Algorithms: COMP3121/9101

Aleks Ignjatović, ignjat@cse.unsw.edu.au
office: 504 (CSE building K 17)
Admin: Song Fang, cs3121@cse.unsw.edu.au

School of Computer Science and Engineering
University of New South Wales Sydney

More Dynamic Programming Practice Problems

1. You are traveling by a canoe down a river and there are n trading posts along the way. Before starting your journey, you are given for each $1 \leq i < j \leq n$ 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 algorithms which produces the sequence of trading posts where you change your canoe which minimizes the total rental cost.

Solution:

- We solve the following subproblems: *Find the minimum cost it would take to reach post i.*
- The base case is $\text{opt}(1) = 0$.
- The recursion is:

$$\text{opt}(i) = \min\{\text{opt}(j) + F(j,i) : 1 \leq j < i\}, \qquad i > 1,$$

- To reconstruct the sequence of trading posts the canoe had to have visited, we define the following function:

$$\text{from}(i) = \arg\min_{1 \leq j < i}\{\text{opt}(j) + F(j,i)\}, \qquad i > 1.$$

  (Note: $\arg\min$ returns the value of j that produces the minimal value of $\text{opt}(j) + F(j,i)$).
- The minimum cost is $\text{opt}(n)$.
- To get the sequence, we backtrack from post n giving the sequence $\{n, \text{from}(n), \text{from}(\text{from}(n)), ..., 1\}$. Reverse this sequence.
- The complexity is $O(n^2)$ because there are n subproblems, and each subproblem takes $O(n)$ to find the best previous trading post.

Solution:

- We solve the following subproblems: *Find the minimum cost it would take to reach post i.*
- The base case is $\text{opt}(1) = 0$.
- The recursion is:

$$\text{opt}(i) = \min\{\text{opt}(j) + F(j,i) : 1 \leq j < i\}, \qquad i > 1,$$

- To reconstruct the sequence of trading posts the canoe had to have visited, we define the following function:

$$\text{from}(i) = \arg\min_{1 \leq j < i}\{\text{opt}(j) + F(j,i)\}, \qquad i > 1.$$

  (Note: $\arg\min$ returns the value of j that produces the minimal value of $\text{opt}(j) + F(j,i)$).
- The minimum cost is $\text{opt}(n)$.
- To get the sequence, we backtrack from post n giving the sequence $\{n, \text{from}(n), \text{from}(\text{from}(n)), ..., 1\}$. Reverse this sequence.
- The complexity is $O(n^2)$ because there are n subproblems, and each subproblem takes $O(n)$ to find the best previous trading post.

Solution:

- We solve the following subproblems: *Find the minimum cost it would take to reach post i.*
- The base case is $\text{opt}(1) = 0$.
- The recursion is:

$$\text{opt}(i) = \min\{\text{opt}(j) + F(j, i) : 1 \leq j < i\}, \qquad i > 1,$$

- To reconstruct the sequence of trading posts the canoe had to have visited, we define the following function:

$$\text{from}(i) = \arg\min_{1 \leq j < i}\{\text{opt}(j) + F(j, i)\}, \qquad i > 1.$$

  (Note: $\arg\min$ returns the value of j that produces the minimal value of $\text{opt}(j) + F(j, i)$).
- The minimum cost is $\text{opt}(n)$.
- To get the sequence, we backtrack from post n giving the sequence $\{n, \text{from}(n), \text{from}(\text{from}(n)), ..., 1\}$. Reverse this sequence.
- The complexity is $O(n^2)$ because there are n subproblems, and each subproblem takes $O(n)$ to find the best previous trading post.

Solution:

- We solve the following subproblems: *Find the minimum cost it would take to reach post i.*
- The base case is $\text{opt}(1) = 0$.
- The recursion is:

$$\text{opt}(i) = \min\{\text{opt}(j) + F(j, i) : 1 \leq j < i\}, \qquad i > 1,$$

- To reconstruct the sequence of trading posts the canoe had to have visited, we define the following function:

$$\text{from}(i) = \arg\min_{1 \leq j < i}\{\text{opt}(j) + F(j, i)\}, \qquad i > 1.$$

(Note: $\arg\min$ returns the value of j that produces the minimal value of $\text{opt}(j) + F(j, i)$).

- The minimum cost is $\text{opt}(n)$.
- To get the sequence, we backtrack from post n giving the sequence $\{n, \text{from}(n), \text{from}(\text{from}(n)), ..., 1\}$. Reverse this sequence.
- The complexity is $O(n^2)$ because there are n subproblems, and each subproblem takes $O(n)$ to find the best previous trading post.

Solution:

- We solve the following subproblems: *Find the minimum cost it would take to reach post i.*
- The base case is $\text{opt}(1) = 0$.
- The recursion is:

$$\text{opt}(i) = \min\{\text{opt}(j) + F(j, i) : 1 \leq j < i\}, \qquad i > 1,$$

- To reconstruct the sequence of trading posts the canoe had to have visited, we define the following function:

$$\text{from}(i) = \arg\min_{1 \leq j < i}\{\text{opt}(j) + F(j, i)\}, \qquad i > 1.$$

  (Note: $\arg\min$ returns the value of j that produces the minimal value of $\text{opt}(j) + F(j, i)$).

- The minimum cost is $\text{opt}(n)$.
- To get the sequence, we backtrack from post n giving the sequence $\{n, \text{from}(n), \text{from}(\text{from}(n)), ..., 1\}$. Reverse this sequence.
- The complexity is $O(n^2)$ because there are n subproblems, and each subproblem takes $O(n)$ to find the best previous trading post.

Solution:

- We solve the following subproblems: *Find the minimum cost it would take to reach post i.*
- The base case is $\text{opt}(1) = 0$.
- The recursion is:

$$\text{opt}(i) = \min\{\text{opt}(j) + F(j, i) : 1 \leq j < i\}, \qquad i > 1,$$

- To reconstruct the sequence of trading posts the canoe had to have visited, we define the following function:

$$\text{from}(i) = \arg\min_{1 \leq j < i}\{\text{opt}(j) + F(j, i)\}, \qquad i > 1.$$

  (Note: $\arg\min$ returns the value of j that produces the minimal value of $\text{opt}(j) + F(j, i)$).
- The minimum cost is $\text{opt}(n)$.
- To get the sequence, we backtrack from post n giving the sequence $\{n, \text{from}(n), \text{from}(\text{from}(n)), ..., 1\}$. Reverse this sequence.
- The complexity is $O(n^2)$ because there are n subproblems, and each subproblem takes $O(n)$ to find the best previous trading post.

Solution:

- We solve the following subproblems: *Find the minimum cost it would take to reach post i.*
- The base case is $\text{opt}(1) = 0$.
- The recursion is:

$$\text{opt}(i) = \min\{\text{opt}(j) + F(j, i) : 1 \leq j < i\}, \qquad i > 1,$$

- To reconstruct the sequence of trading posts the canoe had to have visited, we define the following function:

$$\text{from}(i) = \arg\min_{1 \leq j < i}\{\text{opt}(j) + F(j, i)\}, \qquad i > 1.$$

  (Note: $\arg\min$ returns the value of j that produces the minimal value of $\text{opt}(j) + F(j, i)$).
- The minimum cost is $\text{opt}(n)$.
- To get the sequence, we backtrack from post n giving the sequence $\{n, \text{from}(n), \text{from}(\text{from}(n)), ..., 1\}$. Reverse this sequence.
- The complexity is $O(n^2)$ because there are n subproblems, and each subproblem takes $O(n)$ to find the best previous trading post.

Solution:

- We solve the following subproblems: *Find the minimum cost it would take to reach post i.*
- The base case is $\text{opt}(1) = 0$.
- The recursion is:

$$\text{opt}(i) = \min\{\text{opt}(j) + F(j, i) : 1 \leq j < i\}, \qquad i > 1,$$

- To reconstruct the sequence of trading posts the canoe had to have visited, we define the following function:

$$\text{from}(i) = \arg\min_{1 \leq j < i}\{\text{opt}(j) + F(j, i)\}, \qquad i > 1.$$

  (Note: $\arg\min$ returns the value of j that produces the minimal value of $\text{opt}(j) + F(j, i)$).
- The minimum cost is $\text{opt}(n)$.
- To get the sequence, we backtrack from post n giving the sequence $\{n, \text{from}(n), \text{from}(\text{from}(n)), ..., 1\}$. Reverse this sequence.
- The complexity is $O(n^2)$ because there are n subproblems, and each subproblem takes $O(n)$ to find the best previous trading post.

2. 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.

Solution:

- First, since each box type has six different box rotations (there are three faces and each face can be rotated once, exchanging width and depth), it is easier to treat these rotations as being separate boxes entirely.
- This gives $6n$ boxes in total, and allows us to assume that boxes cannot be rotated.
- We now order these boxes in a decreasing order of the surface area of their base.
- Notice that in this way if a box B_1 can go on top of a box B_0 then B_0 precedes box B_1 in such an ordering.
- We aim to solve the following subproblems: *What is the maximum height possible for a stack if the top most box is box number i?*

Solution:

- First, since each box type has six different box rotations (there are three faces and each face can be rotated once, exchanging width and depth), it is easier to treat these rotations as being separate boxes entirely.
- This gives $6n$ boxes in total, and allows us to assume that boxes cannot be rotated.
- We now order these boxes in a decreasing order of the surface area of their base.
- Notice that in this way if a box B_1 can go on top of a box B_0 then B_0 precedes box B_1 in such an ordering.
- We aim to solve the following subproblems: *What is the maximum height possible for a stack if the top most box is box number i?*

Solution:

- First, since each box type has six different box rotations (there are three faces and each face can be rotated once, exchanging width and depth), it is easier to treat these rotations as being separate boxes entirely.
- This gives $6n$ boxes in total, and allows us to assume that boxes cannot be rotated.
- We now order these boxes in a decreasing order of the surface area of their base.
- Notice that in this way if a box B_1 can go on top of a box B_0 then B_0 precedes box B_1 in such an ordering.
- We aim to solve the following subproblems: *What is the maximum height possible for a stack if the top most box is box number i?*

Solution:

- First, since each box type has six different box rotations (there are three faces and each face can be rotated once, exchanging width and depth), it is easier to treat these rotations as being separate boxes entirely.
- This gives $6n$ boxes in total, and allows us to assume that boxes cannot be rotated.
- We now order these boxes in a decreasing order of the surface area of their base.
- Notice that in this way if a box B_1 can go on top of a box B_0 then B_0 precedes box B_1 in such an ordering.
- We aim to solve the following subproblems: *What is the maximum height possible for a stack if the top most box is box number i?*

Solution:

- First, since each box type has six different box rotations (there are three faces and each face can be rotated once, exchanging width and depth), it is easier to treat these rotations as being separate boxes entirely.
- This gives $6n$ boxes in total, and allows us to assume that boxes cannot be rotated.
- We now order these boxes in a decreasing order of the surface area of their base.
- Notice that in this way if a box B_1 can go on top of a box B_0 then B_0 precedes box B_1 in such an ordering.
- We aim to solve the following subproblems: *What is the maximum height possible for a stack if the top most box is box number i?*

- The recursion is:

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

- The final solution of the problem is the maximum value returned by any of these subproblems, i.e., $\max_{1 \le i \le 6n} \text{opt}(i)$.
- The complexity is $O(n^2)$. There are $6n$ different subproblems, and each subproblem requires us to search through $O(n)$ boxes to find ones that have a base large enough to stack the current box on top.

- The recursion is:

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

- The final solution of the problem is the maximum value returned by any of these subproblems, i.e., $\max_{1 \le i \le 6n} \text{opt}(i)$.
- The complexity is $O(n^2)$. There are $6n$ different subproblems, and each subproblem requires us to search through $O(n)$ boxes to find ones that have a base large enough to stack the current box on top.

- The recursion is:

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

- The final solution of the problem is the maximum value returned by any of these subproblems, i.e., $\max_{1 \le i \le 6n} \text{opt}(i)$.
- The complexity is $O(n^2)$. There are $6n$ different subproblems, and each subproblem requires us to search through $O(n)$ boxes to find ones that have a base large enough to stack the current box on top.

3. 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 100, as well as the price of each candy. Your task is to chose 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 $O(Mn)$.

3. 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 100, as well as the price of each candy. Your task is to chose 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 $O(Mn)$.

4. Consider a 2-D map with a horizontal river passing through its centre. There are n cities on the southern bank with x-coordinates $a_1 \ldots a_n$ and n cities on the northern bank with x-coordinates $b_1 \ldots b_n$. You want to connect as many north-south pairs of cities as possible, with bridges such that no two bridges cross. When connecting cities, you are only allowed to connect the the i^{th} city on the northern bank to the i^{th} city on the southern bank.

Solution: Sort the cities on the south bank according to their x coordinates and then re-enumerate them so that they appear as $\{1, 2, 3, \ldots\}$, and then apply the same permutation to the cities on the north bank. Now the problem reduces to finding a maximal increasing sequence of indices of cities on the north bank.

4. Consider a 2-D map with a horizontal river passing through its centre. There are n cities on the southern bank with x-coordinates $a_1 \ldots a_n$ and n cities on the northern bank with x-coordinates $b_1 \ldots b_n$. You want to connect as many north-south pairs of cities as possible, with bridges such that no two bridges cross. When connecting cities, you are only allowed to connect the the i^{th} city on the northern bank to the i^{th} city on the southern bank.

Solution: Sort the cities on the south bank according to their x coordinates and then re-enumerate them so that they appear as $\{1, 2, 3, \ldots\}$, and then apply the same permutation to the cities on the north bank. Now the problem reduces to finding a maximal increasing sequence of indices of cities on the north bank.

5. You are given a boolean expression consisting of a string of the symbols *true* and *false* and with exactly one operation *and*, *or*, *xor* between any two consecutive truth values. Count the number of ways to place brackets in the expression such that it will evaluate to true. For example, there is only 1 way to place parentheses in the expression *true* and *false* xor *true* such that it evaluates to true.

Solution:

- Let there be n symbols, and $n-1$ operations between them.
- We solve the following two subproblems:
  How many ways (denoted $T(l,r)$) are there to place brackets to make the expression starting from at the l^{th} symbol and ending at r^{th} symbol evaluate to true (T)?
  and
  How many ways (denoted $F(l,r)$) are there to place brackets to make the expression starting from at the l^{th} symbol and ending at r^{th} symbol evaluate to false (F)?
- Problems are solved in the order of $r-l$ breaking evens arbitrarily.

5. You are given a boolean expression consisting of a string of the symbols *true* and *false* and with exactly one operation *and*, *or*, *xor* between any two consecutive truth values. Count the number of ways to place brackets in the expression such that it will evaluate to true. For example, there is only 1 way to place parentheses in the expression *true* and *false* xor *true* such that it evaluates to true.

Solution:

- Let there be n symbols, and $n-1$ operations between them.
- We solve the following two subproblems:
  How many ways (denoted $T(l,r)$) are there to place brackets to make the expression starting from at the l^{th} symbol and ending at r^{th} symbol evaluate to true (T)?
  and
  How many ways (denoted $F(l,r)$) are there to place brackets to make the expression starting from at the l^{th} symbol and ending at r^{th} symbol evaluate to false (F)?
- Problems are solved in the order of $r-l$ breaking evens arbitrarily.

5. You are given a boolean expression consisting of a string of the symbols *true* and *false* and with exactly one operation *and, or, xor* between any two consecutive truth values. Count the number of ways to place brackets in the expression such that it will evaluate to true. For example, there is only 1 way to place parentheses in the expression *true* and *false* xor *true* such that it evaluates to true.

Solution:

- Let there be n symbols, and $n-1$ operations between them.
- We solve the following two subproblems:
  How many ways (denoted $T(l,r)$) are there to place brackets to make the expression starting from at the l^{th} symbol and ending at r^{th} symbol evaluate to true (T)?
  and
  How many ways (denoted $F(l,r)$) are there to place brackets to make the expression starting from at the l^{th} symbol and ending at r^{th} symbol evaluate to false (F)?
- Problems are solved in the order of $r-l$ breaking evens arbitrarily.

5. You are given a boolean expression consisting of a string of the symbols *true* and *false* and with exactly one operation *and, or, xor* between any two consecutive truth values. Count the number of ways to place brackets in the expression such that it will evaluate to true. For example, there is only 1 way to place parentheses in the expression *true* and *false* xor *true* such that it evaluates to true.

Solution:

- Let there be n symbols, and $n-1$ operations between them.
- We solve the following two subproblems:
  How many ways (denoted $T(l,r)$) are there to place brackets to make the expression starting from at the l^{th} symbol and ending at r^{th} symbol evaluate to true (T)?
  and
  How many ways (denoted $F(l,r)$) are there to place brackets to make the expression starting from at the l^{th} symbol and ending at r^{th} symbol evaluate to false (F)?
- Problems are solved in the order of $r-l$ breaking evens arbitrarily.

- The base case is that $\mathrm{T}(i, i)$ is 1 if symbol i is *true*, and 0 if symbol i is *false*. The reverse applies to $\mathrm{F}(i, i)$.
- Otherwise, for each subproblem, we 'split' the expression around an operator 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 two smaller expressions.
- We then evaluate the subproblems on each of the two sides, and combine the results together depending on the type of operator we are splitting by, and whether we want the result to evaluate to true or false.
- We solve both subproblems in parallel:

$$\mathrm{T}(l, r) = \sum_{m=l}^{r-1} \mathrm{TSplit}(l, m, r)$$

$$\mathrm{F}(l, r) = \sum_{m=l}^{r-1} \mathrm{FSplit}(l, m, r)$$

- The base case is that $\mathrm{T}(i, i)$ is 1 if symbol i is *true*, and 0 if symbol i is *false*. The reverse applies to $\mathrm{F}(i, i)$.
- Otherwise, for each subproblem, we 'split' the expression around an operator 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 two smaller expressions.
- We then evaluate the subproblems on each of the two sides, and combine the results together depending on the type of operator we are splitting by, and whether we want the result to evaluate to true or false.
- We solve both subproblems in parallel:

$$\mathrm{T}(l, r) = \sum_{m=l}^{r-1} \mathrm{TSplit}(l, m, r)$$

$$\mathrm{F}(l, r) = \sum_{m=l}^{r-1} \mathrm{FSplit}(l, m, r)$$

- The base case is that $\mathrm{T}(i, i)$ is 1 if symbol i is *true*, and 0 if symbol i is *false*. The reverse applies to $\mathrm{F}(i, i)$.
- Otherwise, for each subproblem, we 'split' the expression around an operator 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 two smaller expressions.
- We then evaluate the subproblems on each of the two sides, and combine the results together depending on the type of operator we are splitting by, and whether we want the result to evaluate to true or false.
- We solve both subproblems in parallel:

$$\mathrm{T}(l, r) = \sum_{m=l}^{r-1} \mathrm{TSplit}(l, m, r)$$

$$\mathrm{F}(l, r) = \sum_{m=l}^{r-1} \mathrm{FSplit}(l, m, r)$$

- The base case is that $\mathrm{T}(i, i)$ is 1 if symbol i is *true*, and 0 if symbol i is *false*. The reverse applies to $\mathrm{F}(i, i)$.
- Otherwise, for each subproblem, we 'split' the expression around an operator 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 two smaller expressions.
- We then evaluate the subproblems on each of the two sides, and combine the results together depending on the type of operator we are splitting by, and whether we want the result to evaluate to true or false.
- We solve both subproblems in parallel:

$$\mathrm{T}(l, r) = \sum_{m=l}^{r-1} \mathrm{TSplit}(l, m, r)$$

$$\mathrm{F}(l, r) = \sum_{m=l}^{r-1} \mathrm{FSplit}(l, m, r)$$

- The base case is that $\mathrm{T}(i, i)$ is 1 if symbol i is *true*, and 0 if symbol i is *false*. The reverse applies to $\mathrm{F}(i, i)$.
- Otherwise, for each subproblem, we 'split' the expression around an operator 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 two smaller expressions.
- We then evaluate the subproblems on each of the two sides, and combine the results together depending on the type of operator we are splitting by, and whether we want the result to evaluate to true or false.
- We solve both subproblems in parallel:

$$\mathrm{T}(l, r) = \sum_{m=l}^{r-1} \mathrm{TSplit}(l, m, r)$$

$$\mathrm{F}(l, r) = \sum_{m=l}^{r-1} \mathrm{FSplit}(l, m, r)$$

$$\mathrm{TSplit}(l,m,r) = \begin{cases} \mathrm{T}(l,m) \times \mathrm{T}(m+1,r) & \text{if operator } m \text{ is 'and',} \\ \mathrm{T}(l,m) \times \mathrm{F}(m+1,r) + \mathrm{T}(l,m) \times \mathrm{T}(m+1,r) + & \\ \mathrm{F}(l,m) \times \mathrm{T}(m+1,r) & \text{if operator } m \text{ is 'or',} \\ \mathrm{T}(l,m) \times \mathrm{F}(m+1,r) + \mathrm{F}(l,m) \times \mathrm{T}(m+1,r) & \\ \qquad \text{if operator } m \text{ is 'xor'.} & \end{cases}$$

$$\mathrm{FSplit}(l,m,r) = \begin{cases} \mathrm{T}(l,m) \times \mathrm{F}(m+1,r) + \mathrm{F}(l,m) \times \mathrm{F}(m+1,r) + & \\ \mathrm{F}(l,m) \times \mathrm{T}(m+1,r) & \text{if operator } m \text{ is 'and',} \\ \mathrm{F}(l,m) \times \mathrm{F}(m+1,r) & \text{if operator } m \text{ is 'or',} \\ \mathrm{T}(l,m) \times \mathrm{T}(m+1,r) + \mathrm{F}(l,m) \times \mathrm{F}(m+1,r) & \\ \qquad \text{if operator } m \text{ is 'xor'.} & \end{cases}$$

The complexity is $O(n^3)$. There are $O(n^2)$ different ranges that l and r could cover, and each needs the evaluations of TSplit or FSplit at up to $n-1$ different splitting points.

6. 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:

- Let us denote by $T(i)$ the subtree of the tree T of all employees which is rooted at an employee i.
- For each such subtree we will compute two quantities, $I(i)$ and $E(i)$. $I(i)$ is the maximal sum of fun factors $\text{fun}(i)$ of employees selected from subtree $T(i)$ which satisfies the constraint and which must include the root i.
- $E(i)$ is the maximal sum of fun factors of employees selected from the subtree $T(i)$ but which does NOT include i.

- These two quantities are easily computed by recursion on subtrees, starting from the leaves.
- For every employee i whose (immediate) subordinates are $j_1, \ldots, j_m$ we have

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

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

- Notice that in the definition of $E(i)$ we have the option to either include the children of i or exclude them, whatever produces larger value of $E(i)$.
- The final answer is $\max(I(n), E(n))$ where n is the root of the corporate tree T.
- Clearly such algorithm runs in time linear in the number of all employees.

- These two quantities are easily computed by recursion on subtrees, starting from the leaves.
- For every employee i whose (immediate) subordinates are $j_1, \ldots, j_m$ we have

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

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

- Notice that in the definition of $E(i)$ we have the option to either include the children of i or exclude them, whatever produces larger value of $E(i)$.
- The final answer is $\max(I(n), E(n))$ where n is the root of the corporate tree T.
- Clearly such algorithm runs in time linear in the number of all employees.

- These two quantities are easily computed by recursion on subtrees, starting from the leaves.
- For every employee i whose (immediate) subordinates are $j_1, \ldots, j_m$ we have

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

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

- Notice that in the definition of $E(i)$ we have the option to either include the children of i or exclude them, whatever produces larger value of $E(i)$.
- The final answer is $\max(I(n), E(n))$ where n is the root of the corporate tree T.
- Clearly such algorithm runs in time linear in the number of all employees.

- These two quantities are easily computed by recursion on subtrees, starting from the leaves.
- For every employee i whose (immediate) subordinates are $j_1, \ldots, j_m$ we have

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

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

- Notice that in the definition of $E(i)$ we have the option to either include the children of i or exclude them, whatever produces larger value of $E(i)$.
- The final answer is $\max(I(n), E(n))$ where n is the root of the corporate tree T.
- Clearly such algorithm runs in time linear in the number of all employees.

- These two quantities are easily computed by recursion on subtrees, starting from the leaves.
- For every employee i whose (immediate) subordinates are $j_1, \ldots, j_m$ we have

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

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

- Notice that in the definition of $E(i)$ we have the option to either include the children of i or exclude them, whatever produces larger value of $E(i)$.
- The final answer is $\max(I(n), E(n))$ where n is the root of the corporate tree T.
- Clearly such algorithm runs in time linear in the number of all employees.

7. 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, using as few bins as possible.

Solution:

- We will solve subproblems $P(i, j)$ of packing i many items of size s_1 and j many items of size s_2 for all $1 \leq i \leq n_1$ and all $1 \leq j \leq n_2$.
- Let $C/s_1 = K$. The recursion step is essentially an exhaustive search:

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

- Thus, 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.
- The algorithm runs in time $O(K\, n_1\, n_2)$.
- Thus, 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.

7. 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, using as few bins as possible.

Solution:

- We will solve subproblems $P(i, j)$ of packing i many items of size s_1 and j many items of size s_2 for all $1 \leq i \leq n_1$ and all $1 \leq j \leq n_2$.
- Let $C/s_1 = K$. The recursion step is essentially an exhaustive search:

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

- Thus, 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.
- The algorithm runs in time $O(K\, n_1\, n_2)$.
- Thus, 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.

7. 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, using as few bins as possible.

Solution:

- We will solve subproblems $P(i, j)$ of packing i many items of size s_1 and j many items of size s_2 for all $1 \leq i \leq n_1$ and all $1 \leq j \leq n_2$.
- Let $C/s_1 = K$. The recursion step is essentially an exhaustive search:

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

- Thus, 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.
- The algorithm runs in time $O(K\, n_1\, n_2)$.
- Thus, 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.

7. 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, using as few bins as possible.

Solution:

- We will solve subproblems $P(i, j)$ of packing i many items of size s_1 and j many items of size s_2 for all $1 \leq i \leq n_1$ and all $1 \leq j \leq n_2$.
- Let $C/s_1 = K$. The recursion step is essentially an exhaustive search:

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

- Thus, 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.
- The algorithm runs in time $O(K\, n_1\, n_2)$.
- Thus, 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.

7. 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, using as few bins as possible.

Solution:

- We will solve subproblems $P(i, j)$ of packing i many items of size s_1 and j many items of size s_2 for all $1 \leq i \leq n_1$ and all $1 \leq j \leq n_2$.
- Let $C/s_1 = K$. The recursion step is essentially an exhaustive search:

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

- Thus, 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.
- The algorithm runs in time $O(K\, n_1\, n_2)$.
- Thus, 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.

7. 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, using as few bins as possible.

Solution:

- We will solve subproblems $P(i, j)$ of packing i many items of size s_1 and j many items of size s_2 for all $1 \leq i \leq n_1$ and all $1 \leq j \leq n_2$.
- Let $C/s_1 = K$. The recursion step is essentially an exhaustive search:

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

- Thus, 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.
- The algorithm runs in time $O(K\, n_1\, n_2)$.
- Thus, 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.

7. 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, using as few bins as possible.

Solution:

- We will solve subproblems $P(i, j)$ of packing i many items of size s_1 and j many items of size s_2 for all $1 \leq i \leq n_1$ and all $1 \leq j \leq n_2$.
- Let $C/s_1 = K$. The recursion step is essentially an exhaustive search:

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

- Thus, 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.
- The algorithm runs in time $O(K\, n_1\, n_2)$.
- Thus, 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.

8. You are given n activities and for each activity i you are given its starting time s_i, its finishing time f_i and the profit p_i which you get if you schedule this activity. Only one activity can take place at any time. Your task is to design an algorithm which produces a subset S of those n activities so that no two activities in S overlap and such that the sum of profits of all activities in S is maximised.

Solution:

- Sort all activities by finishing time f_i.
- We solve the following subproblems for all i: *What is the maximum profit* $\text{opt}(i)$ *we can make if we are to choose between activities* $a_1, a_2, \ldots, a_i$ *and such that activity* a_i *is the last activity we do?*
- Recursion is simple:

$$\text{opt}(i) = \max\{\text{opt}(j) \,:\, f_j < s_i\} + p_i$$
$$\text{prev}(i) = \arg\max\{\text{opt}(j) \,:\, f_j < s_i\}$$

- Finally, the solution to the original problem is profit of $\max_{1 \le i \le n} \text{opt}(i)$ and the sequence of jobs can be obtained starting with $m = \arg\max_{1 \le i \le n} \text{opt}(i)$ and then backtracking via $m, \text{prev}(m), \text{prev}(\text{prev}(m)), \ldots$.

Solution:

- Sort all activities by finishing time f_i.
- We solve the following subproblems for all i: *What is the maximum profit* $\text{opt}(i)$ *we can make if we are to choose between activities* $a_1, a_2, \ldots, a_i$ *and such that activity* a_i *is the last activity we do?*
- Recursion is simple:

$$\text{opt}(i) = \max\{\text{opt}(j) \,:\, f_j < s_i\} + p_i$$
$$\text{prev}(i) = \arg\max\{\text{opt}(j) \,:\, f_j < s_i\}$$

- Finally, the solution to the original problem is profit of $\max_{1 \le i \le n} \text{opt}(i)$ and the sequence of jobs can be obtained starting with $m = \arg\max_{1 \le i \le n} \text{opt}(i)$ and then backtracking via $m, \text{prev}(m), \text{prev}(\text{prev}(m)), \ldots$.

Solution:

- Sort all activities by finishing time f_i.
- We solve the following subproblems for all i: *What is the maximum profit* $\text{opt}(i)$ *we can make if we are to choose between activities* $a_1, a_2, \ldots, a_i$ *and such that activity* a_i *is the last activity we do?*
- Recursion is simple:

$$\text{opt}(i) = \max\{\text{opt}(j) \,:\, f_j < s_i\} + p_i$$
$$\text{prev}(i) = \arg\max\{\text{opt}(j) \,:\, f_j < s_i\}$$

- Finally, the solution to the original problem is profit of $\max_{1 \le i \le n} \text{opt}(i)$ and the sequence of jobs can be obtained starting with $m = \arg\max_{1 \le i \le n} \text{opt}(i)$ and then backtracking via $m, \text{prev}(m), \text{prev}(\text{prev}(m)), \ldots$.

Solution:

- Sort all activities by finishing time f_i.
- We solve the following subproblems for all i: *What is the maximum profit* $\text{opt}(i)$ *we can make if we are to choose between activities* $a_1, a_2, \ldots, a_i$ *and such that activity* a_i *is the last activity we do?*
- Recursion is simple:

$$\text{opt}(i) = \max\{\text{opt}(j) \,:\, f_j < s_i\} + p_i$$
$$\text{prev}(i) = \arg\max\{\text{opt}(j) \,:\, f_j < s_i\}$$

- Finally, the solution to the original problem is profit of $\max_{1 \le i \le n} \text{opt}(i)$ and the sequence of jobs can be obtained starting with $m = \arg\max_{1 \le i \le n} \text{opt}(i)$ and then backtracking via $m, \text{prev}(m), \text{prev}(\text{prev}(m)), \ldots$.

9. Your shipping company has just received N individual 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. Devise an algorithm to select jobs which will bring you the largest possible amount of money.

Solution:

- This is just the standard knapsack problem with t_i being the size of the i^{th} 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 is $O(NT)$.

9. Your shipping company has just received N individual 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. Devise an algorithm to select jobs which will bring you the largest possible amount of money.

Solution:

- This is just the standard knapsack problem with t_i being the size of the i^{th} 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 is $O(NT)$.

9. Your shipping company has just received N individual 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. Devise an algorithm to select jobs which will bring you the largest possible amount of money.

Solution:

- This is just the standard knapsack problem with t_i being the size of the i^{th} 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 is $O(NT)$.

9. Your shipping company has just received N individual 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. Devise an algorithm to select jobs which will bring you the largest possible amount of money.

Solution:

- This is just the standard knapsack problem with t_i being the size of the i^{th} 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 is $O(NT)$.

10. Again your shipping company has just received N individual 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. 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.
- For each triplet (e, t, i) such that $e \leq E,\ t \leq T, i \leq N$ we solve the following subproblem: *Choose a sub-collection of items* $1 \ldots i$ *that fits in a knapsack of capacity* e *units of weight and* t *units of volume, which is of largest possible value.*

10. Again your shipping company has just received N individual 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. 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.
- For each triplet (e, t, i) such that $e \leq E,\ t \leq T, i \leq N$ we solve the following subproblem: *Choose a sub-collection of items* $1 \ldots i$ *that fits in a knapsack of capacity* e *units of weight and* t *units of volume, which is of largest possible value.*

10. Again your shipping company has just received N individual 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. 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.
- For each triplet (e, t, i) such that $e \leq E,\ t \leq T, i \leq N$ we solve the following subproblem: *Choose a sub-collection of items* $1 \ldots i$ *that fits in a knapsack of capacity* e *units of weight and* t *units of volume, which is of largest possible value.*

10. Again your shipping company has just received N individual 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. 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.
- For each triplet (e, t, i) such that $e \leq E,\ t \leq T, i \leq N$ we solve the following subproblem: *Choose a sub-collection of items* $1 \ldots i$ *that fits in a knapsack of capacity* e *units of weight and* t *units of volume, which is of largest possible value.*

- We put 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).
- 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 $O(NET)$

- We put 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).
- 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 $O(NET)$

- We put 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).
- 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 $O(NET)$

11. Because of the recent droughts, n proposals have been made to dam the Murray Darling river. The i^{th} 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). What is the largest number of dams that can be built? You may assume that $x_i < x_{i+1}$.

Solution:

- We solve this by finding the maximum value among the following subproblems for every $i \leq n$: *Find the largest possible number of dams that can be built among proposals* $1 \ldots i$*, such that the* i^{th} *dam is built.*
- The base case is $\text{opt}(1) = 1$.
- The recursion is:

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

- We do an $O(n)$ search at every dam proposal i. Thus, the total complexity is $O(n^2)$.

11. Because of the recent droughts, n proposals have been made to dam the Murray Darling river. The i^{th} 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). What is the largest number of dams that can be built? You may assume that $x_i < x_{i+1}$.

Solution:

- We solve this by finding the maximum value among the following subproblems for every $i \leq n$: *Find the largest possible number of dams that can be built among proposals* $1 \ldots i$*, such that the* i^{th} *dam is built.*
- The base case is $\text{opt}(1) = 1$.
- The recursion is:

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

- We do an $O(n)$ search at every dam proposal i. Thus, the total complexity is $O(n^2)$.

11. Because of the recent droughts, n proposals have been made to dam the Murray Darling river. The i^{th} 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). What is the largest number of dams that can be built? You may assume that $x_i < x_{i+1}$.

Solution:

- We solve this by finding the maximum value among the following subproblems for every $i \leq n$: *Find the largest possible number of dams that can be built among proposals* $1 \ldots i$*, such that the* i^{th} *dam is built.*
- The base case is $\text{opt}(1) = 1$.
- The recursion is:

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

- We do an $O(n)$ search at every dam proposal i. Thus, the total complexity is $O(n^2)$.

11. Because of the recent droughts, n proposals have been made to dam the Murray Darling river. The i^{th} 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). What is the largest number of dams that can be built? You may assume that $x_i < x_{i+1}$.

Solution:

- We solve this by finding the maximum value among the following subproblems for every $i \leq n$: *Find the largest possible number of dams that can be built among proposals* $1 \ldots i$*, such that the* i^{th} *dam is built.*
- The base case is $\text{opt}(1) = 1$.
- The recursion is:

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

- We do an $O(n)$ search at every dam proposal i. Thus, the total complexity is $O(n^2)$.

11. Because of the recent droughts, n proposals have been made to dam the Murray Darling river. The i^{th} 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). What is the largest number of dams that can be built? You may assume that $x_i < x_{i+1}$.

Solution:

- We solve this by finding the maximum value among the following subproblems for every $i \leq n$: *Find the largest possible number of dams that can be built among proposals* $1 \ldots i$*, such that the* i^{th} *dam is built.*
- The base case is $\text{opt}(1) = 1$.
- The recursion is:

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

- We do an $O(n)$ search at every dam proposal i. Thus, the total complexity is $O(n^2)$.

12. 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) (Very Tricky!) 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 $O(n^2)$.

12. 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) (Very Tricky!) 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 $O(n^2)$.

12. 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) (Very Tricky!) 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 $O(n^2)$.

12. 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.

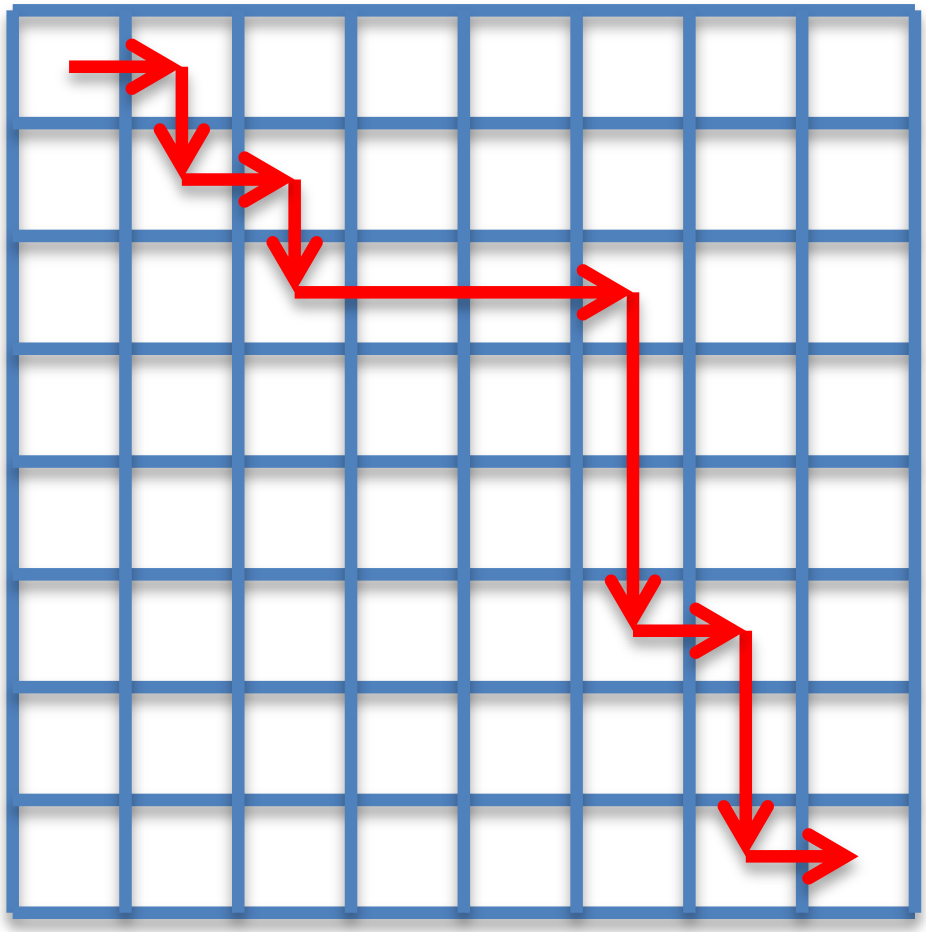
b) Describe an algorithm which always correctly finds a minimal sum path and runs in time n^2.

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

d) (Very Tricky!) 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 $O(n^2)$.

n

m



Solution: a) Start at the top left square, and always go to the immediate square below or to the right that has the smallest integer. This algorithm fails with the following example:

0	1	1
5	1000	1000
5	5	0

The algorithm will get a score of 1002, instead of the correct answer of 15.

b) We solve all subproblems of the form *"What is the best score we can get arriving at the cell at row i and column j"?* The base case is that $\text{opt}(1,1) = \text{board}(1,1)$, and $\text{opt}(i,j) = \infty$ for all i and j that are off the board. The recursion is:

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

The complexity is $O(n^2)$.

Solution: a) Start at the top left square, and always go to the immediate square below or to the right that has the smallest integer. This algorithm fails with the following example:

0	1	1
5	1000	1000
5	5	0

The algorithm will get a score of 1002, instead of the correct answer of 15.

b) We solve all subproblems of the form *"What is the best score we can get arriving at the cell at row i and column j"?* The base case is that $\text{opt}(1,1) = \text{board}(1,1)$, and $\text{opt}(i,j) = \infty$ for all i and j that are off the board. The recursion is:

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

The complexity is $O(n^2)$.

Solution: a) Start at the top left square, and always go to the immediate square below or to the right that has the smallest integer. This algorithm fails with the following example:

0	1	1
5	1000	1000
5	5	0

The algorithm will get a score of 1002, instead of the correct answer of 15.

b) We solve all subproblems of the form *"What is the best score we can get arriving at the cell at row i and column j"?* The base case is that $\text{opt}(1,1) = \text{board}(1,1)$, and $\text{opt}(i,j) = \infty$ for all i and j that are off the board. The recursion is:

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

The complexity is $O(n^2)$.

Solution: a) Start at the top left square, and always go to the immediate square below or to the right that has the smallest integer. This algorithm fails with the following example:

0	1	1
5	1000	1000
5	5	0

The algorithm will get a score of 1002, instead of the correct answer of 15.

b) We solve all subproblems of the form *"What is the best score we can get arriving at the cell at row i and column j"?* The base case is that $\text{opt}(1,1) = \text{board}(1,1)$, and $\text{opt}(i,j) = \infty$ for all i and j that are off the board. The recursion is:

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

The complexity is $O(n^2)$.

Solution: a) Start at the top left square, and always go to the immediate square below or to the right that has the smallest integer. This algorithm fails with the following example:

0	1	1
5	1000	1000
5	5	0

The algorithm will get a score of 1002, instead of the correct answer of 15.

b) We solve all subproblems of the form *"What is the best score we can get arriving at the cell at row i and column j"?* The base case is that $\text{opt}(1,1) = \text{board}(1,1)$, and $\text{opt}(i,j) = \infty$ for all i and j that are off the board. The recursion is:

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

The complexity is $O(n^2)$.

c) Solve the following subproblem: What is the minimum number of ways to reach row i and column j with the score $\text{opt}(i, j)$? The base case is $\text{ways}(1, 1) = 1$ and $\text{ways}(i, j) = 0$ for all i and j that are off the board. The recursion is:

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

The complexity is once again $O(n^2)$

- **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.

- **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.

- **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.

- **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
  $O(n \times n + n \times n/2 + n \times n/4 + \ldots) = O(n \times 2n) = O(n^2)$

- 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
$O(n \times n + n \times n/2 + n \times n/4 + \ldots) = O(n \times 2n) = O(n^2)$

- 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
$O(n \times n + n \times n/2 + n \times n/4 + \ldots) = O(n \times 2n) = O(n^2)$

- 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
  $O(n \times n + n \times n/2 + n \times n/4 + \ldots) = O(n \times 2n) = O(n^2)$

13. A palindrome is a sequence of letters which reads equally from left to right and from right to left. Given a sequence of letters, find efficiently its longest subsequence (not necessarily contiguous) which is a palindrome. Thus, we are looking for a longest palindrome which can be obtained by crossing out some of the letters of the initial sequence without permuting the remaining letters.

Solution:

- Let S_i denote the i^{th} letter of the string.
- Solve the subproblems: *What is the longest palindrome within the substring starting at letter i and ending at j.*
- Subproblems are solved in order of $j - i$.
- The base case is that $\text{opt}(i, i) = 1$, as a single letter is a palindrome by itself.
- We solve this using the recursion:

$$\text{opt}(i,j) = \begin{cases} 1 & \text{if } i = j, \\ 2 & \text{if } i+1 = j \text{ and } S_i = S_j, \\ \text{opt}(i+1, j-1) + 2 & \text{if } S_i = S_j \text{ and } i < j-2, \\ \max\{\text{opt}(i, j-1), \text{opt}(i+1, j)\} & \text{else} \end{cases}$$

13. A palindrome is a sequence of letters which reads equally from left to right and from right to left. Given a sequence of letters, find efficiently its longest subsequence (not necessarily contiguous) which is a palindrome. Thus, we are looking for a longest palindrome which can be obtained by crossing out some of the letters of the initial sequence without permuting the remaining letters.

Solution:

- Let S_i denote the i^{th} letter of the string.
- Solve the subproblems: *What is the longest palindrome within the substring starting at letter i and ending at j.*
- Subproblems are solved in order of $j - i$.
- The base case is that $\text{opt}(i, i) = 1$, as a single letter is a palindrome by itself.
- We solve this using the recursion:

$$\text{opt}(i,j) = \begin{cases} 1 & \text{if } i = j, \\ 2 & \text{if } i+1 = j \text{ and } S_i = S_j, \\ \text{opt}(i+1, j-1) + 2 & \text{if } S_i = S_j \text{ and } i < j-2, \\ \max\{\text{opt}(i, j-1), \text{opt}(i+1, j)\} & \text{else} \end{cases}$$

13. A palindrome is a sequence of letters which reads equally from left to right and from right to left. Given a sequence of letters, find efficiently its longest subsequence (not necessarily contiguous) which is a palindrome. Thus, we are looking for a longest palindrome which can be obtained by crossing out some of the letters of the initial sequence without permuting the remaining letters.

Solution:

- Let S_i denote the i^{th} letter of the string.
- Solve the subproblems: *What is the longest palindrome within the substring starting at letter i and ending at j.*
- Subproblems are solved in order of $j - i$.
- The base case is that $\text{opt}(i, i) = 1$, as a single letter is a palindrome by itself.
- We solve this using the recursion:

$$\text{opt}(i,j) = \begin{cases} 1 & \text{if } i = j, \\ 2 & \text{if } i+1 = j \text{ and } S_i = S_j, \\ \text{opt}(i+1, j-1) + 2 & \text{if } S_i = S_j \text{ and } i < j-2, \\ \max\{\text{opt}(i, j-1), \text{opt}(i+1, j)\} & \text{else} \end{cases}$$

13. A palindrome is a sequence of letters which reads equally from left to right and from right to left. Given a sequence of letters, find efficiently its longest subsequence (not necessarily contiguous) which is a palindrome. Thus, we are looking for a longest palindrome which can be obtained by crossing out some of the letters of the initial sequence without permuting the remaining letters.

Solution:

- Let S_i denote the i^{th} letter of the string.
- Solve the subproblems: *What is the longest palindrome within the substring starting at letter i and ending at j.*
- Subproblems are solved in order of $j - i$.
- The base case is that $\text{opt}(i, i) = 1$, as a single letter is a palindrome by itself.
- We solve this using the recursion:

$$\text{opt}(i,j) = \begin{cases} 1 & \text{if } i = j, \\ 2 & \text{if } i+1 = j \text{ and } S_i = S_j, \\ \text{opt}(i+1, j-1) + 2 & \text{if } S_i = S_j \text{ and } i < j-2, \\ \max\{\text{opt}(i, j-1), \text{opt}(i+1, j)\} & \text{else} \end{cases}$$

13. A palindrome is a sequence of letters which reads equally from left to right and from right to left. Given a sequence of letters, find efficiently its longest subsequence (not necessarily contiguous) which is a palindrome. Thus, we are looking for a longest palindrome which can be obtained by crossing out some of the letters of the initial sequence without permuting the remaining letters.

Solution:

- Let S_i denote the i^{th} letter of the string.
- Solve the subproblems: *What is the longest palindrome within the substring starting at letter i and ending at j.*
- Subproblems are solved in order of $j - i$.
- The base case is that $\text{opt}(i, i) = 1$, as a single letter is a palindrome by itself.
- We solve this using the recursion:

$$\text{opt}(i,j) = \begin{cases} 1 & \text{if } i = j, \\ 2 & \text{if } i+1 = j \text{ and } S_i = S_j, \\ \text{opt}(i+1, j-1) + 2 & \text{if } S_i = S_j \text{ and } i < j-2, \\ \max\{\text{opt}(i, j-1), \text{opt}(i+1, j)\} & \text{else} \end{cases}$$

13. A palindrome is a sequence of letters which reads equally from left to right and from right to left. Given a sequence of letters, find efficiently its longest subsequence (not necessarily contiguous) which is a palindrome. Thus, we are looking for a longest palindrome which can be obtained by crossing out some of the letters of the initial sequence without permuting the remaining letters.

Solution:

- Let S_i denote the i^{th} letter of the string.
- Solve the subproblems: *What is the longest palindrome within the substring starting at letter i and ending at j.*
- Subproblems are solved in order of $j - i$.
- The base case is that $\text{opt}(i, i) = 1$, as a single letter is a palindrome by itself.
- We solve this using the recursion:

$$\text{opt}(i,j) = \begin{cases} 1 & \text{if } i = j, \\ 2 & \text{if } i+1 = j \text{ and } S_i = S_j, \\ \text{opt}(i+1, j-1) + 2 & \text{if } S_i = S_j \text{ and } i < j-2, \\ \max\{\text{opt}(i, j-1), \text{opt}(i+1, j)\} & \text{else} \end{cases}$$

and a function from pairs of natural numbers into pairs of natural numbers

$$\text{pred}[(i,j)] = \begin{cases} (i,i) & \text{if } i = j, \\ (i,j) & \text{if } i+1 = j \text{ and } S_i = S_j, \\ (i+1,j-1) & \text{if } S_i = S_j \text{ and } i < j-2, \\ (i,j-1) & \text{if } S_i \neq S_j \text{ and } \text{opt}(i,j-1) \geq \text{opt}(i+1,j) \\ (i+1,j) & \text{else} \end{cases}$$

- To reconstruct the palindrome, backtrack iterating function $\text{pred}[(i,j)]$ starting with $(1,n)$ and recoding values of i and j when $S_i = S_j$.
- The complexity of this algorithm is $O(n^2)$, where n is the length of the string.

and a function from pairs of natural numbers into pairs of natural numbers

$$\text{pred}[(i,j)] = \begin{cases} (i,i) & \text{if } i = j, \\ (i,j) & \text{if } i+1 = j \text{ and } S_i = S_j, \\ (i+1,j-1) & \text{if } S_i = S_j \text{ and } i < j-2, \\ (i,j-1) & \text{if } S_i \neq S_j \text{ and } \text{opt}(i,j-1) \geq \text{opt}(i+1,j) \\ (i+1,j) & \text{else} \end{cases}$$

- To reconstruct the palindrome, backtrack iterating function $\text{pred}[(i,j)]$ starting with $(1,n)$ and recoding values of i and j when $S_i = S_j$.
- The complexity of this algorithm is $O(n^2)$, where n is the length of the string.

and a function from pairs of natural numbers into pairs of natural numbers

$$\text{pred}[(i,j)] = \begin{cases} (i,i) & \text{if } i = j, \\ (i,j) & \text{if } i+1 = j \text{ and } S_i = S_j, \\ (i+1, j-1) & \text{if } S_i = S_j \text{ and } i < j-2, \\ (i, j-1) & \text{if } S_i \neq S_j \text{ and } \text{opt}(i,j-1) \geq \text{opt}(i+1,j) \\ (i+1,j) & \text{else} \end{cases}$$

- To reconstruct the palindrome, backtrack iterating function $\text{pred}[(i,j)]$ starting with $(1,n)$ and recoding values of i and j when $S_i = S_j$.
- The complexity of this algorithm is $O(n^2)$, where n is the length of the string.

14. 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 $p_1 + \ldots + p_t = n$.

1. Find 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 $1 \leq k \leq n$.
2. Find the total number of partitions of n.

Solution:

- **(a)** Let numpart(k, m) denote the number of partitions of m in which every part is at most k.
- Then for all m numpart$(1, m) = 1$ because the only way to represent m with $k = 1$ would be as a sum of m ones.
- For $k \geq 2$ the value of numpart(k, m) satisfies the recursion

$$\text{numpart}(k, m) = \text{numpart}(k-1, m) + \text{numpart}(k, m-k)$$

- The first term on the right counts number of partitions of m when no part is equal k and the second term counts the number of partitions when there exists at least one part equal to k, which we subtract from m and look at the number of partitions of the remainder $m - k$ in which all the parts are also at most k.
- Note that numpart(k, m) are computed in order of the value of the sum $m + k$.
- **(b)** We run the previous algorithm; the solution is numpart(n, n).

Solution:

- **(a)** Let numpart(k, m) denote the number of partitions of m in which every part is at most k.
- Then for all m numpart$(1, m) = 1$ because the only way to represent m with $k = 1$ would be as a sum of m ones.
- For $k \geq 2$ the value of numpart(k, m) satisfies the recursion

$$\text{numpart}(k, m) = \text{numpart}(k-1, m) + \text{numpart}(k, m-k)$$

- The first term on the right counts number of partitions of m when no part is equal k and the second term counts the number of partitions when there exists at least one part equal to k, which we subtract from m and look at the number of partitions of the remainder $m - k$ in which all the parts are also at most k.
- Note that numpart(k, m) are computed in order of the value of the sum $m + k$.
- **(b)** We run the previous algorithm; the solution is numpart(n, n).

Solution:

- **(a)** Let numpart(k, m) denote the number of partitions of m in which every part is at most k.
- Then for all m numpart$(1, m) = 1$ because the only way to represent m with $k = 1$ would be as a sum of m ones.
- For $k \geq 2$ the value of numpart(k, m) satisfies the recursion

$$\text{numpart}(k, m) = \text{numpart}(k-1, m) + \text{numpart}(k, m-k)$$

- The first term on the right counts number of partitions of m when no part is equal k and the second term counts the number of partitions when there exists at least one part equal to k, which we subtract from m and look at the number of partitions of the remainder $m - k$ in which all the parts are also at most k.
- Note that numpart(k, m) are computed in order of the value of the sum $m + k$.
- **(b)** We run the previous algorithm; the solution is numpart(n, n).

Solution:

- **(a)** Let numpart(k, m) denote the number of partitions of m in which every part is at most k.
- Then for all m numpart$(1, m) = 1$ because the only way to represent m with $k = 1$ would be as a sum of m ones.
- For $k \geq 2$ the value of numpart(k, m) satisfies the recursion

$$\text{numpart}(k, m) = \text{numpart}(k - 1, m) + \text{numpart}(k, m - k)$$

- The first term on the right counts number of partitions of m when no part is equal k and the second term counts the number of partitions when there exists at least one part equal to k, which we subtract from m and look at the number of partitions of the remainder $m - k$ in which all the parts are also at most k.
- Note that numpart(k, m) are computed in order of the value of the sum $m + k$.
- **(b)** We run the previous algorithm; the solution is numpart(n, n).

Solution:

- **(a)** Let numpart(k, m) denote the number of partitions of m in which every part is at most k.
- Then for all m numpart$(1, m) = 1$ because the only way to represent m with $k = 1$ would be as a sum of m ones.
- For $k \geq 2$ the value of numpart(k, m) satisfies the recursion

$$\text{numpart}(k, m) = \text{numpart}(k-1, m) + \text{numpart}(k, m-k)$$

- The first term on the right counts number of partitions of m when no part is equal k and the second term counts the number of partitions when there exists at least one part equal to k, which we subtract from m and look at the number of partitions of the remainder $m - k$ in which all the parts are also at most k.
- Note that numpart(k, m) are computed in order of the value of the sum $m + k$.
- **(b)** We run the previous algorithm; the solution is numpart(n, n).

Solution:

- **(a)** Let numpart(k, m) denote the number of partitions of m in which every part is at most k.
- Then for all m numpart$(1, m) = 1$ because the only way to represent m with $k = 1$ would be as a sum of m ones.
- For $k \geq 2$ the value of numpart(k, m) satisfies the recursion

$$\text{numpart}(k, m) = \text{numpart}(k - 1, m) + \text{numpart}(k, m - k)$$

- The first term on the right counts number of partitions of m when no part is equal k and the second term counts the number of partitions when there exists at least one part equal to k, which we subtract from m and look at the number of partitions of the remainder $m - k$ in which all the parts are also at most k.
- Note that numpart(k, m) are computed in order of the value of the sum $m + k$.
- **(b)** We run the previous algorithm; the solution is numpart(n, n).

15. We say that a sequence of Roman letters A occurs as a subsequence of a sequence of Roman letters B if we can obtain A by deleting some of the symbols of B. Design an algorithm which for every two sequences 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. For example, the sequence ba has three occurrences in the sequence baba: baba, baba, baba.

Solution:

- Let A_i be the i^{th} letter of A, and B_j be the j^{th} letter of B.
- Solve the subproblems: *How many times do the first i letters of A appear as a subsequence of the first j letters of B.*
- The base case is $\text{ways}(1,1) = 1$ if $A_1 = B_1$ and 0 otherwise; also, clearly, $\text{ways}(i,j) = 0$ if $i > j$.

15. We say that a sequence of Roman letters A occurs as a subsequence of a sequence of Roman letters B if we can obtain A by deleting some of the symbols of B. Design an algorithm which for every two sequences 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. For example, the sequence ba has three occurrences in the sequence baba: baba, baba, baba.

Solution:

- Let A_i be the i^{th} letter of A, and B_j be the j^{th} letter of B.
- Solve the subproblems: *How many times do the first i letters of A appear as a subsequence of the first j letters of B.*
- The base case is ways$(1, 1) = 1$ if $A_1 = B_1$ and 0 otherwise; also, clearly, ways$(i, j) = 0$ if $i > j$.

15. We say that a sequence of Roman letters A occurs as a subsequence of a sequence of Roman letters B if we can obtain A by deleting some of the symbols of B. Design an algorithm which for every two sequences 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. For example, the sequence ba has three occurrences in the sequence baba: baba, baba, baba.

Solution:

- Let A_i be the i^{th} letter of A, and B_j be the j^{th} letter of B.
- Solve the subproblems: *How many times do the first i letters of A appear as a subsequence of the first j letters of B.*
- The base case is ways$(1, 1) = 1$ if $A_1 = B_1$ and 0 otherwise; also, clearly, ways$(i, j) = 0$ if $i > j$.

15. We say that a sequence of Roman letters A occurs as a subsequence of a sequence of Roman letters B if we can obtain A by deleting some of the symbols of B. Design an algorithm which for every two sequences 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. For example, the sequence ba has three occurrences in the sequence baba: baba, baba, baba.

Solution:

- Let A_i be the i^{th} letter of A, and B_j be the j^{th} letter of B.
- Solve the subproblems: *How many times do the first i letters of A appear as a subsequence of the first j letters of B.*
- The base case is $\text{ways}(1,1) = 1$ if $A_1 = B_1$ and 0 otherwise; also, clearly, $\text{ways}(i,j) = 0$ if $i > j$.

The recursion is, for all $i < j$

$$\text{opt}(i,j) = \begin{cases} 0, & \text{if } i > j \\ \text{opt}(i, j-1), & \text{if } A_i \neq B_j \\ \text{opt}(i-1, j-1) + \text{opt}(i, j-1) & \text{if } A_i = B_j \end{cases}$$

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.

The complexity is $O(nm)$ where n is the length of A and m is the length of B.

16. 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). Give an O(n)-time algorithm for computing an optimal placement.

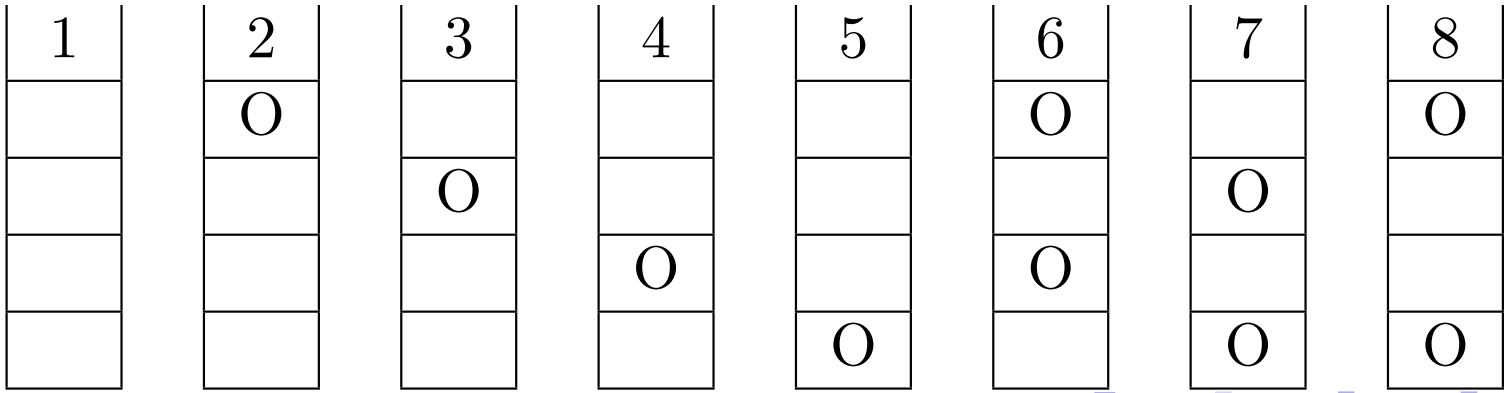
Solution:

- There are 8 patterns, listed below, which are legal and that can occur in any column (in isolation, ignoring the pebbles in adjacent columns):

16. 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). Give an O(n)-time algorithm for computing an optimal placement.

Solution:

- There are 8 patterns, listed below, which are legal and that can occur in any column (in isolation, ignoring the pebbles in adjacent columns):



- 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 $1 \leq k \leq n$.
- Each sub-problem can be assigned a type, which is the pattern occurring in the last column.
- 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.
- 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.

- 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 $1 \leq k \leq n$.
- Each sub-problem can be assigned a type, which is the pattern occurring in the last column.
- 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.
- 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.

- 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 $1 \leq k \leq n$.
- Each sub-problem can be assigned a type, which is the pattern occurring in the last column.
- 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.
- 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.

- 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 $1 \leq k \leq n$.
- Each sub-problem can be assigned a type, which is the pattern occurring in the last column.
- 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.
- 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.

- 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 $1 \leq k \leq n$.
- Each sub-problem can be assigned a type, which is the pattern occurring in the last column.
- 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.
- 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.

- 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 $1 \leq k \leq n$.
- Each sub-problem can be assigned a type, which is the pattern occurring in the last column.
- 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.
- 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.

- 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 $1 \leq k \leq n$.
- Each sub-problem can be assigned a type, which is the pattern occurring in the last column.
- 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.
- 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.

- 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 $1 \leq k \leq n$.
- Each sub-problem can be assigned a type, which is the pattern occurring in the last column.
- 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.
- 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.

- 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 $1 \leq k \leq n$.
- Each sub-problem can be assigned a type, which is the pattern occurring in the last column.
- 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.
- 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 compatibilities are as follows:
  (a) pattern 1 is compatible with all 8 patterns (including itself);
  (b) pattern 2 is compatible with all patterns except 2,6 and 8
  (c) pattern 3 is compatible with all patterns except 3 and 7;
  (d) pattern 4 is compatible with all patterns except 4 and 6;
  (e) pattern 5 is compatible with all patterns except 5,7 and 8;
  (f) pattern 6 is compatible with all patterns except 2,6 and 8;
  (g) pattern 7 is compatible with all patterns except 3,5,7 and 8;
  (h) pattern 8 is compatible with all patterns except 2,5,6,7 and 8.

- The complexity is $O(n)$, as the number of patterns is constant.

- The compatibilities are as follows:
  (a) pattern 1 is compatible with all 8 patterns (including itself);
  (b) pattern 2 is compatible with all patterns except 2,6 and 8
  (c) pattern 3 is compatible with all patterns except 3 and 7;
  (d) pattern 4 is compatible with all patterns except 4 and 6;
  (e) pattern 5 is compatible with all patterns except 5,7 and 8;
  (f) pattern 6 is compatible with all patterns except 2,6 and 8;
  (g) pattern 7 is compatible with all patterns except 3,5,7 and 8;
  (h) pattern 8 is compatible with all patterns except 2,5,6,7 and 8.

- The complexity is $O(n)$, as the number of patterns is constant.

17. Skiers go fastest with skis whose length is about their height. Your team consists of n members, with heights $h_1, h_2, \ldots, h_n$. Your team gets a delivery of $m \geq n$ pairs of skis, with lengths $l_1, l_2, \ldots, l_m$. Your goal is to 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 he got, i.e., to minimise

$$\sum_{1 \leq i \leq n} |h_i - l_{s(i)}|$$

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

- First observe that if we have two skiers i and j such that $h_i < h_j$, then $s(i) < s(j)$. If this were not the case, we could swap the skis assigned to i and j, which would lower the sum of differences.
- This implies that we may initially sort the skiers by height, and sort the skis by length, and find some sort of pairing such that there are no crossovers (similar to question 4).
- Also, if you have equal number of skiers and skis, you would give i^{th} skier i^{th} pair of skis.
- We now solve the following subproblem: *What is the minimum cost of matching the first i skiers with the first $j > i$ skis such that each of the first i skiers gets a ski?*
- The base case is $\text{opt}(i, i) = \sum_{k=1}^{i} |h_k - l_k|$.
- The recursion for $j > i$ is:

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

- First observe that if we have two skiers i and j such that $h_i < h_j$, then $s(i) < s(j)$. If this were not the case, we could swap the skis assigned to i and j, which would lower the sum of differences.
- This implies that we may initially sort the skiers by height, and sort the skis by length, and find some sort of pairing such that there are no crossovers (similar to question 4).
- Also, if you have equal number of skiers and skis, you would give i^{th} skier i^{th} pair of skis.
- We now solve the following subproblem: *What is the minimum cost of matching the first i skiers with the first $j > i$ skis such that each of the first i skiers gets a ski?*
- The base case is $\text{opt}(i, i) = \sum_{k=1}^{i} |h_k - l_k|$.
- The recursion for $j > i$ is:

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

- First observe that if we have two skiers i and j such that $h_i < h_j$, then $s(i) < s(j)$. If this were not the case, we could swap the skis assigned to i and j, which would lower the sum of differences.
- This implies that we may initially sort the skiers by height, and sort the skis by length, and find some sort of pairing such that there are no crossovers (similar to question 4).
- Also, if you have equal number of skiers and skis, you would give i^{th} skier i^{th} pair of skis.
- We now solve the following subproblem: *What is the minimum cost of matching the first i skiers with the first $j > i$ skis such that each of the first i skiers gets a ski?*
- The base case is $\text{opt}(i, i) = \sum_{k=1}^{i} |h_k - l_k|$.
- The recursion for $j > i$ is:

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

- First observe that if we have two skiers i and j such that $h_i < h_j$, then $s(i) < s(j)$. If this were not the case, we could swap the skis assigned to i and j, which would lower the sum of differences.
- This implies that we may initially sort the skiers by height, and sort the skis by length, and find some sort of pairing such that there are no crossovers (similar to question 4).
- Also, if you have equal number of skiers and skis, you would give i^{th} skier i^{th} pair of skis.
- We now solve the following subproblem: *What is the minimum cost of matching the first i skiers with the first $j > i$ skis such that each of the first i skiers gets a ski?*
- The base case is $\text{opt}(i, i) = \sum_{k=1}^{i} |h_k - l_k|$.
- The recursion for $j > i$ is:

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

- First observe that if we have two skiers i and j such that $h_i < h_j$, then $s(i) < s(j)$. If this were not the case, we could swap the skis assigned to i and j, which would lower the sum of differences.
- This implies that we may initially sort the skiers by height, and sort the skis by length, and find some sort of pairing such that there are no crossovers (similar to question 4).
- Also, if you have equal number of skiers and skis, you would give i^{th} skier i^{th} pair of skis.
- We now solve the following subproblem: *What is the minimum cost of matching the first i skiers with the first $j > i$ skis such that each of the first i skiers gets a ski?*
- The base case is $\text{opt}(i, i) = \sum_{k=1}^{i} |h_k - l_k|$.
- The recursion for $j > i$ is:

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

- First observe that if we have two skiers i and j such that $h_i < h_j$, then $s(i) < s(j)$. If this were not the case, we could swap the skis assigned to i and j, which would lower the sum of differences.
- This implies that we may initially sort the skiers by height, and sort the skis by length, and find some sort of pairing such that there are no crossovers (similar to question 4).
- Also, if you have equal number of skiers and skis, you would give i^{th} skier i^{th} pair of skis.
- We now solve the following subproblem: *What is the minimum cost of matching the first i skiers with the first $j > i$ skis such that each of the first i skiers gets a ski?*
- The base case is $\text{opt}(i, i) = \sum_{k=1}^{i} |h_k - l_k|$.
- The recursion for $j > i$ is:

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

- To retrieve the assignment, 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 $1 \leq k \leq i$.
- The complexity of the initial recursion $O(nm)$. The complexity of retrieving the assignment is $O(m)$, as each time we run the algorithm, j decreases by exactly 1.

- To retrieve the assignment, 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 $1 \leq k \leq i$.
- The complexity of the initial recursion $O(nm)$. The complexity of retrieving the assignment is $O(m)$, as each time we run the algorithm, j decreases by exactly 1.

- To retrieve the assignment, 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 $1 \leq k \leq i$.
- The complexity of the initial recursion $O(nm)$. The complexity of retrieving the assignment is $O(m)$, as each time we run the algorithm, j decreases by exactly 1.

- To retrieve the assignment, 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 $1 \leq k \leq i$.
- The complexity of the initial recursion $O(nm)$. The complexity of retrieving the assignment is $O(m)$, as each time we run the algorithm, j decreases by exactly 1.

18. 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 algorithm to find the minimum possible cutting cost for any given stick.

- Let $x(i)$ be the distance along the stick the ith cutting point is at.
- Solve the following subproblems: *What is the minimum cost of cutting up he stick completely from cutting point i to cutting point j.*
- The base case is $\text{opt}(i, i+1) = 0$.
- The recursion is:

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

- The complexity is $O(n^3)$, where n is the number of cutting points on the stick.
- Note: always cutting the stick as close to its middle as possible does not always produce an optimal solution; try making a counter example.

- Let $x(i)$ be the distance along the stick the ith cutting point is at.
- Solve the following subproblems: *What is the minimum cost of cutting up he stick completely from cutting point i to cutting point j.*
- The base case is $\text{opt}(i, i+1) = 0$.
- The recursion is:

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

- The complexity is $O(n^3)$, where n is the number of cutting points on the stick.
- Note: always cutting the stick as close to its middle as possible does not always produce an optimal solution; try making a counter example.

- Let $x(i)$ be the distance along the stick the ith cutting point is at.
- Solve the following subproblems: *What is the minimum cost of cutting up he stick completely from cutting point i to cutting point j.*
- The base case is $\text{opt}(i, i+1) = 0$.
- The recursion is:

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

- The complexity is $O(n^3)$, where n is the number of cutting points on the stick.
- Note: always cutting the stick as close to its middle as possible does not always produce an optimal solution; try making a counter example.

- Let $x(i)$ be the distance along the stick the ith cutting point is at.
- Solve the following subproblems: *What is the minimum cost of cutting up he stick completely from cutting point i to cutting point j.*
- The base case is $\text{opt}(i, i+1) = 0$.
- The recursion is:

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

- The complexity is $O(n^3)$, where n is the number of cutting points on the stick.
- Note: always cutting the stick as close to its middle as possible does not always produce an optimal solution; try making a counter example.

- Let $x(i)$ be the distance along the stick the ith cutting point is at.
- Solve the following subproblems: *What is the minimum cost of cutting up he stick completely from cutting point i to cutting point j.*
- The base case is $\text{opt}(i, i+1) = 0$.
- The recursion is:

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

- The complexity is $O(n^3)$, where n is the number of cutting points on the stick.
- Note: always cutting the stick as close to its middle as possible does not always produce an optimal solution; try making a counter example.

- Let $x(i)$ be the distance along the stick the ith cutting point is at.
- Solve the following subproblems: *What is the minimum cost of cutting up he stick completely from cutting point i to cutting point j.*
- The base case is $\text{opt}(i, i+1) = 0$.
- The recursion is:

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

- The complexity is $O(n^3)$, where n is the number of cutting points on the stick.
- Note: always cutting the stick as close to its middle as possible does not always produce an optimal solution; try making a counter example.

19. For bit strings $X = x_1 \dots x_m$, $Y = y_1 \dots y_n$ and $Z = z_1 \dots z_{m+n}$ 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.

For example if X = 101 and Y = 01 then $x_1x_2y_1x_3y_2 = 10011$ is an interleaving of X and Y, whereas 11010 is not. Give an efficient algorithm to determine if Z is an interleaving of X and Y.

Solution:

- Solve the following subproblems: *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?*
- The base case is $\text{opt}(0, 0) = \text{true}$, and $\text{opt}(i, j) = \text{false}$ if $i < 0$ or $j < 0$.
- The recursion is:

$$\text{opt}(i, j) = (\text{opt}(i-1, j) \text{ AND } z_{i+j} = x_i) \text{ OR } (\text{opt}(i, j-1) \text{ AND } z_{i+j} = y_j).$$

- The problems are solved in order of the size of $i + j$.
- The complexity is $O(nm)$.

19. For bit strings $X = x_1 \dots x_m$, $Y = y_1 \dots y_n$ and $Z = z_1 \dots z_{m+n}$ 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.

For example if X = 101 and Y = 01 then $x_1x_2y_1x_3y_2 = 10011$ is an interleaving of X and Y, whereas 11010 is not. Give an efficient algorithm to determine if Z is an interleaving of X and Y.

Solution:

- Solve the following subproblems: *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?*
- The base case is $\text{opt}(0, 0) = \text{true}$, and $\text{opt}(i, j) = \text{false}$ if $i < 0$ or $j < 0$.
- The recursion is:

$$\text{opt}(i, j) = (\text{opt}(i-1, j) \text{ AND } z_{i+j} = x_i) \text{ OR } (\text{opt}(i, j-1) \text{ AND } z_{i+j} = y_j).$$

- The problems are solved in order of the size of $i + j$.
- The complexity is $O(nm)$.

19. For bit strings $X = x_1 \dots x_m$, $Y = y_1 \dots y_n$ and $Z = z_1 \dots z_{m+n}$ 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.

For example if X = 101 and Y = 01 then $x_1x_2y_1x_3y_2 = 10011$ is an interleaving of X and Y, whereas 11010 is not. Give an efficient algorithm to determine if Z is an interleaving of X and Y.

Solution:

- Solve the following subproblems: *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?*
- The base case is $\text{opt}(0, 0) = \text{true}$, and $\text{opt}(i, j) = \text{false}$ if $i < 0$ or $j < 0$.
- The recursion is:

$$\text{opt}(i, j) = (\text{opt}(i-1, j) \text{ AND } z_{i+j} = x_i) \text{ OR } (\text{opt}(i, j-1) \text{ AND } z_{i+j} = y_j).$$

- The problems are solved in order of the size of $i + j$.
- The complexity is $O(nm)$.

19. For bit strings $X = x_1 \dots x_m$, $Y = y_1 \dots y_n$ and $Z = z_1 \dots z_{m+n}$ 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.

For example if X = 101 and Y = 01 then $x_1x_2y_1x_3y_2 = 10011$ is an interleaving of X and Y, whereas 11010 is not. Give an efficient algorithm to determine if Z is an interleaving of X and Y.

Solution:

- Solve the following subproblems: *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?*
- The base case is $\text{opt}(0, 0) = \text{true}$, and $\text{opt}(i, j) = \text{false}$ if $i < 0$ or $j < 0$.
- The recursion is:

$$\text{opt}(i, j) = (\text{opt}(i-1, j) \text{ AND } z_{i+j} = x_i) \text{ OR } (\text{opt}(i, j-1) \text{ AND } z_{i+j} = y_j).$$

- The problems are solved in order of the size of $i + j$.
- The complexity is $O(nm)$.

19. For bit strings $X = x_1 \dots x_m$, $Y = y_1 \dots y_n$ and $Z = z_1 \dots z_{m+n}$ 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.

For example if X = 101 and Y = 01 then $x_1x_2y_1x_3y_2 = 10011$ is an interleaving of X and Y, whereas 11010 is not. Give an efficient algorithm to determine if Z is an interleaving of X and Y.

Solution:

- Solve the following subproblems: *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?*
- The base case is $\text{opt}(0, 0) = \text{true}$, and $\text{opt}(i, j) = \text{false}$ if $i < 0$ or $j < 0$.
- The recursion is:

$$\text{opt}(i, j) = (\text{opt}(i-1, j) \text{ AND } z_{i+j} = x_i) \text{ OR } (\text{opt}(i, j-1) \text{ AND } z_{i+j} = y_j).$$

- The problems are solved in order of the size of $i + j$.
- The complexity is $O(nm)$.

19. For bit strings $X = x_1 \dots x_m$, $Y = y_1 \dots y_n$ and $Z = z_1 \dots z_{m+n}$ 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.

For example if X = 101 and Y = 01 then $x_1x_2y_1x_3y_2 = 10011$ is an interleaving of X and Y, whereas 11010 is not. Give an efficient algorithm to determine if Z is an interleaving of X and Y.

Solution:

- Solve the following subproblems: *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?*
- The base case is $\text{opt}(0, 0) = \text{true}$, and $\text{opt}(i, j) = \text{false}$ if $i < 0$ or $j < 0$.
- The recursion is:

$$\text{opt}(i, j) = (\text{opt}(i-1, j) \text{ AND } z_{i+j} = x_i) \text{ OR } (\text{opt}(i, j-1) \text{ AND } z_{i+j} = y_j).$$

- The problems are solved in order of the size of $i + j$.
- The complexity is $O(nm)$.

20. 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 IQs are decreasing. Design an algorithm which given the weights and IQs of n elephants, will find a longest sequence of elephants such that their weights are increasing but IQs are decreasing.

Solution: Sort the elephants in order of decreasing IQs. The problem now becomes finding the longest increasing subsequence of elephant weights.

21. 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: Observe that the minimum and the maximum number of illegal workers you could possible have on the evening of day i are $\max(K - i, 0)$ and $K + i$.

We solve the following subproblems: *What is the minimum number of inspections I have failed on day i assuming that on that day I have j many illegal workers?*

21. 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: Observe that the minimum and the maximum number of illegal workers you could possible have on the evening of day i are $\max(K - i, 0)$ and $K + i$.

We solve the following subproblems: *What is the minimum number of inspections I have failed on day i assuming that on that day I have j many illegal workers*?

- The base case is $\text{opt}(0, K) = 0$, since we start with 0 failed inspections before the first day begins.
- Let $\text{failed}(i, j)$ return 1 if the j falls out of the range of $[l_i, r_i]$, and return 0 otherwise.
- The recursion is: for all i such that $1 \leq i \leq N$ and all j such that $\max(0, K - i) \leq j \leq K + i$,

- The base case is $\text{opt}(0, K) = 0$, since we start with 0 failed inspections before the first day begins.
- Let $\text{failed}(i, j)$ return 1 if the j falls out of the range of $[l_i, r_i]$, and return 0 otherwise.
- The recursion is: for all i such that $1 \leq i \leq N$ and all j such that $\max(0, K - i) \leq j \leq K + i$,

- The base case is $\text{opt}(0, K) = 0$, since we start with 0 failed inspections before the first day begins.
- Let $\text{failed}(i, j)$ return 1 if the j falls out of the range of $[l_i, r_i]$, and return 0 otherwise.
- The recursion is: for all i such that $1 \leq i \leq N$ and all j such that $\max(0, K - i) \leq j \leq K + i$,

$$\text{opt}(i,j) =$$

$$\text{failed}(i,j) + \min \begin{cases} \begin{cases} \text{opt}(i-1,j-1) & \text{if } j-1 \geq \max(0, K-(i-1)) \\ \infty & \text{if } j-1 < \max(0, K-(i-1)) \end{cases} \\ \text{opt}(i-1,j), \\ \begin{cases} \text{opt}(i-1,j+1) & \text{if } j+1 \leq K+i-1 \\ \infty & \text{if } j+1 > K+i-1 \end{cases} \end{cases}$$

- 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 on the previous day was achievable over the period of $i-1$ days (you start with K workers, so after $i-1$ days you must have at least $K-(i-1)$ workers which happens if you kept firing one worker every day);
- the second corresponds to keeping the same number of illegal workers as on the previous day;
- the third option corresponds to firing a worker from the previous day.

$$\text{opt}(i,j) =$$

$$\text{failed}(i,j) + \min \begin{cases} \begin{cases} \text{opt}(i-1,j-1) & \text{if } j-1 \geq \max(0, K-(i-1)) \\ \infty & \text{if } j-1 < \max(0, K-(i-1)) \end{cases} \\ \text{opt}(i-1,j), \\ \begin{cases} \text{opt}(i-1,j+1) & \text{if } j+1 \leq K+i-1 \\ \infty & \text{if } j+1 > K+i-1 \end{cases} \end{cases}$$

- 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 on the previous day was achievable over the period of $i-1$ days (you start with K workers, so after $i-1$ days you must have at least $K-(i-1)$ workers which happens if you kept firing one worker every day);
- the second corresponds to keeping the same number of illegal workers as on the previous day;
- the third option corresponds to firing a worker from the previous day.

$$\text{opt}(i,j) =$$

$$\text{failed}(i,j) + \min \begin{cases} \begin{cases} \text{opt}(i-1,j-1) & \text{if } j-1 \geq \max(0, K-(i-1)) \\ \infty & \text{if } j-1 < \max(0, K-(i-1)) \end{cases} \\ \text{opt}(i-1,j), \\ \begin{cases} \text{opt}(i-1,j+1) & \text{if } j+1 \leq K+i-1 \\ \infty & \text{if } j+1 > K+i-1 \end{cases} \end{cases}$$

- 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 on the previous day was achievable over the period of $i-1$ days (you start with K workers, so after $i-1$ days you must have at least $K-(i-1)$ workers which happens if you kept firing one worker every day);
- the second corresponds to keeping the same number of illegal workers as on the previous day;
- the third option corresponds to firing a worker from the previous day.

$$\text{opt}(i,j) =$$

$$\text{failed}(i,j) + \min \begin{cases} \begin{cases} \text{opt}(i-1,j-1) & \text{if } j-1 \geq \max(0, K-(i-1)) \\ \infty & \text{if } j-1 < \max(0, K-(i-1)) \end{cases} \\ \text{opt}(i-1,j), \\ \begin{cases} \text{opt}(i-1,j+1) & \text{if } j+1 \leq K+i-1 \\ \infty & \text{if } j+1 > K+i-1 \end{cases} \end{cases}$$

- 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 on the previous day was achievable over the period of $i-1$ days (you start with K workers, so after $i-1$ days you must have at least $K-(i-1)$ workers which happens if you kept firing one worker every day);
- the second corresponds to keeping the same number of illegal workers as on the previous day;
- the third option corresponds to firing a worker from the previous day.

- The final answer is equal to $\min_{\max(K-N,0) \le j \le K+N} \text{opt}(N, j)$.
- The complexity is $O(N^2)$, as there are N days, and at most $2N + 1$ possible values of illegal worker each day.

- The final answer is equal to $\min_{\max(K-N,0) \le j \le K+N} \text{opt}(N, j)$.
- The complexity is $O(N^2)$, as there are N days, and at most $2N + 1$ possible values of illegal worker each day.

22. 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.

Solution:

- Solve the subproblems: *What is the number of ways I can split the first i elements into contiguous blocks of sum at most K.*
- The base case is $\text{opt}(0) = 1$.
- For $1 \leq j < i$ let $\text{sum}(j, i)$ is the sum of all numbers from $A[j]$ to $A[i]$ inclusive.
- The recursion is:

$$\text{opt}(i) = \sum \{\text{opt}(j-1) : 1 \leq j \leq i, \text{sum}(j, i) \leq K\},$$

- The complexity is $O(n^2)$, as each subproblem requires a $O(n)$ search.

22. 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.

Solution:

- Solve the subproblems: *What is the number of ways I can split the first i elements into contiguous blocks of sum at most K.*
- The base case is $\text{opt}(0) = 1$.
- For $1 \leq j < i$ let $\text{sum}(j, i)$ is the sum of all numbers from $A[j]$ to $A[i]$ inclusive.
- The recursion is:

$$\text{opt}(i) = \sum \{\text{opt}(j-1) : 1 \leq j \leq i, \text{sum}(j, i) \leq K\},$$

- The complexity is $O(n^2)$, as each subproblem requires a $O(n)$ search.

22. 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.

Solution:

- Solve the subproblems: *What is the number of ways I can split the first i elements into contiguous blocks of sum at most K.*
- The base case is $\text{opt}(0) = 1$.
- For $1 \leq j < i$ let $\text{sum}(j, i)$ is the sum of all numbers from $A[j]$ to $A[i]$ inclusive.
- The recursion is:

$$\text{opt}(i) = \sum \{\text{opt}(j-1) : 1 \leq j \leq i, \text{sum}(j, i) \leq K\},$$

- The complexity is $O(n^2)$, as each subproblem requires a $O(n)$ search.

22. 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.

Solution:

- Solve the subproblems: *What is the number of ways I can split the first i elements into contiguous blocks of sum at most K.*
- The base case is $\text{opt}(0) = 1$.
- For $1 \leq j < i$ let $\text{sum}(j, i)$ is the sum of all numbers from $A[j]$ to $A[i]$ inclusive.
- The recursion is:

$$\text{opt}(i) = \sum \{\text{opt}(j-1) : 1 \leq j \leq i, \text{sum}(j, i) \leq K\},$$

- The complexity is $O(n^2)$, as each subproblem requires a $O(n)$ search.

22. 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.

Solution:

- Solve the subproblems: *What is the number of ways I can split the first i elements into contiguous blocks of sum at most K.*
- The base case is $\text{opt}(0) = 1$.
- For $1 \leq j < i$ let $\text{sum}(j, i)$ is the sum of all numbers from $A[j]$ to $A[i]$ inclusive.
- The recursion is:

$$\text{opt}(i) = \sum \{\text{opt}(j-1) : 1 \leq j \leq i, \text{sum}(j, i) \leq K\},$$

- The complexity is $O(n^2)$, as each subproblem requires a $O(n)$ search.

22. 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.

Solution:

- Solve the subproblems: *What is the number of ways I can split the first i elements into contiguous blocks of sum at most K.*
- The base case is $\text{opt}(0) = 1$.
- For $1 \leq j < i$ let $\text{sum}(j, i)$ is the sum of all numbers from $A[j]$ to $A[i]$ inclusive.
- The recursion is:

$$\text{opt}(i) = \sum \{\text{opt}(j-1) : 1 \leq j \leq i, \text{sum}(j, i) \leq K\},$$

- The complexity is $O(n^2)$, as each subproblem requires a $O(n)$ search.

23. Given a 2D $R \times C$ grid of squares representing elevation of terrain, find the path going only down and right that minimises the number of moves from lower elevation to higher elevation.

Solution:

- Let a move from lower elevation to higher elevation be called a 'climb'. We solve the subproblems: *What is the minimum number of 'climbs' to reach row i and column j.*
- The base case is $\text{opt}(0,0) = 0$, and $\text{opt}(i,j) = 0$ if i and j lie off the board.
- The recursion is:

$$\text{opt}(i,j) = \min \begin{cases} \begin{cases} \text{opt}(i-1,j) & \text{if height}(i-1,j) \geq \text{height}(i,j) \\ \text{opt}(i-1,j)+1 & \text{if height}(i-1,j) < \text{height}(i,j) \end{cases} \\ \begin{cases} \text{opt}(i,j-1) & \text{if height}(i,j-1) \geq \text{height}(i,j) \\ \text{opt}(i,j-1)+1 & \text{if height}(i,j-1) < \text{height}(i,j) \end{cases} \end{cases}$$

- The complexity is $O(RC)$.

23. Given a 2D $R \times C$ grid of squares representing elevation of terrain, find the path going only down and right that minimises the number of moves from lower elevation to higher elevation.

Solution:

- Let a move from lower elevation to higher elevation be called a 'climb'. We solve the subproblems: *What is the minimum number of 'climbs' to reach row i and column j.*
- The base case is $\text{opt}(0,0) = 0$, and $\text{opt}(i,j) = 0$ if i and j lie off the board.
- The recursion is:

$$\text{opt}(i,j) = \min \begin{cases} \begin{cases} \text{opt}(i-1,j) & \text{if height}(i-1,j) \geq \text{height}(i,j) \\ \text{opt}(i-1,j)+1 & \text{if height}(i-1,j) < \text{height}(i,j) \end{cases} \\ \begin{cases} \text{opt}(i,j-1) & \text{if height}(i,j-1) \geq \text{height}(i,j) \\ \text{opt}(i,j-1)+1 & \text{if height}(i,j-1) < \text{height}(i,j) \end{cases} \end{cases}$$

- The complexity is $O(RC)$.

23. Given a 2D $R \times C$ grid of squares representing elevation of terrain, find the path going only down and right that minimises the number of moves from lower elevation to higher elevation.

Solution:

- Let a move from lower elevation to higher elevation be called a 'climb'. We solve the subproblems: *What is the minimum number of 'climbs' to reach row i and column j.*
- The base case is $\text{opt}(0,0) = 0$, and $\text{opt}(i,j) = 0$ if i and j lie off the board.
- The recursion is:

$$\text{opt}(i,j) = \min \begin{cases} \begin{cases} \text{opt}(i-1,j) & \text{if height}(i-1,j) \geq \text{height}(i,j) \\ \text{opt}(i-1,j)+1 & \text{if height}(i-1,j) < \text{height}(i,j) \end{cases} \\ \begin{cases} \text{opt}(i,j-1) & \text{if height}(i,j-1) \geq \text{height}(i,j) \\ \text{opt}(i,j-1)+1 & \text{if height}(i,j-1) < \text{height}(i,j) \end{cases} \end{cases}$$

- The complexity is $O(RC)$.

23. Given a 2D $R \times C$ grid of squares representing elevation of terrain, find the path going only down and right that minimises the number of moves from lower elevation to higher elevation.

Solution:

- Let a move from lower elevation to higher elevation be called a 'climb'. We solve the subproblems: *What is the minimum number of 'climbs' to reach row i and column j.*
- The base case is $\text{opt}(0,0) = 0$, and $\text{opt}(i,j) = 0$ if i and j lie off the board.
- The recursion is:

$$\text{opt}(i,j) = \min \begin{cases} \begin{cases} \text{opt}(i-1,j) & \text{if height}(i-1,j) \geq \text{height}(i,j) \\ \text{opt}(i-1,j)+1 & \text{if height}(i-1,j) < \text{height}(i,j) \end{cases} \\ \begin{cases} \text{opt}(i,j-1) & \text{if height}(i,j-1) \geq \text{height}(i,j) \\ \text{opt}(i,j-1)+1 & \text{if height}(i,j-1) < \text{height}(i,j) \end{cases} \end{cases}$$

- The complexity is $O(RC)$.

23. Given a 2D $R \times C$ grid of squares representing elevation of terrain, find the path going only down and right that minimises the number of moves from lower elevation to higher elevation.

Solution:

- Let a move from lower elevation to higher elevation be called a 'climb'. We solve the subproblems: *What is the minimum number of 'climbs' to reach row i and column j.*
- The base case is $\text{opt}(0,0) = 0$, and $\text{opt}(i,j) = 0$ if i and j lie off the board.
- The recursion is:

$$\text{opt}(i,j) = \min \begin{cases} \begin{cases} \text{opt}(i-1,j) & \text{if height}(i-1,j) \geq \text{height}(i,j) \\ \text{opt}(i-1,j)+1 & \text{if height}(i-1,j) < \text{height}(i,j) \end{cases} \\ \begin{cases} \text{opt}(i,j-1) & \text{if height}(i,j-1) \geq \text{height}(i,j) \\ \text{opt}(i,j-1)+1 & \text{if height}(i,j-1) < \text{height}(i,j) \end{cases} \end{cases}$$

- The complexity is $O(RC)$.

24. 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:

- The subproblems are: *Can I reach the i^{th} level after playing exactly k levels?*
- The base case is $\text{opt}(0, 0) = \text{true}$.
- The recursion is:

$$\text{opt}(i, k) = \text{opt}(i-1, k-1) \text{ OR } (\exists j < i-1)(\text{opt}(j, k-1) \text{ AND } \text{link}(j, i))$$

- Here $\text{link}(j, i)$ is true if and only if level j has a secret exit to level i.
- The final answer is $\text{opt}(K, N)$. The time complexity is $O(N^2)$.

24. 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:

- The subproblems are: *Can I reach the i^{th} level after playing exactly k levels?*
- The base case is $\text{opt}(0, 0) = \text{true}$.
- The recursion is:

$$\text{opt}(i, k) = \text{opt}(i-1, k-1) \text{ OR } (\exists j < i-1)(\text{opt}(j, k-1) \text{ AND } \text{link}(j, i))$$

- Here $\text{link}(j, i)$ is true if and only if level j has a secret exit to level i.
- The final answer is $\text{opt}(K, N)$. The time complexity is $O(N^2)$.

24. 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:

- The subproblems are: *Can I reach the i^{th} level after playing exactly k levels?*
- The base case is $\text{opt}(0, 0) = \text{true}$.
- The recursion is:

$$\text{opt}(i, k) = \text{opt}(i-1, k-1) \text{ OR } (\exists j < i-1)(\text{opt}(j, k-1) \text{ AND } \text{link}(j, i))$$

- Here $\text{link}(j, i)$ is true if and only if level j has a secret exit to level i.
- The final answer is $\text{opt}(K, N)$. The time complexity is $O(N^2)$.

24. 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:

- The subproblems are: *Can I reach the i^{th} level after playing exactly k levels?*
- The base case is $\text{opt}(0, 0) = \text{true}$.
- The recursion is:

$$\text{opt}(i, k) = \text{opt}(i-1, k-1) \text{ OR } (\exists j < i-1)(\text{opt}(j, k-1) \text{ AND } \text{link}(j, i))$$

- Here $\text{link}(j, i)$ is true if and only if level j has a secret exit to level i.
- The final answer is $\text{opt}(K, N)$. The time complexity is $O(N^2)$.

24. 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:

- The subproblems are: *Can I reach the i^{th} level after playing exactly k levels?*
- The base case is $\text{opt}(0, 0) = \text{true}$.
- The recursion is:

$$\text{opt}(i, k) = \text{opt}(i-1, k-1) \text{ OR } (\exists j < i-1)(\text{opt}(j, k-1) \text{ AND } \text{link}(j, i))$$

- Here $\text{link}(j, i)$ is true if and only if level j has a secret exit to level i.
- The final answer is $\text{opt}(K, N)$. The time complexity is $O(N^2)$.

24. 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:

- The subproblems are: *Can I reach the i^{th} level after playing exactly k levels?*
- The base case is $\text{opt}(0, 0) = \text{true}$.
- The recursion is:

$$\text{opt}(i, k) = \text{opt}(i-1, k-1) \text{ OR } (\exists j < i-1)(\text{opt}(j, k-1) \text{ AND } \text{link}(j, i))$$

- Here $\text{link}(j, i)$ is true if and only if level j has a secret exit to level i.
- The final answer is $\text{opt}(K, N)$. The time complexity is $O(N^2)$.

25. You are on vacation for N days at a resort that has three possible activities. For each day, for each activity, you've determined how much enjoyment you will get out of that activity. However, you are not allowed to do the same activity two days in a row. What is the maximum total enjoyment possible?

Solution:

- We solve the following subproblems: *What is the maximum enjoyment by day i if I do activity j on that day?*
- The base case is $\text{opt}(0, j) = 0$.
- Let the enjoyment of activity j be e_j. The recursion is:

$$\text{opt}(i, j) = e_j + \max\{\text{opt}(i-1, k) : k \neq j\}.$$

- The solution to the stated problem is $\max_{1 \leq j \leq 3} \text{opt}(N, j)$.
- As there are only three activities to consider, the complexity is $O(N)$.

25. You are on vacation for N days at a resort that has three possible activities. For each day, for each activity, you've determined how much enjoyment you will get out of that activity. However, you are not allowed to do the same activity two days in a row. What is the maximum total enjoyment possible?

Solution:

- We solve the following subproblems: *What is the maximum enjoyment by day i if I do activity j on that day?*
- The base case is $\text{opt}(0, j) = 0$.
- Let the enjoyment of activity j be e_j. The recursion is:

$$\text{opt}(i, j) = e_j + \max\{\text{opt}(i-1, k) : k \neq j\}.$$

- The solution to the stated problem is $\max_{1 \leq j \leq 3} \text{opt}(N, j)$.
- As there are only three activities to consider, the complexity is $O(N)$.

25. You are on vacation for N days at a resort that has three possible activities. For each day, for each activity, you've determined how much enjoyment you will get out of that activity. However, you are not allowed to do the same activity two days in a row. What is the maximum total enjoyment possible?

Solution:

- We solve the following subproblems: *What is the maximum enjoyment by day i if I do activity j on that day?*
- The base case is $\text{opt}(0, j) = 0$.
- Let the enjoyment of activity j be e_j. The recursion is:

$$\text{opt}(i, j) = e_j + \max\{\text{opt}(i-1, k) : k \neq j\}.$$

- The solution to the stated problem is $\max_{1 \leq j \leq 3} \text{opt}(N, j)$.
- As there are only three activities to consider, the complexity is $O(N)$.

25. You are on vacation for N days at a resort that has three possible activities. For each day, for each activity, you've determined how much enjoyment you will get out of that activity. However, you are not allowed to do the same activity two days in a row. What is the maximum total enjoyment possible?

Solution:

- We solve the following subproblems: *What is the maximum enjoyment by day i if I do activity j on that day?*
- The base case is $\text{opt}(0, j) = 0$.
- Let the enjoyment of activity j be e_j. The recursion is:

$$\text{opt}(i, j) = e_j + \max\{\text{opt}(i-1, k) : k \neq j\}.$$

- The solution to the stated problem is $\max_{1 \leq j \leq 3} \text{opt}(N, j)$.
- As there are only three activities to consider, the complexity is $O(N)$.

25. You are on vacation for N days at a resort that has three possible activities. For each day, for each activity, you've determined how much enjoyment you will get out of that activity. However, you are not allowed to do the same activity two days in a row. What is the maximum total enjoyment possible?

Solution:

- We solve the following subproblems: *What is the maximum enjoyment by day i if I do activity j on that day?*
- The base case is $\text{opt}(0, j) = 0$.
- Let the enjoyment of activity j be e_j. The recursion is:

$$\text{opt}(i, j) = e_j + \max\{\text{opt}(i-1, k) : k \neq j\}.$$

- The solution to the stated problem is $\max_{1 \leq j \leq 3} \text{opt}(N, j)$.
- As there are only three activities to consider, the complexity is $O(N)$.

25. You are on vacation for N days at a resort that has three possible activities. For each day, for each activity, you've determined how much enjoyment you will get out of that activity. However, you are not allowed to do the same activity two days in a row. What is the maximum total enjoyment possible?

Solution:

- We solve the following subproblems: *What is the maximum enjoyment by day i if I do activity j on that day?*
- The base case is $\text{opt}(0, j) = 0$.
- Let the enjoyment of activity j be e_j. The recursion is:

$$\text{opt}(i, j) = e_j + \max\{\text{opt}(i-1, k) : k \neq j\}.$$

- The solution to the stated problem is $\max_{1 \leq j \leq 3} \text{opt}(N, j)$.
- As there are only three activities to consider, the complexity is $O(N)$.

26. Given a sequence of n positive or negative integers $A_1, A_2, ...A_n$, determine a contiguous subsequence A_i to A_j for which the sum of elements in the subsequence is maximised.

Solution:

- We solve the subproblems: *What is the maximum sum of elements ending with integer i?*
- The base case is $\text{opt}(0) = 0$.
- The recursion is:

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

- Here, we take the maximum of 0 and $\text{opt}(i-1)$ because we may either choose to add A_i to the previous block, or start a completely new block.

26. Given a sequence of n positive or negative integers $A_1, A_2, ...A_n$, determine a contiguous subsequence A_i to A_j for which the sum of elements in the subsequence is maximised.

Solution:

- We solve the subproblems: *What is the maximum sum of elements ending with integer i?*
- The base case is $\text{opt}(0) = 0$.
- The recursion is:

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

- Here, we take the maximum of 0 and $\text{opt}(i-1)$ because we may either choose to add A_i to the previous block, or start a completely new block.

26. Given a sequence of n positive or negative integers $A_1, A_2, ...A_n$, determine a contiguous subsequence A_i to A_j for which the sum of elements in the subsequence is maximised.

Solution:

- We solve the subproblems: *What is the maximum sum of elements ending with integer i?*
- The base case is $\text{opt}(0) = 0$.
- The recursion is:

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

- Here, we take the maximum of 0 and $\text{opt}(i-1)$ because we may either choose to add A_i to the previous block, or start a completely new block.

26. Given a sequence of n positive or negative integers $A_1, A_2, ...A_n$, determine a contiguous subsequence A_i to A_j for which the sum of elements in the subsequence is maximised.

Solution:

- We solve the subproblems: *What is the maximum sum of elements ending with integer i?*
- The base case is $\text{opt}(0) = 0$.
- The recursion is:

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

- Here, we take the maximum of 0 and $\text{opt}(i-1)$ because we may either choose to add A_i to the previous block, or start a completely new block.

26. Given a sequence of n positive or negative integers $A_1, A_2, ...A_n$, determine a contiguous subsequence A_i to A_j for which the sum of elements in the subsequence is maximised.

Solution:

- We solve the subproblems: *What is the maximum sum of elements ending with integer i?*
- The base case is $\text{opt}(0) = 0$.
- The recursion is:

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

- Here, we take the maximum of 0 and $\text{opt}(i-1)$ because we may either choose to add A_i to the previous block, or start a completely new block.

- To determine the actual block, we may also solve the following recursive function for all i:

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

- To get the block, we simply find the element i such that $opt(i)$ is maximised. The block with maximum sum is $[\text{start}(i), i]$.
- The complexity is $O(n)$.

- To determine the actual block, we may also solve the following recursive function for all i:

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

- To get the block, we simply find the element i such that $opt(i)$ is maximised. The block with maximum sum is $[\text{start}(i), i]$.
- The complexity is $O(n)$.

- To determine the actual block, we may also solve the following recursive function for all i:

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

- To get the block, we simply find the element i such that $opt(i)$ is maximised. The block with maximum sum is $[\text{start}(i), i]$.
- The complexity is $O(n)$.

27. Consider a row of n coins of values $v_1, v_2, ...v_n$, 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:

- Let $\text{opt}(i, j)$ for i, j two integers such that $j - i + 1$ an even number and $1 \leq i < j \leq n$ denote the maximal amount of money we can definitely win if we play on the subsequence between coins at the position i and position j.
- Then $\text{opt}(i, j)$ satisfies the following recursion

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

27. Consider a row of n coins of values $v_1, v_2, ...v_n$, 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:

- Let $\text{opt}(i, j)$ for i, j two integers such that $j - i + 1$ an even number and $1 \leq i < j \leq n$ denote the maximal amount of money we can definitely win if we play on the subsequence between coins at the position i and position j.
- Then $\text{opt}(i, j)$ satisfies the following recursion

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

27. Consider a row of n coins of values $v_1, v_2, ...v_n$, 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:

- Let $\text{opt}(i, j)$ for i, j two integers such that $j - i + 1$ an even number and $1 \leq i < j \leq n$ denote the maximal amount of money we can definitely win if we play on the subsequence between coins at the position i and position j.
- Then $\text{opt}(i, j)$ satisfies the following recursion

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

- 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.
- The problems to find $\text{opt}(i, j)$ are solved in order of the size of $j - i$.
- The complexity is $O(n^2)$, because 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.

- 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.
- The problems to find $\text{opt}(i, j)$ are solved in order of the size of $j - i$.
- The complexity is $O(n^2)$, because 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.

- 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.
- The problems to find $\text{opt}(i, j)$ are solved in order of the size of $j - i$.
- The complexity is $O(n^2)$, because 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.