Dynamic Programming

1 Least Weight Subsequence

Problem 1: Maximum sum among nonadjacent subsequences

Find an efficient algorithm for the following problem:

We are given an array of real numbers $V[1..n]$. We wish to find a subset of array positions, $S \subseteq [1...n]$ that maximizes $\sum_{i \in S} V[i]$ subject to no two consecutive array positions being in S. For example, say $V = [10, 14, 12, 6, 13, 4]$, the best solution is to take elements $1, 3, 5$ to get a total of $10 + 12 + 13 = 35$. If instead, we try to take the 14 in position 2, we must exclude the 10 and 12 in positions 1 and 3, leaving us with the second best choice $2, 5$ giving a total of $14 + 13 = 27$.

Problem 2: Maximum Sum Descent

Some positive integers are arranged in a triangle like the one shown in Fig. 1. Design an algorithm (more efficient than an exhaustive search, of course) to find the largest sum in a descent from its apex to the base through a sequence of adjacent numbers, one number per each level. Start by solving the problem in Fig. 1, then generalize the solution to a N levels problem.

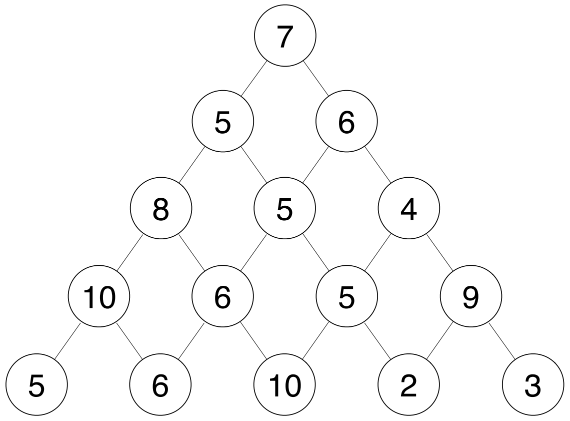

Figure 1: Example of numbers in the Maximum Sum Descent problem.

Problem 3: Longest Pattern Meeting Regular Description

We are given an n bit string w over $0, 1$, and an m-state deterministic finite state automaton D over the alphabet $0, 1$ (represented by its state transition table, start state, and an array saying for each state whether it is accepting.) Find the longest substring accepted by D

Give the fastest algorithm you can to solve this problem assuming $n \geq m$. (The time should be a function of both n and m.) (Hint: An $O(n^2)$ algorithm is trivial)

Problem 4: Picking Up Coins

Some coins are spread in the cells of an n x m board, one coin per cell. A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell. On each step, the robot can move either one cell to the right or one cell down from its current location. When the robot visits a cell with a coin, it picks up that coin. Devise an algorithm to find the maximum number of coins the robot can collect and a path it needs to follow to do this.

Problem 5: Long trip (DPV)

You are going on a long trip. You start on the road at mile post 0 Along the way there are n hotels at mile posts $a_1 < a_2 < \cdots < a_n$., where each a_i is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance a_n), which is your destination.

You'd ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the *penalty* for that day is $(200 - x)^2$. You want to plain your trip so as to minimize the total penalty - that is, the sum, over all travel days, of the daily penalties. Give an efficient algorithm that determines the optimal sequence of hotels at which to stop.

Problem 6: Longest increasing subsequence

The input is a sequence of numbers $a_1, \ldots, a_n$. A *subsequence* is any subset of these numbers taken in order. An *increasing subsequence* is one in which the numbers are getting larger. The task is to find the length of the longest increasing subsequence.

Problem 7: Optimized increasing subsequence (CLRS)

Give an $O(n \log n)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers. (Hint: Observe that the last element of a candidate subsequence of length i is at least as large as the last element of a candidate subsequence of length $i - 1$. Maintain candidate subsequences by linking them through the input sequence.)

Problem 8: Printing neatly (CLRS)

Consider the problem of neatly printing a paragraph on a printer. The input text is a sequence of n words of lengths $l_1, l_2, \ldots, l_n$, measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. Our criterion of "neatness" is as follows. If a given line contains words i through j, where $i \leq j$, and we leave exactly one space between words, the number of extra space characters at the end of the line is $M - j + i - \sum_{k=i}^{j} l_k$, which must be nonnegative so that the words fit on the line. We wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines. Give a dynamic-programming algorithm to print a paragraph of n words neatly on a printer. Analyze the running time and space requirements of your algorithm.

Problem 9: Restaurants along a highway

Yuckdonald's is considering opening a series of restaurants along Quaint Valley Highway (QVH). The n possible locations are along a straight line, and the distances of these locations from the start of the QVH are, in miles and in increasing order, $m_1, m_2, \cdots, m_n$. The constraints are as follows:

- At each location, Yuckdonald's may open at most one restaurant. The expected profit from opening a restaurant at location i is p_i, where $p_i > 0$ and $i = 1, 2, \cdots, n$.
- Any two restaurants should be at least k miles apart, where k is a positive integer.

Give an efficient algorithm to compute the maximum expected total profit subject to the given constraints.

Problem 10: Maximum Length Chain of Subwords

We are given a set of n strings of length at most k over a finite alphabet σ. A sequence of strings that form a chain under the (consecutive) subword relation; i.e., if the output is $w_1, w_2, \ldots, w_t$ then we can write $w_{i+1} = uw_iv$ for some strings u, v.

Find a chain of maximum size.

Problem 11: Divide into words (DPV)

You are given a string of n characters $s[1 \ldots n]$, which you believe to be a corrupted text document in which all punctuation has vanished (so that it looks something like "itwasthebestoftimes..."). You wish to reconstruct the document using a dictionary, which is available in the form of a Boolean function dict($\cdot$): for any string w,

$$\text{dict}(w) = \begin{cases} \text{true} & \text{if } w \text{ is a valid word} \\ \text{false} & \text{otherwise} \end{cases}$$

1. Give a dynamic programming algorithm that determines whether the string $s[\cdot]$ can be reconstituted as a sequence of valid words. The running time should be at most $O(n^2)$, assuming that calls to dict take unit time.

2. In the event that the string is valid, make your algorithm output the corresponding sequence of words.

Problem 12: Untangling signal superposition (KT 6.19)

You're consulting for a group of people (who would prefer not to be mentioned here by name) whose jobs consist of monitoring and analyzing electronic signals coming from ships in coastal Atlantic waters. They want a fast algorithm for a basic primitive that arises frequently: "untangling" a superposition of two known signals. Specifically, they're picturing a situation in which each of two ships is emitting a short sequence of 0s and 1s over and over, and they want to make sure that the signal they're hearing is simply an *interleaving* of these two emissions, with nothing extra added in.

This describes the whole problem; we can make it a little more explicit as follows. Given a string x consisting of 0s and 1s, we write x^k to denote k copies of x concatenated together. We say that a string x' is a *repetition* of x if it is a prefix of x^k for some number k. So $x' = 10110110110$ is a repetition of $x = 101$.

We say that a string s is an *interleaving* of x and y if its symbols can be partitioned into two (not necessarily contiguous) subsequences s' and s'', so that s' is a repetition of x and s'' is a repetition of y. (So each symbol in s must belong to exactly one of s' or s''.) For example, if $x = 101$ and $y = 00$, then $s = 100010101$ is an interleaving of x and y, since characters $1, 2, 5, 7, 8, 9$ form 101101–a repetition of x–and the remaining characters $3, 4, 6$ form 000–a repetition of y.

In terms of our application, x and y are the repeating sequences from the two ships, and s is the signal we're listening to: We want to make sure s "unravels" into simple repetitions of x and y. Give an efficient algorithm that takes strings s, x, and y and decides if s is an interleaving of x and y.

Problem 13: Data compression

Consider the following data compression technique. We have a table of m text strings, each of length at most k. We want to encode a data string D of length n using as few text strings as possible. For example, if our table contains $(a, ba, abab, b)$ and the data string is *bababbaababa*, the best way to encode it is $(b, abab, ba, abab, a)$ — a total of five code words. Give an $O(nmk)$ algorithm to find the length of the best encoding. You may assume that the string has an encoding in terms of the table.

Problem 14: Single Word Reusable Boggle (SWoRB)

You are given an $n \times n$ matrix of letters from a finite alphabet Σ, and a target word w in Σ of length m. You want to determine whether there is a (not necessarily simple, i.e., letters can be reused) path in the grid so that the letters along that path are w. Each step in the path can go from a point in the grid to any of its 8 neighboring points, including diagonal moves. Give an efficient algorithm to play SWoRB.

Problem 15: Single Machine Maximal Weighted Schedule

We are given a set of n jobs $J_i = < s_i, f_i, w_i >, 1 \leq i \leq n$, where $s_i < f_i$ and $0 < w_i$. s_i is called the start time of the job, f_i, the finish time, and w_i the weight. A solution is a subset S of the jobs such that S cannot contain two jobs J_i and J_k, $i \neq k$, that overlap, i.e., where $s_i \leq s_k \leq f_i$.

A solution S has total weight $W(S) = \sum J_i w_i$. Give the best algorithm to find a solution with the maximum total weight. (Hint: It is possible to solve the problem in time close to linear time.)

Problem 16: Reliability

We are given a directed graph $G = (V, E)$ on which each edge $(u, v) \in E$ has an associated value $r(u, v)$, which is a real number in the range $0 \leq r(u, v) \leq 1$ that represents the reliability of a communication channel from vertex u to vertex v. We interpret $r(u, v)$ as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent. Give an **efficient** algorithm to find the most reliable path between two given vertices. Write the complete algorithm. Analyze its time complexity.

Problem 17: Speech recognition

We can use dynamic programming on a directed graph $G = (V, E)$ for speech recognition. Each edge $(u, v) \in E$ is labeled with a sound $\sigma(u, v)$ from a finite set Σ of sounds. The labeled graph is a formal model of a person speaking a restricted language. Each path in the graph starting from a distinguished vertex $v_0 \in V$ corresponds to a possible sequence of sounds produced by the model. The label of a directed path is defined to be the concatenation of the labels of the edges on that path.

Describe an efficient algorithm that, given an edge-labeled graph G with distinguished vertex v_0 and a sequence $s = (\sigma_1, \cdots, \sigma_k)$ of characters from Σ, returns a path in G that begins at v_0 and has s as its label, if any such path exists. Otherwise, the algorithm should return NO-SUCH-PATH. Analyze the running time of the algorithm. Clearly write any dynamic programming formulation you may use to solve this problem.

Problem 18: Speech recognition

We can use dynamic programming on a directed graph $G = (V, E)$ for speech recognition. Each edge $(u, v) \in E$ is labeled with a sound $\sigma(u, v)$ from a finite set Σ of sounds. The labeled graph is a formal model of a person speaking a restricted language. Each path in the graph starting from a distinguished vertex $v_0 \in V$ corresponds to a possible sequence of sounds produced by the model. The label of a directed path is defined to be the concatenation of the labels of the edges on that path.

Describe an efficient algorithm that, given an edge-labeled graph G with distinguished vertex v_0 and a sequence $s = (\sigma_1, \cdots, \sigma_k)$ of characters from Σ, returns a path in G that begins at v_0 and has s as its label, if any such path exists. Otherwise, the algorithm should return NO-SUCH-PATH. Analyze the running time of the algorithm. Clearly write any dynamic programming formulation you may use to solve this problem.

Problem 19: Distributed data structure

You have a data structure S that is too large to store on one machine, so you plan to store the n different elements on k identical machines. The elements are ordered, and, to locate easily, each machine will store a consecutive sequence of elements. Some elements e_i have pointers to other elements e_j. You want to minimize the number of pointers that go between different machines. At most t elements can fit on any one machine, where $tk \geq n$.

a. Formalize the above problem specification as a combinatorial optimization problem, e.g., give the Instance, Solution Format, Constraints, and Objective.

b. Give an efficient algorithm for the problem. Assume that there is a (possibly empty) linked list of pointers associated with each element. Assume that m is the total number of pointers. You may want to use both n and m to analyze the time complexity.

Problem 20: Weighted Unit intervals

Describe an efficient algorithm that, given a set $\{x_1, x_2, \ldots, x_n\}$ of points on the real line and a set of closed intervals $[s_i, f_i]$ for $1 \leq i \leq d$ where each the cost of the i-th interval is $c_i > 0$, determines the least costly set of closed intervals that contains all of the given points. Assume that there is a set of intervals to cover all the points.

Problem 21: Book packing

Consider the problem of storing n books on shelves in a library. The order of the books is fixed by the cataloging system and so cannot be rearranged. Therefore, we can speak of a book b_i, where $1 \leq i \leq n$, that has a thickness t_i and height h_i. The length of each bookshelf at this library is L.

1. Suppose all the books have the same height h and the shelves are all separated by a distance of greater than h, so any book fits on any shelf. The greedy algorithm would fill the first shelf with as many books as we can until we get the smallest i such that b_i does not fit, and then repeat with the subsequent shelves. Show that the greedy algorithm always finds the optimal shelf placement, and analyze its time complexity.

2. Now consider the case where the height of the books is not constant, but we have the freedom to adjust the height of each shelf to that of the tallest book on the shelf. Thus the cost of a particular layout is the sum of the heights of the largest book on each shelf. Give an example to show that the greedy algorithm of stuffing each shelf as full as possible does not always give the minimum overall height.

   Give an algorithm for this problem, and analyze its time complexity.

Problem 22: Shortest wireless path sequence (KT 6.14)

A large collection of mobile wireless devices can naturally form a network in which the devices are the nodes, and two devices x and y are connected by an edge if they are able to directly communicate with each other (e.g., by a short-range radio link). Such a network of wireless devices is a highly dynamic object, in which edges can appear and disappear over time as the devices move around. For instance, an edge (x, y) might disappear as x and y move far apart from each other and lose the ability to communicate directly.

In a network that changes over time, it is natural to look for efficient ways of *maintaining* a path between certain designated nodes. There are two opposing concerns in maintaining such a path: we want paths that are short, but we also do not want to have to change the path frequently as the network structure changes. (That is, we'd like a single path to continue working, if possible, even as the network gains and loses edges.) Here is a way we might model this problem.

Suppose we have a set of mobile nodes V, and at a particular point in time there is a set E_0 of edges among these nodes. As the nodes move, the set of edges changes from E_0 to E_1, then to E_2, then to E_3, and so on, to an edge set E_b. For $i = 0, 1, 2, \ldots, b$, let G_i denote the graph (V, E_i). So if we were to watch the structure of the network on the nodes V as a "time lapse", it would look precisely like the sequence of graphs $G_0, G_1, G_2, \ldots, G_{b-1}, G_b$. We will assume that each of these graphs G_i is connected.

Now consider two particular nodes $s, t \in V$. For an s-t path P in one of the graphs G_i, we define the *length* of P to be simply the number of edges in P, and we denote this $\ell(P)$. Our goal is to produce a sequence of paths $P_0, P_1, \ldots, P_b$ so that for each i, P_i is an s-t path in G_i. We want the paths to be relatively short. We also do not want there to be too many *changes*–points at which the identity of the path switches. Formally, we define *changes*$(P_0, P_1, \ldots, P_b)$ to be the number of indices i $(0 \leq i \leq b-1)$ for which $P_i \neq P_{i+1}$.

Fix a constant $K > 0$. We define the cost of the sequence of paths $P_0, P_1, \ldots, P_b$ to be

$$cost(P_0, P_1, \ldots, P_b) = \sum_{i=0}^{b} \ell(P_i) + K \cdot changes(P_0, P_1, \ldots, P_b).$$

1. Suppose it is possible to choose a single path P that is an s-t path in each of the graphs $G_0, G_1, \ldots, G_b$. Give a polynomial-time algorithm to find the shortest such path.

2. Give a polynomial-time algorithm to find a sequence of paths $P_0, P_1, \ldots, P_b$ of minimum cost, where P_i is an s-t path in G_i for $i = 0, 1, \ldots, b$.

2 Least Weight Subsequence with Compact Representation

Problem 23: Selling shares of stock (KT 6.25)

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 algorithms 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.

Problem 24: Redundancy (DPV)

A mission-critical production system has n stages that have to be performed sequentially; stage i is performed by machine M_i. Each machine M_i has a probability r_i of functioning reliably and a probability $1 - r_i$ of failing (and the failures are independent). Therefore, if we implement each stage with a single machine, the probability that the whole system words is $r_1 \cdot r_2 \cdots r_n$. To improve this probability we add redundancy, by having m_i copies of the machine M_i that performs stage i. The probability that all m_i copies fail simultaneously is only $(1 - r_i)^{m_i}$, so the probability that stage i is completed correctly is $1 - (1 - r_i)^{m_i}$ and the probability that the whole system works is $\prod_{i=1}^{n}(1 - (1 - r_i)^{m_i})$. Each machine M_i has cost c_i, and there is a total budget of B to buy machines. (Assume that B and c_i are positive integers.)

Given the probabilities $r_1, \ldots, r_n$, the costs $c_1, \ldots, c_n$, and the budget B, find the redundancies $m_1, \ldots, m_n$ that are withing the available budget and that maximize the probability that the system words correctly.

You can assume the costs c_i and the budget B are integers.

Problem 25: Memory allocation

We have n files $f_1, \ldots, f_n$ we need to store on disk. Each file f_i is $s_i \leq B$ bytes long, where a page of disk stores B bytes. We can store each file entirely on a page, or split it between two consecutive pages.

Consecutive files must be stored on the same or consecutive pages. We'll be accessing the files in order and performing a computation on each. Each time we access a new file that is not entirely on the same page as the previous file, we have an overhead of T_{page} for reading the page into main memory. In addition, if the file f_i is split between two pages, we have an overhead of $T_{split} * s_i$ for cache misses during the computation. We wish to find an efficient algorithm for storing files in memory that minimizes the total overhead time.

An algorithms for this problem is considered efficient if it runs in time $O(nB)$.

Problem 26: Coin change with repetition

Given an unlimited supply of coins of denominations $x_1, x_2, \ldots, x_n$, we wish to make change for a value v; that is, we wish to find a set of coins whose total value is v. This might not be possible; for example, if the denominations are 5 and 10 then we can make change for 15 but not for 12. Give an $O(nv)$ dynamic programming algorithm for the following problem.

Input: $x_1, x_2, \ldots, x_n; v$

Question: Is it possible to make change for v using coins of denominations $x_1, \ldots, x_n$?

Problem 27: Unique-coin change (DPV)

Consider the following variation on the change-making problem: you are given denominations $x_1, x_2, \ldots x_n$, and you want to make change for a value v, but you are allowed to use each denomination *at most once.* For instance, if the denominations are 1, 5, 10, 20, then you can make change for $16 = 1 + 15$ and for $31 = 1 + 10 + 20$ but not for 40 (because you can't use 20 twice).

Input: Positive integers $x_1, x_2, \ldots, x_n$; another integer v. Output: Can you make change for v, using each denomination x_i at most once?

Show how to solve this problem in time $O(nv)$.

Problem 28: Difference of sums

Give an algorithm for the following problem. Given a list of n distinct positive integers, partition the list into two sublists, each of size $n/2$, such that the difference between the sums of the integers in the two sublists is minimized. Formulate a dynamic programming definition for the problem. Develop an algorithm based on your definition. What is the time complexity of your algorithm? You can assume that n is a power of 2.

Problem 29: Scheduling tasks for two workers

Suppose we are given N tasks where task i requires exactly t_i hours and t_i is a positive integer between 1 and K. Each task is to be assigned to one of two workers. Our goal is to find an assignment that minimizes the difference in total time assigned to the two workers. (Ideally, we would assign the tasks so both workers have the same amount of work; if that isn't possible, we want to make the loads as equal as possible.)

Give a dynamic programming algorithm for finding the best assignment, and give a time analysis. The running time should be a polynomial in N and K.

Problem 30: 0-1 knapsack (CLRS)

Give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(nW)$ time, where n is the number of items and W is the maximum weight of items that the thief can put in his knapsack.

Problem 31: Knapsack *with* repetition

During a robbery, a burglar finds much more loot than he had expected and has to decide what to take. His bag (or knapsack) will hold a total weight of at most W pounds. There are n items to pick from, of weight $w_1, ...w_n$ and dollar value $v_1, ..., v_n$. What's the most valuable combination of items he can fit into his bag?

With repetition: we suppose there is an infinite number of each item (we can repeat each item as many time as we want).

Problem 32: Knapsack *without* repetition

Same as previous except that there is only one item of each type (*without* repetition).

Note that if there was a limited number of each item, we would just have to deal with each item as it was unique.

3 Alignment

Problem 33: Edit distance

The *cost of an alignment* of 2 strings is the number of columns in which the letters differ. And the *edit distance* between two strings is the cost of their best possible alignment. The task is to find the minimum edit distance between 2 given strings $x[1..m]$ and $y[1..n]$.

```
S - N O W Y   - S N O W - Y
S U N N - Y   S U N - - N Y
   cost 3         cost 5
```

Problem 34: Longest common subsequence (DPV)

Given two strings $x = x_1x_2 \cdots x_n$ and $y = y_1y_2 \cdots y_m$, we wish to find the length of their *longest common subsequence*, that is, the largest k for which there are indices $i_1 < i_2 < \cdots < i_k$ and $j_1 < j_2 < \cdots < j_k$ with $x_{i_1}x_{i_2} \cdots x_{i_k} = y_{j_1}y_{j_2} \cdots y_{j_k}$. Show how to do this in time $O(mn)$.

Problem 35: Protein Bonding

Let Σ be a finite set of amino acids, and let $w = w_1....w_n$ be a sequence of acids from Σ. For $\sigma, \sigma' \in \Sigma$, let $b(\sigma, \sigma')$ be the strength of a bond between the two types of acids, a non-negative real number. A *bonding* of the sequence is a partial matching between positions in the word so that matched pairs can be connected with lines drawn below the word without lines crossing. Equivalently, it should satisfy : there are no two bonded pairs i_1, j_1 and i_2, j_2 with $i_1 \leq i_2 \leq j_1 \leq j_2$. The *total bond strength* is the sum over all bonded positions i, j of the bond strength $b(w_i, w_j)$. Give an efficient algorithm to find the bonding of a protein sequence that maximizes the total bond strength.

4 Parenthesization

Problem 36: Optimal binary search tree (DPV)

Suppose we know the frequency with which keywords occur in programs of a certain language, for instance:

begin	5%
do	40%
else	8%
end	4 %
if	10%
then	10%
while	23%

We want to organize then in a *binary search tree*, so that the keyword in the root is alphabetically bigger than all the keywords in the left subtree and smaller than all the keywords in the right subtree (and this holds for all nodes).

Give an efficient algorithm for the following task.

Input: n words (in sorted order); frequencies of these words: $p_1, p_2, \ldots, p_n$.

Output: The binary search tree of lowest cost (defined as the expected number of comparisons in looking up a word).

Problem 37: Multiplication with nonassociative costs

Let $a_1, a_2, \ldots, a_n$ be a sequence of positive integers. Our goal is to compute the product of these numbers. However, the cost of multiplying two numbers is the product of the two numbers. Fully parenthesize the sequence so as to minimize the total cost of all multiplications.

5 Context-free Grammar Parsing

Problem 38: Multiplication

Consider the problem of examining a string $x = x_1x_2\ldots x_n$ of characters from an alphabet of k symbols, and a multiplication table over this alphabet, and deciding whether or not it is possible to parenthesize x in such a way that the value of the resulting expression is a, where a belongs to the alphabet. The multiplication tables is neither commutative nor associative, so the order of multiplication matters.

Give an algorithm, with time in polynomial in n and k, to decide whether such a parenthesization exists for a given string, multiplication table, and goal element.

Problem 39: Multiplication, specific table (DPV)

Let us define a multiplication operation on three symbols a,b, c according to the following table: thus $ab = b$, $ba = c$, and so on. Notice that the multiplication is neither associative nor commutative.

Table 1: Multiplication Table

	a	b	c
a	b	b	a
b	c	b	a
c	a	c	c

Find an efficient algorithm that examines a string of these symbols, say, *bbbbac*, and decides whether or not it is possible to parenthesize the string in such a way that the value of the resulting expression is a. For example, on input *bbbbac* your algorithm should return *yes* because $((b(bb))(ba))c = a$.

6 All Pairs Shortest Paths

Problem 40: Tree distances

Let $G = (V, E)$ be a binary tree with n vertices. We want to construct an $n \times n$ matrix whose ij-th entry is equal to the distance between vertex v_i and vertex v_j. Design an $O(n^2)$ algorithm to construct such a matrix for a tree that is given in the adjacency-list representation.

7 Miscellaneous

Problem 41: Descending partitions

A descending partition of positive integer N is a sequence of positive integers $A_1 > A_2 > ... > A_k$ with $\sum_{i=1}^{i=K} A_i = N$. Give an efficient (poly-time in N) algorithm that, given N, computes the number of decreasing partitions of N. For example, if N=6, the decreasing partitions are: $(6); (5,1); (4,2); (3,2,1)$ so your algorithm, on input 6 should return 4.

Problem 42: Pivoting vertices

Let T be a rooted directed tree, not necessarily binary. There is a non-negative real weight associated with each vertex, such that the weight of a vertex is greater than the weight of the vertex's parent. Each vertex can be designated as either a *regular* or a *pivot* vertex. The cost of a pivot vertex is the same as its weight. Regular vertices get discounts: their cost is their weight minus the weight of the closest ancestor that is a pivot vertex. Thus, selecting a vertex as a pivot vertex may increase its cost, but it will decrease the costs of some of its descendants. If a regular vertex has no ancestor which is a pivot vertex, then its cost is its weight itself.

There is no limit on the number of pivot vertices. Design an efficient algorithm to designate every vertex as either a regular vertex or a pivot vertex, such that the total cost of all vertices is minimized.

Problem 43: Warehouse problem

There are two warehouses V and W from which widgets are to be shipped to destinations D_i, $1 \le i \le n$. Let $d_i \ge 1$ be the demand at D_i, for $1 \le i \le n$, and r_V, r_W be the number of widgets available at V and W, respectively. Let $\bar{d} = \sum_{i=1}^{n} d_i$ denote total demand of all the destinations. Assume that there are enough widgets available to fill the demand, that is,

$$r_V + r_W = \bar{d}$$

Let v_i be the cost of shipping a widget from warehouse V to destination D_i, and w_i be the cost of shipping a widget from warehouse W to destination D_i, for $1 \le i \le n$. The *warehouse problem* is the problem of finding x_i, $y_i \in \mathbf{N}$ for $1 \le i \le n$ such that when x_i widgets are sent from V to D_i and y_i widgets are sent from W to D_i:

- the demand at D_i is satisfied, that is , $x_i + y_i = d_i$,
- the inventory at V is sufficient, $\sum_{i=1}^{n} x_i = r_V$,
- the inventory at W is sufficient, $\sum_{i=1}^{n} y_i = r_W$,

and the total cost of shipping the widgets,

$$\sum_{i=1}^{n} (v_i x_i + w_i y_i)$$

is minimized.

Use dynamic programming to design an $O(\bar{d}^2)$ algorithm. Provide a clear description of your dynamic programming definitions and recursive formulations. Make sure that your definitions and recursive formulations deal with the determination of x_i and y_i for each destination as well as the minimum total cost. Prove the correctness of your algorithm.

Present pseudocode to implement your algorithm and analyze its time complexity.

You will get partial credit if you present an algorithm whose complexity is worse than $\bar{d}^2$.

Problem 44: Dice pools

This problem arises from calculating success probabilities for certain role-playing games, where players roll dice in proportion to their character's abilities, and each die is either a "Success ", a "Failure" or "Neutral", and the outcome is determined by the number of successes minus the number of failures. (For example, in one game, dice take random values from 1 to 10, with 1 being a "Failure" and 8-10 being a "Success".) Abstractly, the problem is: there are n independent random variables, $X_1 ... X_n$. Each variable is $+1$ with probability p, -1 with probability q and 0 otherwise, where $0 \leq p, q \leq 1$ and $p + q \leq 1$. (In the above example, $p = 3/10, q = 1/10$.) We want to calculate, given n, an array of probabilities: for all k with $-n \leq k \leq n$ compute the probability that $\sum_{i=1}^{i=n} X_i = k$. Your algorithm should be polynomial-time in n. Assume arithmetic operations are constant time.

Problem 45: Scheduling with Availability Constraints

The input is a set of n jobs, each with a positive real availability time, a_i, and a positive real duration, d_i. You want to schedule the jobs on a single processor without interrupts, under the constraint that no job can start before its availability time. in other words, each job gets assigned an interval of time (s_i, f_i) where s_i is called the start time and $f_i = s_i + d_i$ is the finish time. Since the processor can only do one job at a time, we have the constraint that all the intervals are disjoint, i.e., for each pair of jobs $i \neq j$ either $s_i \geq f_j$ or $s_j \geq f_i$. Also, since we cannot start jobs before they arrive, $a_i \leq s_i$ for each i. you wish to minimize the last finish time, i.e., the time when all jobs are completed. Describe a fast algorithm for this problem.

Problem 46: Shortest Path Counting

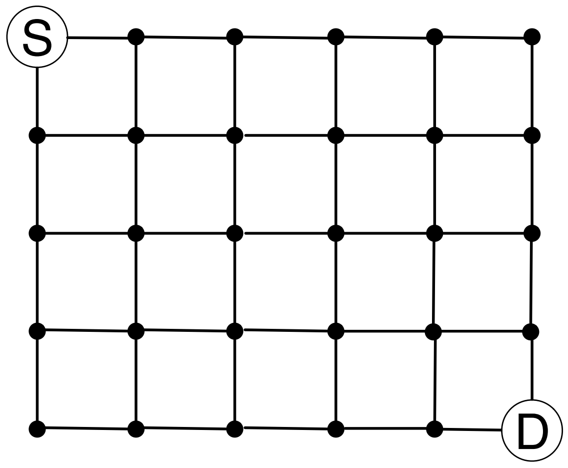

Figure 2: The city streets of the Shortest Path Counting problem.

Find the number of the shortest paths from the starting point S to the destination D in a city with perfectly horizontal streets and vertical avenues shown in Figure 2.

Problem 47: Palindrome Counting

In how many different ways can the palindrome "WAS IT A CAT I SAW" be read in the diamond-shaped arrangement shown in Tab 2? You may start at any W and go in any direction on each step up, down, left, or right through adjacent letters. The same letter can be used more than once in the same sequence.

Problem 48: Soccer Strategy

						W						
					W	A	W					
				W	A	S	A	W				
			W	A	S	I	S	A	W			
		W	A	S	I	T	I	S	A	W		
	W	A	S	I	T	A	T	I	S	A	W	
W	A	S	I	T	A	C	A	T	I	S	A	W
	W	A	S	I	T	A	T	I	S	A	W	
		W	A	S	I	T	I	S	A	W		
			W	A	S	I	S	A	W			
				W	A	S	A	W				
					W	A	W					
						W						

Table 2: Letter Arrangement

The coach of the US soccer team decided to place the players in the soccer field as in the Figure, where the goalkeeper is at the bottom of the Figure and the striker is on the top. Currently, the goalkeeper has the ball. A player can pass the ball only to the players connected to him with an edge.

Find the number of different shortest paths from the goalkeeper to the striker, so that the striker can make a goal.

need figure here

Problem 49: Blocked Paths

Find the number of different shortest paths from the starting point S to the destination D in a city with perfectly horizontal streets and vertical avenues as shown in Figure 3. No path can cross the fenced off area shown in grey in the figure.

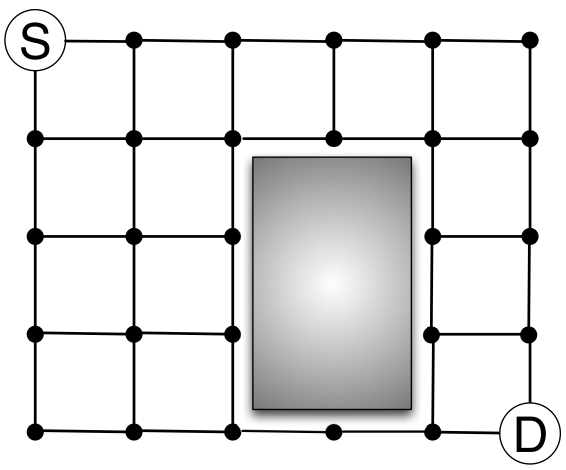

Figure 3: The city streets of the Blocked Paths problem.

Problem 50: Pile Splitting

1. Given n counters in a pile, split the counters into two smaller piles and compute the product of the numbers of the counters in the two piles obtained. Continue to split each pile into two smaller piles and to compute the products until there are n piles of size one. Once there are n piles, sum all the products computed. How should one split the piles to maximize the sum of the products? What is this maximal sum equal to?

2. How does the solution to the puzzle change if we are to compute the sum of the numbers of the counters in the two piles obtained after every split and have a goal of maximizing the total of such sums?

Problem 51: Bitonic euclidean traveling-salesperson problem (CLRS)

The *euclidean traveling-salesman problem* is the problem of determining the shortest closed tour that connects a given set of n points in the plane. The general problem is NP-complete, and its solution is therefore believed to require more than polynomial time.

J. L. Bentley has suggested that we simplify the problem by restricting our attention to *bitonic tours*, that is, tours that start at the leftmost point, go strictly left to right to the rightmost point, and then go strictly right to left back to the starting point. In this case, a polynomial-time algorithm is possible.

Describe an $O(n^2)$-time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same $x - coordinate$. (Hint: Scan left to right, maintaining optimal possibilities for the two parts of the tour.)

8 Solved Problems

Problem 52: Longest increasing subsequence

The input is a sequence of numbers $a_1, \ldots, a_n$. A *subsequence* is any subset of these numbers taken in order. An *increasing subsequence* is one in which the numbers are getting larger. The task is to find the length of the longest increasing subsequence.

Solution: Longest increasing subsequence

Dynamic Programming definitions. For each suffix, we compute the length of the longest increasing subsequence that begins with the first letter of the suffix.

$$a_1 \quad a_2 \quad \cdots \quad \cdots \quad \boxed{a_i \quad a_{i+1} \quad \cdots \quad \cdots \quad a_n}$$

We define $\text{opt}(i)$ = the length of the longest increasing subsequence *starting* at a_i for $1 \leq i \leq n$.

The length of the longest increasing subsequence of the given input $a_1 \ldots a_n$ is $\max_{1 \leq i \leq n} \text{opt}(i)$. We will compute the optimal subsequence by computing for each i, the index of the succeeding element (if any) in the longest increasing subsequence that starts at a_i.

For $1 \leq i \leq n$, we define $T(i)$ to be the index of the succeeding element in the longest increasing subsequence that begins at a_i if there is a succeeding element. Otherwise $T(i)$ will have a null value.

Recursive formulation. Imagine the longest increase subsequence S starting at a_i for $1 \leq i < n$. If there is no next item in the longest increasing subsequence starting at a_i, then its length is 1. Otherwise, let a_j be the next item in the sequence. Then we have $j > i$ and $a_j \geq a_i$.

We write the sequence $S : a_i \to S'$ where $S' : a_j \to \cdots$ is an increasing subsequence starting at a_j. Since S is the longest among subsequences that starts with a_i, S' must be the longest increasing subsequence starting at a_j and therefore its length must be $\text{opt}(j)$. So, if a_j is the next item, the length of S will be:

$$\underbrace{1}_{\substack{\text{for the step} \\ a_i \to a_j}} \quad + \quad \underbrace{\text{opt}(j)}_{\substack{\text{length of \textbf{the} longest increasing} \\ \text{subsequence starting at } a_j}}$$

Since we don't know j, we select j by searching through all $j > i$ such that $a_j \geq a_i$ and $\text{opt}(j)$ is the largest.

$$\text{opt}(i) \quad = \quad 1 \quad + \quad \max_{\substack{j>i \\ \text{if } a_j \geq a_i}} \text{opt}(j)$$

Note that $\text{opt}(i) = 1$ if there is no j satisfying the conditions mentioned above.

The base case is $\text{opt}(n) = 1$ as the longest increasing subsequence starting at index n is only a_n.

Similarly $T(i) = j$ where j is the index that maximizes $\text{opt}(j)$ subject to the conditions $i < j \leq n$ and $a_i \leq a_j$. If there is no such j, we set $T(i)$ to be a null value. Also $T(n)$ is a null value.

Pseudocode. If we just have to return the length of the longest increasing subsequence, we don't need back pointers.

```
// base case
OPT(n) = 1

// main loop
for i = n-1..1:
    OPT[i] = 1 + max([OPT[j] for j > i if a[i] ≤ a[j]])

// final result
return max([OPT[i] for i = 1..n])
```

If we have to return the actual longest increasing subsequence, we need back pointers.

```
// base case
OPT(n) = 1
T(n) = null

// main loop
for i = n-1..1:
    max = 0
    for j = i+1..n:
        if a[i] ≤ a[j] and max < OPT[j]:
            max = OPT[j]
            T[i] = j
    OPT[i] = 1 + max

// final result
i = argmax([OPT[i] for i = 1..n])
seq = [a[i]]
```

```
while T[i] not null:
    i = T[i]
    seq.append(a[i])
return seq
```

Time complexity.

- We have $O(n)$ subproblems,
- each subproblem takes time $O(n)$.

Therefore the overall complexity is $O(n^2)$.

Correctness. We show by reverse induction on the value i that the recursive formulation for opt(i) correctly computes the length of the longest increasing subsequence starting at a_i. Either the longest increasing subsequence starting at a_i is of length 1 or it has a next element at location j for some $i+1 \leq j \leq n$. In the latter case, the length of the longest increasing subsequence starting at i is $1 + \text{opt}(j)$ since opt(j) is the length of the longest increasing subsequence starting at a_j (by induction). Since we are searching through all possible $j > i$ for the next element and selecting j such that opt(j) is maximized, opt(i) will in fact be the length of the longest increasing subsequence starting at a_i.

The base case of the induction happens when $i = n$ in which case $\text{opt}(n) = 1$ as it should be.

Printing the longest increasing subsequence Students are asked to describe the algorithm for printing the longest increasing subsequence (based on the earlier recursive formulation) and prove its correctness.