List of Abbreviations and Symbols

$A[1..n]$	An array indexed from 1 to n of n elements.
$\mathbb{N}$	Set of all natural numbers, i.e., $\{1, 2, 3, \dots\}$.
$\mathbb{R}$	Set of all real numbers.
$\mathbb{Z}$	Set of all integers, i.e., $\{\dots, -2, -1, 0, 1, 2, \dots\}$.
$\exists$	symbol meaning *there exists*.
$\forall$	symbol meaning *for all*.

Modifiers

To help you with what problems to try, problems marked with **[K]** are key questions that tests you on the core concepts, please do them first. Problems marked with **[H]** are harder problems that we recommend you to try after you complete all other questions (or perhaps you prefer a challenge). Good luck!!!

Contents

§1 Optimisation problems

Exercise 1.1. [K] You are given n intervals on an axis. The ith interval $[l_i, r_i)$ has integer endpoints $l_i < r_i$ and has a score of s_i. Your task is to select a set of disjoint intervals with maximum total score. Note that if intervals i and j satisfy $r_i = l_j$ then they are still disjoint. Design an algorithm which solves this problem and runs in $O(n^2)$ time.

The input consists of the positive integer n, as well as $2n$ integers $l_1, r_1, \ldots, l_n, r_n$ and n positive real numbers $s_1, \ldots, s_n$. The output is the set of intervals chosen, organised in any format or data structure.

Example

For example, if $n = 4$ and the intervals are:

$l_1 = 0$	$r_1 = 3$	$s_1 = 2$
$l_2 = 1$	$r_2 = 3$	$s_2 = 1$
$l_3 = 2$	$r_3 = 4$	$s_3 = 4$
$l_4 = 3$	$r_4 = 5$	$s_4 = 3$

then you should select only the first and fourth intervals, for a maximum total score of 5. Note that interval 3 is not disjoint with any other interval.

Solution. Sort the intervals by increasing order of endpoint r_i and relabel accordingly. Henceforth we assume $r_1 \leq \ldots \leq r_n$. We then proceed by dynamic programming.

Subproblems: for $0 \leq i \leq n$, let $P(i)$ be the problem of determining $\operatorname{opt}(i)$, the maximum total score of a set of disjoint intervals where the last chosen interval is the ith, and $m(i)$, the second largest interval index in one such set. If the set consists of only one interval, then $m(i)$ will be zero.

Recurrence: for $1 \leq i \leq n$,

$$m(i) = \operatorname*{argmax}_{j \,:\, r_j \leq l_i} \operatorname{opt}(j)$$

and

$$\operatorname{opt}(i) = s_i + \operatorname{opt}(m(i)).$$

The solution for i must include interval i, so we extend the best solution with last chosen interval j finishing at or before the start of interval i.

Base case: $\operatorname{opt}(0) = 0$ and $m(0)$ is undefined.

Order of computation: subproblem $P(i)$ depends only on earlier subproblems ($P(j)$, where $j < i$), so we can solve the subproblems in increasing order of i.

Final answer: The maximum total score is

$$\max_{1 \leq i \leq n} \operatorname{opt}(i).$$

To recover the set which yields this score, we let

$$i^* = \operatorname*{argmax}_{1 \leq i \leq n} \operatorname{opt}(i),$$

and backtrack through the m array to obtain the set $\{i^*, m(i^*), m(m(i^*)), \ldots\}$.

Time complexity: There are $O(n)$ subproblems, each solved in $O(n)$, and constructing the final answer also takes $O(n)$. Thus the overall time complexity is $O(n^2)$. ■

Exercise 1.2. [K] Due to the recent droughts, n proposals have been made to dam the Murray river. The ith proposal asks to place a dam x_i meters from the head of the river (i.e., from the source of the river) and requires that there is not another dam within r_i metres (upstream or downstream). Design an algorithm that returns the maximal number of dams that can be built, you may assume that $x_i < x_{i+1}$ for all $i = 1, \ldots, n-1$.

Solution.

Subproblems: We can solve this problem by considering the subproblem $P(i)$: *For every $i \leq n$, what is* $\text{opt}(i)$*, the largest possible number of dams that can be built among proposals* $1, \ldots, i$ *such that the ith dam is built.*

Base case: Our base case is then $\text{opt}(1) = 1$ as only dam 1 can be built.

Recurrence: Our recurrence is then

$$\text{opt}(i) = 1 + \max_{j<i} \{\text{opt}(j) \,:\, x_i - x_j > \max(r_i, r_j)\}$$

as we exhaustively search for the optimal previous dam j such that it maximizes our total number of dams (given that dam i must be built) and it satisfies the spacing requirement.

Order of computation: subproblem $P(i)$ depends only on earlier subproblems ($P(j)$, where $j < i$), so we can solve the subproblems in increasing order of i.

Final answer: By our construction, the maximal number of dams that can be built will then be the quantity under $\text{opt}(n)$.

Time complexity: As we need to consider the placement of all $j < i$ possible dams per $P(i)$, the total complexity of our algorithm is $O(n^2)$. ■

Exercise 1.3. [K] You are given a set of n types of rectangular boxes, where the i^{th} box has height h_i, width w_i and depth d_i. You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box. Design an $O(n^2)$ algorithm that returns the maximum height of the stack of boxes.

Hint — *How can we reduce this problem into a simpler problem without rotations?*

Solution. To simplify the problem, we distinguish among all of the rotations. For each box, there are six rotations since for each face, we can exchange its width and height and there are three faces. Thus, consider the problem of having $6n$ types of rectangular boxes, which allows us to assume that there are no rotations involved. We now solve the original problem.

Using merge sort, we can order the boxes in decreasing order based on the surface area of their base (remember that each box is fixed and there are no rotations). In this way, if B_1 can be stacked on top of B_0, then B_0 must appear before B_1 when we ordered the boxes.

Subproblems: We now solve the following subproblem $P(i)$: *what is* $\text{opt}(i)$*, the maximum height possible for a stack if the top box is box number i?*

Base case: $\text{opt}(i) = h_i$ for each i if we cannot place a box below B_i.

Recurrence: Observe that we look at *all* boxes whose base is strictly larger than box B_i and look at what the maximum height can be produced from placing box B_i at the top. That is, for each $1 \leq i \leq 6n$:

$$\text{opt}(i) = \max\{\text{opt}(j) + h_i : \text{ over all } j \text{ such that } w_j > w_i,\ d_j > d_i\}.$$

Order of subproblems: Subproblem $P(i)$ depends on previous subproblems, so we compute these subproblems in increasing order of w_i and d_i.

Final answer: The final solution is simply $\max_{1 \leq i \leq 6n} \text{opt}(i)$.

Time complexity: Ordering the boxes using merge sort takes $O(n \log n)$ time. For the dynamic programming component, we have $6n$ subproblems and for each subproblem, we perform a $O(n)$ search to find boxes whose base is large enough to stack the current box. Thus, we have an $O(n \log n) + O(6n^2) = O(n^2)$ algorithm. ■

Exercise 1.4. [K] You have n_1 items of size s_1 and n_2 items of size s_2. You would like to pack all of these items into bins, each of capacity $C > s_1, s_2$, using as few bins as possible. Design an algorithm that returns the minimal number of bins required.

Solution. Consider the following:

Subproblems: we consider the subproblems $P(i,j)$ of finding $\text{opt}(i,j)$, *the minimal number of bins required to pack i many items of size s_1 and j many items of size s_2.*

Recurrence: For all $1 \leq i \leq n_1$ and all $1 \leq j \leq n_2$. Let $C/s_1 = K$ and $\text{opt}(i,j)$ denote the solution to $P(i,j)$, then our recurrence step is essentially an exhaustive search

$$\text{opt}(i,j) = 1 + \min_{0 \leq k \leq K} \left\{ \text{opt}\left(i - k,\ j - \left\lfloor \frac{C - k\,s_1}{s_2} \right\rfloor \right) \right\}$$

Then, we try all options of placing between 0 and K many items of size s_1 into one single box and then fill the box to capacity with items of size s_2, and optimally packing the remaining items by checking previous values of the subproblems.

Base case: We can compute the base case of $\text{opt}(1,1)$ easily via considering the magnitude of s_1, s_2 and C.

ORder of computation: We can compute the subproblems in increasing orders of j then i as each (i,j) depends on earlier subproblems (similar to knapsack).

Final answer: The minimal number of bins is then just $\text{opt}(n_1, n_2)$.

Time complexity: As for each (i,j) we consider K many possible placements, our algorithm runs in time $O(K\,n_1\,n_2) = O(C\,n_1\,n_2)$.

Note — Such an algorithm runs in exponential time in the size of the input, because input takes only $O(\log_2 C + \log_2 n_1 + \log_2 n_2 + \log_2 s_1 + \log_2 s_2)$ many bits to represent.

■

Exercise 1.5. [K] Consider a row of n coins of values $v_1, v_2, \ldots v_n$, where n is even. We play a game against an opponent by alternating turns. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can definitely win if we move first.

Solution. From this problem, we can see that dynamic programming has a very important role in the field of game theory.

Subproblems: Let us consider the subproblem $P(i, j)$: *What is* $\text{opt}(i, j)$*, the maximal amount of money we can definitely win if we play on the subsequence between coins at the position i and* j. We also add the requirement that such that $j - i + 1$ is an even number and $1 \leq i < j \leq n$.

Recurrence: Our recurrence is then

$$\text{opt}(i, j) = \max \begin{cases} v_i + \min\{\text{opt}(i+2, j), \text{opt}(i+1, j-1)\}, \\ v_j + \min\{\text{opt}(i+1, j-1), \text{opt}(i, j-2)\} \end{cases}$$

The options inside the min functions represent the choice your opponent takes, either to continue taking from the same end as you took or from the opposite end of the sequence.

Order of computation: The problems to find $\text{opt}(i, j)$ are solved in order of the size of $j - i$.

Final answer: The maximum amount of money we can win is $\text{opt}(1, n)$.

Time complexity: The total complexity is $O(n^2)$ as this is how many pairs of i, j we have and each stage of the recursion has only constant number of steps (4 table lookups and 2 additions plus two min and one max operations). ■

Exercise 1.6. [K] Given a sequence of n positive or negative integers $A_1, A_2, \ldots, A_n$, determine a contiguous subsequence A_i to A_j for which the sum of elements in the subsequence is maximised.

Solution.

Subproblems: For each $1 \leq i \leq n$, We solve the following subproblem $P(i)$: *what is* $\text{opt}(i)$*, the maximum sum of elements ending with integer* i*?*

Base case: If there are no elements chosen, the maximum sum of elements is 0; hence, $\text{opt}(0) = 0$.

Recurrence: Suppose that we have chosen the sequence of elements that ends in $i - 1$, this is what we compute when we compute $\text{opt}(i - 1)$. Now, if we are forced to choose A_i (since we always end with integer i), either it'll increase $\text{opt}(i - 1)$ *or* A_i begins a new chain of elements. Thus, we have

$$\text{opt}(i) = A_i + \max(\text{opt}(i-1), 0).$$

To determine the contiguous subsequence, we also keep track of the start element; to do so, we solve the following recursive function. That is, for all $1 \leq i \leq n$,

$$\text{start}(i) = \begin{cases} \text{start}(i-1) & \text{if } \text{opt}(i-1) > 0, \\ i & \text{if } \text{opt}(i-1) \leq 0. \end{cases}$$

Order of subproblems: Since each subproblem depends on the previous subproblems, we solve the subproblems in increasing order of i.

Final answer: We can determine the end of the subsequence by solving the initial subproblems and returning the index i such that $\max_{1 \leq i \leq n} \text{opt}(i)$. To return the contiguous subsequence, we return $[\text{start}(i), i]$.

Time complexity: Each subproblem can be solved in $O(1)$ since we just determine the maximum of $\text{opt}(i-1)$ and 0. There are n subproblems; therefore, the time complexity is $O(n)$. ■

Exercise 1.7. [H] Skiers go fastest with skis whose length is close to their height. Consider n skiers with heights $h_1, h_2, \ldots, h_n$, gets a delivery of $m \geq n$ pairs of skis, with lengths $l_1, l_2, \ldots, l_m$. Design an algorithm to assign to each skier one pair of skis to minimise the sum of the absolute differences between the height h_i of the skier and the length of the corresponding ski they got, i.e., to minimise

$$S = \sum_{i=1}^{n} \left| h_i - l_{s(i)} \right|$$

where $s(i)$ is the index of the skies assigned to the skier of height h_i.

Solution. To start, we observe that

Claim — For any 2 skiers i and j with $h_i < h_j$, then our preferred assignment will have $l_{s(i)} < l_{s(j)}$ (we call this the case of crossover). If this were not the case, we could swap the skis assigned to i and j, which would lower the sum of differences.

Proof. We produce a similar argument to *Tutorial 3 question 4.6.* As the $|\cdot|$ is the distance between 2 points on a number line, we can consider all possible orientation of h_i, h_j and $l_{s(i)}, l_{s(j)}$ and consider the sum of distance between them. After considering all orientations, we will see that swapping the assignment will always produce a lower or equal sum of difference. (*try it yourself!!*) ■

This implies that we may initially sort the skiers by height, sort the skis by length, and find some sort of pairing such that there are no crossovers. We now relabel the sorted height and skis as $h_1, h_2, \ldots, h_n$ and $l_1, l_2, \ldots, l_m$.

Subproblems: Consider the following subproblem $P(i,j)$: *What is* $\text{opt}(i,j)$*, the minimum cost of matching the first i skiers with the first j skies such that each of the first i skiers gets a ski?*

Base case: Our base case is then for all $i = j$,

$$\text{opt}(i,i) = \sum_{k=1}^{i} |h_k - l_k|$$

as this this the case of i skiers are "*forced*" to pick i skis.

Recurrence: Our recurrence for $j > i$ is then

$$\text{opt}(i,j) = \min\{\text{opt}(i,j-1),\, \text{opt}(i-1,j-1) + |h_i - l_j|\}$$

as we consider assigning the skier i with l_j and picking the more optimal option.

Order of computation: Note that for this recurrence we need to again be careful of the order of computation for our subproblems. For the discrete grid domain of $\text{opt}(\cdot)$, for each (i,j), we require $(i, j-1)$ and $(i-1, j-1)$. Therefore, one such valid order is to compute the subproblems in increasing order of j then increasing order of i.

Final answer: We can now use this value to compute our assignment, by starting at $(i,j) = (n,m)$, we start at $(i,\, j)$ where $i = n$ and $j = m$. If $\text{opt}(i-1, j-1) + |h_i - l_j| < \text{opt}(i, j-1)$ then $s(i) = j$ and we try again at $(i-1, j-1)$. Otherwise, we try again at $(i,\, j-1)$. If at any point we reach $i = j$ (our base case), we simply assign $s(k) = k$ for all $1 \leq k \leq i$.

Time complexity: The complexity of our recurrence is then $O(nm)$ for all i, j until $\text{opt}(n,m)$. The complexity of retrieval is $O(m)$, as each step of our retrieval, j decreases by exactly 1. The total complexity of our algorithm is then $O(mn)$. ■

Exercise 1.8. [H] You have been handed responsibility for a business in Texas for the next n days. Initially, you have K illegal workers. At the beginning of each day, you may hire an illegal worker, keep the number of illegal workers the same or fire an illegal worker. At the end of each day, there will be an inspection. The inspector on the i^{th} day will check that you have between l_i and r_i illegal workers (inclusive). If you do not, you will fail the inspection. Design an algorithm that determines the fewest number of inspections you will fail if you hire and fire illegal employees optimally.

Solution. We start by noticing that:

Note — Minimum and maximum illegal workers you could possible have on the evening of day i are $\max(K-i, 0)$ and $K+i$.

Subproblems: we can consider the subproblem $P(i, j)$: *What is* $\text{opt}(i, j)$*, the minimum number of inspections I have failed on day i assuming that on that day I have j many illegal workers?*

Base case: Our base case is $\text{opt}(0, K) = 0$, since we start with 0 failed inspections before the first day begins.

Recurrence: Our recurrence for $1 \le i \le n$ and all j such that $\max(0, K-i) \le j \le K+i$,

$$\text{opt}(i, j) = \text{failed}(i, j) + \min \begin{cases} \text{opt}(i-1, j-1) & \text{if } j-1 \ge \max(0, K-(i-1))), \\ \text{opt}(i-1, j), \\ \text{opt}(i-1, j+1) & \text{if } j+1 \le K+i-1. \end{cases}$$

Here, $\text{failed}(i, j)$ returns 1 if the j falls out of the range of $[l_i, r_i]$, and returns 0 otherwise. Note that the first option in the cases corresponds to hiring an illegal worker on the morning of day i, providing that the number of workers the second corresponds to keeping the same number of illegal workers as on the previous day and the third option corresponds to firing a worker from the previous day.

Order of computation: we can compute the subproblems in increasing order of j then i.

Final answer: The final answer is then

$$\min\{\text{opt}(n, j) : \max(K-n, 0) \le j \le K+n\}.$$

Time complexity: The total complexity is $O(n^2)$, as there are n days, and at most $2n+1$ possible values of illegal worker each day. ■

Exercise 1.9. [H] A company is organising a party for its employees. The organisers of the party want it to be a fun party, and so have assigned a fun rating to every employee. The employees are organised into a strict hierarchy, i.e. a tree rooted at the president. There is one restriction, though, on the guest list to the party: an employee and their immediate supervisor (parent in the tree) cannot both attend the party (because that would be no fun at all). Give an algorithm that makes a guest list for the party that maximises the sum of the fun ratings of the guests.

Solution. Suppose that $T(i)$ defines the subtree of the tree T of all employees that is rooted by employee i. Then, for each subtree, we will look at the maximum sum of the fun ratings in two ways: one which includes employee i and one that does not include employee i.

For each subtree, we define the following quantities:

- $I(i)$ is the maximal sum of fun factors $\text{fun}(i)$ that satisfies all of the constraints but *includes* root i,

- $E(i)$ is the maximal sum of fun factors fun(i) that satisfies all of the constraints but *excludes* root i.

With these quantities defined, we now perform the dynamic programming approach.

Subproblems: We solve the following subproblem $P(i)$: *What is* opt(i)*, the maximum sum of the fun ratings of* $T(i)$?

Base case: For each i that is a leaf node, $\text{opt}(i) = \text{fun}(i)$.

Recursion: For the recursion, we need to compute two quantities, $I(i)$ and $E(i)$. For each non-leaf node, we need to consider excluding the subordinates (because i is the immediate supervisor of these workers) if we choose to include employee i, or if we exclude i, then we can either choose to exclude the children or include the children of i, whichever value is greater. Thus, we compute $I(i)$ by

$$I(i) = \text{fun}(i) + \sum_{1\le k\le m} E(j_k),$$

where $j_1, \dots, j_m$ are the subordinates of i.

We compute $E(i)$ by

$$E(i) = \sum_{1\le k\le m} \max\left(I(j_k), E(j_k)\right),$$

where $j_1, \dots, j_m$ are defined as above. In this way, *for each* child of i, we can either include them or exclude them.

Then solving $P(i)$ is simply a matter of $\max(I(i), E(i))$.

Order of subproblems and Final answer: We solve the subproblems from the leaves of the tree to the root of the leaves, where the final solution is simply $\text{opt}(n) = \max(I(n), E(n))$ where n is the root of the tree.

Time complexity: The time complexity is linear in the number of employees because we only need to scan through each of the employees once. Each employee only has one parent. In the worst case, we need to ask every employee exactly once. ■

Exercise 1.10. [H] You have to cut a wood stick into several pieces. The most affordable company, Analog Cutting Machinery (ACM), charges money according to the length of the stick being cut. Their cutting saw allows them to make only one cut at a time. It is easy to see that different cutting orders can lead to different prices. For example, consider a stick of length 10 m that has to be cut at 2, 4, and 7 m from one end. There are several choices. One can cut first at 2, then at 4, then at 7. This leads to a price of $10 + 8 + 6 = 24$ because the first stick was of 10 m, the resulting stick of 8 m, and the last one of 6 m. Another choice could cut at 4, then at 2, then at 7. This would lead to a price of $10 + 4 + 6 = 20$, which is better for us. Your boss demands that you design an $O(n^2)$ algorithm to find the minimum possible cutting cost for any given stick.

Solution. Define $x(i)$ to be the distance along the stick where the ith cut occurs. We, then, approach with dynamic programming.

Subproblems: We solve the following subproblem $P(i, j)$: *what is* opt(i, j)*, the minimum cost of cutting up the stick from the cutting point* i *to cutting point* j?

Base case: Fixing i, the base case occurs when $j = i + 1$ in which case, $\text{opt}(i, i + 1) = 0$ for each i.

Recursion: Choosing to cut at points i and j will decrease the distance to $x(j)-x(i)$. In this interval, we then need to find the smallest possible cut(s) in terms of cost; therefore, we need to compute $\text{opt}(i, k) + \text{opt}(k, j)$

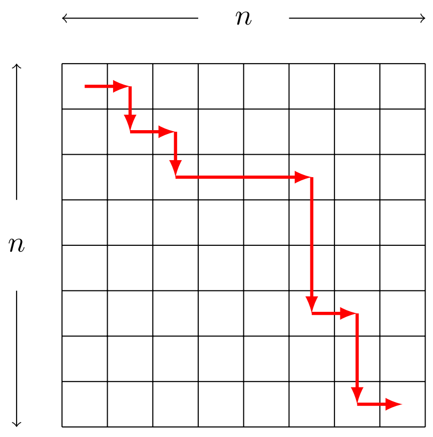
for each $i < k < j$. Thus, we have

$$\text{opt}(i,j) = x(j) - x(i) + \min\{\text{opt}(i,k) + \text{opt}(k,j) : 1 < k < j\}.$$

Order of computation and Final answer: We solve each subproblem in increasing order of j and then increasing order of i with the final solution being $\text{opt}(1,n)$.

Time complexity: For each i, we need to solve $n-1$ subproblems. Thus, there are $O(n^2)$ many subproblems, each of which take constant time. Thus, the time complexity is $O(n^2)$. ■

Exercise 1.11. [H] You are given an $n \times n$ chessboard with an integer in each of its n^2 squares. You start from the top left corner of the board; at each move you can go either to the square immediately below or to the square immediately to the right of the square you are at the moment; you can never move diagonally. The goal is to reach the right bottom corner so that the sum of integers at all squares visited is minimal.



(a) Describe a greedy algorithm which attempts to find such a minimal sum path and show by an example that such a greedy algorithm might fail to find such a minimal sum path.

(b) Describe an algorithm which always correctly finds a minimal sum path and runs in time n^2.

(c) Describe an algorithm which computes the number of such minimal paths.

(d) Assume now that such a chessboard is stored in a read only memory. Describe an algorithm which always correctly finds a minimal sum path and runs in linear space (i.e., amount of read/write memory used is $O(n)$) and in time $O(n^2)$.

> **Hint** — *Combine divide and conquer with dynamic programming.*

Solution.

(a) Consider the following greedy solution. Starting at the top left square, we move to the square below or to the right that contains the smallest integer.

Using this solution, consider the following counterexample.

0	1	1
5	1000	1000
5	5	0

The algorithm would choose the path 1-1-1000-0 for a score of 1002, while the correct path would be 5-5-5-0 for a score of 15.

(b) We define the quantity board(i, j) to be the integer inside the cell located on row i and column j. We then begin with the dynamic programming approach.

Subproblem: We solve the subproblem $P(i, j)$: *what is* opt(i, j)*, the best score we can get if we arrive at the cell* (i, j)?

Base case: The base case is $\text{opt}(1, 1) = \text{board}(1, 1)$. To ensure that we never consider illegal moves (i.e. moves that land outside of the board), we also have that $\text{opt}(i, j) = \infty$ if cell (i, j) does not exist (or is off the board).

Recursion: If we land at cell (i, j), observe that the previous move must have been either $(i - 1, j)$ (i.e. we move to the right to land at the cell) or $(i, j - 1)$ (i.e. we move downwards to land at the cell). Thus, we need to check which way yielded us with the minimum value *so far*. Another way to see this is that, if we land at the final square (n, n), then the path that we would need to trace needs to be the path where it is minimal ending at either cell $(n - 1, n)$ or $(n, n - 1)$, and we repeat this process until we land back at the original square.

Thus, the recursion is

$$\text{opt}(i, j) = \text{board}(i, j) + \min\{\text{opt}(i - 1, j), \text{opt}(i, j - 1)\}.$$

Order of subproblem: Observe that, in order to solve opt(i, j), we need to know opt$(i - 1, j)$ and opt$(i, j - 1)$. Thus, the order in which we solve the problem is that we solve the subproblems in increasing order of i, and then increasing order of j to break ties.

Final solution: The final solution is simply solving opt(n, n). To return the actual path, we can make a note of the choice that the recursion makes by choosing either opt$(i - 1, j)$ or opt$(i, j - 1)$ by backtracking through the subproblems that are chosen. This gives us a path from (n, n) to $(1, 1)$ which yields the path taken.

Time complexity: To analyse the time complexity, we require two pieces of information: the number of subproblems and the time taken to compute each subproblem. There are n^2 subproblems for each $1 \leq i, j \leq n$. Now, for each subproblem $P(i, j)$, we would have computed opt$(i - 1, j)$ and opt$(i, j - 1)$ previously. Hence, we have $O(1)$ lookup time and the computation for opt(i, j) is simply a comparison and an arithmetic operation, all of which takes $O(1)$ time. Thus, the overall time complexity is $n^2 \cdot O(1) = O(n^2)$.

(c) We approach this similar to part (b); in fact, we will use opt(i, j) as defined above to our advantage.

Subproblem: We solve the subproblem $Q(i, j)$: *what is* ways(i, j)*, the minimum number of ways to reach cell* (i, j) *with the score of* opt(i, j)? Recall that opt(i, j) refers to the *minimum path* landing at cell (i, j).

Base case: The base case is $\text{ways}(1, 1) = 1$ since there is only one way to get to cell $(1, 1)$. We also define $\text{ways}(i, j) = 0$ for all cells that fall outside of the grid since there are no ways to get to those cells.

Recursion: In much the same way, if we land at cell (i, j), the previous move must have been either from $(i - 1, j)$ or $(i, j - 1)$. We, therefore, need to check the minimum path from both of these directions. There are three cases to consider:

- If the minimum path happens to come from $(i - 1, j)$, then we ignore all of the paths from $(i, j - 1)$ because those paths will not be minimal.

- If the minimum path happens to come from $(i, j-1)$, then we ignore all of the paths from $(i-1, j)$.
- However, if they are equal, then we consider *all* possible paths to land at (i, j).

With this in mind, it is easy to see that the recursion becomes

$$\text{ways}(i,j) = \begin{cases} \text{ways}(i-1,j) & \text{if } \text{opt}(i-1,j) < \text{opt}(i,j-1), \\ \text{ways}(i,j-1) & \text{if } \text{opt}(i-1,j) > \text{opt}(i,j-1), \\ \text{ways}(i-1,j) + \text{ways}(i,j-1) & \text{if } \text{opt}(i-1,j) = \text{opt}(i,j-1). \end{cases}$$

Order of subproblem: We solve the problems in increasing order of i, and then increasing order of j to break ties for the same reason as part (b).

Final solution: The final solution is simply solving ways(n, n).

Time complexity: For the same reason as part (b), the time complexity is $O(n^2)$ since there are still n^2 subproblems and each subproblem takes $O(1)$ time to compute.

(d) This is a very tricky one. The idea is to combine divide and conquer with dynamic programming. Note that to generate optimal scores at a row i you only need the optimal scores of the previous row. You start running the previous algorithm from the top left cell to the middle row $\lfloor n/2 \rfloor$, keeping in memory only the previous row. You now run the algorithm from the bottom right corner until you reach the middle row, always going either up one cell or to the left one cell. Once you reach the middle row you sum up the scores obtained by moving down and to the right from the top left cell and the scores obtained by moving up and to the left from the bottom right cell and you pick the cell $C(n/2, m)$ in that row with the minimal sum. This clearly is the cell on the optimal trajectory. You now store the coordinates of that cell and proceed with the same strategy applied to the top left region for which $C(n/2, m)$ is bottom right cell, and also applied to the bottom right region of the board for which $C(n/2, m)$ is the top left cell. The run time is $O(n \times n + n \times n/2 + n \times n/4 + \ldots) = O(n \times 2n) = O(n^2)$.

■

§2 Enumeration problems

Exercise 2.1. [K] Some people think that the bigger an elephant is, the smarter it is. To disprove this, you want to analyse a collection of elephants and place as large a subset of elephants as possible into a sequence whose weights are increasing but their IQ's are decreasing. Design an $O(n \log n)$ algorithm which, given the weights and IQ's of n elephants, will find a longest sequence of elephants such that their weights are increasing but IQ's are decreasing.

Hint — *How does this relate to the longest subsequence problem?*

Solution. Using merge sort (or equivalent), sort the elephants in decreasing order of IQ. This problem now reduces to the longest increasing subsequence problem in terms of the weight of the elephants. Sorting takes $O(n \log n)$ and the longest increasing subsequence problem also takes $O(n \log n)$. Hence, the overall time complexity is $O(n \log n)$. ■

Exercise 2.2. [K] There are n levels to complete in a video game. Completing a level takes you to the next level, however each level has a secret exit that lets you skip to another level later in the game. Determine if there is a path through the game that plays exactly K levels.

Solution.

Subproblems: We consider $P(i, k)$: *Can I reach the ith level after playing exactly k levels ?* Let $\text{opt}(i, k)$ denote the solution for $P(i, k)$.

Base case: Our base case is then $\text{opt}(0, 0) = T$ as the reaching 0 levels is trivial.

Recurrence: Our recurrence is then

$$\text{opt}(i, k) = \text{opt}(i - 1, k - 1) \vee (\exists j < i - 1 : (\text{opt}(j, k - 1) \wedge \text{link}(j, i)))$$

for the logical operators *and* ($\wedge$) and *or* ($\vee$). Here $\text{link}(j, i)$ is true if and only if level j has a secret exit to level i. Our recurrence is then just considering the optimal option considering skipping the level or proceeding normally.

Final answer: Therefore, the final answer is $\text{opt}(K, n)$.

Time complexity: There are $O(nK)$ subproblems in total, and it appears at first that we need to iterate through all $j < i$ for each one. However, we can do better. Since only n links exist, we can precompute for each level i which earlier levels j link to it, and iterate through only this list. In this way, all subproblems $O(\cdot, k)$ take a total of $O(n)$ time, and therefore the total time complexity is $O(nK)$. ■

Exercise 2.3. [K] A partition of a number n is a sequence $\langle p_1, p_2, \ldots, p_t\rangle$ (we call the p_k *parts*) such that $1 \leq p_1 \leq p_2 \leq \ldots \leq p_t \leq n$ and such that $p_1 + \ldots + p_t = n$. Define $\text{numpart}(k, n)$ to be the number of partitions of n such that every part is at most k.

(a) Explain why $\text{numpart}(1, n) = 1$ for each n.

(b) Devise a recursion to determine the number of partitions of n in which every part is smaller or equal than k, where n, k are two given numbers such that $2 \leq k \leq n$.

(c) Hence, find the total number of partitions of n.

Solution.

(a) Note that $\text{numpart}(1, n)$ counts the number of ways of expressing n such that every part is at most 1. This can be done precisely when every part is 1.

(b) The recursion is done similar to Exercise 1.9 – we split the cases into the case where the greatest part is precisely k and the case where the greatest part is not k.

For the first case, suppose that we fix some part to be k. This gives us something like this:

$$p_1 + \cdots + k + \cdots + p_t = n.$$

By subtracting k from both sides, we are now looking at the number of partitions of $n - k$ whose greatest part is at most k. This is precisely $\text{numparts}(k, n - k)$.

For the second case, the greatest part is at most $k - 1$ and we want to partition n into parts at most $k - 1$. Thus, the second case is simply $\text{numpart}(k - 1, n)$.

Since each of these cases are disjoint, cases will not overlap (i.e. you can't have a case where the greatest part is k and not k simultaneously). Therefore, the number of partitions is simply

$$\text{numparts}(k, n) = \text{numparts}(k, n - k) + \text{numparts}(k - 1, n).$$

We solve these subproblems in the increasing order of $n + k$.

(c) The total number of partitions does not restrict ourselves to what the greatest part of the partition is; thus, the total number of partitions is simply numparts(n, n) which is computed as before.

■

Exercise 2.4. [K] Given an array of n positive integers, find the number of ways of splitting the array up into contiguous blocks of sum at most k in $O(n^2)$ time.

Solution.

Subproblems: We solve the following subproblem $P(i)$ of finding opt(i): *what is the number of ways that I can split the first i elements into contiguous blocks of size k?*

Base case: opt$(0) = 1$ since there's exactly 1 way of splitting 0 elements into contiguous blocks of size k (or less).

Recursion: Define sum(j, i) to be the sum of all numbers from j to i (inclusive). The recursion becomes

$$\text{opt}(i) = \sum_{\substack{1 \le j \le i, \\ \text{sum}(j,i) \le k}} \text{opt}(j-1).$$

Final answer: The final solution is simply opt(n).

Time complexity: The time complexity is $O(n^2)$ since there are n subproblems and for each subproblem, we make an $O(n)$ search. ■

Exercise 2.5. [H] You are given a boolean expression consisting of a string of the symbols *true* (T) and *false* (F) and with exactly one operation of *and* $(\wedge)$, *or* $(\vee)$ or *xor* $(\oplus)$ between any two consecutive symbols(truth values). Count the number of ways to place brackets in the expression such that it will evaluate to true.

Example 2.1 (Simple expression)

There are 2 ways to place parentheses in the expression

$$T \wedge F \oplus T$$

such that it evaluates to true, namely $T \wedge (F \oplus T) \iff T$ and $(T \wedge F) \oplus T \iff T$.

Solution. We start by defining some notations, let $s_1, \dots, s_n$ denote the given n symbols and let $o_1, \dots, o_{n-1}$ denote the given $n - 1$ operations.

Subproblems: Then we can solve this problem by considering the 2 subproblems,

(a) $P(l, r)$: *how many ways are there to make the expression starting from at the s_l and ending at s_r (including all operations in between s_l and s_r) evaluate to T?*

(b) $Q(l, r)$: *by inheriting the setup of l, r from above, how many ways evaluates to F?*

We then now let $p(l, r)$ denote the solution of $P(l, r)$ and $q(l, r)$ for the solution of $Q(l, r)$.

Example 2.2 (Definition p)

For $T \wedge F \oplus T$, $P(1,2)$ will denote the number of ways of making $T \wedge F$ evaluate to T with the correct bracketing, which in this case $p(1,2) = 0$.

Base case: We can now define our base case, note that if $l = r$, then $p(\cdot)$ and $q(\cdot)$ can be easily determined basing on the value of the symbol, hence $\forall i : l = r = i$, then

$$p(i,i) = \begin{cases} 1 & s_i = T \\ 0 & s_i = F \end{cases} \quad q(i,i) = \begin{cases} 1 & s_i = F \\ 0 & s_i = T \end{cases}$$

if we consider $p(\cdot)$ and $q(\cdot)$'s value on a grid in $\mathbb{N}^2$, then our base cases will fill the value for the diagonal of our grid. Also, note that we will only be filling for values $l < r$, hence our grid will be upper-triangular.

Recurrence: For each subproblem, we also notice that as each of the operators is binary, we can "*split*" the expression around an operator o_m so that everything to the left of the operator is in its own bracket, and everything to the right of the operator is in its own bracket to form 2 smaller expressions. We give an illustration via

$$\underbrace{(s_1\ o_1 \ldots s_{m-1}\ o_{m-1}\ s_m)}_{\text{subproblem 1}}\ o_m\ \underbrace{(s_{m+1}\ o_{m+1} \ldots o_{n-1}\ s_n)}_{\text{subproblem 2}}.$$

Example 2.3

For a simple example, the expression $T \wedge F \oplus T$, if we split around $\oplus$ we have $(T \wedge F) \oplus (T)$.

Now we have a way to extend our solution to the general case of (l,r)! We can evaluate the subproblems on each of the 2 sides, and combine the results depending on the type of o_m and whether we want the result to evaluate to T or F (function value of p or q). Note that we actually require both P and Q to combine the result for the general case!! Let us define 2 operators that helps us in determining the overall solution,

$$\Gamma_p(l,m,r) = \begin{cases} p(l,m) \times p(m+1,r) & o_m = \wedge \\ p(l,m) \times p(l,m) \times q(m+1,r) + p(l,m) \times p(m+1,r) + q(l,m) \times p(m+1,r) & o_m = \vee \\ p(l,m) \times q(m+1,r) + q(l,m) \times p(m+1,r) & o_m = \oplus \end{cases}$$

$$\Gamma_q(l,m,r) = \begin{cases} p(l,m) \times q(m+1,r) + q(l,m) \times q(m+1,r) + q(l,m) \times p(m+1,r) & o_m = \wedge \\ q(l,m) \times q(m+1,r) & o_m = \vee \\ p(l,m) \times p(m+1,r) + q(l,m) \times q(m+1,r) & o_m = \oplus \end{cases}$$

We here note that $\Gamma_p(\cdot)$ is the value of $p(l,r)$ if we decide to split on o_m (same for $\Gamma_q(\cdot)$ but for $q(l,r)$). Each of those expressions of Γ is a result based on the truth table of o_m. We will here give reasoning behind one case for the expression of Γ_p, the rest of them will be *left as exercise*. For the case of $o_m = \wedge$, both subsections can evaluate to true $\iff$

- the way the brackets are placed on left sections belong to one of the ways considered by $p(l,m)$
- the way the brackets are placed on right sections belong to one of the ways considered by $p(m+1,n)$

Therefore, we can simply count the number of permutation by setting $p(l,m) \times p(m+1,n)$. Lastly, we can simply consider all the possible *splits* on each of the existing operations and combine them. Our final expression is then

$$p(l,r) = \sum_{m=l}^{r-1} \Gamma_p(l,m,r) \quad q(l,r) = \sum_{m=l}^{r-1} \Gamma_q(l,m,r).$$

Order of computation: However, we should be careful of 2 things:

- to ensure that the recurrence can be given correct, we need to compute $p(l,r)$ and $q(l,r)$ in parallel
- order that we compute our subproblems is also important as each $p(l,r)$ requires all of $p(l,l+1),\ldots,p(l,r-1)$ and $p(r-1,r),\ldots,p(l+1,r)$ (same for $q(l,r)$). Therefore, one valid order of computation is solving our subproblem column by column from down to up (with no values considered below the diagonal) on our grid.

Final answer: By our problem construction, the final solution is just the quantity $p(1,n)$.

Time complexity: Lastly, we see that the time complexity of our algorithm is $O(n^3)$ as there are n^2 different ranges that l and r could cover, and each needs the evaluations of $\Gamma_p(\cdot)$ or $\Gamma_q(\cdot)$ at up to n different splitting points. ■

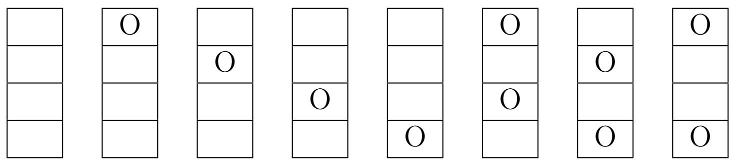
Exercise 2.6. [H] We are given a checkerboard which has 4 rows and n columns, and has an integer written in each square. We are also given a set of $2n$ pebbles, and we want to place some or all of these on the checkerboard (each pebble can be placed on exactly one square) so as to maximise the sum of the integers in the squares that are covered by pebbles. There is one constraint: for a placement of pebbles to be legal, no two of them can be on horizontally or vertically adjacent squares (diagonal adjacency is fine).

(a) Determine the number of legal *patterns* that can occur in any column (in isolation, ignoring the pebbles in adjacent columns) and describe these patterns.

Call two patterns *compatible* if they can be placed on adjacent columns to form a legal placement. Let us consider sub-problems consisting of the first k columns $1 \le k \le n$. Each sub-problem can be assigned a type, which is the pattern occurring in the last column.

(b) Using the notions of compatibility and type, give an $O(n)$-time algorithm for computing an optimal placement.

Solution. **(a)** There are 8 patterns, listed below.



(b) Let t denote the type of pattern, so that t ranges from 1 to 8. We solve our subproblem by choosing k columns from our set of patterns, such that each column is compatible with the one before and after. The subproblem is: What is the maximum score we can get by only placing pebbles in the first k columns, such the k^{th} column has pattern t. The base case is that $\text{opt}(0) = 0$. The recursion is:

$$\text{opt}(k,t) = \text{score}(k,t) + \max\{\text{opt}(k-1,s) : s \text{ is compatible with } t\}.$$

Here, $\text{score}(k,t)$ is the score obtained by using pattern t on column k. The complexity is $O(n)$, as the number of patterns is constant.

■

§3 Knapsack and related problems

Exercise 3.1. [K] You have an amount of money M and you are in a candy store. There are n kinds of candies and for each candy, you know how much pleasure you get by eating it, which is a number between 1 and K, as well as the price of each candy. Your task is to choose which candies you are going to buy to maximise the total pleasure you will get by gobbling them all.

Solution. This is a knapsack problem with duplicated values. The pleasure score is the value of the item, the cost of a particular type of candy is its weight, and the money M is the capacity of the knapsack. The complexity is $O(Mn)$. ■

Exercise 3.2. [K] Your shipping company has just received N shipping requests (jobs). For each request i, you know it will require t_i trucks to complete, paying you d_i dollars. You have T trucks in total. Out of these N jobs you can take as many as you would like, as long as no more than T trucks are used total. Devise an efficient algorithm to select jobs that will bring you the largest possible amount of money.

Solution. We can recognize that this is nothing but the standard knapsack problem with t_i being the size of the ith item, d_i its value and with T as the capacity of the knapsack. Since each transportation job can be executed only once, it is a knapsack problem with no duplicated items allowed. The complexity follows the knapsack problem which results to be $O(NT)$. ■

Exercise 3.3. [K] Again your shipping company has just received N shipping requests (jobs). This time, for each request i, you know it will require e_i employees and t_i trucks to complete, paying you d_i dollars. You have E employees and T trucks in total. Out of these N jobs you can take as many of them as you would like, as long as no more than E employees and T trucks are used in total. Devise an efficient algorithm to select jobs which will bring you the largest possible amount of money.

Solution. This is a slight modification of the knapsack problem with two constraints on the total size of all jobs; think of a knapsack which can hold items of total weight not exceeding E units of weight and total volume not exceeding T units of volume, with item i having a weight of e_i integer units of weight and t_i integer units of volume.

Subproblems: For each triplet $e \leq E,\ t \leq T, i \leq N$ we solve the following subproblem: choose a subcollection of items $1 \dots i$ that fits in a knapsack of capacity e units of weight and t units of volume, which is of largest possible value, putting such a value in a 3D table of size $E \times T \times N$ where N is the number of items (in this case jobs).

Recurrence: These values are obtained using the following recursion:

$$\text{opt}(i, e, t) = \max\{\text{opt}(i-1, e, t),\ \text{opt}(i-1, e-e_i, t-t_i) + d_i\}.$$

The complexity is $O(NET)$. ■

§4 Strings

Exercise 4.1. [K] We say that a string A occurs as a subsequence of another string B if we can obtain A by deleting some of the letters of B. Design an algorithm that for two strings A and B gives the number of different occurrences of A in B, i.e., the number of ways one can delete some of the symbols of B to get A.

Example 4.1

The string $A =$ "ba" has three occurrences as a subsequence of string $B =$ "$baba$":

$$\text{“}\underline{ba}ba,\ ba\underline{ba},\ \underline{b}ab\underline{a}\text{”}.$$

Solution. We again begin by defining some notations, let A_i denote the ith letter of A and B_j as the jth letter of B with $|A| = n$ and $|B| = m$.

Subproblems: We consider the subproblem $P(i, j)$: *How many times do the first i letters of A appears as a subsequence of the first j letters of B?* Let $\text{ways}(i, j)$ denote the solution to $P(i, j)$.

Base case: Our base case is then

- $\text{ways}(0, j) = 1$ for $1 \le j \le m$ (empty string for A)
- $\text{ways}(i, 0) = 0$ for $1 \le i \le n$ (B being empty)

Recurrence: Our recurrence is then

$$\text{ways}(i, j) = \begin{cases} \text{ways}(i, j-1), & \text{if } A_i \neq B_j \\ \text{ways}(i-1, j-1) + \text{ways}(i, j-1) & \text{if } A_i = B_j. \end{cases}$$

Note that in the second case when $A_i = B_j$ we have two options: we can map A_i into B_j because they match, but we can also map the first i many letters of A into $j - 1$ many letters of B, so we have the sum of these two options.

Order of computation: We compute the subproblems in increasing order of j then i.

Final answer: Our final solution is then $\text{ways}(n, m)$.

Time complexity: As we need to compute all possible values of (i, j), our final complexity is then $O(mn)$. ■

Exercise 4.2. [K] For bit strings $X = x_1 \dots x_n$, $Y = y_1 \dots y_m$ and $Z = z_1 \dots z_{n+m}$, we say that Z is an interleaving of X and Y if it can be obtained by interleaving the bits in X and Y in a way that maintains the left-to-right order of the bits in X and Y.

Example

If $X = 101$ and $Y = 01$ then $x_1\, x_2\, y_1\, x_3\, y_2 = 10011$ is an interleaving of X and Y, whereas 11010 is not.

Design an efficient algorithm to determine if Z is an interleaving of X and Y.

Solution.

Subproblems: We consider the following subproblem $P(i, j)$: *Do the first i bits of X and the first j bits of Y form an interleaving in the first $i + j$ bits of Z?* Let $\text{opt}(i, j)$ be the boolean solution (T or F) of $P(i, j)$.

Base case: Our base case is $\text{opt}(0, 0) = T$ for the empty bit string. Note that we also set $\text{opt}(i, j) = F$ if $i < 0$ or $j < 0$.

Recurrence: Our recurrence is then

$$\text{opt}(i, j) = (\text{opt}(i-1, j) \wedge (z_{i+j} = x_i)) \vee (\text{opt}(i, j-1) \wedge (z_{i+j} = y_j))$$

for the logical operators *and* ($\wedge$) and *or* ($\vee$). Note our recurrence works as we check both cases of z_{i+j} being the extension of x_i or y_j.

Order of computation: We compute the subproblems in increasing order of j then i.

Final answer: Our final solution is then $\text{opt}(m, n)$.

Time complexity: Since we need to compute all of the subproblems up to $\text{opt}(m, n)$, our total complexity is $O(nm)$. ■

Exercise 4.3. [H] A palindrome is a sequence of at least two letters which reads equally from left to right and from right to left.

Example 4.2
The string "*ababa*" is a palindrome of size 5.

Given a sequence of letters (a string), find efficiently its longest subsequence (not necessarily contiguous) that is a palindrome. In other words, we are looking for the longest palindrome which can be obtained by crossing out some of the letters of the initial sequence without permuting the remaining letters.

Solution. We begin by defining some notations, let S denote the given string with length of S be n (i.e., $|S| = n$), s_i denote the ith letter.

Subproblems: We then consider the subproblem $P(l, r)$: *What is the longest palindrome within the substring starting at s_l ending at s_j?* Let the length of the solution to $P(l, r)$ be $\text{opt}(l, r)$,

Base case: Our base case is $\text{opt}(1, 1) = 1$ as any string of length 1 is a palindrome.

Recurrence: Our recurrence is then

$$\text{opt}(l, r) = \begin{cases} 1 & \text{if } l = r, \\ 2 & \text{if } l + 1 = r \text{ and } s_l = s_r, \\ \text{opt}(l + 1, r - 1) + 2 & \text{if } s_l = s_r \text{ and } l < r - 1, \\ \max\{\text{opt}(l, r - 1), \text{opt}(l + 1, r)\} & \text{otherwise} \end{cases}$$

Again, we can deduce the expression for the cases via the definition, but majorly, the 2nd and 3rd cases represent the cases where we can extend the maximum length by 2 as the characters at the end of our subsequences matches. If we cannot extend our maximal sequence, then we simply take the maximum of the 2 cases of extending l or r.

Order of computation: From the definition of our $\text{opt}(\cdot)$, we need to compute them in the increasing order of i then j.

Final answer: As we know the maximal length, we now setup a process to retrieve the indexes of the actual palindrome, consider

$$\text{pred}[(l, r)] = \begin{cases} (l, l) & \text{if } l = r, \\ (l, r) & \text{if } l + 1 = r \text{ and } s_l = s_r, \\ (l + 1, r - 1) & \text{if } s_l = s_r \text{ and } l < r - 1, \\ (l, r - 1) & \text{if } s_l \neq s_r \text{ and } \text{opt}(l, r - 1) \geq \text{opt}(l + 1, r) \\ (l + 1, r) & \text{otherwise} \end{cases}$$

we can then backtrack starting with $\text{pred}(1, n)$ and recording values of l and r when $s_l = s_r$.

Time complexity: Computing $\text{opt}(l, r)$ requires us to consider half of the grid given by the value combination of (l, r) and the retrieval process will produce a path in the grid of (l, r). This tells us the total complexity of our algorithm is then $O(n^2)$. ■

Exercise 4.4. [K] Consider a 2-D map with a horizontal river passing through its centre. There are n cities on the southern bank with x-coordinates $a_1 \dots a_n$ and n cities on the northern bank with x-coordinates $b_1 \dots b_n$. You want to connect as many north-south pairs of cities as possible, with bridges such that no two bridges cross. Design an $O(n^2)$ algorithm to determine the maximum number of such pairs.

Hint — *Consider how the longest common subsequence problem can help us here.*

Exercise 4.5. [H] A *context-free grammar* is a tuple $\langle V, \Sigma, R, S\rangle$, where:

- V is a set of variables;
- Σ is a finite set of symbols used to define characters of a word;
- R is a set of production rules that determines how the strings can be generated by the grammar;
- S is the start variable.

The rules of a grammar come in the form:

- $A \to \alpha$ where A is a variable and α is a symbol,
- $A \to BC$ where A, B, C are variables and B, C are not the start variable,
- $S \to \epsilon$ where ϵ is the empty character.

To generate a string, we use the production rules given above and keep performing substitutions until we get to a point where there are no variables. That is, we repeatedly replace variables with the corresponding symbol or variable given by the production rules.

(a) Consider the context-free grammar, $G = \langle V, \Sigma, R, S\rangle$, where

- $V = \{A, B, C\}$,
- $\Sigma = \{0, 1\}$,
- R is given by the production rules:
  - $A \to \epsilon$ or $A \to BC$,
  - $B \to 0$ or $B \to BB$,
  - $C \to 1$ or $C \to CC$,
- $S = A$.

We can generate the string 01 because, starting with $A \to BC$, we can replace B with 0 and C with 1. Thus, we generate $A \to 01$ and so, G generates 01. Using these production rules, find another string that can be generated by G.

We now attempt to solve the membership problem; that is, given an arbitrary context-free grammar G and a string w, does G generate w?

(b) Suppose that w is a symbol. Devise an $O(1)$ algorithm to determine if G generates w.

(c) Now suppose that w is a string of length 2. Devise a recursion to determine if G generates w.

(d) Now suppose that w is a string of length n. Devise an $O(n^3)$ algorithm to determine if G generates w.

> **Note** — Note that this formally depends on the size of the grammar but the size of the grammar is independent of the length of the string, so we treat the size of the grammar as a fixed constant.

Solution. **(a)** We can generate $w = 0011$. Starting with A, we can either generate ϵ or BC. If we generate BC, then we can either generate 0 or BB. Thus, a derivation is as follows:

Derivation	String
$A \to BC$	$w = BC$
$B \to BB$, $C \to CC$	$w = BBCC$
$B \to 0$, $C \to 1$	$w = 0011$

(b) Start by constructing a size 1×1 table for the character in w. For each variable A in the grammar G, if $A \to w$ is a production rule, we record A in the cell $(1, 1)$. Then we just check if S belongs in the cell. If so, we accept w; otherwise, we reject. This formally requires one scan of the entire grammar so the time complexity is $O(|G|)$; however, the size of the grammar is independent of the input length of w.

(c) If w is a string of size 2, then we proceed similarly to a string of size 1. Construct a 2×2 table. For each rule of the form $A \to BC$, if we're not in a (i, i) cell, we check the next cell across and the cell directly above. If the (i, k) cell contains B and $(k + 1, j)$ contains C, then (i, j) contains A. To determine if $w \in G$, we check to see if the start variable S belongs in the cell $(1, 2)$. If so, we accept w; otherwise, reject w.

(d) We repeat this process until we reduce it down to the case where we have no strings left to consider. Construct an $n \times n$ table.

For each rule of the form $A \to BC$, we check if (i, k) contains B and $(k + 1, j)$ contains C. If so, we include A into the cell (i, j). We accept $w \in G$ if cell $(1, n)$ contains the start variable; otherwise, we reject.

There are $O(n^2)$ number of cells to consider, and for each cell, we do an $O(n)$ search to check if (i, k) and $(k + 1, j)$ contain the appropriate elements in their respective cells. Thus, the time complexity is $O(n^3)$.

■

§5 Shortest paths

Exercise 5.1. [K] You are travelling by canoe down a river and there are n trading posts along the way. Before starting your journey, you are given for each $1 \le i < j \le n$ the fee $F(i, j)$ for renting a canoe from post i to post j. These fees are arbitrary, for example it is possible that $F(1, 3) = 10$ and $F(1, 4) = 5$. You begin at trading post 1 and must end at trading post n (using rented canoes). Your goal is to design an efficient algorithm that produces the sequence of trading posts where you change your canoe which minimizes the total rental cost.

Solution.

Subproblems: We start by defining our subproblem $P(i)$: *What is* $\mathrm{opt}(i)$*, the minimum cost it would take to reach post* i*?*

Base case: Since we start at trading post 1 we have the base case $\mathrm{opt}(1) = 0$.

Recurrence: For $i > 1$, our recurrence is then

$$\mathrm{opt}(i) = \min\{\mathrm{opt}(j) + F(j, i) : 1 \leq j < i\}.$$

Out recurrence works as $\mathrm{opt}(j)$ marks the minimal cost of $1 \to j$, we note that to get to i we must come from some $j < i$ taking the cost $F(i, j)$. So, simply taking minimum suffices for us to compute the minimal cost of $1 \to i$.

Order of computation: we can compute the subproblems in increasing order of i.

Final answer: We then reconstruct the sequence of trading posts the canoe had to have visited. For $i > 1$ we define the following function:

$$\mathrm{from}(i) = \underset{1 \leq j < i}{\mathrm{argmin}}\{\mathrm{opt}(j) + F(j, i)\}.$$

(Note: argmin returns the value of j that produces the minimal value of $\mathrm{opt}(j) + F(j, i)$). The minimum cost by our definition is then simply $\mathrm{opt}(n)$.To get the sequence, we backtrack from post n giving the sequence $\{n, \mathrm{from}(n), \mathrm{from}(\mathrm{from}(n)), \ldots, 1\}$. Reverse this to get the boat's journey.

Time complexity: The complexity is $O(n^2)$ because there are n subproblems, and each subproblem takes $O(n)$ to find the best previous trading post. ■