Model the problem as a graph where each of the n cereal boxes form the vertices of the graph. A directed edge with weight w is present from node i to node j if cereal box i gives a discount of w for box j. Observe that each node in the graph has outdegree equal to one. Label each node with the price of the cereal box. Let p_v denote the price of the node v and let w_{uv} denote the discount offered by box u towards the purchase of box v.
2. As long as there is a node u with no incoming edges, buy the box represented by the node, delete the unique outgoing edge (u, v), and update the price of v to be $\max(0, p_v - w_{uv})$.
3. At this point since we have no nodes with indegree zero and every node has outdegree 1, the graph must be a union of disjoint cycles.
4. The effective discount of any node u in a cycle is $\min(p_u, w_{vu})$ where (v, u) is the unique incoming edge into u. For each cycle, purchase the nodes in the same order as given by the cycle starting with the node with the smallest effective discount.

Correctness. In the first case, where box X has no coupons for it, let S be any schedule which does not buy box X first. Then let S' be the schedule where we buy X first, and then buy the other boxes in the same order as in S. Since X had no coupon, we pay the same price for X in both orders, namely full price. Now, for all other boxes, the price in S' is at most the price in S, the only difference being the box we got a coupon for by buying X, which may have dropped in price. Therefore, the total cost of S' is at most that of S.

In the second case, where we have a cycle, then the box X we buy is the one with the minimum real discount value. Again let S be any schedule that does not buy X first, and define S' as first buying all the boxes in X's cycle in order starting from X and then buying the rest as in S. Let Y be the first box in X's cycle bought in S. No box outside of X's cycle has changed price. For each box in the cycle except X and Y , we pay the full price - real value of the coupon in S', and at least that in S, since there is only one coupon. The price of X in S' may have increased over that in S by the real value of its coupon, but the price of Y has decreased: i.e in S, we pay full price but in S' we deduct the real value of $Y's$ coupon. Since X has the smallest real coupon value, this means the total price for S' is at most that of S.

Thus in both cases, we see that the greedy strategy does at least as well as the so called optimal strategy if not better and hence this algorithm is correct.

Time complexity. All the precomputation like the number of coupons for each box etc. takes $O(n)$. For all boxes with no coupons we do constant number of operations and number of such boxes is at most n. Therefore this step also takes $O(n)$. Now for the cycles, we randomly start at any point in a cycle and we go through the cycle at most twice. The cycle length can be at most only n. Moreover we do only constant-time operations for each node in the cycle. Therefore time complexity for this part of algorithm is $O(n)$.

Therefore total time complexity for the algorithm is $O(n)$.

Problem 46: Scheduling to minimize the number of rooms

Let $E = \{(s_i, f_i) | 1 \le i \le n\}$ be a set of events to be scheduled in as few rooms as possible without conflicts. Find a schedule that minimizes the number of rooms used in the schedule.

Solution: Scheduling to minimize the number of rooms

Solution Sketch: Assume that the sequence $\{s_i\}$ is nondecreasing. If not, sort it in $O(n \log n)$ time. For room $l \ge 1$, let S_j^l denote the set of events scheduled in room l at the end of step j of the algorithm. For $l \ge 1$, let F_j^l denote the largest finish time of the events scheduled in room l at the end of step j. $S_0^l = \varnothing$ and $F_0^l = -\infty$ for all $l \ge 1$.

Assume that at the end of step j, the algorithm produces the partial schedule $\{S_j^l\}$. Let $\{F_j^l\}$ denote the corresponding largest finish times. Our inductive hypothesis is that the partial schedule $\{S_j^l\}$ is extendable to an optimal schedule $\{T^l\}$ where for all additional events $(s_i, f_i) \in T^l - S_j^l$, $s_i \ge F_j^l$.

In step $j+1$, the algorithm processes the event (s_{j+1}, f_{j+1}) and schedules it in room l_1 where l_1 is the smallest l for which $s_{j+1} \ge F_j^l$. Set $S_{j+1}^{l_1} = S_j^{l_1} \cup \{(s_{j+1}, f_{j+1})\}$ and $S_{j+1}^l = S_j^l$ for all $l \ne l_1$. Also set $F_{j+1}^{l_1} = f_{j+1}$ and $F_{j+1}^l = F_j^l$ for all $l \ne l_1$. $\{S_{j+1}^l\}$ is the partial schedule at the end of step $j+1$.

We need to show that the partial schedule $\{S_{j+1}^l\}$ satisfies the inductive hypothesis. For this, we will exhibit an appropriate extension. Let $\{T^l\}$ be the extension guaranteed by the inductive hypothesis where $S_j^l \subseteq T^l$ for all $l \ge 1$ and for each $(s_i, f_i) \in T^l - S_j^l$, $s_i \ge F_j^l$. Let l_2 be the room in which the event (s_{j+1}, f_{j+1}) is scheduled in $\{T^l\}$, that is, $(s_{j+1}, f_{j+1}) \in T^{l_2}$. If $l_2 = l_1$, $\{T^l\}$ is the desired extension. Verify this. This is where you need to use the fact that $\{s_i\}$ is a nondecreasing sequence.

If not, $l_2 > l_1$ since l_1 is the smallest l such that $s_{j+1} \ge F_j^l$. Let $R^{l_2} \subseteq T^{l_2}$ be the set of events (s_i, f_i) such that $s_i \ge s_{j+1}$. That is, R^{l_2} is the set events in room l_2 which start on or after s_{j+1} according to schedule $\{T^l\}$. Also let $R^{l_1} \subseteq T^{l_1}$ be the set of events (s_i, f_i) such that $s_i \ge F_j^l$. In other words, $R^{l_1} = T^l - S_j^l$.

Let $\{U^l\}$ be a new schedule where $U^{l_1} = S_j^{l_1} \cup R^{l_2}$, $U^{l_2} = (T^{l_2} - R^{l_2}) \cup R^{l_1}$ and $U^l = T^l$ for all other l. Verify that the events in U^{l_1} are not conflicting among themselves. Similarly verify that the events in U^{l_2} are not in conflict. Proceed then to prove that U^l is an extension of the partial schedule S^l_{j+1} satisfying the inductive hypothesis. Also verify that the number of nonempty rooms in $\{U^l\}$ is less or equal to the number of nonempty rooms in $\{T^l\}$. Where did we use the fact that the next event is scheduled in the room with the smallest index?