Textbook DP Problems

Sijian Tan

1 Problem 1

Let $G = (V, E)$ be an undirected graph with n nodes. Recall that a subset of the nodes is called an *independent set* if no two of them are joined by an edge. Finding large independent sets is difficult in general; but here we'll see that it can be done efficiently if the graph is "simple" enough.

Call a graph $G = (V, E)$ a *path* if its nodes can be written as $v_1, v_2, \ldots, v_n$, with an edge between v_i and v_j if and only if the numbers i and j differ by exactly 1. With each node v_i, we associate a positive integer weight w_i.

Consider, for example, the five-node path drawn in Figure 6.28. The *weights* are the numbers drawn inside the nodes.

The goal in this question is to solve the following problem:

Find an independent set in a path G whose total weight is as large as possible.

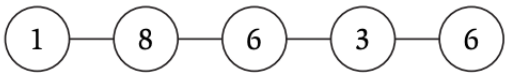

Figure 6.28 A paths with weights on the nodes. The maximum weight of an independent set is 14.

Figure 1: Problem 1

Give an algorithm that takes an n-node path G with weights and returns an independent set of maximum total weight. The running time should be polynomial in n, independent of the values of the weights.

1.1 Strategy

Don't overthink this problem. It is just about whether we choose the i^{th} element. If we choose this, then we could not choose the $(i-1)^{th}$ element. If we don't choose this, then we could consider the previous $(i-1)$ elements.

1.2 Recurrence Relation

Subproblem: Define $OPT[i]$ be the maximum weight we could get up to i^{th} element.

Recurrence relation could be expressed as:

$$OPT[i] = \max(OPT[i-1], w_i + OPT[i-2]) \tag{1}$$

The base cases include:

$$OPT[1] = w_1, \;\; OPT[2] = \max(w_1, w_2) \tag{2}$$

2 Problem 2

If you select a low-stress job for your team in week i, then you get a revenue of $\ell_i > 0$ dollars; if you select a high-stress job, you get a revenue of $h_i > 0$ dollars. The catch, however, is that in order for the team to take on a high-stress job in week i, it's required that they do no job (of either type) in week $i-1$; they need a full week of prep time to get ready for the crushing stress level. On the other hand, it's okay for them to take a low-stress job in week i even if they have done a job (of either type) in week $i-1$.

So, given a sequence of n weeks, a *plan* is specified by a choice of "low-stress," "high-stress," or "none" for each of the n weeks, with the property that if "high-stress" is chosen for week $i > 1$, then "none" has to be chosen for week $i-1$. (It's okay to choose a high-stress job in week 1.) The *value* of the plan is determined in the natural way: for each i, you add ℓ_i to the value if you choose "low-stress" in week i, and you add h_i to the value if you choose "high-stress" in week i. (You add 0 if you choose "none" in week i.)

Problem: Given sets of values $\ell_1, \ell_2, \ldots, \ell_n$ and $h_1, h_2, \ldots, h_n$, find a plan of maximum value. (Such a plan will be called *optimal*.)

Example: Suppose $n = 4$, and the values of ℓ_i and h_i are given by the following table. Then the plan of maximum value would be to choose "none" in week 1, a high-stress job in week 2, and low-stress jobs in weeks 3 and 4. The value of this plan would be $0 + 50 + 10 + 10 = 70$.

	Week 1	Week 2	Week 3	Week 4
ℓ	10	1	10	10
h	5	50	5	1

Figure 2: Problem 2

Give an efficient algorithm that takes values for $\ell_1, \ell_2, \ldots, \ell_n$ and $h_1, h_2, \ldots, h_n$ and returns the *value* of an optimal plan.

2.1 Strategy

Very straightforward binary choice (compatible) problem.

2.2 Recurrence Relation

Subproblem: Define $OPT[i]$ be the maximum value we could get up to i^{th} week.
Recurrence relation could be expressed as:

$$OPT[i] = \max(OPT[i-1] + l_i, OPT[i-2] + h_i) \tag{3}$$

And the base cases include:

$$OPT[1] = \max(l_1, h_1) \tag{4}$$

$$OPT[2] = \max(OPT[1] + l_2, h_2) \tag{5}$$

3 Problem 3

Let $G = (V, E)$ be a directed graph with nodes $v_1, \ldots, v_n$. We say that G is an *ordered graph* if it has the following properties.

(i) Each edge goes from a node with a lower index to a node with a higher index. That is, every directed edge has the form (v_i, v_j) with $i < j$.

(ii) Each node except v_n has at least one edge leaving it. That is, for every node v_i, $i = 1, 2, \ldots, n-1$, there is at least one edge of the form (v_i, v_j).

The length of a path is the number of edges in it. The goal in this question is to solve the following problem (see Figure 6.29 for an example).

Given an ordered graph G, find the length of the longest path that begins at v_1 and ends at v_n.

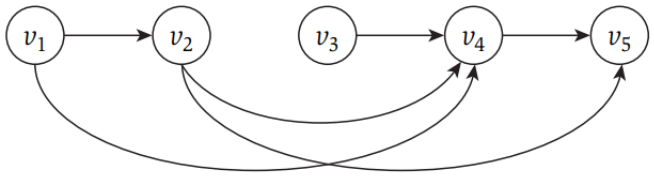

Figure 6.29 The correct answer for this ordered graph is 3: The longest path from v_1 to v_n uses the three edges (v_1, v_2),(v_2, v_4), and (v_4, v_5).

Figure 3: Problem 3

Give an efficient algorithm that takes an ordered graph G and returns the *length* of the longest path that begins at v_1 and ends at v_n. (Again, the *length* of a path is the number of edges in the path.)

3.1 Strategy

When we consider the longest path to a specific node, we need to consider all the previous nodes connected to this node, and then just plus one. Notice that there may be some nodes do not have previous nodes connected, we also need to consider this case.

3.2 Recurrence Relation

Subproblem: Define $OPT[i]$ be the longest path to reach i^{th} vertex.
Recurrence relation could be expressed as:

$$OPT[i] = \max(OPT[j] + 1) \tag{6}$$

Here, j will be the indices of all vertices go into i^{th} vertex. If no vertex is connected to i^{th} vertex, then just set $OPT[i] = 0$.

4 Problem 4

Suppose you're running a lightweight consulting business—just you, two associates, and some rented equipment. Your clients are distributed between the East Coast and the West Coast, and this leads to the following question.

Each month, you can either run your business from an office in New York (NY) or from an office in San Francisco (SF). In month i, you'll incur an *operating cost* of N_i if you run the business out of NY; you'll incur an operating cost of S_i if you run the business out of SF. (It depends on the distribution of client demands for that month.)

However, if you run the business out of one city in month i, and then out of the other city in month $i+1$, then you incur a fixed *moving cost* of M to switch base offices.

Given a sequence of n months, a *plan* is a sequence of n locations—each one equal to either NY or SF—such that the ith location indicates the city in which you will be based in the ith month. The *cost* of a plan is the sum of the operating costs for each of the n months, plus a moving cost of M for each time you switch cities. The plan can begin in either city.

Problem: Given a value for the moving cost M, and sequences of operating costs $N_1, \ldots, N_n$ and $S_1, \ldots, S_n$, find a plan of minimum cost. (Such a plan will be called *optimal*.)

Example: Suppose $n = 4$, $M = 10$, and the operating costs are given by the following table.

	Month 1	Month 2	Month 3	Month 4
NY	1	3	20	30
SF	50	20	2	4

Then the plan of minimum cost would be the sequence of locations

$$[NY, NY, SF, SF],$$

with a total cost of $1 + 3 + 2 + 4 + 10 = 20$, where the final term of 10 arises because you change locations once.

Figure 4: Problem 4

Give an efficient algorithm that takes values for n, M, and sequences of operating costs $N_1, \ldots, N_n$ and $S_1, \ldots, S_n$, and returns the *cost* of an optimal plan.

4.1 Strategy

This is a very classical two ends question. We need to consider the cases ending with different situation, and combine them together.

4.2 Recurrence Relation

Subproblem: Define $OPT[i]$ as the minimum cost up to i^{th} month. Define $OPT_N[i]$ as the minimum cost up to i^{th} month and ending with NY. Define $OPT_S[i]$ as the minimum cost up to i^{th} month and ending with SF.

Recurrence Relation could be expressed as:

$$OPT_N[i] = N_i + \min(OPT_N[i-1], OPT_S[i-1] + M) \tag{7}$$

$$OPT_S[i] = S_i + \min(OPT_S[i-1], OPT_N[i-1] + M) \tag{8}$$

$$OPT[i] = \min(OPT_N[i], OPT_S[i]) \tag{9}$$

5 Problem 6

In a word processor, the goal of "pretty-printing" is to take text with a ragged right margin, like this:

```
Call me Ishmael.
Some years ago,
never mind how long precisely,
having little or no money in my purse,
and nothing particular to interest me on shore,
I thought I would sail about a little
and see the watery part of the world.
```

Figure 5: Ragged Text

and turn it into text whose right margin is as "even" as possible, like this:

```
Call me Ishmael. Some years ago, never
mind how long precisely, having little
or no money in my purse, and nothing
particular to interest me on shore, I
thought I would sail about a little
and see the watery part of the world.
```

Figure 6: Even Text

To make this precise enough for us to start thinking about how to write a pretty-printer for text, we need to figure out what it means for the right margins to be "even." So suppose our text consists of a sequence of words, $W = \{w_1, w_2, \ldots, w_n\}$, where w_i consists of c_i characters. We have a maximum line length of L. We will assume we have a fixed-width font and ignore issues of punctuation or hyphenation.

A *formatting* of W consists of a partition of the words in W into *lines*. In the words assigned to a single line, there should be a space after each word except the last; and so if $w_j, w_{j+1}, \ldots, w_k$ are assigned to one line, then we should have

$$\left[\sum_{i=j}^{k-1}(c_i + 1)\right] + c_k \leq L. \tag{10}$$

We will call an assignment of words to a line *valid* if it satisfies this inequality. The difference between the left-hand side and the right-hand side will be called the *slack* of the line—that is, the number of spaces left at the right margin.

Give an efficient algorithm to find a partition of a set of words W into valid lines, so that the sum of the *squares of the slacks* of all lines (including the last line) is minimized.

5.1 Strategy

This is a very hard problem, but it is actually similar with the bookself problem in previous Quals. We don't need to actually care about the lines, let DP care about it.

We only need to care about the last line, and consider the addition of the slacks.

5.2 Recurrence Relation

Subproblem: Define $OPT[i]$ to be the minimum spaces of the optimal solution on the set of words $W_i = [w_1, w_2, ..., w_i]$.

Recurrence Relation: For any $a \leq b$, let S_{ab} denote the **slack of a line containing the words** $w_a, w_{a+1}, ..., w_b$. Define $S_{a,b} = \infty$ if these words exceed total length L.

The optimal solution must begin the last line somewhere (at word w_j), and solve the sub-problem on the earlier lines optimally:

$$OPT[i] = \min_{1 \leq j \leq i}(S_{j,i}^2 + OPT[j-1]) \tag{11}$$

6 Problem 7

As a solved exercise in Chapter 5, we gave an algorithm with $O(n \log n)$ running time for the following problem. We're looking at the price of a given stock over n consecutive days, numbered $i = 1, 2, \ldots, n$. For each day i, we have a price $p(i)$ per share for the stock on that day. (We'll assume for simplicity that the price was fixed during each day.) We'd like to know: How should we choose a day i on which to buy the stock and a later day $j > i$ on which to sell it, if we want to maximize the profit per share, $p(j) - p(i)$? (If there is no way to make money during the n days, we should conclude this instead.)

In the solved exercise, we showed how to find the optimal pair of days i and j in time $O(n \log n)$. But, in fact, it's possible to do better than this. Show how to find the optimal numbers i and j in time $O(n)$.

6.1 Strategy and Recurrence

This is a very inspiring question. Let $OPT[i]$ denote the maximum possible return the investors can make if they sell the stock on day i. Node that $OPT[1] = 0$. Now, in the optimal way of selling the stock on day i, **the investors were either holding it on day $i-1$ or there were not.** If they did not hold it, then $OPT[i] = 0$. If they held it, then $OPT[i] = OPT[i-1] + (p(i) - p(i-1))$. Thus, we have:

$$OPT[i] = \max(0, OPT[i-1] + (p(i) - p(i-1))) \tag{12}$$

7 Problem 8

The residents of the underground city of Zion defend themselves through a combination of kung fu, heavy artillery, and efficient algorithms. Recently they have become interested in automated methods that can help fend off attacks by swarms of robots.

Here's what one of these robot attacks looks like.

- A swarm of robots arrives over the course of n seconds; in the ith second, x_i robots arrive. Based on remote sensing data, you know this sequence $x_1, x_2, \ldots, x_n$ in advance.
- You have at your disposal an *electromagnetic pulse* (EMP), which can destroy some of the robots as they arrive; the EMP's power depends on how long it's been allowed to charge up. To make this precise, there is a function $f(\cdot)$ so that if j seconds have passed since the EMP was last used, then it is capable of destroying up to $f(j)$ robots.
- So specifically, if it is used in the kth second, and it has been j seconds since it was previously used, then it will destroy $\min(x_k, f(j))$ robots. (After this use, it will be completely drained.)
- We will also assume that the EMP starts off completely drained, so if it is used for the first time in the jth second, then it is capable of destroying up to $f(j)$ robots.

The problem: Given the data on robot arrivals $x_1, x_2, \ldots, x_n$, and given the recharging function $f(\cdot)$, choose the points in time at which you're going to activate the EMP so as to destroy as many robots as possible.

Example: Suppose $n = 4$, and the values of x_i and $f(i)$ are given by the following table.

i	**1**	**2**	**3**	**4**
x_i	1	10	10	1
$f(i)$	1	2	4	8

Figure 7: Problem 8

The best solution would be to activate the EMP in the 3rd and the 4th seconds. In the 3rd second, the EMP has gotten to charge for 3 seconds, and so it destroys $\min(10, 4) = 4$ robots; In the 4th second, the EMP has only gotten to charge for 1 second since its last use, and it destroys $\min(1, 1) = 1$ robot. This is a total of 5.

Give an efficient algorithm that takes the data on robot arrivals $x_1, x_2, \ldots, x_n$, and the recharging function $f(\cdot)$, and returns the maximum number of robots that can be destroyed by a sequence of EMP activations.

7.1 Strategy

To determine when to charge the EMP is the same as where to cut the string (quality problem), so it is actually a segmentation problem.

7.2 Recurrence Relation

Subproblem: Define $OPT[i]$ as the maximum robots we could destroy up to i^{th} second.

Recurrence Relation could be expressed as:

$$OPT[i] = \max_{1 \le j \le i}(OPT[j] + \min(x_i, f(i-j))) \tag{13}$$

8 Problem 9

You're helping to run a high-performance computing system capable of processing several terabytes of data per day. For each of n days, you're presented with a quantity of data; on day i, you're presented with x_i terabytes. For each terabyte you process, you receive a fixed revenue, but any unprocessed data becomes unavailable at the end of the day (i.e., you can't work on it in any future day).

You can't always process everything each day because you're constrained by the capabilities of your computing system, which can only process a fixed number of terabytes in a given day. In fact, it's running some one-of-a-kind software that, while very sophisticated, is not totally reliable, and so the amount of data you can process goes down with each day that passes since the most recent reboot of the system. On the first day after a reboot, you can process s_1 terabytes, on the second day after a reboot, you can process s_2 terabytes, and so on, up to s_n; we assume $s_1 > s_2 > s_3 > \cdots > s_n > 0$. (Of course, on day i you can only process up to x_i terabytes, regardless of how fast your system is.) To get the system back to peak performance, you can choose to reboot it; but on any day you choose to reboot the system, you can't process any data at all.

The problem. Given the amounts of available data $x_1, x_2, \ldots, x_n$ for the next n days, and given the profile of your system as expressed by $s_1, s_2, \ldots, s_n$ (and starting from a freshly rebooted system on day 1), choose the days on which you're going to reboot so as to maximize the total amount of data you process.

Example. Suppose $n = 4$, and the values of x_i and s_i are given by the following table.

	Day 1	Day 2	Day 3	Day 4
x	10	1	7	7
s	8	4	2	1

Figure 8: Problem 9

The best solution would be to reboot on day 2 only; this way, you process 8 terabytes on day 1, then 0 on day 2, then 7 on day 3, then 4 on day 4, for a total of 19. (Note that if you didn't reboot at all, you'd process $8 + 1 + 2 + 1 = 12$; and other rebooting strategies give you less than 19 as well.)

Give an efficient algorithm that takes values for $x_1, x_2, \ldots, x_n$ and $s_1, s_2, \ldots, s_n$ and returns the total *number* of terabytes processed by an optimal solution.

8.1 Strategy

To determine when is the best time to reboot is the same as the segmentation problem, and to determine whether we do the reboot or not is the same as the binary

choice problem. But make sure to use correct indices to cover all the calculations.

8.2 Recurrence Relation

Subproblem: Define $OPT[i]$ as the maximum terabytes can be processed up to i^{th} day.

Recurrence relation could be expressed as:

$$OPT[i] = \max_{1 \le j < i}(OPT[i-j-1] + \sum_{k=1}^{j} \min(s_k, x_{i-j+k}), OPT[i-1] + \min(s_i, x_i)) \quad (14)$$

9 Problem 10

Here's the problem you face. Your job can only run on one of the machines in any given minute. Over each of the next n minutes, you have a "profile" of how much processing power is available on each machine. In minute i, you would be able to run $a_i > 0$ steps of the simulation if your job is on machine A, and $b_i > 0$ steps of the simulation if your job is on machine B. You also have the ability to move your job from one machine to the other; but doing this costs you a minute of time in which no processing is done on your job.

So, given a sequence of n minutes, a *plan* is specified by a choice of A, B, or "move" for each minute, with the property that choices A and B cannot appear in consecutive minutes. For example, if your job is on machine A in minute i, and you want to switch to machine B, then your choice for minute $i + 1$ must be *move*, and then your choice for minute $i + 2$ can be B. The *value* of a plan is the total number of steps that you manage to execute over the n minutes: so it's the sum of a_i over all minutes in which the job is on A, plus the sum of b_i over all minutes in which the job is on B.

The problem. Given values $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$, find a plan of maximum value. (Such a strategy will be called *optimal*.) Note that your plan can start with either of the machines A or B in minute 1.

Example. Suppose $n = 4$, and the values of a_i and b_i are given by the following table.

	Minute 1	Minute 2	Minute 3	Minute 4
A	10	1	1	10
B	5	1	20	20

Figure 9: Problem 10

Then the plan of maximum value would be to choose A for minute 1, then *move* for minute 2, and then B for minutes 3 and 4. The value of this plan would be $10 + 0 + 20 + 20 = 50$.

Give an efficient algorithm that takes values for $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$ and returns the *value* of an optimal plan.

9.1 Strategy

This is also a very classical end with two cases problem.

9.2 Recurrence Relation

Subproblem: Define $OPT[i]$ as the maximum values up to i^{th} minute. Define $OPT_A[i]$ as the maximum values up to i^{th} minute and ending with machine A. Define $OPT_B[i]$ as the maximum values up to i^{th} minute and ending with machine B.

Recurrence relation could be expressed as:

$$OPT_A[i] = \max(OPT_A[i-1] + a_i, OPT_B[i-2] + a_i) \tag{15}$$

$$OPT_B[i] = \max(OPT_B[i-1] + b_i, OPT_A[i-2] + b_i) \tag{16}$$

$$OPT[i] = \max(OPT_A[i], OPT_B[i]) \tag{17}$$

10 Problem 15

On most clear days, a group of your friends in the Astronomy Department gets together to plan out the astronomical events they're going to try observing that night. We'll make the following assumptions about the events.

- There are n events, which for simplicity we'll assume occur in sequence separated by exactly one minute each. Thus event j occurs at minute j; if they don't observe this event at exactly minute j, then they miss out on it.
- The sky is mapped according to a one-dimensional coordinate system (measured in degrees from some central baseline); event j will be taking place at coordinate d_j, for some integer value d_j. The telescope starts at coordinate 0 at minute 0.
- The last event, n, is much more important than the others; so it is required that they observe event n.

The Astronomy Department operates a large telescope that can be used for viewing these events. Because it is such a complex instrument, it can only move at a rate of one degree per minute. Thus they do not expect to be able to observe all n events; they just want to observe as many as possible, limited by the operation of the telescope and the requirement that event n must be observed.

We say that a subset S of the events is *viewable* if it is possible to observe each event $j \in S$ at its appointed time j, and the telescope has adequate time (moving at its maximum of one degree per minute) to move between consecutive events in S.

The problem. Given the coordinates of each of the n events, find a viewable subset of maximum size, subject to the requirement that it should contain event n. Such a solution will be called *optimal*.

Example. Suppose the one-dimensional coordinates of the events are as shown here.

Event	1	2	3	4	5	6	7	8	9
Coordinate	1	−4	−1	4	5	−4	6	7	−2

Figure 10: Problem 15

Then the optimal solution is to observe events 1, 3, 6, 9. Note that the telescope has time to move from one event in this set to the next, even moving at one degree per minute.

Give an efficient algorithm that takes values for the coordinates $d_1, d_2, \ldots, d_n$ of the events and returns the *size* of an optimal solution.

10.1 Strategy

The relation is straightforward, if the last event satisfies the condition, then we just add one.

10.2 Recurrence Relation

Subproblem: Define $OPT[i]$ as the maximum number of event up to i^{th} minutes.
Recurrence relation could be expressed as:

$$OPT[i] = 1 + \max_{1 \leq j < i}(OPT[j]) \tag{18}$$

Here j must satisfy the following relation:

$$|d_i - d_j| \leq i - j \tag{19}$$

11 Problem 16

There are many sunny days in Ithaca, New York; but this year, as it happens, the spring ROTC picnic at Cornell has fallen on a rainy day. The ranking officer decides to postpone the picnic and must notify everyone by phone. Here is the mechanism she uses to do this.

Each ROTC person on campus except the ranking officer reports to a unique *superior officer*. Thus the reporting hierarchy can be described by a tree T, rooted at the ranking officer, in which each other node v has a parent node u equal to his or her superior officer. Conversely, we will call v a *direct subordinate* of u. See Figure 6.30, in which A is the ranking officer, B and D are the direct subordinates of A, and C is the direct subordinate of B.

To notify everyone of the postponement, the ranking officer first calls each of her direct subordinates, one at a time. As soon as each subordinate gets the phone call, he or she must notify each of his or her direct subordinates, one at a time. The process continues this way until everyone has been notified. Note that each person in this process can only call direct subordinates on the phone; for example, in Figure 6.30, A would not be allowed to call C.

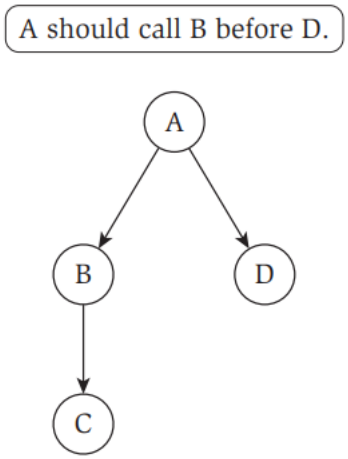

Figure 6.30 A hierarchy with four people. The fastest broadcast scheme is for A to call B in the first round. In the second round, A calls D and B calls C. If A were to call D first, then C could not learn the news until the third round.

Figure 11: Problem 16

11.1 Strategy and Solution

The ranking officer must notify her subordinates in some sequence, after which they will recursively broadcast the message as quickly as possible to their subtrees. This is just like the homework problem on triathalon scheduling from the chapter on greedy algorithms: the subtrees must be "started" one at a time, after which they complete recursively in parallel. Using the solution to that problem, she should talk to the subordinates in decreasing order of the time it takes for their subtrees (recursively) to be notified.

Hence, we have the following set of sub-problems: for each subtree T' of T, we define $x(T')$ to be the number of rounds it takes for everyone in T' to be notified, once the root has the message. Suppose now that T' has child subtrees $T_1, \ldots, T_k$, and we label them so that $x(T_1) \geq x(T_2) \geq \cdots \geq x(T_k)$. Then by the argument in the above paragraph, we have the recurrence

$$x(T') = \min_j \left[j + x(T_j) \right].$$

If T' is simply a leaf node, then we have $x(T') = 0$.

Figure 12: Problem 16 Solution

12 Problem 17

Your friends have been studying the closing prices of tech stocks, looking for interesting patterns. They've defined something called a *rising trend*, as follows.

They have the closing price for a given stock recorded for n days in succession; let these prices be denoted $P[1], P[2], \ldots, P[n]$. A rising trend in these prices is a subsequence of the prices $P[i_1], P[i_2], \ldots, P[i_k]$, for days $i_1 < i_2 < \ldots < i_k$, so that

- $i_1 = 1$, and
- $P[i_j] < P[i_{j+1}]$ for each $j = 1, 2, \ldots, k-1$.

Thus a rising trend is a subsequence of the days—beginning on the first day and not necessarily contiguous—so that the price strictly increases over the days in this subsequence.

They are interested in finding the longest rising trend in a given sequence of prices.

Example. Suppose $n = 7$, and the sequence of prices is

$$10, 1, 2, 11, 3, 4, 12.$$

Then the longest rising trend is given by the prices on days 1, 4, and 7. Note that days 2, 3, 5, and 6 consist of increasing prices; but because this subsequence does not begin on day 1, it does not fit the definition of a rising trend.

12.1 Strategy

This problem is also a straightforward binary problem. We either choose this value (then we need to start from the last compatible one) or not choose this value.

12.2 Recurrence Relation

Subproblem: Define $OPT[i]$ as number of longest rising trend up to i^{th} day.

Recurrence Relation could be expressed as:

$$OPT[i] = \max(OPT[j] + 1, OPT[i-1]) \tag{20}$$

Here, j is the index of largest previous value smaller than $p[i]$, such that $p[j] < p[i]$.

13 Problem 20

Suppose it's nearing the end of the semester and you're taking n courses, each with a final project that still has to be done. Each project will be graded on the following scale: It will be assigned an integer number on a scale of 1 to $g > 1$, higher numbers being better grades. Your goal, of course, is to maximize your average grade on the n projects.

You have a total of $H > n$ hours in which to work on the n projects cumulatively, and you want to decide how to divide up this time. For simplicity, assume H is a positive integer, and you'll spend an integer number of hours on each project. To figure out how best to divide up your time, you've come up with a set of functions $\{f_i : i = 1, 2, \ldots, n\}$

(rough estimates, of course) for each of your n courses; if you spend $h \leq H$ hours on the project for course i, you'll get a grade of $f_i(h)$. (You may assume that the functions f_i are *nondecreasing*: if $h < h'$, then $f_i(h) \leq f_i(h')$.)

So the problem is: Given these functions $\{f_i\}$, decide how many hours to spend on each project (in integer values only) so that your average grade, as computed according to the f_i, is as large as possible. In order to be efficient, the running time of your algorithm should be polynomial in n, g, and H; none of these quantities should appear as an exponent in your running time.

13.1 Strategy

This question is a classical two constraints problem. We need to maximize the grade, but at the same time the total hours could not be larger than H.

13.2 Recurrence Relation

Subproblem: We define $OPT[i, H']$ as the maximum grades we could get up to i^{th} course, and with the remaining hours as H'.

Recurrence Relation could be expressed as:

$$OPT[i, H'] = \max_{0 \leq h \leq H'}\{OPT[i-1, H'-h] + f(h)\} \tag{21}$$

14 Problem 21

Some time back, you helped a group of friends who were doing simulations for a computation-intensive investment company, and they've come back to you with a new problem. They're looking at n consecutive days of a given stock, at some point in the past. The days are numbered $i = 1, 2, \ldots, n$; for each day i, they have a price $p(i)$ per share for the stock on that day.

For certain (possibly large) values of k, they want to study what they call *k-shot strategies*. A k-shot strategy is a collection of m pairs of days $(b_1, s_1), \ldots, (b_m, s_m)$, where $0 \leq m \leq k$ and

$$1 \leq b_1 < s_1 < b_2 < s_2 < \cdots < b_m < s_m \leq n.$$

We view these as a set of up to k nonoverlapping intervals, during each of which the investors buy 1,000 shares of the stock (on day b_i) and then sell it (on day s_i). The *return* of a given k-shot strategy is simply the profit obtained from the m buy-sell transactions, namely,

$$1000 \sum_{i=1}^{m} \left(p(s_i) - p(b_i)\right).$$

The investors want to assess the value of k-shot strategies by running simulations on their n-day trace of the stock price. Your goal is to design an efficient algorithm that determines, given the sequence of prices, the k-shot strategy with the maximum possible return. Since k may be relatively large in these simulations, your running time should be polynomial in both n and k; it should not contain k in the exponent.

14.1 Strategy and Solution

By *transaction* (i, j), we mean the single transaction that consists of buying on day i and selling on day j. Let $P[i, j]$ denote the monetary return from transaction (i, j). Let $Q[i, j]$ denote the maximum profit obtainable by executing a single transaction somewhere in the interval of days between i and j. Note that the transaction achieving the maximum in $Q[i, j]$ is either the transaction (i, j), or else it fits into one of the intervals $[i, j-1]$ or $[i+1, j]$. Thus we have

$$Q[i, j] = \max\{P[i, j], Q[i, j-1], Q[i+1, j]\}.$$

Using this formula, we can build up all values of $Q[i, j]$ in time $O(n^2)$. (By going in order of increasing $i + j$, spending constant time per entry.)

Now, let us say that an *m-exact strategy* is one with exactly m non-overlapping buy-sell transactions. Let $M[m, d]$ denote the maximum profit obtainable by an m-exact strategy on days $1, \ldots, d$, for $0 \leq m \leq k$ and $0 \leq d \leq n$. We will use $-\infty$ to denote the profit obtainable if there isn't room in days $1, \ldots, d$ to execute m transactions. (E.g. if $d < 2m$.) We can initialize $M[m, 0] = -\infty$ and $M[0, d] = 0$ for each m and each d.

In the optimal m-exact strategy on days $1, \ldots, d$, the final transaction occupies an interval that begins at i and ends at j, for some $1 \leq i \leq j \leq d$; and up to day $i - 1$ we then have an $(m-1)$-exact strategy. Thus we have

$$M[m, d] = \max_{1 \leq i < j \leq d} \{Q[i, j] + M[m-1, i-1]\}.$$

We can fill in these entries in order of increasing $m + d$. The time spent per entry is $O(n)$, since we've already computed all $Q[i, j]$. Since there are $O(kn)$ entries, the total time is therefore $O(kn^2)$. We can determine the strategy associated with each entry by maintaining a pointer to the entry that produced the maximum, and tracing back through the dynamic programming table using these pointers.

Finally, the optimal k-shot strategy is, by definition, an m-exact strategy for some $m \leq k$; thus, the optimal profit from a k-shot strategy is

$$\max_{0 \leq m \leq k} M[m, n].$$

15 Problem 22

To assess how "well-connected" two nodes in a directed graph are, one can not only look at the length of the shortest path between them, but can also count the *number* of shortest paths.

This turns out to be a problem that can be solved efficiently, subject to some restrictions on the edge costs. Suppose we are given a directed graph $G = (V, E)$, with costs on the edges; the costs may be positive or negative, but every cycle in the graph has strictly positive cost. We are also given two nodes $v, w \in V$. Give an efficient algorithm that computes the number of shortest v-w paths in G. (The algorithm should not list all the paths; just the number suffices.)

15.1 Strategy and Solution

Let c_e denote the cost of the edge e and we will overload the notation and write c_{st} to denote the cost of the edge between the nodes s and t.

This problem is by its nature quite similar to the shortest path problem. Let us consider a two-parameter function $Opt(i, s)$ denoting the optimal cost of the shortest path to s using exactly i edges, and let $N(i, s)$ denote the number of such paths.

We start by setting $Opt(i, v) = 0$ and $Opt(i, v') = \infty$ for all $v' \neq v$. Also set $N(i, v) = 1$ and $N(i, v') = 0$ for all $v' \neq v$. Intuitively this means that the source v is reachable with cost 0 and there is currently one path to achieve this.

Then we compute the following recurrence:

$$Opt(i, s) = \min_{t,(t,s)\in E}\{Opt(i-1, t) + c_{ts}\}.$$

The above recurrence means that in order to travel to node s using exactly i edges, we must travel to a predecessor node t using exactly $i - 1$ edges and then take the edge connecting t to s. Once of course the optimal cost value has been computed, the number of paths that achieve this optimum would be computed by the following recurrence:

$$N(i, s) = \sum_{\substack{t,(t,s)\in E \\ \text{and } Opt(i,s)=Opt(i-1,t)+c_{ts}}} N(i-1, t).$$

In other words, we look at all the predecessors from which the optimal cost path may be achieved and add all the counters.

The above recurrences can be calculated by a double loop, where the outside loops over i and the inside loops over all the possible nodes s. Once the recurrences have been solved, our target optimal path to w is obtained by taking the minimum of all the paths of different lengths to w—that is:

$$Opt_{final}(w) = \min_i\{Opt(i, w)\}.$$

And the number of such paths can be computed by adding up the counters of all the paths which achieve the minimal cost.

$$N_{final}(w) = \sum_{i,Opt(i,w)=Opt(w)} N(i, w).$$

16 Problem 25

Consider the problem faced by a stockbroker trying to sell a large number of shares of stock in a company whose stock price has been steadily falling in value. It is always hard to predict the right moment to sell stock, but owning a lot of shares in a single company adds an extra complication: the mere act of selling many shares in a single day will have an adverse effect on the price.

Since future market prices, and the effect of large sales on these prices, are very hard to predict, brokerage firms use models of the market to help them make such decisions. In this problem, we will consider the following simple model. Suppose we need to sell x shares of stock in a company, and suppose that we have an accurate model of the

market: it predicts that the stock price will take the values $p_1, p_2, \ldots, p_n$ over the next n days. Moreover, there is a function $f(\cdot)$ that predicts the effect of large sales: if we sell y shares on a single day, it will permanently decrease the price by $f(y)$ from that day onward. So, if we sell y_1 shares on day 1, we obtain a price per share of $p_1 - f(y_1)$, for a total income of $y_1 \cdot (p_1 - f(y_1))$. Having sold y_1 shares on day 1, we can then sell y_2 shares on day 2 for a price per share of $p_2 - f(y_1) - f(y_2)$; this yields an additional income of $y_2 \cdot (p_2 - f(y_1) - f(y_2))$. This process continues over all n days. (Note, as in our calculation for day 2, that the decreases from earlier days are absorbed into the prices for all later days.)

Design an efficient algorithm that takes the prices $p_1, \ldots, p_n$ and the function $f(\cdot)$ (written as a list of values $f(1), f(2), \ldots, f(x)$) and determines the best way to sell x shares by day n. In other words, find natural numbers $y_1, y_2, \ldots, y_n$ so that $x = y_1 + \ldots + y_n$, and selling y_i shares on day i for $i = 1, 2, \ldots, n$ maximizes the total income achievable. You should assume that the share value p_i is monotone decreasing, and $f(\cdot)$ is monotone increasing; that is, selling a larger number of shares causes a larger drop in the price. Your algorithm's running time can have a polynomial dependence on n (the number of days), x (the number of shares), and p_1 (the peak price of the stock).

Example Consider the case when $n = 3$; the prices for the three days are 90, 80, 40; and $f(y) = 1$ for $y \leq 40,000$ and $f(y) = 20$ for $y > 40,000$. Assume you start with $x = 100,000$ shares. Selling all of them on day 1 would yield a price of 70 per share, for a total income of 7,000,000. On the other hand, selling 40,000 shares on day 1 yields a price of 89 per share, and selling the remaining 60,000 shares on day 2 results in a price of 59 per share, for a total income of 7,100,000.

16.1 Strategy

This is also a two constraints problem. Notice that this question assumes that $f(x) + f(y) = f(x + y)$, which could simplify the problem.

16.2 Recurrence Relation

Subproblem: Define $OPT[i, x']$ as the maximum income we could get up to i^{th} day and the remaining x' shares.

Recurrence Relation could be expressed as:

$$OPT[i, x'] = \max_{0 \leq y \leq x'}(OPT[i-1, x'-y] + y(p_i - f(y))) \qquad (22)$$

17 Problem 26

Consider the following inventory problem. You are running a company that sells some large product (let's assume you sell trucks), and predictions tell you the quantity of sales to expect over the next n months. Let d_i denote the number of sales you expect in month i. We'll assume that all sales happen at the beginning of the month, and trucks that are not sold are *stored* until the beginning of the next month. You can store at most S trucks, and it costs C to store a single truck for a month. You receive shipments of trucks by placing orders for them, and there is a fixed ordering fee of K

each time you place an order (regardless of the number of trucks you order). You start out with no trucks. The problem is to design an algorithm that decides how to place orders so that you satisfy all the demands $\{d_i\}$, and minimize the costs. In summary:

- There are two parts to the cost: (1) storage—it costs C for every truck on hand that is not needed that month; (2) ordering fees—it costs K for every order placed.
- In each month you need enough trucks to satisfy the demand d_i, but the number left over after satisfying the demand for the month should not exceed the inventory limit S.

Give an algorithm that solves this problem in time that is polynomial in n and S.

17.1 Strategy

This problem is actually straight forward, but we need to set a custom variable, the amount of truck that would be left in day i. Now we need to consider the cases if the left trucks are enough for the next day's demand. If not we need to order; if yes, we need to consider if order is more cheaper.

17.2 Recurrence Relation

Subproblem: Define $OPT[i, s]$ as the minimum cost up to i^{th} months with s trucks left over:

Recurrence Relation could be expressed as:

- If $s + d_i > S$:
$$OPT[i, s] = OPT[i - 1, 0] + K \tag{23}$$
- Else:
$$OPT[i, s] = \min(OPT[i - 1, s + d_i], OPT[i - 1, 0] + K) \tag{24}$$

18 Problem 27

The owners of an independently operated gas station are faced with the following situation. They have a large underground tank in which they store gas; the tank can hold up to L gallons at one time. Ordering gas is quite expensive, so they want to order relatively rarely. For each order, they need to pay a fixed price P for delivery in addition to the cost of the gas ordered. However, it costs c to store a gallon of gas for an extra day, so ordering too much ahead increases the storage cost.

They are planning to close for a week in the winter, and they want their tank to be empty by the time they close. Luckily, based on years of experience, they have accurate projections for how much gas they will need each day until this point in time. Assume that there are n days left until they close, and they need g_i gallons of gas for each of the days $i = 1, \ldots, n$. Assume that the tank is empty at the end of day 0. Give an algorithm to decide on which days they should place orders, and how much to order so as to minimize their total cost.

18.1 Strategy

This problem is very similar with the last one. However, we need to maker sure everything is empty at the last day.

18.2 Recurrence Relation

Subproblem: Define $OPT[i, s]$ as the minimum cost we could reach up to i^{th} day with s gallons left. Notice that here $0 \leq i \leq n-1$.

Recurrence Relation could be expressed as:

- If $s + g_i > L$:
$$OPT[i, s] = OPT[i-1, 0] + P \tag{25}$$
- Else:
$$OPT[i, s] = \min(OPT[i-1, c(s+g_i)], OPT[i-1, 0] + P) \tag{26}$$