Documentation

Lucas Guesser Targino da Silva

April 11, 2023

Contents

Todo list

Write this theorem appropriately and prove it. 15

0.1 Stairway to Heaven

0.1.1 Problem

You want to reach the heaven, which is at the top of a staircase. The staircase has n steps. At each step, you can climb either one step or two steps further. In how many ways can you reach heaven?

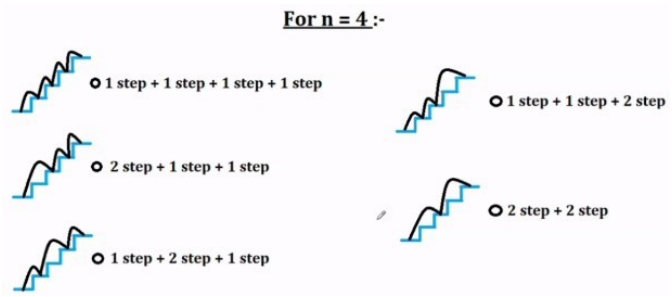


Figure 1: Example of all possible ways for n = 4

0.1.2 Solution - Recursion

Let W be the function which associates a number of steps with the number of **W**ays you can reach heaven.

Algorithm 1 Opt

```
1: W(n) = W(n - 1) + W(n - 2)
2: W(0) = 1                    ▷ if you are already there, then there is one way, stay where you are
3: W(1) = 1
4: W(2) = 2
```

Such solution allows memoization.

0.1.3 Solution - Sum

Algorithm 2 Opt

1: limit = quotient(n, 2)
2: $W(n) = \sum_{i=0}^{limit} \frac{(n-i)!}{i!(n-2i)!}$

That is not particularly obvious, but if you spend some time analysing how one constructs all possibilities and if you apply a bit of combinatorial analysis, you get there...

0.1.4 Solution - Fibonacci

Notice that the solution to this problem is a Fibonacci number.

Chapter 1

Share Market

1.1 Basic Definitions

Definition 1 (Natural)**.** Given $v \in \mathbb{N}$, we define:

$$\mathbb{N}_v = \{n \in \mathbb{N} : n \geqslant v\} \tag{1.1}$$

Definition 2 (Range)**.** Given $n \in \mathbb{N}_2$, we define the **Range of** n as:

$$[[n]] = \{0, \ldots, n-1\} = \{i \in \mathbb{N} : i < n\} \tag{1.2}$$

Definition 3 (Square)**.** Given $n \in \mathbb{N}_2$, we define the **Square of** n as:

$$sq(n) = [[n]] \times [[n]] \tag{1.3}$$

1.2 Problem Definitions

You are given an array in which the ith element is the price of a given stock on the day i. You are permitted to complete at most 1 transaction (i.e. buy once and sell once). What is the maximum profit you can gain?

Notice that you cannot sell a stock before buying it.

1.2.1 Definitions

Definition 4 (Price Tuple)**.** Given $n \in \mathbb{N}_2$, a **Price Tuple** p is a positive real tuple with n elements:

$$p = \langle p_0, \ldots, p_{n-1} \rangle \in \mathbb{R}_+^n \tag{1.4}$$

Definition 5 (Operation Pair)**.** Given $n \in \mathbb{N}_2$, an **Operation Pair** ω is a pair:

$$\omega = \langle b(\omega), s(\omega) \rangle \in sq(n) \quad , b < s \tag{1.5}$$

when the context is clear enough, we will simply write $\omega = \langle b(\omega), s(\omega) \rangle = \langle b, s \rangle$.

Definition 6 (Ascending Operation Pair)**.** Given $n \in \mathbb{N}_2$ and a Price Tuple p, an Operation Pair $\omega = \langle b, s \rangle$ is said to be **Ascending** when:

$$p_b \leqslant p_{b+1} \leqslant \cdots \leqslant p_{s-1} \leqslant p_s \tag{1.6}$$

Definition 7 (Maximal Operation Pair)**.** Given $n \in \mathbb{N}_2$ and a Price Tuple p, we say that an Ascending Operation Pair $\omega = \langle b, s \rangle$ is a **Maximal Operation Pair**, or simply that ω is **Maximal**, when it is Ascending and it satisfies the conditions:

$$p_s - p_b \geqslant p_{s+1} - p_b \quad \text{, if } s+1 < n \tag{1.7}$$
$$p_s - p_b \geqslant p_s - p_{b-1} \quad \text{, if } b-1 > 0 \tag{1.8}$$

Theorem 1. *Given $n \in \mathbb{N}_2$ and a Price Tuple p, an Operation Pair $\omega = \langle b, s \rangle$ is Maximal if and only if the following conditions are satisfied:*

$$p_s \geqslant p_{s+1} \quad \text{, if } s+1 < n \tag{1.9}$$
$$p_b \leqslant p_{b-1} \quad \text{, if } b-1 > 0 \tag{1.10}$$

Proof. It comes directly from the inequalities of the Definition 7. □

Definition 8 (Independent Operation Pairs)**.** Given two Operation Pairs ω, ω', we say that ω, ω' are **Independent** when:

$$s(\omega) < b(\omega') \vee s(\omega') < b(\omega) \tag{1.11}$$

moreover, given a set of Operation Pairs $S = \{\omega_0, \ldots, \omega_{m-1}\}$, we say that S is independent when it satisfies:

$$\forall\omega\forall\omega' \left(\omega \in S \wedge \omega' \in S \rightarrow \omega, \omega' \text{ are independent}\right) \tag{1.12}$$

i.e. all Operation Pairs $\omega, \omega' \in S$ are pairwise Independent.

Definition 9 (Operation Set)**.** Given $n \in \mathbb{N}_2$, an **Operation Set** $S \subseteq sq(n)$ is a set which satisfies:

1. $\forall\omega \left(\omega \in S \rightarrow \omega \text{ is an Operation Pair}\right)$
2. S is independent

Moreover, we denote the set of all possible Operations Set by:

$$\Omega_n = \{S \subseteq sq(n) : S \text{ is an Operation Set}\} \tag{1.13}$$

Definition 10. Given an $n \in \mathbb{N}_2$, a Price Tuple p, and an Operation Set $S \in \Omega_n$, the price $\rho(S)$ of S is:

$$\rho(S) = \sum_{\omega \in S} p_{s(\omega)} - p_{b(\omega)} \tag{1.14}$$

Lemma 1. *Given an $n \in \mathbb{N}_2$ and a Price Tuple p, if a Operation Set $S^* \in \Omega_n$ is has optimal price, then all Operation Pairs of S^* are Maximal, i.e.:*

$$\rho(S^*) = \max_{S \in \Omega_n} \left[\rho(S)\right] \rightarrow \forall\omega \left(\omega \in S^* \rightarrow \omega \textit{ is Maximal}\right) \tag{1.15}$$

Proof. Suppose $S^* \in \Omega_n$ is a Operation Set with optimal price. Suppose by contradiction that $\exists\omega \left(\omega \in S^* \wedge \omega \text{ is not Maximal}\right)$, and let $\omega = \langle b, s \rangle$ be such Operation Pair. By Theorem 1, one of the following cases must be true:

1. $p_s < p_{s+1} \quad \text{, if } s+1 < n$
2. $p_b > p_{b-1} \quad \text{, if } b-1 > 0$

Case 1

Suppose that $p_s < p_{s+1}$, $s+1 < n$.

Case 1.1

Suppose in addition that $\omega' = \langle b, s+1\rangle$ is independent of all $\omega \in S^* \setminus \{\omega\}$. Let $S' = (S^* \setminus \{\omega\}) \cup \{\omega'\}$. Notice that

$$\rho(\omega) = p_{s(\omega)} - p_{b(\omega)} < p_{s(\omega')} - p_{b(\omega')} = \rho(\omega') \tag{1.16}$$

Therefore $\rho(S^*) < \rho(S')$ (because S^* and S' differ only in the elements above), contradicting the optimality of S^*.

Case 1.2

Suppose in addition that $\exists\omega'(\omega' \in S^* \wedge b(\omega') = s+1)$, and let $\omega' = \langle s+1, s'\rangle$ be such Operation Pair. Let $\omega'' = \langle b, s'\rangle$ and $S'' = (S^* \cup \omega'') \setminus \{\omega, \omega'\}$. Notice that

$$\begin{aligned} \rho(\omega) + \rho(\omega') = \\ (p_{s(\omega)} - p_{b(\omega)}) + (p_{s(\omega')} - p_{b(\omega')}) = \\ (p_s - p_b) + (p_{s'} - p_{s+1}) = \\ (p_{s'} - p_b) + (p_s - p_{s+1}) < \\ p_{s'} - p_b = \\ \rho(\omega'') \end{aligned} \tag{1.17}$$

Therefore $\rho(S^*) < \rho(S'')$ (because S^* and S' differ only in the elements above), contradicting the optimality of S^*.

Case 1 - Conclusion

The hypothesis $p_s < p_{s+1}$, $s+1 < n$ leads to a contradiction. Therefore, if S^* is optimal, then $p_s \geqslant p_{s+1}$, $s+1 < n$, as we wanted to prove.

Case 2

Suppose that $p_b > p_{b-1}, b-1 > 0$. The proof of this case is similar to the proof of the Case 1.2.1, but this uses b and $b-1$ instead of s and $s-1$. □

Definition 11. Given an $n \in \mathbb{N}_2$, let $\omega = \langle b, s\rangle \in sq(n)$ be an Operation Pair. We say that i is included in ω, and denote by $i \triangleright \omega$, when $b \leqslant i \leqslant s$.

Definition 12. Given an $n \in \mathbb{N}_2$ and a Price Tuple p, we say that an Operation Set $S \in \Omega_n$ is Great when:

$$\forall i\big((i \in [[n-1]] \wedge p_i < p_{i+1}) \rightarrow (\exists\omega\,(\omega \in S \wedge i \triangleright \omega \wedge (i+1) \triangleright \omega))\big) \tag{1.18}$$

i.e. the indices of all ascending pairs of p are included in ω.

Lemma 2. *Given an $n \in \mathbb{N}_2$ and a Price Tuple p, if a Operation Set $S^* \in \Omega_n$ is has optimal price, then S^* is Great.*

Proof. Suppose by contradiction that S^* is not Great, i.e.

$$\exists i\big((i \in [[n-1]] \wedge p_i < p_{i+1}) \wedge (\nexists\omega\,(\omega \in S \wedge i \triangleright \omega \wedge (i+1) \triangleright \omega))\big) \tag{1.19}$$

and let i be such value.

Case 1

Suppose that $i \triangleright \omega \wedge \neg(i+1 \triangleright \omega)$ for some $\omega \in S^*$. If $i+1 \triangleright \omega'$ for some ω', then join ω and ω' to get an Operation Set with cost greater than S^*. If that is not the case, extend ω to include $i+1$ to get an Operation Set with cost greater than S^*. In all cases, the original hypothesis contradicts the optimality of S^*.

Case 2

Suppose that $\neg(i \triangleright \omega) \wedge i+1 \triangleright \omega$ for some $\omega \in S^*$. This case is similar to the previous one, except that the Operation Pair has to be extended backwards instead of forwards.

Case 3

Suppose that $\forall\omega(\omega \in S^* \rightarrow (\neg(i \triangleright \omega) \wedge \neg(i+1 \triangleright \omega)))$. Create a new Operation Set $S' = S^* \Cup \{\langle i, i+1\rangle\}$, which has greater cost, contradicting the optimality of S^*.

Conclusion

The cases above cover all possible cases. Therefore, the lemma has been proven by contradiction. □

Lemma 3. *Given an $n \in \mathbb{N}_2$ and a Price Tuple p, if an Operation Set $S^* \in \Omega_n$ satisfy:*

1. $\forall\omega\,(\omega \in S^* \rightarrow \omega$ *is Maximal*$)$
2. S^* *is Great*

then S has optimal price, i.e $\rho\,(S) = \max\limits_{S' \in \Omega_n} [\rho\,(S')]$.

It is tiresome to write this proof so I won't.

Theorem 2. *Given an $n \in \mathbb{N}_2$ and a Price Tuple p, an Operation Set $S^* \in \Omega_n$ has optimal price if and only if it satisfies:*

1. $\forall\omega\,(\omega \in S^* \rightarrow \omega$ *is Maximal*$)$
2. S^* *is Great*

Proof. Lemmas 1 and 2 prove the $\Rightarrow$ part. Lemma 3 proves the $\Leftarrow$. □

1.3 Problem Description

Input a Price Tuple p

Output a value $z \in \mathbb{R}$

Goal $\max z$

1.4 Solution - Dynamic Programming

Given that you buy on a day i, while the value does not decrease, you keep it. If it will drop the next day, you sell it.

1.4.1 Initial State

Find the first pair $\langle b, s \rangle$ for which the price increases, i.e. the first pair of consecutive indices for which $p_s - p_b > 0$.

1.4.2 Optimal Substructure

Let:

1. $\langle b_l, s_l \rangle$ be the last operation;
2. p_{s_l} be the price of the last sell;
3. i the index of the current day;
4. p_i the stock price of the current day;
5. p_{i-1} the stock price of the previous day;

Cases:

1. if $(s_l == i - 1) \wedge (p_{s_l} \leqslant p_i)$
   (a) replace $\langle b_l, s_l \rangle$ by $\langle b_l, i \rangle$
   (b) rationale: if the stock price is increasing, you keep it;
2. if $(s_l < i - 1) \wedge (p_{i-1} \leqslant p_i)$
   (a) add $\langle i - 1, i \rangle$
   (b) rationale: if you have no stock and it will increase, you buy and sell it;
3. the others are cases in which the stock price drops, and there is nothing to do;

Complexity: $\mathcal{O}(n)$

1.5 Solution - Simple

Algorithm 3 Simple-Algoruithm

1: $\Delta p \leftarrow [\langle p_{i+1} - p_i \rangle \ for \ i \in \{0, \ldots, n-2\}]$
2: $\Delta p_{>} \leftarrow filter(\Delta p, (>= 0))$
3: $r \leftarrow sum(\Delta p_{>})$
4: **return** r

The filter takes care of removing the drops on the price, while the sum of the differences computes the gains.

Complexity: $\mathcal{O}(n)$

Chapter 2

Sum of the Range

2.1 Problem Definition

2.1.1 Input

1. two natural numbers $m, n \in \mathbb{N}$

2. an array of values $v \in \mathbb{R}^n$

3. an set of queries $Q = \{\langle i, j\rangle : i, j < n\}^m$

2.1.2 Output

The output $a : Q \to \mathbb{R}^m$ is the answer function of all queries Q. The answer $a(q)$ to a query $q = \langle i, j\rangle$ is given by:

$$a(q) = \sum_{k=i}^{j} v[k] \tag{2.1}$$

2.2 Example

$$\begin{aligned} n &= 6 && (2.2)\\ m &= 3 && (2.3)\\ v &= \langle 1, -2, 3, 10, -8, 0\rangle && (2.4)\\ q &= \langle \langle 0, 2\rangle, \langle 1, 4\rangle, \langle 3, 3\rangle \rangle && (2.5)\\ a &= \langle 2, 3, 10\rangle = \langle 1 - 2 + 3, -2 + 3 + 10 - 8, 10\rangle && (2.6) \end{aligned}$$

2.3 Solution Naive

Algorithm 4 Naive

Require: $m \in \mathbb{N}, n \in \mathbb{N}, v \in \mathbb{R}^n, Q = \{\langle i, j\rangle : i, j < n\}^m$

1: $a = zeros(m)$
2: **for** $k \in \{0, \ldots, m-1\}$ **do**
3: $\quad \langle i, j\rangle \leftarrow Q[k]$
4: $\quad a[k] \leftarrow sum(\langle v[i], \ldots, v[j]\rangle)$
5: **return** a

2.4 Solution Optimized

Notice that:

$$a(\langle i, j\rangle) = \begin{cases} a(\langle 0, j\rangle) - a(\langle 0, i-1\rangle) & , \text{ if } i > 0 \\ a(\langle 0, j\rangle) & , \text{ if } j = 0 \end{cases} \tag{2.7}$$

The algorithm is then: compute all values $a(\langle 0, j\rangle), \forall j \in \{0, \ldots, n-1\}$ and then answer all queries using the formula above.

Algorithm 5 Opt

Require: $m \in \mathbb{N}, n \in \mathbb{N}, v \in \mathbb{R}^n, Q = \{\langle i, j\rangle : i, j < n\}^m$

1: $\Delta s \leftarrow \text{zeros}(n+1)$
2: **for** $i \in \{0, \ldots, n-1\}$ **do**
3: $\quad \Delta s[i+1] \leftarrow \Delta s[i] + v[i]$
4: $a = zeros(m)$
5: **for** $k \in \{0, \ldots, m-1\}$ **do**
6: $\quad \langle i, j\rangle \leftarrow Q[k]$
7: $\quad a[k] = \Delta s[j+1] - \Delta s[i]$
8: **return** a

Chapter 3

Longest Increasing Subsequence

3.1 Basic Definitions

Definition 13 (Sequence)**.** A **Sequence** is a function f from the subset $I \subseteq \mathbb{N}$ of the Natural Numbers into a Codomain Cd:

$$f : I \to Cd \tag{3.1}$$

Denote by $\mathcal{S}(I, Cd)$ the set of all sequences of I into Cd:

$$\mathcal{S}(I, Cd) = \{f : I \to Cd\} \tag{3.2}$$

Definition 14 (Successor)**.** Let $I \subseteq \mathbb{N}$ be a set. The successor is a bijective function:

$$\sigma_I : I \backslash \max(I) \to I \backslash \min(I) \tag{3.3}$$

$$i \mapsto \min\{j \in I : j > i\} \tag{3.4}$$

Definition 15 (Increasing Sequence)**.** Given sets $I \subseteq \mathbb{N}$ and Cd, a sequence $f \in \mathcal{S}(I, Cd)$ is said to be **increasing** when the values of the sequence increase, i.e.:

$$f \text{ is an increasing sequence} \quad \leftrightarrow \quad \forall i\,(i \in I \backslash \max(I) \to f(i) \leqslant f(\sigma_I(i))) \tag{3.5}$$

Definition 16 (Length of a Sequence)**.** Given sets $I \subseteq \mathbb{N}$ and Cd, and a sequence $f \in \mathcal{S}(I, Cd)$, the length of the sequence f, denoted by $\mathfrak{L}(f)$ is the cardinality (number of elements) of its domain I:

$$\mathfrak{L}(f) = |I| \tag{3.6}$$

Definition 17 (Subsequence)**.** Let $f \in \mathcal{S}(I, Cd)$ be a sequence from $I \subseteq \mathbb{N}$ into a Codomain Cd. A sequence $g \in \mathcal{S}(I', Cd')$ is called a subsequence of f, and denoted by $g \preceq f$, when $I' \subseteq I$ and $Cd' \subseteq Cd$:

$$\forall f\,(f \in \mathcal{S}(I, Cd) \to \forall g\,(g \in \mathcal{S}(I', Cd') \to (g \preceq f \leftrightarrow (I' \subseteq I \wedge Cd' \subseteq Cd)))) \tag{3.7}$$

Moreover, denote by:

1. $\mathfrak{s}(f)$ the set of all subsequences of the sequence f;
2. $\mathfrak{d}(g) \subseteq I$ the domain of a subsequence $g \preceq f$;

Theorem 3. *Every sequence of n elements can be indexed using $\{0, \ldots, n-1\}$.*

Write this theorem appropriately and prove it.

3.1.1 Examples

We will write the sequence using ordered pairs. For example, a sequence v with $I = \{0, 1, 3, 10\}$ and $Cd = \mathbb{R}$ could be:

$$v = \{\langle 0, 4.7\rangle, \langle 1, -8.8\rangle, \langle 3, -5.4\rangle, \langle 10, 2.3\rangle\} \tag{3.8}$$

The elements of the sequence will be written using common function notation:

$$\begin{aligned} v(0) &= 4.7 \\ v(1) &= -8.8 \\ v(3) &= -5.4 \\ v(10) &= 2.3 \end{aligned}$$

In general, we are interested in sequences of n elements for which $I = \{0, \ldots, n-1\}$. In such cases, we will write the sequence simply as a tuple:

$$\begin{aligned} v &= \{\langle 0, 4.7\rangle, \langle 1, -8.8\rangle, \langle 2, -5.4\rangle, \langle 3, 2.3\rangle\} \\ &\equiv \langle 4.7, -8.8, -5.4, 2.3\rangle \end{aligned} \tag{3.9}$$

3.2 Problem Definition

3.2.1 Input

1. a natural number $n \in \mathbb{N}$;

2. A sequence $v \in \mathcal{S}(\{0, \ldots, n-1\}, \mathbb{R})$[1]

3.2.2 Output

Define $\mathcal{I}(v) = \{s \in \mathfrak{s}(v) : s \text{ is increasing}\}$ the set of all increasing subsequences of v. An output is any element $s \in \mathcal{I}(v)$.

3.2.3 Goal

Find the longest increasing subsequence of v:

$$s^* = \underset{s \in \mathcal{I}(v)}{\arg\max}\ \mathfrak{L}(s) \tag{3.10}$$

[1]Notice that here we are indexing using the naturals $\{0, \ldots, n-1\}$ and not a general $I \subseteq \mathbb{N}$. This is because of the Theorem 3.

3.3 Naive Algorithm

Algorithm 6 Naive

Require: $n \in \mathbb{N}, v \in \mathbb{R}^n$

1: $S \leftarrow$ generate_all_subsequences(v)
2: $S' \leftarrow$ filter(is_increasing_sequence, s)
3: $s^* \leftarrow \underset{s \in S'}{\arg\max}\ \mathfrak{L}(s)$
4: **return** s^*

3.4 Recursive Algorithm

Define:

1. $B(i)$: the set of indices lower than i for which the correspondent element in the sequence is smaller than $v(i)$. This is going to be used to compose the subproblems.

$$B(i, v) = \{j \in \mathfrak{d}(v) : j < i \ \wedge \ v(j) < v(i)\} \tag{3.11}$$

2. $L(i)$: the longest increasing subsequence ending at the index i, so that its last element is $v(i)$. It can be defined recursively by the Equation 3.12. In it, $\oplus$ is the concatenation of two sequences and when $B = \varnothing$, the output of the arg max is an empty sequence.

$$L(i, v) = \underset{j \in B(i,v)}{\arg\max}\ \mathfrak{L}(L(j)) \oplus \langle v(i) \rangle \tag{3.12}$$

The recursive and the dynamic programming algorithms differ in only one thing: the latter stores the intermediate values (memoization) so that no recomputation is made. The former recomputes the solutions of the subproblems every time they are needed.

Algorithm 7 Dynamic Programming / Recursive

Require: $n \in \mathbb{N}, v \in \mathbb{R}^n$

1: $S \leftarrow \langle L(i) \ : \ i \in \mathfrak{d}(v) \rangle$
2: $s^* \leftarrow \underset{s \in S}{\arg\max}\ \mathfrak{L}(s)$
3: **return** s^*

The complexity of the Recursive algorithm is exponential while the Dynamic Programming algorithm is $\mathcal{O}(n^2)$ (for each element of the sequence, in the worst case scenario it visits all the previous values, so it is quadratic).

Chapter 4

Domino Arrangements

4.1 Problem Definition

1. Input:

   (a) an natural number $n \in \mathbb{N}$;

2. Output:

   (a) a list of all the ways the dominos can be arranged in the $2 \times n$ grid;

4.2 Examples

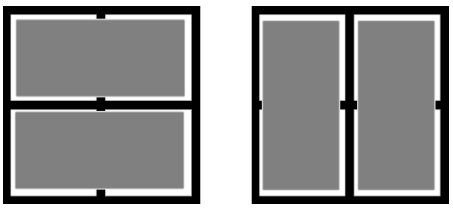


Figure 4.1: all arrangments in a 2×2 grid.

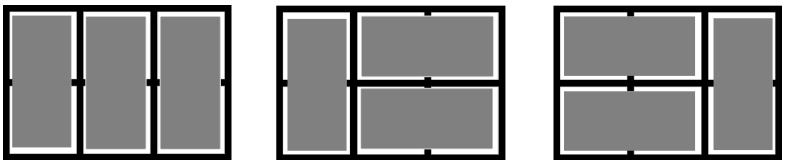


Figure 4.2: all arrangments in a 2×3 grid.

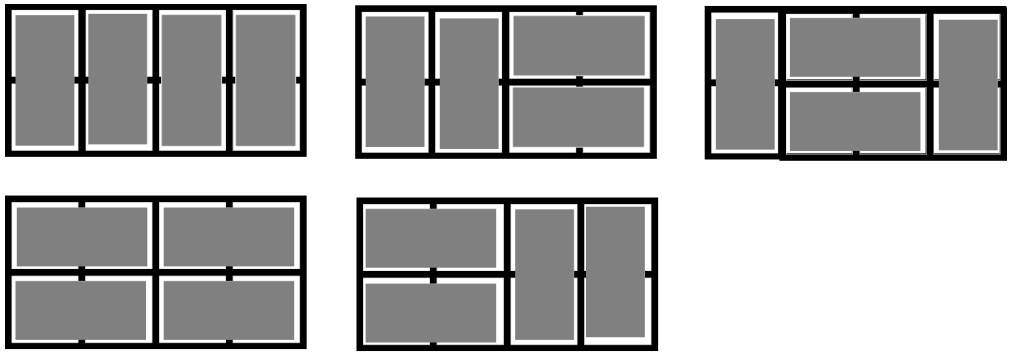


Figure 4.3: all arrangments in a 2×4 grid.

4.3 Algorithm

Given $n \in \mathbb{N}$, let $S(n)$ be the solution (all arrangements). $S(n)$ can be obtained from two recursive calls:

1. one vertical domino combined with $S(n-1)$;

2. two horizontal dominos combined with $S(n-2)$;

Such procedure is shown in the Figure

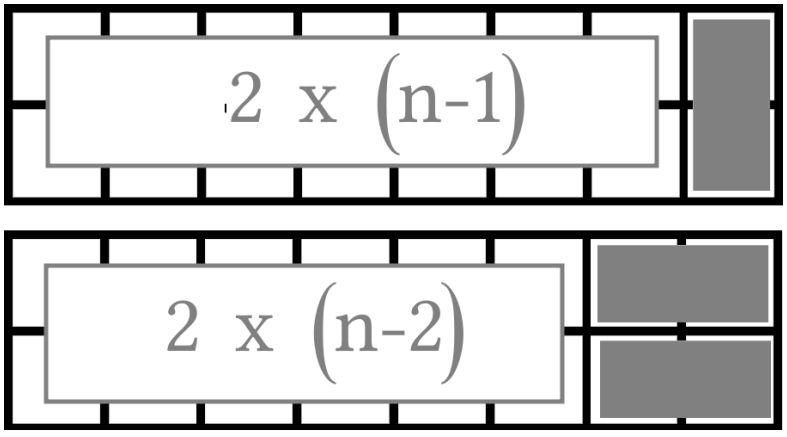


Figure 4.4: recursive solution for a $2 \times n$ grid.

4.4 Algorithm

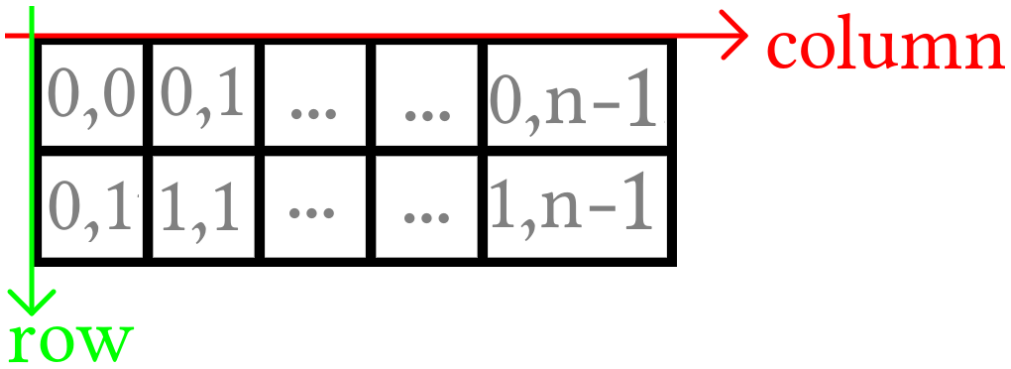


Figure 4.5: coordinate system for a $2 \times n$ grid.

Algorithm 8 domino_arrangements

Require: $n \in \mathbb{N}$

1: **if** $n == 1$ **then**

2: **return**

$$\left\langle \langle \textbf{Domino}\left[\langle 0,0\rangle, \textit{vertical}\right]\rangle \right\rangle$$

3: **else if** $n == 2$ **then**

4: **return**

$$\Big\langle \langle \textbf{Domino}\left[\langle 0,0\rangle, \textit{vertical}\right], \textbf{Domino}\left[\langle 0,1\rangle, \textit{vertical}\right]\rangle,$$
$$\langle \textbf{Domino}\left[\langle 0,0\rangle, \textit{vertical}\right], \textbf{Domino}\left[\langle 1,0\rangle, \textit{vertical}\right]\rangle \Big\rangle$$

5: **else**

6: **return**

$$\text{domino\_arrangements}(n-1) \oplus \langle \textbf{Domino}\left[\langle 0,0\rangle, \textit{vertical}\right]\rangle,$$
$$\text{domino\_arrangements}(n-2) \oplus \langle \textbf{Domino}\left[\langle 0,0\rangle, \textit{vertical}\right], \textbf{Domino}\left[\langle 1,0\rangle, \textit{vertical}\right]\rangle$$

▷ $\oplus$ is the concatenation of the list on the right with all the lists on the left.

4.5 Correctness of the Algorithm

Base Case 1

By inspection, one notices that for a $2 \times n$ grid, there are two possible arrangements. Therefore, the line 2 of the algorithm returns the optimal solution of the case $n = 1$.

Base Case 2

By inspection, one notices that for a $2 \times n$ grid, there are two possible arrangements. Therefore, the line 4 of the algorithm returns the optimal solution of the case $n = 2$.

Recursion

First of all, notice that, if a domino is in the horizontal position, then there must be a domino right over (or below) it. In other words, arrangements such as the one of the Figure 4.6.

Now, for the last domino, there is two possibilities: either it is in the vertical or the horizontal (and so there is another one below it) position. But for both cases, the problem of arranging the other dominos is the same problem but with a smaller grid:

1. the last domino is in the vertical position: solve the problem for the grid $2 \times (n-1)$
2. the last domino is in the horizontal position: solve the problem for the grid $2 \times (n-2)$

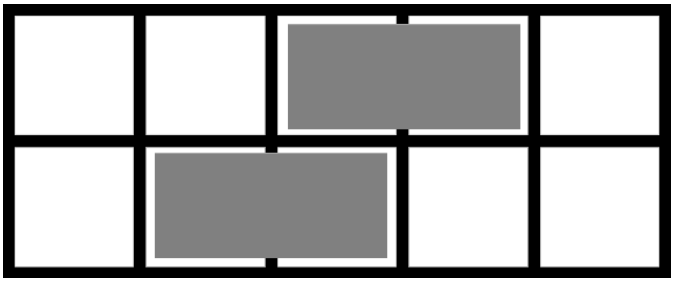


Figure 4.6: invalid configuration of dominos. A horizontal domino must have another right over or below it.

Chapter 5

Stairway to Heaven 2

5.1 Basic Definitions

Definition 18 (Step Domain)**.** Given $l \in \mathbb{N}_{>0}$, define the Step Domain $\mathbb{D}(l)$ as:

$$\mathbb{D}(l) = \{1, \ldots, l\} \tag{5.1}$$

Definition 19 (Sequence Set)**.** Given $n \in \mathbb{N}_{>0}$ and $l \in \mathbb{N}_{>0}$, the set $\mathcal{S}(n, l)$, known as Sequence Set, of all sequences that lead to heaven with steps in the Step Domain $\mathbb{D}(l)$, is defined as:

$$\mathcal{S}(n, l) = \left\{ x \in \mathbb{D}(l)^k : k \leqslant n \wedge \sum_{i=0}^{|x|-1} x(i) = n \right\} \tag{5.2}$$

where $|x|$ is the size of the sequence x.

Definition 20 (Sequence Cost)**.** Given $n \in \mathbb{N}_{>0}$, $l \in \mathbb{N}_{>0}$, $s \in \mathcal{S}(n, l)$, and $f \in \mathbb{R}^n$, the cost $\sigma_f(s)$ of the sequence s with the fees f is defined as:

$$\sigma_f(s) = f(0) + \sum_{i=1}^{|s|-1} f\left(\sum_{j=0}^{i-1} s(j) \right) \tag{5.3}$$

5.2 Problem Definition

1. Input:
   (a) the number of steps: a natural number $n \in \mathbb{N}_{>0}$;
   (b) the largest step: a natural number $l \in \mathbb{N}_{>0}$;
   (c) the fees of each step: a vector of values $f \in \mathbb{R}^n$;
2. Output: a sequence $s \in \mathcal{S}(n, l)$;
3. Goal: Minimize the cost of the sequence $\sigma_f(s)$;

5.3 Naive Algorithm

The algorithm simply generates all sequences and find the one with the lowest cost by checking everyone.

Algorithm 9 Naive

Require: $n \in \mathbb{N}_{>0}, l \in \mathbb{N}_{>0}, f \in \mathbb{R}^n$

1: $S \leftarrow \text{generate\_sequences}(n, l)$
2: $s^* \leftarrow \arg\min_{s \in S} \sigma_f(s)$
3: **return** s^*

5.4 Dynamic Programming Algorithm

5.4.1 Subproblem

Let $i \in \mathbb{N}_{>0} \wedge i \leqslant n$. Given that one has the optimal solution for any $j \in \mathbb{N}_{>0} \wedge j \leqslant i$, how does one find the optimal solution for when the heaven is at i? There are l[1] options to consider:

1: the optimal solution for $j = i - 1$ plus a step of size 1

2: the optimal solution for $j = i - 2$ plus a step of size 2

...

l**:** the optimal solution for $j = i - l$ plus a step of size l

The optimal solution for i is, among the l options above, the one with the lowest cost. Executing that procedure for all $i \leqslant n$, one gets a $\mathcal{O}(n)$ (linear complexity) algorithm.

5.4.2 Algorithm

Algorithm 10 DP

Require: $n \in \mathbb{N}_{>0}, l \in \mathbb{N}_{>0}, f \in \mathbb{R}^n$

1: $v \leftarrow \langle \text{Object}\{sequence : \langle\rangle, cost : f(0)\}\rangle$ ▷ v is a sequence of objects
2: **for** $i \in \{1, \ldots, n\}$ **do**
3: $C \leftarrow \text{get\_all\_options}(v, l)$
4: $c^* \leftarrow \arg\min_{c \in C} c.cost$
5: $v.append(c^*)$
6: **return** $v.back()$

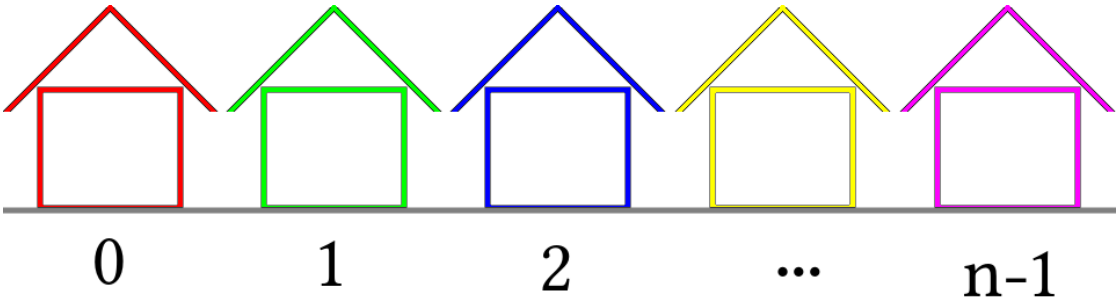
[1] A **constant** number of options.

Chapter 6

Burglar's Night Out

6.1 Problem Statement

A burglar has come to your neighborhood at night. Your neighborhood has many houses arranged in a linear fashion and each house has some amount of money. The burglar wishes to take as much money as possible. However, he cannot rob two consecutive houses otherwise the security bell will ring and he will get caught.



6.2 Basic Definitions

Definition 21 (Binary Sequence and Real Sequence)**.** A **Binary Sequence** is a sequence of values of the binary set $\mathbb{B} = \{True, False\}$. A **Real Sequence** is a sequence of values of the real set $\mathbb{R}$.

Definition 22 (Cost of a Sequence)**.** Given a Binary Sequence b and a real sequence r, both with the same length, the **Cost** $\sigma_r(b)$ **of the Sequence** b with values r is given by:

$$\sigma_r(b) = \sum_{i \in \{i \in [[n]] : b[i] = True\}} r[i] \tag{6.1}$$

Definition 23 (True Alternate Binary Sequence)**.** A Binary Sequence b is called **True Alternate Binary Sequence** or simply **True Alternate** if there are no two consecutive values in

which both are $True$. In other words:

$$\begin{aligned} &\forall i \in [[n]] \nexists j \in [[n]] \, (j = i + 1 \wedge i = True \wedge j = True) \quad \equiv \quad (6.2) \\ &\forall i \Big((i \in [[n]] \wedge i = True) \rightarrow \nexists j \, (j \in [[n]] \wedge j = i + 1 \wedge j = True) \Big) \end{aligned}$$

6.3 Problem Definition

1. Input:
   (a) the number of houses: a natural number $n \in \mathbb{N}$;
   (b) the cost sequence: a Real Sequence r of length n;
2. Output: a True Alternate Binary Sequence b of length n;
3. Goal: Maximize the cost of the sequence $\sigma_r \, (b)$;

6.4 Naive Algorithm

Algorithm 11 Naive

Require: $n \in \mathbb{N}, r \in \mathbb{R}^n$

$S \leftarrow$ generate_all_binary_sequences(n)
$S' \leftarrow$ filter_true_alternate_sequences(S)
$s^* \leftarrow \arg\max_{s \in S'} \sigma_r \, (s)$
return s^*

6.5 Dynamic Programming

6.5.1 Initialization

Initialize the problem with two optimal solutions for the problem of size 1:

1. $\langle true \rangle$ (i.e. the optimal solution which uses the value $r[0]$);
2. $\langle false \rangle$ (i.e. the optimal solution which does not use the value $r[0]$);

6.5.2 Optimal Subproblem Structure

Suppose one has two optimal solutions for the subproblem of size $i < n$:

1. the optimal solution which uses the value $r[i]$ (i.e. $b[i] = True$);
2. the optimal solution which does not use the value $r[i]$ (i.e. $b[i] = False$);

The solution for the subproblem of size $i + 1$ is the best one among the following:

1. do not use $r[i]$; use $r[i + 1]$;
2. use $r[i]$, do not use $r[i + 1]$;

3. do not use $r[i]$; do not use $r[i+1]$[1];

1 is the best solution which uses $r[i+1]$. The best among 2 and 3 is the best solution which does not use $r[i+1]$.

Finally, the best solution for the problem of size n (the original problem) is the best between the two: the one which uses $r[n-1]$ and the one which doesn't use it.

6.5.3 Algorithm Description

Algorithm 12 data structure *SubproblemSolutions*.

```
object SubproblemSolutions {
  lastIncluded: BinarySequence
  lastExcluded: BinarySequence
}
```

Algorithm 13 Dynamic Programming

Require: $n \in \mathbb{N}, r \in \mathbb{R}^n$

1: // Initialization
2: $S \leftarrow vector < SubproblemSolutions >$
3: $s \leftarrow SubproblemSolutions(lastIncluded = \langle true \rangle, lastExcluded = \langle false \rangle)$
4: $S.append(s)$
5: // Rescursive solver
6: **for** $i \in \{1, \ldots, n-1\}$ **do**
7: // last included
8: $lastIncluded \leftarrow S[i-1].lastExcluded$
9: $lastIncluded.append(True)$
10: // last excluded
11: $lastExcluded \leftarrow \arg\max_{s \in S[i-1]} \sigma_r(s)$
12: $lastExcluded.append(False)$
13: // next entry
14: $S.append(SubproblemSolutions(lastIncluded, lastExcluded))$
15: // Optimal solution of the original problem
16: $s^* \leftarrow \arg\max_{s \in S[n-1]} \sigma_r(s)$
17: **return** s^*

[1]This case does not have to be here if the cost sequence r has only non-negative values. But since one allowed negative costs, this case has to be considered.

Chapter 7

Number Splitting

7.1 Problem Definition

1. Input: a number $n \in \mathbb{N}$;

2. Output: a partition of natural values $P = \langle p_0, \ldots, p_m \rangle$ such that:

   (a) $m \geqslant 2$;

   (b) $\sum_{p \in P} p = n$;

3. Goal: Maximize the product of the elements of P:

$$\max \prod_{p \in P} p \tag{7.1}$$

7.2 Naive Algorithm

The naive algorithm is pretty simple: generate all partitions of the number n with at least two numbers and select the one with the largest product.

Below, the Algorithm 14 specify how to generate all partitions using a recursive algorithm, and the Algorithm 15 is the actual naive algorithm described above.

Algorithm 14 Generate all partitions

```
1: function PARTITION(n, excludeLast)
2:     P ← {}
3:     if n = 0 then
4:         P.insert(⟨⟩)
5:     else if n = 1 then
6:         P.insert(⟨1⟩)
7:     for i = 1, ..., n do
8:         for p ∈ Partition(n − i, false) do
9:             p+ ← sorted([i] + p)
10:            P.insert(p+)
11:    if excludeLast then
12:        P.remove_if(λ(p) p == ⟨n⟩)
13:    return P
```

Algorithm 15 Naive algorithm

```
1: function NAIVE(n)
2:     P ← Partition(n, true)
3:     p* = arg max_{p∈P} ∏_{i∈p} i
4:     return p*
```

7.3 Dynamic Programming Algorithm

Suppose one has the optimal solution $\forall k < n$. Then the optimal solution for n can be found by inspection on $\left\lfloor \frac{n}{2} \right\rfloor + 1$ options: $n, \langle 1, n-1 \rangle, \langle 2, n-2 \rangle, \ldots, \left\langle \left\lfloor \frac{n}{2} \right\rfloor, n - \left\lfloor \frac{n}{2} \right\rfloor \right\rangle$. Since one knows the optimal solution for all those values, the optimal for n is the one with the highest product.

The solution with one element only is not valid as a final solution. That is why the Algorithm 18 specify the $includeLast = false$.

Algorithm 16 data structure *SubproblemSolution*.

```
object SubproblemSolution {
    cost: Natural
    partition: Partition
    }
```

Algorithm 17 Dynamic Programming Recursive Solver

```
1: function DP_RECURSIVE(n, includeLast)
2:     if n = 1 then
3:         return SubproblemSolution {1, ⟨1⟩}
4:     else
5:         C ← {}                                              ▷ List of candidates
6:         if includeLast then
7:             C.add(SubproblemSolution {n, ⟨n⟩})
8:         for i = 1, ..., ⌊n/2⌋ do
9:             lhs ← DP_Recursive (i, true)
10:            rhs ← DP_Recursive (n − 1, true)
11:            C.add (SubproblemSolution {lhs.cost · rhs.cost, lhs.partition + rhs.partition})
12:        c* = arg max_{c∈C} c.cost                           ▷ Select the best candidate
13:        return c*
```

Algorithm 18 Dynamic Programming Solver

```
1: function DYNAMIC_PROGRAMMING(n, includeLast)
2:     return DP_Recursive (n, false) .partition
```