COMP3121/9101: Algorithm Design and Analysis

Problem Set 5 – Dynamic Programming

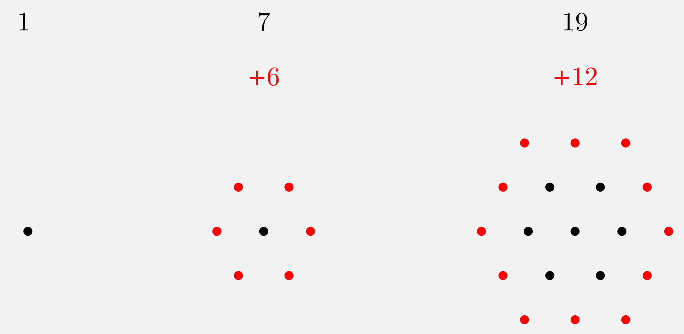
[K] – key questions **[H]** – harder questions **[E]** – extended questions

☆ – contains sample solutions

Contents

1. Memoisation 2
2. Optimisation 4
3. Counting 6
4. Selected Solutions 7

☆ **[K] Exercise 1.** The *nth centred hexagonal number* is the number of vertices required to fill a hexagon of radius n. The first few hexagonal numbers are given with an illustration.



Design an $O(n)$ algorithm that returns the nth centred hexagonal number. Can you find a closed form for the nth centred hexagonal number, and thus, design an $O(1)$ algorithm for the nth centred hexagonal number?

[K] Exercise 2. You are given a $2 \times n$ rectangular board and identical 1×2 tiles. Each tile can be laid either horizontally or vertically.

(a) Design an $O(n)$ algorithm to count the number of ways to completely cover the board with the 1×2 tiles. Can you find a closed form?

(b) We now build up a way to generate the number of ways to completely tile a $3 \times n$ rectangular board. Let $f(n)$ be the number of ways to tile a $3 \times n$ board with 1×2 tiles.

(i) Explain why $f(2k+1) = 0$ for each $k \in \mathbb{N}$. For the remainder of this problem, we will let $f(n)$ be the number of ways to tile a $3 \times 2n$ board with 1×2 tiles.

(ii) Show that $f(1) = 3$ and $f(2) = 11$.

(iii) Generate a suitable recursion based on the tilings found in $f(1)$.

Hint. *To ensure that your recursion is correct, check that* $f(2) = 11$.

(iv) Hence, design an $O(n)$ algorithm that counts the number of ways to completely cover the $3 \times 2n$ board with the tiles.

As you found, constructing a general recurrence for an $m \times n$ *board is quite difficult.*

[K] Exercise 3. In this problem, we introduce the *Tower of Hanoi* game. For students who might not be familiar with the game, we have three pegs and n disks. Each disk has a unique radius. In other words, no two distinct disks share the same radius. The rules of the game are as follows:

- The game state initially starts out with all disks on the first peg. The disks are stacked in increasing order of radius with the smallest disk on the top of the stack.
- In each move, you can only move one disk from one peg to another. You can only move one disk from one peg to another peg if the top of the stack of the peg has a bigger radius. In other words, you cannot stack a disk on top of a smaller disk.
- The game ends when all disks are placed in the third peg, in increasing order of radius with the smallest disk on the top of the stack of the third peg.

To familiarise yourself with the game, feel free to click here to play a game! In this problem, we will design an algorithm that finds the minimal number of moves to win the game, given n disks.

Let $f(n)$ be the minimum number of moves to win the *Tower of Hanoi* game with n disks.

(a) Find $f(1)$, $f(2)$, $f(3)$, and $f(4)$.

(b) For $n \geq 2$, explain why $f(n) = 2f(n-1) + 1$.

(c) Hence, design an $O(n)$ algorithm that computes the minimum number of moves to win the game with n disks.

☆ [K] **Exercise 4**. You are given n types of denominations of values with $v(1) < v(2) < \cdots < v(n)$ and $v(1) = 1$. You are additionally given a positive integer C. Design an $O(nC)$ algorithm which makes change of value C using as few coins as possible. You may assume that you have an infinite supply of such coins.

Hint. *Provide a suitable subproblem that allows you to define a simple recursion.*

[K] **Exercise 5**. You are given string A of size n and string B of size m, and you want to transform A to B. You are allowed to insert a character, delete a character, and replace a character with another. However, each operation comes with a cost.

- Inserting a character costs c_I;
- Deleting a character costs c_D;
- Replacing a character costs c_R.

Design an $O(mn)$ algorithm to find the lowest total cost to transform A into B.

Hint. How should you relate strings A and B together? Provide a suitable subproblem that allows you to do so.

☆ [K] **Exercise 6**. A *palindrome* is a word that can be read the same both forwards and backwards. For example, the word “kayak” is a palindrome as reading it forwards is the same as reading it backwards. Similarly, “kaayak” is not a palindrome because reading it backwards reads “kayaak” which is not the same as “kaayak”. Given a string of length n, design an $O(n^2)$ algorithm that finds the minimum number of characters to delete such that the resulting string after deletion is a palindrome.

For example, “kaayak” only requires one deletion to form a palindrome, while the string “abccab” requires two deletions.

☆ [K] **Exercise 7**. 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 to a problem without rotations?

[K] **Exercise 8**. 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 $i = 1, \ldots, n-1$.

☆ [K] **Exercise 9**. 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.

[H] Exercise 10. 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.

☆ **[H] Exercise 11**. 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.

☆ **[K] Exercise 12**. 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.

[K] Exercise 13. Let A and B be two strings of lengths n and m respectively. 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. Given strings A and B, design an $O(mn)$ algorithm that gives the number of different occurrences of A in B; that is, we want to compute the number of ways one can delete some of the symbols of B to get A.

☆ **[K] Exercise 14**. A partition of a number n is a sequence $\langle p_1, p_2, \dots, p_k \rangle$ such that $1 \le p_1 \le p_2 \le \cdots \le p_k \le n$ and $p_1 + p_2 + \ldots + p_k = n$. We call each p_i a *part*. Define $\textsc{numpart}(n, k)$ to be the number of partitions of n such that each part is at most k.

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

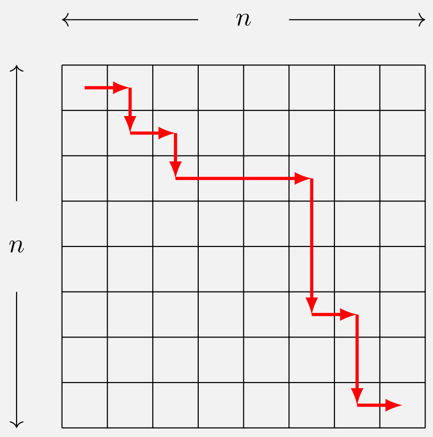
(b) Devise a recurrence to determine the number of partitions of n in which every part is at most k, where n, k are two given integers such that $2 \le k \le n$.

- To obtain the right recursion, consider two different cases depending on the largest possible value of each part.

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

(d) How many subproblems are there, and for each subproblem, what is the time complexity to compute? Hence, analyse the time complexity of the algorithm.

[K] Exercise 15. 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.

4. Selected Solutions

Exercise 1. We define each subproblem $P(i)$: *let* $\text{OPT}(i)$ *be the ith centred hexagonal number.*

Let $H(n)$ be the nth centred hexagonal number. To get the nth layer of the hexagon, we need to add n vertices to each edge. Therefore, there are $6n$ vertices we need to add. However, there are exactly 6 vertices that belong on two edges (these are the corner vertices of the hexagon). Therefore, we need to subtract the double counting of the corner vertices. In other words, our recursion becomes

$$H(n) = H(n-1) + 6n - 6 = H(n-1) + 6(n-1),$$

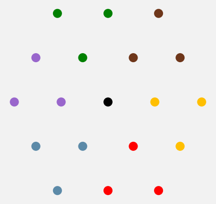
with $H(1) = 1$. This gives us a nice procedure for our recursive algorithm.

If $n = 1$, then our algorithm returns $H(1) = 1$. Otherwise, for each i from $2, \dots, n$, we return

$$H(i) = H(i-1) + 6(i-1).$$

We cache each intermediary subproblem into an array of length n so that the lookup time is $O(1)$.

We can actually partition the vertices (except for the centre) of the hexagon into six triangles as follows.



These form the $(n-1)$ triangular numbers which have the closed form

$$1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}.$$

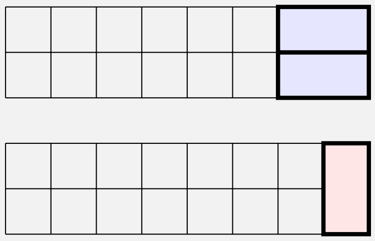
Therefore, the nth centred hexagonal number can be thought of six $(n-1)$ triangular numbers with the centre vertex which gives the closed form

$$\begin{aligned} H(n) &= 6\left(\frac{n(n-1)}{2}\right) + 1 \\ &= 3n(n-1) + 1 \\ &= 3n^2 - 3n + 1. \end{aligned}$$

In other words, we can output the nth centred hexagonal number in constant time by returning $H(n) = 3n^2 - 3n + 1$.

Exercise ??.

(a)
- **Subproblem**: Let $\text{NUM}(i)$ be the number of ways to tile a $2 \times i$ rectangular board using 1×2 tiles.
- **Recurrence**: We observe that, to tile a $2 \times i$ tile, we have the two ending possibilities as follows:



If we place one of the blue tiles, the other 1×2 blue tile must appear to complete the grid. Therefore, in this case, it suffices to count the number of ways to tile the $1 \times (i-2)$ board. If we decide to place a red tile, then we see that it suffices to count the number of ways to tile the $1 \times (i-1)$ board. Therefore, we obtain the natural recurrence

$$\textsc{num}(i) = \textsc{num}(i-1) + \textsc{num}(i-2).$$

- **Base cases**: Since the recurrence relies on $\textsc{num}(i-1)$ and $\textsc{num}(i-2)$, we require the two base cases:

$$\textsc{num}(1) = 1, \quad \textsc{num}(2) = 2.$$

- **Order of Computation and Final Solution**: By our recurrence, we observe that each subproblem relies on computing smaller sized boards. Therefore, the natural ordering is to solve the subproblems in increasing order of board size; that is, we compute $\textsc{num}(1), \textsc{num}(2), \textsc{num}(3), \textsc{num}(4), \ldots, \textsc{num}(n)$, with the final solution being $\textsc{num}(n)$.
- **Time Complexity**: There are n subproblems, each of which can be computed in $O(1)$ time. Therefore, the time complexity is $O(n)$.

If we define $\textsc{num}(0) = 1$, then we observe that we obtain the Fibonacci sequence by our recurrence; specifically,

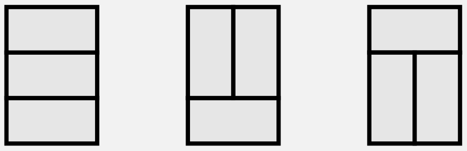
$$\begin{aligned}\textsc{num}(0) &= 1 = F_1,\\ \textsc{num}(1) &= 1 = F_2,\\ \textsc{num}(2) &= 2 = F_3,\\ \textsc{num}(3) &= 3 = F_4,\\ &\ldots\\ \textsc{num}(n) &= F_{n-1}.\end{aligned}$$

Therefore, we can simply return $\textsc{num}(n) = F_{n-1}$, which has a well-known closed form; that is,

$$\textsc{num}(n) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}}{\sqrt{5}}.$$

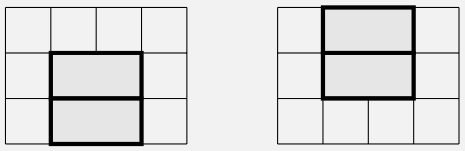
(b) (i) We see that every 1×2 tile must cover exactly two distinct squares of the board. Therefore, to cover a board with 1×2 tiles, the number of squares in the board must be even. However, any $3 \times (2k+1)$ board will contain an odd number of squares. Therefore, any tiling will have at least one square remaining if no tiles can intersect.

(ii) We first show that $f(1) = 3$. This is easy to list out. We have

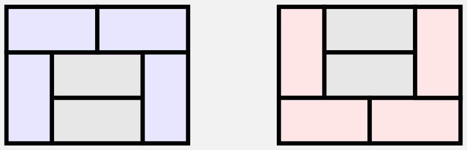


Therefore, we have $f(1) = 3$. To derive an expression for $f(2)$, we can break this into two 3×2 boards. Each board has exactly three configurations as shown above. Therefore, there are $3 \times 3 = 9$ configurations if tiles do not overlap between the two 3×2 boards. We now look at all configurations where tiles can overlap over the two boards.

Consider the following configurations.



This forces the remaining tiles to be placed in the following positions.



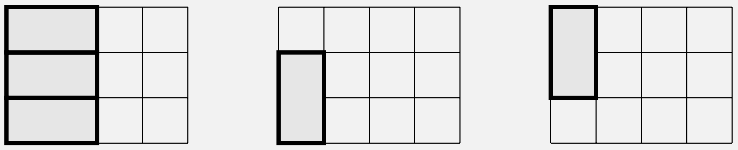
Note that placing three 1×2 tiles in the middle of the board reduces the problem to a 3×1 board which has no solution. Therefore, we note that these are the only additional solutions that aren't accounted for in the previous disjoint case. Thus, we obtain $f(2) = 9 + 2 = 11$ number of ways to tile a 3×4 board.

(iii) Let $f(n)$ denote the number of ways to tile a $3 \times 2n$ board and $g(n)$ denote the number of ways to tile a $3 \times (2n + 1)$ board with its first square missing.

To tile a $3 \times 2n$ board, we can either:

- tile it so that the three left-most squares are all covered by the horizontal 1×2 tiles, leaving us with a $3 \times (2n - 2)$ board,
- or tile two of the left-most squares leaving one square vacant.

The following are illustrated below.



This gives $f(n) = f(n - 1) + 2g(n - 1)$.

In a similar light, to tile a $3 \times (2n + 1)$ board with one tile missing, the other two squares have to be covered by either:

- a single tile, leaving a $3 \times 2n$ board,
- or two tiles leaving a missing square in the next column.

This gives $g(n) = f(n) + g(n - 1)$.

Therefore the recurrence becomes

$$\begin{aligned} f(n) &= f(n-1) + 2g(n-1) \\ &= f(n-1) + 2\left(f(n-1) + g(n-2)\right) \\ &= f(n-1) + 2\left(f(n-1) + \frac{f(n-1) - f(n-2)}{2}\right) \\ &= f(n-1) + 2f(n-1) + f(n-1) - f(n-2) \\ &= 4f(n-1) - f(n-2). \end{aligned}$$

(c) We therefore devise a dynamic programming solution based on the recurrence above.

- **Subproblem**: Let $f(n)$ denote the number of ways to tile a $3 \times 2n$ board using 1×2 dominoes.

- **Recurrence**: The recurrence is given by
$$f(n) = 4f(n-1) - f(n-2).$$
- **Base case**: $f(1) = 3$ and $f(2) = 11$.
- **Order of Computation and Final Solution**: We solve each subproblem in increasing order of i with the final solution being $f(n)$.
- **Time Complexity**: There are n subproblems and each subproblem can be computed in cosntant time giving an $O(n)$ algorithm.

Exercise 3.

(a) Denote $a \to b$ as moving the topmost disk of peg a to peg b. Then:

- for $n = 1$, the solution is $1 \to 3$,
- for $n = 2$, the solution is $1 \to 2$, $1 \to 3$, $2 \to 3$,
- for $n = 3$, the solution is $1 \to 3$, $1 \to 2$, $3 \to 2$, $1 \to 3$, $2 \to 1$, $2 \to 3$, $1 \to 3$, and
- for $n = 4$, the solution is $1 \to 2$, $1 \to 3$, $2 \to 3$, $1 \to 2$, $3 \to 1$, $3 \to 2$, $1 \to 2$, $1 \to 3$, $2 \to 3$, $2 \to 1$, $3 \to 1$, $2 \to 3$, $1 \to 2$, $1 \to 3$, $2 \to 3$.

Therefore $f(1) = 1$, $f(2) = 3$, $f(3) = 7$ and $f(4) = 15$.

(b) In order to move the largest disk to the target position, we must first move all $n-1$ disks above it to peg 2. This takes $f(n-1)$ moves. Once we move the largest disk, we must then move all $n-1$ smaller disks from peg 2 to peg 3, again taking $f(n-1)$ moves. Therefore the task both requires at least $2f(n-1)+1$ moves and can be accomplished in exactly this many moves, completing the proof.

(c) **Subproblems:** for $i \geq 1$, let $f(i)$ be the minimum number of moves to win the *Tower of Hanoi* game with i disks.

Recurrence: for $i > 1$, $f(i) = 2f(i-1) + 1$.

Base case: $f(1) = 1$.

Order of computation: increasing from $i = 1$ to $i = n$.

Final answer: $f(n)$.

Time complexity: each of n subproblems is solved in constant time, for a total time complexity of $O(n)$.

Note: we can also prove (by induction) that $f(n) = 2^n - 1$, and compute this directly for a faster algorithm.

Exercise 4. To provide a formal dynamic programming solution, we need to propose a suitable subproblem for which our recursion falls out naturally.

Let $P(i)$ be the subproblem: *what is the smallest number of coins required to make change of value i?*. Let $\text{OPT}(i)$ be the solution to the subproblem $P(i)$. Note that, to arrive at subproblem $P(i)$, we need to consider what happens if we take out an arbitrary coin. In particular, each coin will only add one to our tally. Therefore, to obtain the minimal number of coins, each previous subproblem needs to also maintain minimality. Therefore, we minimise each of the previous subproblems by considering all of the coin denominations; this gives us the following recursion
$$\text{OPT}(i) = \min\{\text{OPT}(i - v(k)) : 1 \leq k \leq n\} + 1,$$

for $1 \le i \le C$. To obtain the actual change sequence, suppose that the optimal k that was chosen was m. In other words, $\textsc{opt}(i) = \textsc{opt}(i - v(m)) + 1$. We can construct a table where index i stores the value m. Then backtracking to obtain the sequence is a matter of figuring out the right lookup based on the kth denomination for each subproblem.

We can define the base case to be $\textsc{opt}(0) = 0$ since no coins are required to obtain a value of 0. The final solution is $\textsc{opt}(C)$ and the natural order of computing the subproblems is in increasing order of i.

We finally justify the time complexity. There are C subproblems (one for each i) and each subproblem requires a single loop through all of the n denominations. Therefore, each subproblem is solved in $O(n)$ time, which implies that our algorithm runs in $C \cdot O(n) = O(nC)$ time. It is important to note that C is not a constant, but rather part of our input. Therefore, we can't reduce this to $O(n)$.

Exercise 5. The trick behind this exercise is to provide a natural subproblem that allows you to connect strings A and B together. In particular, we want to ensure that the first k letters of both strings are matching, so that we only need to change subsequent characters.

Therefore, we shall let $P(i,j)$ be the subproblem: *what is the lowest total cost to transform the sequence $A[1\ldots i]$ to $B[1\ldots j]$?*. Let $\textsc{opt}(i,j)$ be the solution to subproblem $P(i,j)$.

To obtain the recursion, we need to find all of the ways to obtain $A[1\ldots i]$ and $B[1\ldots j]$. If the current operation is delete, then we must have transformed $A[1\ldots i-1]$ into $B[1\ldots j]$ and then deleted $A[i]$. If the current operation is insert, then we must have transformed $A[1\ldots i]$ into $B[1\ldots j-1]$ and then appended $B[j]$. Otherwise, we must have transformed $A[1\ldots i-1]$ into $B[1\ldots j-1]$. In this case, if $A[i] = B[j]$, there is nothing to do. Otherwise, if $A[i] \ne B[j]$, then we do a simple replacement. This defines our recursion as follows:

$$\textsc{opt}(i,j) = \min \begin{cases} \textsc{opt}(i-1,j) + c_D, & \\ \textsc{opt}(i,j-1) + c_I, & \\ \textsc{opt}(i-1,j-1) & \text{if } A[i] = B[j], \\ \textsc{opt}(i-1,j-1) + c_R & \text{if } A[i] \ne B[j], \end{cases}$$

with $1 \le i \le n$ and $1 \le j \le m$.

The base case is $\textsc{opt}(i,0) = i \cdot c_D$ and $\textsc{opt}(0,j) = j \cdot c_I$. The overall solution is $\textsc{opt}(n,m)$. Note that there are mn subproblems (one for each character of A and one for each character of B). However, updating each of the subproblem takes constant time. Therefore, the overall running time of the algorithm is $O(mn)$.

Exercise 6. Let $s[1,\ldots,n]$ be our input string.

- **Subproblem**: Let $\textsc{opt}(i,j)$ denote the minimum number of characters to delete such that the string $s[i, i+1,\ldots,j]$ is a palindrome.
- **Recurrence**: Now, to derive a recursion, we first observe that $\textsc{opt}(i+1,j-1)$ gives us the smallest number of deletions required such that $s[i+1,\ldots,j-1]$ is a palindrome.

  $$\underbrace{s[i+1]\,s[i+2]\,\ldots\,s[j-1]}_{\textsc{opt}(i+1,j-1) \implies \text{ palindrome}}$$

  To ensure that $s[i,\ldots,j]$ is a palindrome, we now determine whether $s[i] = s[j]$. If so, then there are no extra deletions required to form a palindrome from index i to j. In other words, the number of deletions required is just the number of deletions that forms the palindrome $s[i+1,\ldots,j-1]$; in other words, we have that $\textsc{opt}(i,j) = \textsc{opt}(i+1,j-1)$ in the case where $s[i] = s[j]$.

We now jump to the case where $s[i] \neq s[j]$. In this case, we need to delete one of $s[i]$ or $s[j]$. To determine which character to delete, we compute both subproblems and take the minimum number of deletions among the two. Therefore, under the case where $s[i] \neq s[j]$, we have

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

In other words, the recurrence becomes

$$\textsc{opt}(i,j) = \begin{cases} \textsc{opt}(i+1,j-1) & \text{if } s[i] = s[j], \\ \min\{\textsc{opt}(i+1,j), \textsc{opt}(i,j-1)\} + 1 & \text{if } s[i] \neq s[j]. \end{cases}$$

For the edge case where $j-1 \leq i$, set $\textsc{opt}(i,j) = 0$ since we already have a palindrome.

- **Base case**: For the base case, note that $\textsc{opt}(i,i) = 0$ for each $1 \leq i \leq n$ since a single character is a palindrome.
- **Order of Computation and Final Solution**: The final solution is $\textsc{opt}(1,n)$. To obtain the order of evaluation, observe that each subproblem relies on computing "smaller" length strings. Therefore, we should be solving the subproblems in increasing order of string size (i.e. in increasing order of $j-i$).
- **Time Complexity**: To determine the time complexity, observe that there are n^2 many subproblems. Each subproblem takes $O(1)$ to compute; therefore, the running time of the algorithm is $O(n^2)$.

Exercise 7. 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.

- **Subproblem**: Let $\textsc{opt}(i)$ denote the maximum height possible for a stack if the top box is box number 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 $1 \leq i \leq 6n$:
$$\textsc{opt}(i) = \max\{\textsc{opt}(j) + h_i : \text{ over all } j \text{ such that } w_j > w_i,\ d_j > d_i\}.$$
- **Base case**: $\textsc{opt}(i) = h_i$, for each i if we cannot place a box below B_i.
- **Order of Computation and Final Solution**: By focusing on the recurrence, we observe that each subproblem relies on smaller widths and smaller depth. Therefore, we compute the subproblems in increasing order of w_i and d_i (i.e. in increasing order of $w_i + d_i$). The final solution is $\max_{1\leq i\leq 6n} \textsc{opt}(i)$.
- **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 8.

- **Subproblem**: Let $\textsc{opt}(i)$ denote the maximum number of dams that can be built among proposals $1, \ldots, i$ such that the ith dam is built.
- **Recurrence**: We first observe that, if we build the ith dam, then we cannot place any dam where $|x_i - x_j| \leq r_i$ *and* $|x_i - x_j| \leq r_j$. Therefore, we need to look at all dams such that $|x_i - x_j| > \max\{r_i, r_j\}$. By our subproblem

formulation, we look at all proposals from $1, \dots, i$. Therefore, we look at all possible dams $j < i$ that satisfy the above constraint that maximises the number of dams that we can build. This is equivalent to building dam i and then building dams using proposals $1, \dots, j$, giving us the recurrence

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

- **Base case**: If there is only one dam, then we build the dam. Therefore, $\textsc{opt}(1) = 1$.
- **Order of Computation**: To obtain the appropriate recurrence, we need to look at all proposals from $1, \dots, j, \dots, i$. Therefore, the natural order of evaluation is in increasing order of i.
- **Overall answer**: Considering all possible choices for the last dam to be built, the answer is

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

- **Time Complexity**: There are n subproblems. However, each subproblem requires us to look at $j < i$ subproblems. This gives $1 + 2 + \cdots + n = O(n^2)$ many subproblems to compute altogether. Therefore, the running time of the algorithm is $O(n^2)$.

Exercise 9.

- **Subproblem**: Let $\textsc{opt}(i)$ denote the minimum cost it would take to reach post i.
- **Recurrence**: We note that, to arrive at post i, we must come from some post j. We look at all possible posts that we can come from and choose the post that minimises the overall cost. This gives us

$$\textsc{opt}(i) = \min_{1 \le j \le i}\{\textsc{opt}(j) + F(j, i)\}.$$

- **Base case**: We begin at trading post 1; therefore, the base case is $\textsc{opt}(1) = 0$.
- **Order of Computation and Final Solution**: Since our subproblem recurrence relies on previous values of i, the natural ordering is in increasing order of i with the final solution being $\textsc{opt}(n)$.

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

$$\text{from}(i) = \operatorname*{argmin}_{1 \le j < i}\{\textsc{opt}(j) + F(j, i)\},$$

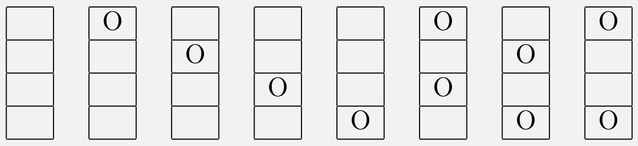
  where argmin returns the value j that minimises $\textsc{opt}(j) + F(j, i)$. To obtain the sequence, we backtrack from n, giving the sequence:

$$\{n, \text{from}(n), \text{from}(\text{from}(n)), \dots, 1\}$$

  and reversing this sequence gives the boat's journey.
- **Time Complexity**: There are n subproblems and each subproblem takes $O(n)$ time to find the previous trading post. Therefore, the overall running time is $O(n^2)$.

Exercise 10.

(a) There are 8 patterns, listed below.



(b) Let t denote the type of pattern so that $1 \le t \le 8$.

- **Subproblem**: Let $\textsc{opt}(k,t)$ denote the maximum score that can be achieved by only placing pebbles in the first k columns such that the kth column contains pattern t.
- **Recurrence**: We first define $\text{score}(k,t)$ to be the score obtained by using pattern t on column k. Then the recurrence becomes
$$\textsc{opt}(k,t) = \text{score}(k,t) + \max\{\textsc{opt}(k-1,s) : s \text{ is compatible with } t\}.$$
- **Base case**: If there is no column, then any pattern on the column gives a score of 0. Therefore, $\textsc{opt}(0,t) = 0$ for each $1 \le t \le 8$.
- **Order of Computation and Final Solution**: For each column, we need to solve for every pattern. Therefore, we solve in increasing order of t and then increasing order of k, with the final solution being $\max_{1\le t\le 8} \textsc{opt}(n,t)$.
- **Time Complexity**: Since the number of patterns is fixed, there are only $O(n)$ many subproblems and each subproblem only needs at most eight many subproblems to check. Therefore, the running time is $O(n)$.

Exercise 11. 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 $\text{fun}(i)$ that satisfies all of the constraints but *excludes* root i.

With these quantities defined, we now compute the dynamic programming solution.

- **Subproblem**: Let $\textsc{opt}(i)$ denote the maximum sum of the fun ratings of $T(i)$.
- **Recurrence**: For the recurrence, 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, we have that
$$\textsc{opt}(i) = \max(I(i), E(i)).$$
- **Base case**: For each i that is a leaf node, $\textsc{opt}(i) = \text{fun}(i)$.
- **Order of Computation and Final Solution**: We solve the subproblems from the leaves of the tree to the root of the leaves, where the final solution is simply $\textsc{opt}(n) = \max(I(n), E(n))$ 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 12.

- **Subproblem**: Let $\text{num}(i)$ denote the number of ways to split the first i elements into contiguous blocks of sum at most k.
- **Recurrence**: We look at all of the possible ways to place a divider such that all elements in the same group of the divider sum to at most k. Denote $\text{sum}(j,i)$ to be the sum of all elements between (and including) j and i. Then we have that
$$\text{num}(i) = \sum_{\substack{1 \le j \le i,\\ \text{sum}(j,i) \le k}} \text{num}(j-1).$$
- **Base case**: There is exactly one way to split 0 elements; therefore, $\text{num}(0) = 1$.
- **Order of Computation and Final Solution**: We solve the subproblems in increasing order of i. For a particular subproblem $\text{num}(i)$, we consider the j values in *decreasing* order, so that we can update $\text{num}(j,i)$ value in constant time by adding one more term at each step until the early exit. The final solution is $\text{num}(n)$.
- **Time Complexity**: There are n subproblems and each subproblem requires an $O(n)$ search. Therefore, the running time of the algorithm is $O(n^2)$.

Exercise 13. We begin by defining some notations. Let A_i denote the ith letter of A and B_j as the jth letter of B.

- **Subproblem**: Let $\text{ways}(i,j)$ denote the number of times the first i letters of A appears as a subsequence in the first j letters of B.
- **Recurrence**: If $A_i \neq B_j$, then this is equivalent to removing the jth character from B and checking if the first i characters appear in the substring $B_1, \dots, B_{j-1}$. This gives $\text{ways}(i, j-1)$ number of ways this could happen.

  On the other hand, if $A_i = B_j$, then we check to see if $A_{i-1} = B_{j-1}$. This gives $\text{ways}(i-1, j-1)$ number of ways. However, the substring A ending at A_i can also be contained entirely inside the string $B_1, \dots, B_{j-1}$. Therefore, we also compute $\text{ways}(i, j-1)$. In other words, we obtain the recurrence
$$\text{ways}(i,j) = \begin{cases} \text{ways}(i, j-1) & \text{if } A_i \neq B_j, \\ \text{ways}(i-1, j-1) + \text{ways}(i, j-1) & \text{if } A_i = B_j. \end{cases}$$
- **Base case**: If A is empty, then $\text{ways}(0,j) = 1$ for each $1 \le j \le m$. If B is empty, then $\text{ways}(i,0) = 0$ for each $1 \le i \le n$.
- **Order of Computation and Final Solution**: We solve each subproblem in increasing order of j and then increasing order of i.
- **Time Complexity**: There are mn many subproblems (one for i and one for j) and each subproblem takes constant time to compute; therefore, the running time of the algorithm is $O(mn)$.

Exercise 14.

(a) Note that $\textsc{numpart}(n, 1)$ 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) We split up the partitions by the value of their largest part p_t. More precisely, we consider the cases where $p_t = k$ and $p_t < k$ separately.

For the first case, we have the equation

$$p_1 + \cdots + p_{t-1} + k = n,$$

and subtracting k from both sides gives

$$p_1 + \ldots + p_{t-1} = n - k.$$

We observe that the remaining sequence $\langle p_1, \ldots, p_{t-1} \rangle$ is a partition of $n - k$ in which no part exceeds k, so there are $\textsc{numpart}(k, n-k)$ such sequences.

For the second case, if $p_t < k$ then every other term of the sequence is also less than k. Therefore $\langle p_1, \ldots, p_t \rangle$ is a partition of n in which no part exceeds $k - 1$, so there are $\textsc{numpart}(n, k-1)$ such sequences.

Note that these 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

$$\textsc{numpart}(n, k) = \textsc{numpart}(n-k, k) + \textsc{numpart}(n, k-1).$$

We can solve these subproblems in lexicographic order, i.e. increasing order of k, breaking ties in increasing order of n.

(c) If we allow p_t to take any value up to n, then any partition of n is allowed. Therefore the total number of (unrestricted) partitions of n is simply $\textsc{numpart}(n, n)$.

(d) Note that there is one subproblem for each number of partitions and one subproblem for the maximum size of each partition. Therefore, there are n^2 many subproblems. For each subproblem, it takes $O(1)$ time to compute. Thus, the overall running time is $O(n^2)$.

Exercise 15.

(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 $\textsc{board}(i, j)$ to be the integer inside the cell located on row i and column j. We then begin with the dynamic programming approach.

- **Subproblem**: Let $\textsc{opt}(i, j)$ denote the best score that can be achieved if we arrive at cell (i, j).
- **Recurrence**: 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

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

- **Base case**: The base case is $\textsc{opt}(1,1) = \textsc{board}(1,1)$. To ensure that we never consider illegal moves (i.e. moves that land outside of the board), we also have that $\textsc{opt}(i,j) = \infty$ if cell (i,j) does not exist (or is off the board).
- **Order of Computation and Final Solution**: Observe that, in order to solve $\textsc{opt}(i,j)$, we need to know $\textsc{opt}(i-1,j)$ and $\textsc{opt}(i,j-1)$. 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. The final solution is simply solving $\textsc{opt}(n,n)$. To return the actual path, we can make a note of the choice that the recursion makes by choosing either $\textsc{opt}(i-1,j)$ or $\textsc{opt}(i,j-1)$ 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 $1 \le i,j \le n$. Now, for each subproblem $P(i,j)$, we would have computed $\textsc{opt}(i-1,j)$ and $\textsc{opt}(i,j-1)$ previously. Hence, we have $O(1)$ lookup time and the computation for $\textsc{opt}(i,j)$ 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 $\textsc{opt}(i,j)$ as defined above to our advantage.

- **Subproblem**: Let $\textsc{ways}(i,j)$ denote the minimum number of ways to reach cell (i,j) with score $\textsc{opt}(i,j)$.
- **Recurrence**: 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 recurrence becomes

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

- **Base case**: The base case is $\textsc{ways}(1,1) = 1$ since there is only one way to get to cell $(1,1)$. We also define $\textsc{ways}(i,j) = 0$ for all cells that fall outside of the grid since there are no ways to get to those cells.
- **Order of Computation and Final Solution**: We solve the problems in increasing order of i, and then increasing order of j to break ties for the same reason as the previous part. The final solution is $\textsc{ways}(n,n)$.
- **Time Complexity**: The time complexity is $O(n^2)$ since there are still n^2 subproblems and each subproblem takes $O(1)$ time to compute.