

---

# Foundations of Computer Science

*Release 0.4*

**Amir Kamil**

**Mar 30, 2024**

# CONTENTS

<b>I</b>	<b>Algorithms</b>	<b>1</b>
<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Text Objectives . . . . .	2
1.2	Tools for Abstraction . . . . .	3
1.3	The First Algorithm: Euclid's GCD . . . . .	4
<b>2</b>	<b>The Potential Method</b>	<b>7</b>
2.1	A Potential Function for Euclid's Algorithm . . . . .	10
<b>3</b>	<b>Divide and Conquer</b>	<b>13</b>
3.1	The Master Theorem . . . . .	14
3.2	Integer Multiplication . . . . .	16
3.3	The Closest-Pair Problem . . . . .	19
<b>4</b>	<b>Dynamic Programming</b>	<b>25</b>
4.1	Implementation Strategies . . . . .	25
4.2	Weighted Task Selection . . . . .	28
4.3	Longest Increasing Subsequence . . . . .	31
4.4	Longest Common Subsequence . . . . .	32
4.5	All-Pairs Shortest Paths . . . . .	37
<b>5</b>	<b>Greedy Algorithms</b>	<b>40</b>
<b>II</b>	<b>Computability</b>	<b>45</b>
<b>6</b>	<b>Introduction to Computability</b>	<b>46</b>
6.1	Formal Languages . . . . .	46
6.2	Overview of Automata . . . . .	49
<b>7</b>	<b>Finite Automata</b>	<b>52</b>
7.1	Formal Definition . . . . .	56
<b>8</b>	<b>Turing Machines</b>	<b>60</b>
8.1	The Language of a Turing Machine . . . . .	84
8.2	Decidable Languages . . . . .	86
8.3	Equivalent Models . . . . .	88
<b>9</b>	<b>Diagonalization</b>	<b>91</b>
9.1	Countable Sets . . . . .	91
9.2	Uncountable Sets . . . . .	94

9.3	The Existence of an Undecidable Language . . . . .	96
<b>10</b>	<b>“Natural” Undecidable Problems</b>	<b>99</b>
10.1	Code as Input . . . . .	99
10.2	The Barber Language . . . . .	100
10.3	The Acceptance Language and Simulation . . . . .	101
10.4	The Halting Problem . . . . .	103
<b>11</b>	<b>Turing Reductions</b>	<b>105</b>
11.1	The Halts-on-Empty Problem . . . . .	107
11.2	More Undecidable Languages and Turing Reductions . . . . .	109
11.3	Wang Tiling . . . . .	112
<b>12</b>	<b>Recognizability</b>	<b>122</b>
12.1	Unrecognizable Languages . . . . .	125
12.2	Dovetailing . . . . .	126
<b>13</b>	<b>Rice’s Theorem</b>	<b>129</b>
13.1	Rice’s Theorem and Program Analysis . . . . .	132
<b>III</b>	<b>Complexity</b>	<b>135</b>
<b>14</b>	<b>Introduction to Complexity</b>	<b>136</b>
14.1	Polynomial Time and the Class P . . . . .	137
14.2	Examples of Efficient Verification . . . . .	139
14.3	Efficient Verifiers and the Class NP . . . . .	142
14.4	P Versus NP . . . . .	146
<b>15</b>	<b>Satisfiability and the Cook-Levin Theorem</b>	<b>148</b>
15.1	Configurations and Tableaus . . . . .	150
15.2	Constructing the Formula . . . . .	152
15.3	Conclusion . . . . .	158
<b>16</b>	<b>NP-Completeness</b>	<b>161</b>
16.1	Polynomial-Time Mapping Reductions . . . . .	161
16.2	NP-Hardness and NP-Completeness . . . . .	163
16.3	Resolving P versus NP . . . . .	164
<b>17</b>	<b>More NP-Complete Problems</b>	<b>168</b>
17.1	3SAT . . . . .	168
17.2	Clique . . . . .	172
17.3	Vertex Cover . . . . .	175
17.4	Set Cover . . . . .	179
17.5	Hamiltonian Cycle . . . . .	181
17.6	Subset Sum . . . . .	183
17.7	Concluding Remarks . . . . .	186
<b>18</b>	<b>Search and Approximation</b>	<b>187</b>
18.1	Approximation for Minimum Vertex Cover . . . . .	190
18.2	Maximum Cut . . . . .	193
18.3	Knapsack . . . . .	196
18.4	Other Approaches to NP-Hard Problems . . . . .	198

<b>IV</b>	<b>Randomness</b>	<b>199</b>
<b>19</b>	<b>Randomized Algorithms</b>	<b>200</b>
19.1	Review of Probability . . . . .	201
19.2	Randomized Approximations . . . . .	205
19.3	Quick Sort . . . . .	210
19.4	Skip Lists . . . . .	215
<b>20</b>	<b>Monte Carlo Methods and Concentration Bounds</b>	<b>220</b>
20.1	Variance and Chebyshev's Inequality . . . . .	221
20.2	Hoeffding's Inequality . . . . .	226
20.3	Polling . . . . .	229
20.4	Load Balancing . . . . .	232
<b>V</b>	<b>Cryptography</b>	<b>235</b>
<b>21</b>	<b>Introduction to Cryptography</b>	<b>236</b>
21.1	Review of Modular Arithmetic . . . . .	237
21.2	One-time Pad . . . . .	242
<b>22</b>	<b>Diffie-Hellman Key Exchange</b>	<b>247</b>
<b>23</b>	<b>RSA</b>	<b>250</b>
23.1	RSA Signatures . . . . .	254
23.2	Quantum Computers and Cryptography . . . . .	255
<b>VI</b>	<b>Supplemental Material</b>	<b>256</b>
<b>24</b>	<b>Supplemental: Algorithms</b>	<b>257</b>
24.1	Non-master-theorem Recurrences . . . . .	257
<b>25</b>	<b>Supplemental: Computability</b>	<b>259</b>
25.1	Applying Rice's Theorem . . . . .	259
25.2	Computable Functions and Kolmogorov Complexity . . . . .	261
<b>26</b>	<b>Supplemental: Randomness</b>	<b>264</b>
26.1	Primality Testing . . . . .	264
26.2	Multiplicative Chernoff Bounds . . . . .	268
26.3	Probabilistic Complexity Classes . . . . .	276
26.4	Amplification for Two-Sided-Error Algorithms . . . . .	281
<b>VII</b>	<b>Appendix</b>	<b>284</b>
<b>27</b>	<b>Appendix</b>	<b>285</b>
27.1	Proof of the Master Theorem . . . . .	285
27.2	Alternative Analysis of Quick Sort . . . . .	290
27.3	Proof of the Simplified Multiplicative Chernoff Bounds . . . . .	294
27.4	Proof of the Upper-Tail Hoeffding's Inequality . . . . .	296
27.5	General Case of Hoeffding's Inequality . . . . .	300

<b>VIII About</b>	<b>304</b>
<b>28 About</b>	<b>305</b>
<b>Index</b>	<b>306</b>

# **Part I**

# **Algorithms**

## INTRODUCTION

Every complex problem has a solution that is clear, simple, and wrong. — H. L. Mencken

Welcome to *Foundations of Computer Science*! This text covers foundational aspects of Computer Science that will help you reason about any computing task. In particular, we are concerned with the following with respect to problem solving:

- What are common, effective approaches to designing an algorithm?
- Given an algorithm, how do we reason about whether it is correct and how efficient it is?
- Are there limits to what problems we can solve with computers, and how do we identify whether a particular problem is solvable?
- What problems are *efficiently* solvable, and how do we determine whether a particular problem is?
- For problems that seem not to be solvable efficiently, can we efficiently find *approximate* solutions, and what are common techniques for doing so?
- Can randomness help us in solving problems?
- How can we exploit problems that are not efficiently solvable to build secure cryptography algorithms?

In order to answer these questions, we must define formal mathematical models and apply a proof-based methodology. Thus, this text will feel much like a math text, but we apply the approach directly to widely applicable problems in Computer Science.

As an example, how can we demonstrate that there is no general algorithm for determining whether or not two programs have the same functionality? A simple but incorrect approach would be to analyze all possible algorithms, and show that none can work. However, there are infinitely many possible algorithms, so we have no hope of this approach working. Instead, we need to construct a model that captures the notion of what is computable by any algorithm, and use that to demonstrate that no such algorithm exists.

### 1.1 Text Objectives

The main purpose of this text is to give you the tools to approach computational problems you've never seen before. Rather than being given a particular algorithm or data structure to implement, approaching a new problem requires reasoning about whether the problem is solvable, how to relate it to problems that you've seen before, what algorithmic techniques are applicable, whether the algorithm you come up with is correct, and how efficient the resulting algorithm is. These are all steps that must be taken *prior* to actually writing code to implement a solution, and these steps are independent of your choice of programming language or framework.

Thus, in a sense, the fact that you will not have to write code (though you are free to do so if you like) is a feature, not a bug. We focus on the prerequisite algorithmic reasoning required before writing any code, and this reasoning is independent of the implementation details. If you were to implement all the algorithms you design in this course, the workload would be far greater, and it would only replicate the coding practice you get in your programming courses.

Instead, we focus on the aspects of problem solving that you have not yet had much experience in. The training we give you in this text will make you a better programmer, as algorithmic design and analysis is crucial to effective programming. This text also provides a solid framework for further exploration of theoretical Computer Science, should you wish to pursue that path. However, you will find the material here useful regardless of which subfields of Computer Science you decide to study.

As an example, suppose your boss tells you that you need to make a business trip to visit several cities, and you must minimize the cost of visiting all those cities. This is an example of the classic [traveling salesperson problem](#)<sup>1</sup>, and you may have heard that it is an intractable problem. What exactly does that mean? Does it mean that it cannot be solved at all? What if we change the problem so that you don't have to minimize the cost, but instead must fit the cost within a given budget (say \$2000)? Does this make the problem any easier? What if we don't require that the total cost be minimized, but that it merely needs to be within a factor of two of the optimal cost?

Before we can reason about the total cost of a trip, we need to know how much it costs to travel between two consecutive destinations in the trip. Suppose we are traveling by air. There may be no direct flight between those two cities, or it might be very expensive. To help us keep our budget down, we need to figure out what the cheapest itinerary between those two cities is, considering intermediate layover stops. And since we don't know a priori in what order we will visit all the cities, we need to know the cheapest itineraries between all pairs of cities. This is an instance of the [all-pairs shortest path problem](#)<sup>2</sup>. Is this an “efficiently solvable” problem, and if so, what algorithmic techniques can we use to find a solution?

We will consider both of the problems above in this text. We will learn what it means for a problem to be solvable or not (and how to prove this), what it means for a problem to be tractable or not (and how to prove that it is, or give strong evidence that it is not), and techniques for designing and analyzing algorithms for tractable problems.

## 1.2 Tools for Abstraction

*Abstraction* is a core principle in Computer Science, allowing us to reason about and use complex systems without needing to pay attention to implementation details. As mentioned earlier, the focus of this text is on reasoning about problems and algorithms independently of specific implementations. To do so, we need appropriate abstract models that are applicable to any programming language or system architecture.

Our abstract model for expressing algorithms is *pseudocode*, which describes the steps in the algorithm at a high level without implementation details. As an example, the following is a pseudocode description of the [Floyd-Warshall algorithm](#) (page 37), which we will discuss later:

### Algorithm 1 (Floyd-Warshall)

**Input:** a weighted directed graph

**Output:** all-pairs (shortest-path) distances in the graph

**function** FLOYDWARSHALL( $G = (V, E)$ )

**for all**  $u, v \in V$  **do**

$d_0(u, v) = \text{weight}(u, v)$

**for**  $k = 1$  to  $|V|$  **do**

**for all**  $u, v \in V$  **do**

$d_k(u, v) = \min\{d_{k-1}(u, v), d_{k-1}(u, k) + d_{k-1}(k, v)\}$

**return**  $d_{|V|}$

This description is independent of how the graph  $G$  is represented, or the two-dimensional matrices  $d_i$ , or the specific syntax of the loops. Yet it should be clear to the intended reader what each step of the algorithm does. Expressing this algorithm in a real-world programming language like C++ would only add unnecessary syntactic and implementation

<sup>1</sup> [https://en.wikipedia.org/wiki/Travelling\\_salesman\\_problem](https://en.wikipedia.org/wiki/Travelling_salesman_problem)

<sup>2</sup> [https://en.wikipedia.org/wiki/Shortest\\_path\\_problem#All-pairs\\_shortest\\_paths](https://en.wikipedia.org/wiki/Shortest_path_problem#All-pairs_shortest_paths)



details that are an artifact of the chosen language and data structures, and not intrinsic to the algorithm itself. Instead, pseudocode gives us a simple means of expressing an algorithm that facilitates understanding and reasoning about its core elements.

We also need abstractions for reasoning about the efficiency of algorithms. The actual real-world running time and space/memory usage of a program depend on many factors, including choice of language and data structures, available compiler optimizations, and characteristics of the underlying hardware. Again, these are not intrinsic to an algorithm itself. Instead, we focus not on concrete real-world efficiency, but on how the algorithm's running time (and sometime memory usage) *scales* with respect to the input size. Specifically, we focus on time complexity in terms of:

1. the number of basic operations performed as a function of input size,
2. asymptotically, i.e., as the input size grows,
3. ignoring leading constants,
4. over worst-case inputs.

We measure space complexity in a similar manner, but with respect to the number of memory cells used, rather than the number of basic operations performed. These measures of time and space complexity allow us to evaluate algorithms in the abstract, rather than with respect to a particular implementation. This is not to say that absolute performance is irrelevant. Rather, the asymptotic complexity gives us a “higher-level” view of an algorithm. An algorithm with poor asymptotic time complexity will be inefficient when run on large inputs, regardless of how good the implementation is.

Later in the text, we will also reason about the intrinsic solvability of a problem. We will see how to express problems in the abstract (e.g. as *languages* and *decision problems*), and we will examine a simple model of computation (*Turing machines*) that captures the essence of computation.

## 1.3 The First Algorithm: Euclid's GCD

One of the oldest known algorithms is Euclid's algorithm for computing the *greatest common divisor* (GCD) of two integers.

**Definition 2 (Divides, Divisor)** Let  $x \in \mathbb{Z}$  be an integer. We say that an integer  $d$  *divides*  $x$  (and is a *divisor* of  $x$ ) if there exists an integer  $k \in \mathbb{Z}$  such that  $d \cdot k = x$ . (When  $d \neq 0$ , this is equivalent to  $x/d$  being an integer.)

Whether  $d$  divides  $x$  is not affected by their signs (positive, negative, or zero), so from now on we restrict our attention to nonnegative integers and divisors.

Note:  $d = 1$  divides *any* integer  $x$ , by taking  $k = x$  (i.e.,  $1 \cdot x = x$ ), and  $d = x$  is the largest divisor of  $x$  when  $x > 0$ . Take care with the special cases involving zero: *any* integer  $d$  divides  $x = 0$ , because  $d \cdot 0 = 0$ . But  $d = 0$  does not divide anything except  $x = 0$ , because  $d \cdot k = 0 \cdot k = 0$  for every  $k$ .

**Definition 3 (Greatest Common Divisor)** Let  $x, y \in \mathbb{Z}$  be nonnegative integers. A *common divisor* of  $x, y$  is an integer that divides both of them, and their *greatest common divisor*, denoted  $\gcd(x, y)$ , is the largest such integer.

For example,  $\gcd(21, 9) = 3$  and  $\gcd(121, 5) = 1$ . Also,  $\gcd(7, 7) = \gcd(7, 0) = 7$ . (Make sure you understand why!) If  $\gcd(x, y) = 1$ , we say that  $x$  and  $y$  are *coprime*. So, 121 and 5 are coprime, but 21 and 9 are not coprime (nor are 7 and 7, nor are 7 and 0).

Take note: as long as  $x, y$  are not both zero,  $\gcd(x, y)$  is well defined, because there is at least one common divisor  $d = 1$ , and no common divisor can be greater than  $\max(x, y)$ . However,  $\gcd(0, 0)$  is not well defined, because every integer divides zero, and there is no largest integer. In this case, it is convenient to define  $\gcd(0, 0) = 0$ , so that  $\gcd(x, 0) = x$  for all  $x$ .

So far we have just defined the GCD mathematically. Now we consider the computational question: given two integers, can we *compute* their GCD, and how efficiently can we do so? As we will see later, this problem turns out to be very important in cryptography and other fields.

Here is a naïve, “brute-force” algorithm for computing the GCD of given integers  $x \geq y$  (we adopt this requirement for convenience, since we can swap the values without changing the answer): try every integer from  $y$  down to 1, check whether it divides both  $x$  and  $y$ , and return the first (and hence largest) such number that does. The algorithm is clearly correct, because the GCD of  $x$  and  $y$  cannot exceed  $y$ , and the algorithm returns the first (and hence largest) value that actually divides both arguments.

**Algorithm 4 (Naïve GCD)**

**Input:** integers  $x \geq y \geq 0$ , not both zero

**Output:** their greatest common divisor  $\text{gcd}(x, y)$

**function** NAIVEGCD( $x, y$ )

**for**  $d = y$  down to 1 **do**

**if**  $d$  divides both  $x$  and  $y$  **then return**  $d$

Here, the mod operation computes the remainder of the first operand divided by the second. For example,  $9 \bmod 6 = 3$  since  $9 = 6 \cdot 1 + 3$ , and  $9 \bmod 3 = 0$  since  $9 = 3 \cdot 3 + 0$ . So, the result of  $x \bmod d$  is an integer in the range  $[0, d - 1]$ , and in particular  $x \bmod 1 = 0$ . The result is not defined when  $d$  is 0.

How efficient is the above algorithm? In the worst case, it performs two mod operations for every integer in the range  $[1, y]$ . Using asymptotic notation, the worst-case number of mod operations is therefore  $\Theta(y)$ . (Recall that this  $\Theta(y)$  notation means: between  $cy$  and  $c'y$  for some positive constants  $c, c'$ , for all “large enough”  $y$ .)

Can we do better?

Here is a key observation: if  $d$  divides both  $x$  and  $y$ , then it also divides  $x - my$  for any integer  $m \in \mathbb{Z}$ . Here is the proof: since  $x = d \cdot a$  and  $y = d \cdot b$  for some integers  $a, b \in \mathbb{Z}$ , then  $x - my = da - mdb = d \cdot (a - mb)$ , so  $d$  divides  $x - my$  as well. By the same kind of reasoning, the converse holds too: if some  $d'$  divides both  $x - my$  and  $y$ , it also divides  $x$ . Thus, the common divisors of  $x$  and  $y$  are exactly the common divisors of  $x - my$  and  $y$ , and hence the *greatest* common divisors of these two pairs are equal. We have just proved the following:

**Lemma 5** For all  $x, y, m \in \mathbb{Z}$ , we have  $\text{gcd}(x, y) = \text{gcd}(x - my, y)$ .

Since any  $m \in \mathbb{Z}$  will do, let’s choose  $m$  to minimize  $x - my$ , without making it negative. As long as  $y \neq 0$ , we can do so by taking  $m = \lfloor x/y \rfloor$ , the integer quotient of  $x$  and  $y$ . Then  $r = x - my = x - \lfloor x/y \rfloor y$  is simply the remainder of  $x$  when divided by  $y$ , i.e.,  $r = x \bmod y$ .

The above results in the following corollary (where the second equality holds because the GCD is symmetric):

**Corollary 6** For all  $x, y \in \mathbb{Z}$  with  $y \neq 0$ , we have  $\text{gcd}(x, y) = \text{gcd}(x \bmod y, y) = \text{gcd}(y, x \bmod y)$ .

(The rightmost expression maintains our convention that the first argument of  $\text{gcd}$  should be greater than or equal to the second.)

We have just derived the key *recurrence relation* for  $\text{gcd}$ , which we will use as the heart of an algorithm. However, we also need base cases. As mentioned previously,  $x \bmod y$  is defined only when  $y \neq 0$ , so we need a base case for  $y = 0$ . As we have already seen,  $\text{gcd}(x, 0) = x$  (even when  $x = 0$ ). (We can also observe that  $\text{gcd}(x, 1) = 1$  for all  $x$ . This latter base case is not technically necessary; some descriptions of Euclid’s algorithm include it, while others do not.)

This leads us to the Euclidean algorithm:

**Algorithm 7 (Euclid’s Algorithm)**

**Input:** integers  $x \geq y \geq 0$ , not both zero

**Output:** their greatest common divisor  $\text{gcd}(x, y)$

**function** EUCLID( $x, y$ )

**if**  $y = 0$  **then return**  $x$

**return** EUCLID( $y, x \bmod y$ )



Here are some example runs of this algorithm:

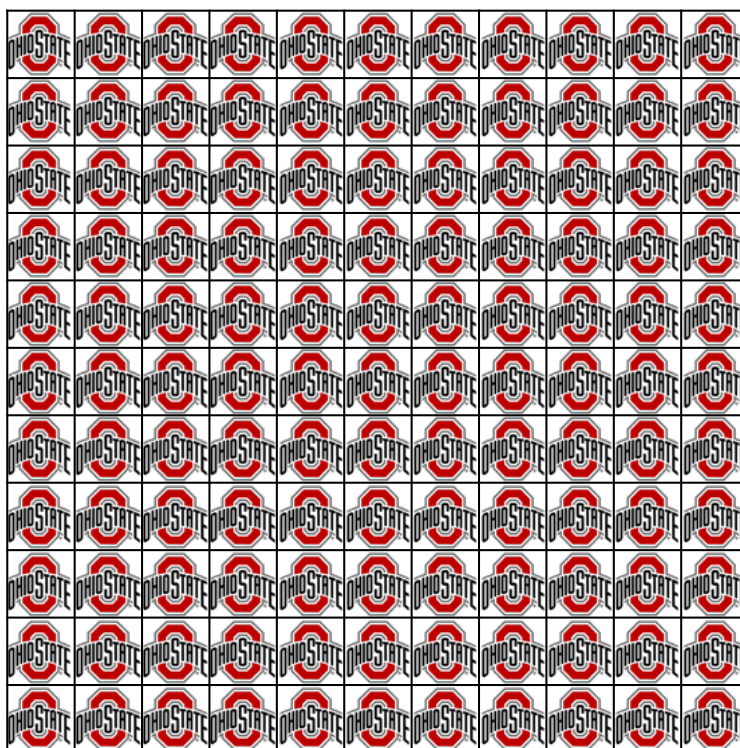
**Example 8**
$$\begin{aligned} & \text{EUCLID}(21, 9) \\ &= \text{EUCLID}(9, 3) \\ &= \text{EUCLID}(3, 0) \\ &= 3 \end{aligned}$$
**Example 9**
$$\begin{aligned} & \text{EUCLID}(30, 19) \\ &= \text{EUCLID}(19, 11) \\ &= \text{EUCLID}(11, 8) \\ &= \text{EUCLID}(8, 3) \\ &= \text{EUCLID}(3, 2) \\ &= \text{EUCLID}(2, 1) \\ &= \text{EUCLID}(1, 0) \\ &= 1 \end{aligned}$$
**Example 10**
$$\begin{aligned} & \text{EUCLID}(376281, 376280) \\ &= \text{EUCLID}(376280, 1) \\ &= \text{EUCLID}(1, 0) \\ &= 1 \end{aligned}$$

How efficient is this algorithm? Clearly, it does one mod operation per iteration—or more accurately, recursive call—but it is no longer obvious how many iterations it performs. For instance, the computation of  $\text{EUCLID}(30, 19)$  does six iterations, while  $\text{EUCLID}(376281, 376280)$  does only two. There doesn't seem to be an obvious relationship between the form of the input and the number of iterations.

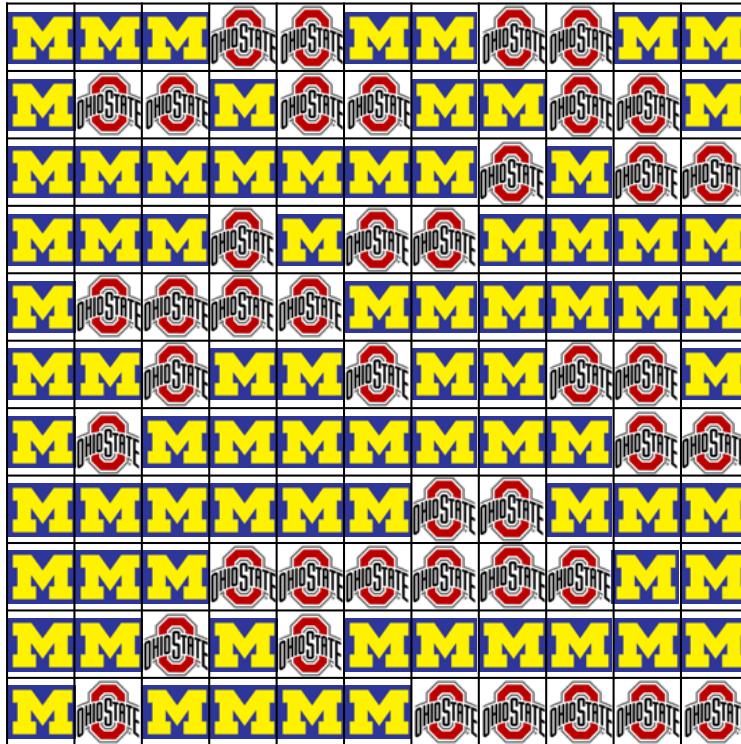
This is an extremely simple algorithm, consisting of just a few lines of code. But that does not make it simple to reason about. We need new techniques to analyze code like this and determine its time complexity.

## THE POTENTIAL METHOD

Let's set aside Euclid's algorithm for the moment and examine a game instead. Consider a "flipping game" that has an  $11 \times 11$  board covered with two-sided chips, say  on the front side and  on the back. Initially, the entirety of the board is covered with every chip face down.



The game pits a row player against a column player, and both take turns flipping an entire row or column, respectively. A row or column may be flipped only if it contains more face-down (OSU) than face-up (UM) chips. The game ends when a player can make no legal moves, and that player loses the game. For example, if the game reaches the following state and it is the column player's move, the column player has no possible moves and therefore loses.



Let's set aside strategy and ask a simpler question: must the game end in a finite number of moves, or is there a way to play the game in a way that continues indefinitely?

An  $11 \times 11$  board is rather large to reason about, so a good strategy is to simplify the problem by considering a  $3 \times 3$  board instead. Let's consider the following game state.



Suppose that it is the row player's turn, and the player chooses to flip the bottom row.



Notice that in general, a move may flip some chips from UM to OSU, and others vice versa. This move in particular flipped the first chip in the row from UM to OSU, and the latter two chips from OSU to UM. The number of chips flipped in each direction depends on the state of a row or column, and it is not generally the case that every move flips three OSU chips to UM in the  $3 \times 3$  board.

Continuing the game, the column player only has a single move available.



Again, one UM chip is flipped to OSU, and two OSU chips are flipped to UM.

It is again the row player's turn, but now no row flips are possible. The game ends, with the victory going to the column player.

We observe that each move flipped both UM and OSU chips, but each move flipped *more* OSU chips to UM than vice versa. Is this always the case? Indeed it is, by rule: a move is legal only if the flipped row or column has more OSU than UM chips. The move flips each OSU chip to a UM one, and each UM chip to an OSU chip. Since there are more OSU than UM chips, more OSU-to-UM flips happen than vice versa. More formally and generally, for an  $n \times n$  board, an individual row or column has  $k$  OSU chips and  $n - k$  UM chips, for some value of  $k$ . A move is legal when  $k > n - k$ . After the flip, the row or column will have  $k$  UM chips and  $n - k$  OSU chips. The net change in the number of UM chips is  $k - (n - k)$ , which is positive because  $k > n - k$ . Thus, each move strictly increases the number of UM chips, and strictly decreases the number of OSU chips.

In the  $3 \times 3$  case, we start with nine OSU chips. No board configuration can have fewer than zero OSU chips. Then because each move decreases the number of OSU chips, no more than nine moves are possible before the game must end. By the same reasoning, no more than 121 moves are possible in the original  $11 \times 11$  game.

Strictly speaking, it will always take fewer than 121 moves to reach a configuration where a player has no moves available, because the first move decreases the number of OSU chips by eleven. But we don't need an exact number to answer our original question. We have proved an upper bound of 121 on the number of moves, so we have established that any valid game must indeed end after a finite number of moves.

The core of the above reasoning is that we defined a special measure of the board's state, namely, the number of OSU chips. We observe that in each step, this measure must decrease (by at least one). The measure of the initial state is finite, and there is a lower bound the measure cannot go below, so eventually that lower bound must be reached.

This pattern of reasoning is called the *potential method*. Formally, given some set  $A$  of "states" (e.g., game states, algorithm states, etc.), let  $s: A \rightarrow \mathbb{R}$  be a function that maps states to numbers. The function  $s$  is a *potential function* if:

1. it strictly decreases with every state transition (e.g., turn in a game, step of an algorithm, etc.)<sup>3</sup>
2. it is lower-bounded by some fixed value: there is some  $\ell \in \mathbb{R}$  for which  $s(a) \geq \ell$  for all  $a \in A$ .

By defining a valid potential function and establishing both its lower bound and how quickly it decreases, we can upper bound the number of steps a complex algorithm may take.

<sup>3</sup> We need to be a bit more precise to ensure that  $s$  reaches its lower bound in a finite number of steps. A sufficient condition is that  $s$  decreases by at least some *fixed constant*  $c$  in each step. In the game example, we established that  $s$  decreases by at least  $c = 1$  in each turn.

## 2.1 A Potential Function for Euclid's Algorithm

Let's return to Euclid's algorithm and try to come up with a potential function we can use to reason about its efficiency. Specifically, we want to determine an upper bound on the number of iterations the algorithm takes for a given input  $x$  and  $y$ .

First observe that  $x$  and  $y$  do not increase from one step to the next. For instance,  $\text{EUCLID}(30, 19)$  calls  $\text{EUCLID}(19, 11)$ , so  $x$  decreases from 30 to 19 and  $y$  decreases from 19 to 11. However, the amount each argument individually decreases can vary. In  $\text{EUCLID}(376281, 376280)$ ,  $x$  decreases by only one in the next iteration, while  $y$  decreases by 376279. In  $\text{EUCLID}(7, 7)$ ,  $x$  does not decrease at all in the next iteration, but  $y$  decreases by 7.

Since the algorithm has two arguments, both of which typically change as the algorithm proceeds, it seems reasonable to define a potential function that takes both into account. Let  $x_i$  and  $y_i$  be the values of the two variables in the  $i$ th iteration (recursive call). So,  $x_0 = x$  and  $y_0 = y$ , where  $x$  and  $y$  are the original inputs to the algorithm;  $x_1, y_1$  are the arguments to the first recursive call; and so on. As a candidate potential function, we try a simple sum of the two arguments:

$$s_i = x_i + y_i .$$

Before we look at some examples, let's first establish that this is a valid potential function. Examining the algorithm, when  $y_i \neq 0$  we see that

$$\begin{aligned} s_{i+1} &= x_{i+1} + y_{i+1} \\ &= y_i + r_i , \end{aligned}$$

where  $r_i = x_i \bmod y_i$ . Given the invariant maintained by the algorithm that  $x_i \geq y_i$ , and that  $x_i \bmod y_i \in [0, y_i - 1]$ , we have that  $r_i < y_i \leq x_i$ . Therefore,

$$\begin{aligned} s_{i+1} &= y_i + r_i \\ &< y_i + x_i \\ &= s_i . \end{aligned}$$

Thus, the potential  $s$  always decreases by at least one (because  $s$  is a natural number) from one iteration to the next, satisfying the first requirement of a potential function. We also observe that  $y_i \geq 0$  for all  $i$ ; coupled with  $x_i \geq y_i$ , and the fact that both arguments are not zero, we get that  $s_i \geq 1$  for all  $i$ . Since we have established a lower bound on  $s$ , it meets the second requirement of a potential function.

At this point, we can conclude that Euclid's algorithm  $\text{EUCLID}(x, y)$  performs *at most*  $x + y$  iterations. (This is because the initial potential is  $x + y$ , each iteration decreases the potential by at least one, and the potential is always greater than zero.) However, this bound is no better than the one we derived for the brute-force GCD algorithm. So, have we just been wasting our time here?

Fortunately, we have not! As we will soon show, this  $x + y$  upper bound for  $\text{EUCLID}(x, y)$  is very *loose*, and in fact the actual number of iterations is much smaller. We will prove this by showing that the potential decreases *much faster* than we previously considered.

As an example, let's look at the values of the potential function for the execution of  $\text{EUCLID}(21, 9)$ :

$$\begin{aligned} s_0 &= 21 + 9 = 30 \\ s_1 &= 9 + 3 = 12 \\ s_2 &= 3 + 0 = 3 . \end{aligned}$$

And the following are the potential values for  $\text{EUCLID}(8, 5)$ :

$$\begin{aligned} s_0 &= 8 + 5 = 13 \\ s_1 &= 5 + 3 = 8 \\ s_2 &= 3 + 2 = 5 \\ s_3 &= 2 + 1 = 3 \\ s_4 &= 1 + 0 = 1 . \end{aligned}$$

The values decay rather quickly for  $\text{Euclid}(21, 9)$ , and somewhat more slowly for  $\text{Euclid}(8, 5)$ . But the key observation is that they appear to decay *multiplicatively* (by some factor), rather than additively. In these examples, the ratio of  $s_{i+1}/s_i$  is largest for  $s_2/s_1$  in  $\text{Euclid}(8, 5)$ , where it is 0.625. In fact, we will prove an upper bound that is not far from that value.

**Lemma 11** *For all valid inputs  $x, y$  to Euclid's algorithm,  $s_{i+1} \leq \frac{2}{3}s_i$  for every iteration  $i$  of  $\text{Euclid}(x, y)$ .*

The recursive case of  $\text{Euclid}(x_i, y_i)$  invokes  $\text{Euclid}(y_i, x_i \bmod y_i)$ , so  $x_{i+1} = y_i$  and  $y_{i+1} = r_i = x_i \bmod y_i$  (the remainder of dividing  $x_i$  by  $y_i$ ). By definition of remainder, we can express  $x_i$  as

$$x_i = q_i \cdot y_i + r_i,$$

where  $q_i = \lfloor x_i/y_i \rfloor$  is the integer quotient of  $x_i$  divided by  $y_i$ . Since  $x_i \geq y_i$ , we have that  $q_i \geq 1$ . Then:

$$\begin{aligned} s_i &= x_i + y_i \\ &= q_i \cdot y_i + r_i + y_i \text{ (substituting } x_i = q_i \cdot y_i + r_i) \\ &= (q_i + 1) \cdot y_i + r_i \\ &\geq 2y_i + r_i \text{ (since } q_i \geq 1). \end{aligned}$$

We are close to what we need to relate  $s_i$  to  $s_{i+1} = y_i + r_i$ , but we would like a common multiplier for both the  $y_i$  and  $r_i$  terms. Let's split the difference by adding  $r_i/2$  and subtracting  $y_i/2$ :

$$\begin{aligned} s_i &\geq 2y_i + r_i \\ &> 2y_i + r_i - \frac{y_i - r_i}{2} \text{ (since } r_i < y_i). \end{aligned}$$

The latter step holds because  $r_i$  is the remainder of dividing  $x_i$  by  $y_i$ , so  $y_i - r_i > 0$ . (And subtracting a positive number makes a quantity smaller.)

Continuing onward, we have:

$$\begin{aligned} s_i &\geq 2y_i + r_i - \frac{y_i - r_i}{2} \\ &= \frac{3}{2}(y_i + r_i) \\ &= \frac{3}{2}s_{i+1}. \end{aligned}$$

Rearranging the inequality, we conclude that  $s_{i+1} \leq \frac{2}{3}s_i$ .

By repeated applications of the above lemma (i.e., induction), starting with  $s_0 = x_0 + y_0 = x + y$ , we can conclude that:

**Corollary 12** *For all valid inputs  $x, y$  to Euclid's algorithm,  $s_i \leq (\frac{2}{3})^i(x + y)$  for all iterations  $i$  of  $\text{Euclid}(x, y)$ .*

We can now prove the following:

**Theorem 13 (Time Complexity of Euclid's Algorithm)** *For any valid inputs  $x, y$ ,  $\text{Euclid}(x, y)$  performs  $O(\log(x + y))$  iterations (and mod operations).*

We have previously shown that  $1 \leq s_i \leq (\frac{2}{3})^i(x + y)$  for the  $i$ th iteration of  $\text{Euclid}(x, y)$ . Therefore,

$$\begin{aligned} 1 &\leq \left(\frac{2}{3}\right)^i(x + y) \\ \left(\frac{3}{2}\right)^i &\leq x + y \\ i &\leq \log_{3/2}(x + y) \text{ (taking the base-}(3/2)\text{ log of both sides).} \end{aligned}$$



We have just established an upper bound on  $i$ , which means that the number of iterations cannot exceed  $\log_{3/2}(x+y) = O(\log(x+y))$ . Indeed, in order for  $i$  to exceed this quantity, it would have to be the case that  $1 > (2/3)^i(x+y)$ , leaving no possible value for the associated potential  $s_i$ —an impossibility! (Also recall that we can change the base of a logarithm by multiplying by a suitable constant, and since  $O$ -notation ignores constant factors, the base in  $O(\log(x+y))$  does not matter. Unless otherwise specified, the base of a logarithm is assumed to be 2 in this text.) Since each iteration does at most one mod operation, the total number of mod operations is also  $O(\log(x+y))$ , completing the proof.

Under our convention that  $x \geq y$ , we have that  $O(\log(x+y)) = O(\log 2x) = O(\log x)$ . Recall that the naïve algorithm did  $\Theta(x)$  iterations and mod operations (in the worst case). This means that Euclid's algorithm is *exponentially faster* than the naïve algorithm!

We have seen that the potential method gives us an important tool in reasoning about the complexity of algorithms, enabling us to establish an upper bound on the runtime of Euclid's algorithm.

## DIVIDE AND CONQUER

The *divide-and-conquer* algorithmic paradigm involves subdividing a large problem instance into smaller instances of the same problem. The subinstances are solved recursively, and then their solutions are combined in some appropriate way to construct a solution for the original larger instance.

Since divide and conquer is a recursive paradigm, the main tool for analyzing divide-and-conquer algorithms is induction. When it comes to complexity analysis, such algorithms generally give rise to recurrence relations expressing the time or space complexity. While these relations can be solved inductively, certain patterns are common enough that higher-level tools have been developed to handle them. We will see one such tool in the form of the master theorem.

As an example of a divide-and-conquer algorithm, the following is a description of the *merge sort* algorithm for sorting mutually comparable items:

**Algorithm 14 (Merge Sort)**

**Input:** an array of elements that can be ordered

**Output:** a sorted array of the same elements

```
function MERGESORT( $A[1, \dots, n]$ )  
  if  $n = 1$  then return  $A$   
   $m = \lfloor n/2 \rfloor$   
   $L = \text{MERGESORT}(A[1, \dots, m])$   
   $R = \text{MERGESORT}(A[m + 1, \dots, n])$   
  return MERGE( $L, R$ )
```

**Input:** two sorted arrays

**Output:** a sorted array of the same elements

```
function MERGE( $L[1, \dots, \ell], R[1, \dots, r]$ )  
  if  $\ell = 0$  then return  $R$   
  if  $r = 0$  then return  $L$   
  if  $L[1] \leq R[1]$  then  
    return  $L[1] : \text{MERGE}(L[2, \dots, \ell], R[1, \dots, r])$   
  else  
    return  $R[1] : \text{MERGE}(L[1, \dots, \ell], R[2, \dots, r])$ 
```

The algorithm sorts an array by first recursively sorting its two halves, then combining the sorted halves with the *merge* operation. Thus, it follows the pattern of a divide-and-conquer algorithm.

6	14	12	1	9	4	8	0	5	13	15	10	7	2	3	11
Sort each half recursively															
0	1	4	6	8	9	12	14	2	3	5	7	10	11	13	15
Merge															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

A naïve algorithm such as [insertion sort](#)<sup>4</sup> has a time complexity of  $\Theta(n^2)$ . How does merge sort compare?

Define  $T(n)$  to be the total number of basic operations (array indexes, element comparisons, etc.) performed by MERGESORT on an array of  $n$  elements. Similarly, let  $S(n)$  be the number of basic operations apart from the recursive calls themselves: testing whether  $n = 1$ , splitting the input array into halves, the cost of MERGE on the two halves, etc. Then we have the following recurrence for  $T(n)$ :

$$T(n) = 2T(n/2) + S(n).$$

This is because on an array of  $n$  elements, MERGESORT makes two recursive calls to itself on some array of  $n/2$  elements, each of which takes  $T(n/2)$  time by definition, and all its other non-recursive work takes  $S(n)$  by definition. (For simplicity, we ignore the floors and ceilings for  $n/2$ , which do not affect the ultimate asymptotic bounds.)

Observe that the merge step does  $n$  comparisons and  $\sim n$  array concatenations. Assuming that each of these operations takes a constant amount of time (that does not grow with  $n$ )<sup>5</sup>, we have that  $S(n) = O(n)$ . So,

$$T(n) = 2T(n/2) + O(n).$$

How can we solve this recurrence, i.e., express  $T(n)$  in a “closed form” that depends only on  $n$  (and does not refer to  $T$  itself)? While we can do so using induction or other tools for solving recurrence relations, this can be a lot of work. Thankfully, there is a special tool called the Master Theorem that directly yields a solution to this recurrence and many others like it.

## 3.1 The Master Theorem

Suppose we have some recursive divide-and-conquer algorithm that solves an input of size  $n$ :

- recursively solving some  $k \geq 1$  smaller inputs,
- each of size  $n/b$  for some  $b > 1$  (as before, ignoring floors and ceilings),
- where the total cost of all the “non-recursive work” (splitting the input, combining the results, etc.) is  $O(n^d)$ .

Then the running time  $T(n)$  of the algorithm follows the recurrence relation

$$T(n) = kT(n/b) + O(n^d).$$

The *Master Theorem* provides the solutions to such recurrences.<sup>6</sup>

<sup>4</sup> [https://en.wikipedia.org/wiki/Insertion\\_sort](https://en.wikipedia.org/wiki/Insertion_sort)

<sup>5</sup> Using a linked-list data structure, adding an element to the front does indeed take a constant amount of time; for arrays, with care it is also possible to implement MERGE in linear time. On the other hand, the assumption of constant-time comparisons typically holds only for fixed-size data types, such as 32-bit integers or 64-bit floating-point numbers. For arbitrary-size numbers or other variable-length data types like strings whose sizes might grow with  $n$ , this assumption does not hold, and the cost of comparisons needs to be considered more carefully.

<sup>6</sup> Refer to the [appendix](#) (page 285) for proofs of this master theorem, as well as a more general version with log factors.

**Theorem 15 (Master Theorem)** Let  $k \geq 1, b > 1, d \geq 0$  be constants that do not vary with  $n$ , and let  $T(n)$  be a recurrence with base case  $T(1) = O(1)$  having the following form, ignoring ceilings/floors on (or more generally, addition/subtraction of any constant to) the  $n/b$  argument on the right-hand side:

$$T(n) = k \cdot T(n/b) + O(n^d).$$

Then this recurrence has the solution

$$T(n) = \begin{cases} O(n^d) & \text{if } k < b^d \\ O(n^d \log n) & \text{if } k = b^d \\ O(n^{\log_b k}) & \text{if } k > b^d. \end{cases}$$

In addition, the above bounds are tight: if the  $O$  in the recurrence is replaced with  $\Theta$ , then it is in the solution as well.

Observe that the test involving  $k$ ,  $b$ , and  $d$  can be expressed in logarithmic form, by taking base- $b$  logarithms and comparing  $\log_b k$  to  $d$ .

In the case of merge sort, we have  $k = b = 2$  and  $d = 1$ , so  $k = b^d$ , so the solution is  $T(n) = O(n \log n)$  (and this is tight). Thus, merge sort is much more efficient than insertion sort! (As always in this text, this is merely an *asymptotic* statement, for large enough  $n$ .)

We emphasize that in order to apply (this version of) the Master Theorem, the values  $k, b, d$  must be *constants* that do not vary with  $n$ . For example, the theorem does not apply to a divide-and-conquer algorithm that recursively solves  $k = \sqrt{n}$  subinstances, or one whose subinstances are of size  $n/\log n$ . In such a case, a different tool is needed to solve the recurrence. Fortunately, the Master Theorem does apply to the vast majority of divide-and-conquer algorithms of interest.

### 3.1.1 Master Theorem with Log Factors

A recurrence such as

$$T(n) = 2T(n/2) + O(n \log n)$$

does not exactly fit the form of the master theorem above, since the additive term  $O(n \log n)$  does not look like  $O(n^d)$  for some constant  $d$ .<sup>7</sup> Such a recurrence can be handled by a more general form of the theorem, as follows.

**Theorem 16** Let  $T(n)$  be the following recurrence, where  $k \geq 1, b > 1, d \geq 0, w \geq 0$  are constants that do not vary with  $n$ :

$$T(n) = kT(n/b) + O(n^d \log^w n).$$

Then:

$$T(n) = \begin{cases} O(n^d \log^w n) & \text{if } k < b^d \\ O(n^d \log^{w+1} n) & \text{if } k = b^d \\ O(n^{\log_b k}) & \text{if } k > b^d. \end{cases}$$

Applying the generalized master theorem to the recurrence

<sup>7</sup> Actually, it can be made to fit this form, since any  $O(n \log n)$  function is also, say,  $O(n^{1.001})$ —because  $O$  represents an *upper bound*, and  $\log n = O(n^\varepsilon)$  for an arbitrary positive constant  $\varepsilon > 0$ . However, this substitution is not *tight*—it does not hold with  $\Theta$  in place of  $O$ —and adding a tiny amount to the exponent is clunky.

$$T(n) = 2T(n/2) + O(n \log n),$$

we have  $k = 2, b = 2, d = 1, w = 1$ , so  $k = b^d$ . Therefore,

$$\begin{aligned} T(n) &= O(n^d \log^{w+1} n) \\ &= O(n \log^2 n). \end{aligned}$$

## 3.2 Integer Multiplication

We now turn our attention to algorithms for integer multiplication. For fixed-size data types, such as 32-bit integers, multiplication can be done in a constant amount of time, and is typically implemented as a hardware instruction for common sizes. However, if we are working with arbitrary  $n$ -bit numbers, we will have to implement multiplication ourselves in software. (As we will see later, multiplication of such “big” integers is essential to many cryptography algorithms.)

Let’s first take a look the standard grade-school long-multiplication algorithm.

$$\begin{array}{r} \phantom{=}\phantom{+}\phantom{+}\phantom{2478} \phantom{\longrightarrow} \phantom{=}\phantom{1}\phantom{0}\phantom{0}\phantom{1}\phantom{1}\phantom{0}\phantom{1}\phantom{0}\phantom{1}\phantom{1}\phantom{1}\phantom{0} \\ \phantom{+}\phantom{+}\phantom{2478} \phantom{\longrightarrow} \phantom{=}\phantom{1}\phantom{0}\phantom{0}\phantom{1}\phantom{1}\phantom{0}\phantom{1}\phantom{0}\phantom{1}\phantom{1}\phantom{1}\phantom{0} \\ \phantom{+}\phantom{2478} \phantom{\longrightarrow} \phantom{=}\phantom{1}\phantom{0}\phantom{0}\phantom{1}\phantom{1}\phantom{0}\phantom{1}\phantom{0}\phantom{1}\phantom{1}\phantom{1}\phantom{0} \\ \phantom{2478} \phantom{\longrightarrow} \phantom{=}\phantom{1}\phantom{0}\phantom{0}\phantom{1}\phantom{1}\phantom{0}\phantom{1}\phantom{0}\phantom{1}\phantom{1}\phantom{1}\phantom{0} \\ \hline 2478 \longrightarrow = 1\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ 1\ 0 \end{array}$$

$\begin{array}{l} 1\ 1\ 1\ 0\ 1\ 1 \leftarrow 59 \\ \times 1\ 0\ 1\ 0\ 1\ 0 \leftarrow 42 \\ \hline 1\ 1\ 1\ 0\ 1\ 1 \quad 59 < 1 \\ 1\ 1\ 1\ 0\ 1\ 1 \quad 59 < 3 \\ 1\ 1\ 1\ 0\ 1\ 1 \quad 59 < 5 \end{array}$

Here, the algorithm is illustrated for binary numbers, but it works the same as for decimal numbers, just in base two. We first multiply the top number by the last (least-significant) digit in the bottom number. We then multiply the top number by the second-to-last digit in the bottom number, but we shift the result leftward by one digit. We repeat this for each digit in the bottom number, adding one more leftward shift with each digit. Once we have done all the multiplications for each digit of the bottom number, we add up the results to compute the final product.

How efficient is this algorithm? If the input numbers are each  $n$  bits long, then each individual multiplication takes linear  $O(n)$  time: we have to multiply each digit in the top number by the single digit in the bottom number (plus a carry if we are working in decimal). Since we have to do  $n$  multiplications, computing the partial results takes  $O(n^2)$  total time. We then need to add the  $n$  partial results. The longest partial result is the last one, which is about  $2n$  digits long. Thus, we add  $n$  numbers, each of which has  $O(n)$  digits. Adding two  $O(n)$ -digit numbers takes  $O(n)$  time, so adding  $n$  of them takes a total of  $O(n^2)$  time. Adding the time for the multiplications and additions leaves us with a total of  $O(n^2)$  time for the entire multiplication. (All of the above bounds are tight, so the running time is in fact  $\Theta(n^2)$ .)

Can we do better? Let’s try to make use of the divide-and-conquer paradigm. We first need a way of breaking up an  $n$ -digit number into smaller pieces. We can do that by splitting it into the first  $n/2$  digits and the last  $n/2$  digits. For the rest of our discussion, we will work with decimal numbers, though the same reasoning applies to numbers in any other base. Assume that  $n$  is even for simplicity (we can ensure this by appending a zero in the most-significant digit, if needed).

$$X = \begin{array}{|c|c|} \hline A & B \\ \hline \end{array}$$

$\xleftarrow{\quad n/2 \text{ digits} \quad} \quad \xrightarrow{\quad n/2 \text{ digits} \quad}$

As an example, consider the number 376280. Here  $n = 6$ , and splitting the number into two pieces gives us 376 and 280. How are these pieces related to the original number? We have:

$$\begin{aligned} 376280 &= 376 \cdot 1000 + 280 \\ &= 376 \cdot 10^3 + 280 \\ &= 376 \cdot 10^{n/2} + 280 . \end{aligned}$$

In general, when we split an  $n$ -digit number  $X$  into two  $n/2$ -digit pieces  $A$  and  $B$ , we have that  $X = A \cdot 10^{n/2} + B$ .

Let's now apply this splitting process to multiply two  $n$ -digit numbers  $X$  and  $Y$ . Split  $X$  into  $A$  and  $B$ , and  $Y$  into  $C$  and  $D$ , so that:

$$\begin{aligned} X &= A \cdot 10^{n/2} + B , \\ Y &= C \cdot 10^{n/2} + D . \end{aligned}$$

$$\begin{array}{l} X = \\ Y = \end{array} \begin{array}{|c|c|} \hline A & B \\ \hline C & D \\ \hline \end{array}$$

$\xleftarrow{\quad n/2 \text{ digits} \quad} \quad \xrightarrow{\quad n/2 \text{ digits} \quad}$

We can then expand  $X \cdot Y$  as:

$$\begin{aligned} X \cdot Y &= (A \cdot 10^{n/2} + B) \cdot (C \cdot 10^{n/2} + D) \\ &= A \cdot C \cdot 10^n + A \cdot D \cdot 10^{n/2} + B \cdot C \cdot 10^{n/2} + B \cdot D \\ &= A \cdot C \cdot 10^n + (A \cdot D + B \cdot C) \cdot 10^{n/2} + B \cdot D . \end{aligned}$$

This suggests a natural divide-and-conquer algorithm for multiplying  $X$  and  $Y$ :

- split them as above,
- *recursively* multiply  $A \cdot C$ ,  $A \cdot D$ , etc.,
- multiply each of these by the appropriate power of 10,
- sum everything up.

How efficient is this computation? First, observe that multiplying a number by  $10^k$  is the same as shifting it to the left by appending  $k$  zeros to the (least-significant) end of the number, so it can be done in  $O(k)$  time. So, the algorithm has the following subcomputations:

- 4 recursive multiplications of  $n/2$ -digit numbers ( $A \cdot C$ ,  $A \cdot D$ ,  $B \cdot C$ ,  $B \cdot D$ ),
- 2 left shifts, each of which takes  $O(n)$  time,
- 3 additions of  $O(n)$ -digit numbers, which take  $O(n)$  time.

Let  $T(n)$  be the time it takes to multiply two  $n$ -digit numbers using this algorithm. By the above analysis, it satisfies the recurrence

$$T(n) = 4T(n/2) + O(n) .$$

Applying the Master Theorem with  $k = 4$ ,  $b = 2$ ,  $d = 1$ , we have that  $k > b^d$ . Therefore, the solution is

$$T(n) = O(n^{\log_2 4}) = O(n^2) .$$

Unfortunately, this is the same as for the long-multiplication algorithm! We did a lot of work to come up with a divide-and-conquer algorithm, and it doesn't do any better than a naïve algorithm. Our method of splitting and recombining wasn't sufficiently "clever" to yield an improvement.

### 3.2.1 The Karatsuba Algorithm

Observe that the  $O(n^2)$  bound above has an exponent of  $\log_2 4 = 2$  because we recursed on *four* separate subinstances of size  $n/2$ . Let's try again, but this time, let's see if we can rearrange the computation so that we have *fewer than four* such subinstances. We previously wrote

$$X \cdot Y = A \cdot C \cdot 10^n + (A \cdot D + B \cdot C) \cdot 10^{n/2} + B \cdot D .$$

This time, we will write  $X \cdot Y$  in a different, more clever way using fewer multiplications of (roughly) "half-size" numbers. Consider the values

$$M_1 = (A + B) \cdot (C + D)$$

$$M_2 = A \cdot C$$

$$M_3 = B \cdot D .$$

Observe that  $M_1 = A \cdot C + A \cdot D + B \cdot C + B \cdot D$ , and we can subtract  $M_2 = A \cdot C$  and  $M_3 = B \cdot D$  to obtain

$$M_1 - M_2 - M_3 = A \cdot D + B \cdot C .$$

This is exactly the "middle" term in the above expansion of  $X \cdot Y$ . Thus:

$$X \cdot Y = M_2 \cdot 10^n + (M_1 - M_2 - M_3) \cdot 10^{n/2} + M_3 .$$

This suggests a different divide-and-conquer algorithm for multiplying  $X$  and  $Y$ :

- split them as above,
- compute  $A + B, C + D$  and *recursively* multiply them to get  $M_1$ ,
- *recursively* compute  $M_2 = A \cdot C$  and  $M_3 = B \cdot D$ ,
- compute  $M_1 - M_2 - M_3$ ,
- multiply by appropriate powers of 10,
- sum up the terms.

This is known as the *Karatsuba algorithm*. How efficient is the computation? We have the following subcomputations:

- Computing  $M_1$  does two additions of  $n/2$ -digit numbers, resulting in two numbers that are up to  $n/2 + 1$  digits each. This takes  $O(n)$  time.
- Then these two numbers are multiplied, which takes essentially  $T(n/2)$  time. (The Master Theorem lets us ignore the one extra digit of input length, just like we can ignore floors and ceilings.)
- Computing  $M_2$  and  $M_3$  each take  $T(n/2)$  time.
- Computing  $M_1 - M_2 - M_3$ , multiplying by powers of 10, and adding up terms all take  $O(n)$  time.

So, the running time  $T(n)$  satisfies the recurrence

$$T(n) = 3T(n/2) + O(n) .$$

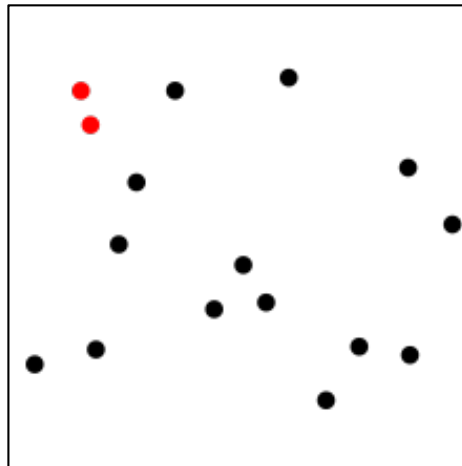
Applying the master theorem with  $k = 3$ ,  $b = 2$ ,  $d = 1$ , we have that  $k > b^d$ . This yields the solution

$$T(n) = O(n^{\log_2 3}) = O(n^{1.585}) .$$

(Note that  $\log_2 3$  is slightly smaller than 1.585, so the second equality is valid because big-O represents an upper bound.) Thus, the Karatsuba algorithm gives us a runtime that is asymptotically much faster than the naïve algorithm! Indeed, it was the first algorithm discovered for integer multiplication that takes “subquadratic” time.

### 3.3 The Closest-Pair Problem

In the *closest-pair problem*, we are given  $n \geq 2$  points in  $d$ -dimensional space, and our task is to find a pair  $p, p'$  of the points whose distance apart  $\|p - p'\|$  is *smallest* among all pairs of the points; such points are called a “closest pair”. Notice that there may be ties among distances between points, so there may be more than one closest pair. Therefore, we typically say *a* closest pair, rather than *the* closest pair, unless we know that the closest pair is unique in some specific situation. This problem has several applications in computational geometry and data mining (e.g. clustering). The following is an example of this problem in two dimensions, where the (unique) closest pair is at the top left in red.



A naïve algorithm compares the distance between every pair of points and returns a pair that is the smallest distance apart; since there are  $\Theta(n^2)$  pairs, the algorithm takes  $\Theta(n^2)$  time. Can we do better?

Let’s start with the problem in the simple setting of one dimension. That is, given a list of  $n$  real numbers  $x_1, x_2, \dots, x_n$ , we wish to find a pair of the numbers that are closest together. In other words, find some  $x_i, x_j$  that minimize  $|x_i - x_j|$ , where  $i \neq j$ .

Rather than comparing every pair of numbers, we can first sort the numbers. Then it must be the case that there is some closest pair of numbers that is adjacent in the sorted list. (Exercise: prove this formally. However, notice that not *every* closest pair must be adjacent in the sorted list, because there can be duplicate numbers.) So, we need only compare each pair of adjacent points to find some closest pair. The following is a complete algorithm:

#### Algorithm 17 (Closest Numbers)

**Input:** an array of  $n \geq 2$  numbers

**Output:** a closest pair of numbers in the array

**function** CLOSESTNUMBERS( $A[1, \dots, n]$ )  
 $S = \text{MERGESORT}(A)$



```

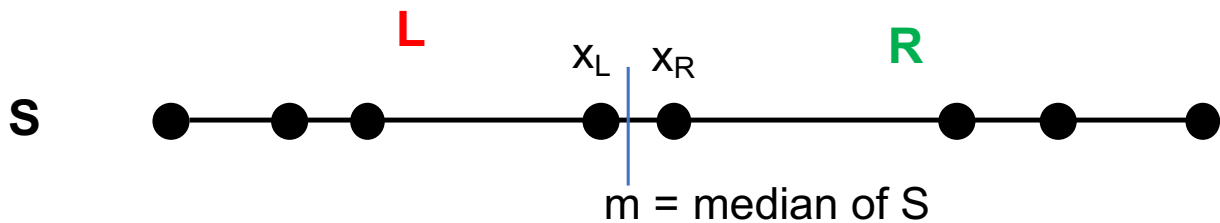
i = 1
for k = 2 to n − 1 do
    if  $|S[k] - S[k + 1]| < |S[i] - S[i + 1]|$  then
        i = k
return  $S[i], S[i + 1]$ 

```

As we saw previously, merge sort takes  $\Theta(n \log n)$  time (assuming fixed-size numbers). The algorithm above also iterates over the sorted list, doing a constant amount of work in each iteration. This takes  $\Theta(n)$  time. Putting the two steps together results in a total running time of  $\Theta(n \log n)$ , which is better than the naïve  $\Theta(n^2)$ .

This algorithm works for one-dimensional points, i.e., real numbers. Unfortunately, it is not clear how to generalize this algorithm to two-dimensional points. While there are various ways we can sort such points, there is no obvious ordering that provides the guarantee that some closest pair of points is adjacent in the resulting ordering. For example, if we sort by  $x$ -coordinate, then a closest pair will be relatively close in their  $x$ -coordinates, but there may be another point with an  $x$ -coordinate between theirs that is very far away in its  $y$ -coordinate.

Let's take another look at the one-dimensional problem, instead taking a divide-and-conquer approach. Consider the median of all the points, and suppose we partition the points into two halves of (almost) equal size, according to which side of the median they lie on. For simplicity, here and below we assume without loss of generality that the median splits the points into halves that are as balanced as possible, by breaking ties between points as needed to ensure balance.



Now consider a closest pair of points in the full set. It must satisfy exactly one of the following three cases:

- both points are from the left half,
- both points are from the right half,
- it “crosses” the halves, with one point in the left half and the other point in the right half.

In the “crossing” case, we can draw a strong conclusion about the two points: they must consist of a *largest* point in the left half, and a *smallest* point in the right half. (For if not, there would be an even closer crossing pair, which would contradict the hypothesis about the original pair.) So, such a pair is the only crossing pair we need to consider when searching for a closest pair.

This reasoning leads naturally to a divide-and-conquer algorithm. We find the median and *recursively* find a closest pair within just the left-half points, and also within just the right-half points. We also consider a largest point on the left with a smallest point on the right. Finally, we return a closest pair among all three of these options. By the three cases above, the output must be a closest pair among all the points. Specifically, in the first case, the recursive call on the left half returns a closest pair for the full set, and similarly for the second case and the recursive call on the right half. And in the third case, by the above reasoning, the specific crossing pair constructed by the algorithm is a closest pair for the full set.

The full algorithm is as follows. For the convenience of the recursive calls, in the case  $n = 1$  we define the algorithm to return a “dummy” output  $(\perp, \perp, \infty)$  representing non-existent points that are infinitely far apart. Therefore, we do not have to check whether each recursive call involves more than one point, and some other pair under consideration will be closer than this dummy result.

**Algorithm 18 (1D Closest Pairs)****Input:** an array of  $n \geq 1$  real numbers**Output:** a closest pair of the points, and their distance apart (or a dummy output with distance  $\infty$ , when  $n = 1$ )

```

function CLOSESTPAIR1D( $A[1, \dots, n]$ )
  if  $n = 1$  then return  $(\perp, \perp, \infty)$ 
  if  $n = 2$  then return  $(A[1], A[2], |A[1] - A[2]|)$ 
  partition  $A$  by its median  $m$  into arrays  $L, R$ 
   $(\ell, \ell', \delta_L) = \text{CLOSESTPAIR1D}(L)$ 
   $(r, r', \delta_R) = \text{CLOSESTPAIR1D}(R)$ 
   $p =$  a largest element in  $L$ 
   $p' =$  a smallest element in  $R$ 
  return one of the triples  $(\ell, \ell', \delta_L), (r, r', \delta_R), (p, p', |p - p'|)$  that has smallest distance

```

Analyzing this algorithm for its running time, we can find the median of a set of points by sorting them and then taking the point in the middle. We can also obtain a largest element in the left side and a largest element in the right right from the sorted list. Partitioning the points by the median also takes  $\Theta(n)$  time; we just compare each point to the median. The non-recursive work is dominated by the  $\Theta(n \log n)$ -time sorting, so we end up with the recurrence:

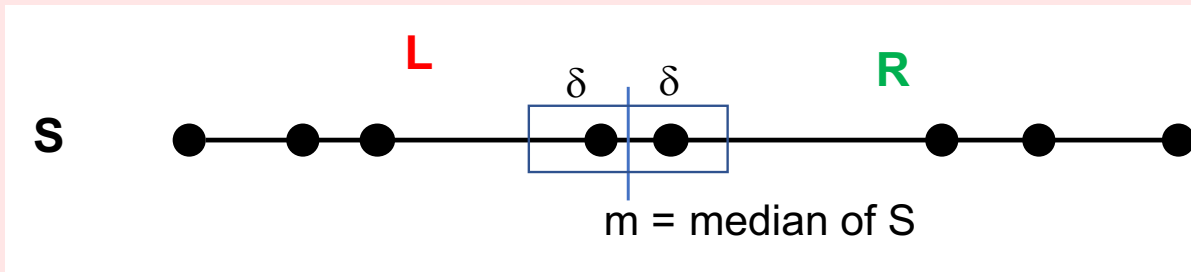
$$T(n) = 2T(n/2) + \Theta(n \log n).$$

This isn't quite covered by the basic form of the Master Theorem, since the additive term is not of the form  $\Theta(n^d)$  for a constant  $d$ . However, it is covered by the *Master Theorem with Log Factors* (page 15), which yields the solution  $T(n) = \Theta(n \log^2 n)$ . (See also a solution using substitution in [Example 247](#).) This means that this algorithm is asymptotically less efficient than our previous one! However, there are two possible modifications we can make:

- Use a  $\Theta(n)$  [median-finding algorithm](#)<sup>8</sup> rather than sorting to find the median.
- Sort the points just once at the beginning, so that we don't need to re-sort in each recursive call.

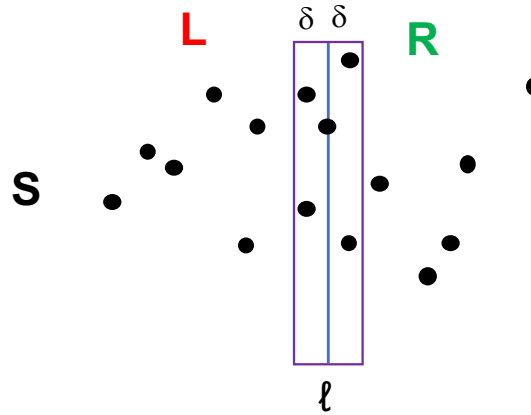
Either modification brings the running time of the non-recursive work down to  $\Theta(n)$ , resulting in a full running time of  $T(n) = \Theta(n \log n)$ . (The second option involves an additional  $\Theta(n \log n)$  on top of this for the presorting, but that still results in a total time of  $\Theta(n \log n)$ .)

**Exercise 19** In the one-dimensional closest-pair algorithm, we computed the median  $m$ , the closest-pair distance  $\delta_L$  on the left, and the closest-pair distance  $\delta_R$  on the right. Let  $\delta = \min\{\delta_L, \delta_R\}$ . How many points can lie in the interval  $[m, m + \delta)$ ? What about the interval  $(m - \delta, m + \delta)$ ?



Now let's try to generalize this algorithm to two dimensions. It's not clear how to split the points according to a median point, or even what a meaningful "median point" would be. So rather than doing that, we instead use a median *line* as defined by the median  $x$ -coordinate.

<sup>8</sup> [https://en.wikipedia.org/wiki/Median\\_of\\_medians](https://en.wikipedia.org/wiki/Median_of_medians)



As before, any closest pair for the full set must satisfy one of the following: both of its points are in the left half, both are in the right half, or it is a “crossing” pair with exactly one point in each half. Similarly to above, we will prove that in the “crossing” case, the pair must satisfy some specific conditions. This means that it will suffice for our algorithm to check *only* those crossing pairs that meet the conditions—since this will find a closest pair, it can ignore all the rest.

In two dimensions we cannot draw as strong of a conclusion about the “crossing” case as we could in one dimension. In particular, the  $x$ -coordinates of the pair may *not* be closest to the median line: there could be another crossing pair whose  $x$ -coordinates are even closer to the median line, but whose  $y$ -coordinates are very far apart, making that pair farther apart overall. Nevertheless, the  $x$ -coordinates are “relatively close” to the median line, as shown in the following lemma.

**Lemma 20** *Let  $\delta_L$  and  $\delta_R$  respectively be the closest-pair distances for just the left and right halves. If a closest pair for the entire set of point is “crossing,” then both of its points must be within distance  $\delta = \min\{\delta_L, \delta_R\}$  of the median line.*

**Proof 21** We prove the contrapositive. If a crossing pair of points has at least one point at distance greater than  $\delta$  from the median line, then the pair of points are more than  $\delta$  apart. Therefore, they cannot be a closest pair for the entire set, because there is another pair that is only  $\delta$  apart.  $\square$

Thus, the only crossing pairs that our algorithm needs to consider are those whose points lie in the “ $\delta$ -strip”, i.e., the space within distance  $\delta$  of the median line (in the  $x$ -coordinate); no other crossing pair can be a closest pair for the entire set. This leads to the following algorithm:

**Algorithm 22 (2D Closest Pair – First Attempt)**

**Input:** an array of  $n \geq 1$  points in the plane

**Output:** a closest pair of the points, and their distance apart (or a dummy output with distance  $\infty$ , when  $n = 1$ )

```

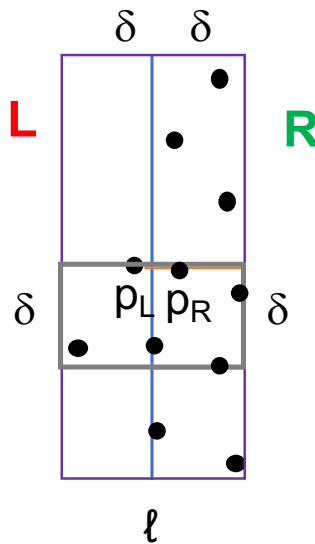
function CLOSESTPAIR2DATTEMPT( $A[1, \dots, n]$ )
    if  $n = 1$  then return  $(\perp, \perp, \infty)$ 
    if  $n = 2$  then return  $(A[1], A[2], \|A[1] - A[2]\|)$ 
    partition  $A$  by its median  $x$ -coordinate  $m$  into arrays  $L, R$ 
     $(\ell, \ell', \delta_\ell) = \text{CLOSESTPAIR2DATTEMPT}(L)$ 
     $(r, r', \delta_r) = \text{CLOSESTPAIR2DATTEMPT}(R)$ 
     $\delta = \min\{\delta_\ell, \delta_r\}$ 
    find a closest pair  $(p, p') \in L \times R$  among the points whose  $x$ -coordinates are within  $\delta$  of  $m$ 
    return one of the triples  $(\ell, \ell', \delta_\ell), (r, r', \delta_r), (p, p', \|p - p'\|)$  that has smallest distance
    
```

Apart from its checks of crossing pairs in the  $\delta$ -strip, the non-recursive work is same as in the one-dimensional algorithm, and it takes  $\Theta(n)$  time. (We can presort the points by  $x$ -coordinate or use a  $\Theta(n)$ -time median-finding algorithm, as before.) How long does it take to check the crossing pairs in the  $\delta$ -strip? A naïve examination would consider every pair of points where one is in the left side of the  $\delta$ -strip and the other is in its right side. But notice that in the worst case, *all* of the points can be in the  $\delta$ -strip! For example, this can happen if the points are close together in the  $x$ -dimension—in the extreme, they all lie on the median line—but far apart in the  $y$ -dimension. So in the worst case, we have  $n/2$  points in each of the left and right parts of the  $\delta$ -strip, leaving us with  $n^2/4 = \Theta(n^2)$  crossing pairs to consider. So we end up with the recurrence and solution

$$T(n) = 2T(n/2) + \Theta(n^2) = \Theta(n^2).$$

This is no better than the naïve algorithm that just compares all pairs of points! We have not found an efficient enough non-recursive “combine” step.

Let’s again consider the case where a closest pair for the whole set is a “crossing” pair, and try to establish some additional stronger properties for it, so that our algorithm will not need to examine as many crossing pairs in its combine step. Let  $p_L = (x_L, y_L)$  and  $p_R = (x_R, y_R)$  respectively be the points from the pair that are on the left and right of the median line, and assume without loss of generality that  $y_L \geq y_R$ ; otherwise, replace  $p_L$  with  $p_R$  in the following analysis. Because this is a closest pair for the entire set of points,  $p_L$  and  $p_R$  are at most  $\delta = \min\{\delta_L, \delta_R\}$  apart, where as in [Lemma 20](#) above,  $\delta_L$  and  $\delta_R$  are respectively the closest-pair distances for just the left and right sides. Therefore,  $0 \leq y_L - y_R \leq \delta$ .



We ask: how many of the given points  $p' = (x', y')$  in the  $\delta$ -strip can satisfy  $0 \leq y_L - y' \leq \delta$ ? Equivalently, any such point is in the  $\delta$ -by- $2\delta$  rectangle of the  $\delta$ -strip whose top edge has  $p_L$  on it. We claim that there can be at most *eight* such points, including  $p_L$  itself. (The exact value of eight is not too important; what matters is that it is a *constant*.) The key to the proof is that every pair of points must be at least  $\delta$  apart, so we cannot fit too many points into the rectangle. We leave the formal proof to [Exercise 25](#) below.

In conclusion, we have proved the following key structural lemma.

**Lemma 23** *If a closest pair for the whole set is a crossing pair, then its two points are in the  $\delta$ -strip, and they are within 7 positions of each other when all the points in the  $\delta$ -strip are sorted by  $y$ -coordinate.*

So, if a closest pair in the whole set is a crossing pair, then it suffices for the algorithm to compare each point in the  $\delta$ -strip with the (up to) seven points in the  $\delta$ -strip that precede it in sorted order by  $y$ -coordinate. By [Lemma 23](#), this will find a closest pair for the entire set, so the algorithm does not need to check any other pairs. The formal algorithm is as follows.

**Algorithm 24 (2D Closest Pairs)**

**Input:** an array of  $n \geq 1$  points in the plane

**Output:** a closest pair of the points, and their distance apart (or a dummy output with distance  $\infty$ , when  $n = 1$ )

**function** CLOSESTPAIR2D( $A[1, \dots, n]$ )

**if**  $n = 1$  **then return**  $(\perp, \perp, \infty)$

**if**  $n = 2$  **then return**  $(A[1], A[2], \|A[1] - A[2]\|)$

    partition  $A$  by its median  $x$ -coordinate  $m$  into arrays  $L, R$

$(\ell, \ell', \delta_\ell) = \text{CLOSESTPAIR2D}(L)$

$(r, r', \delta_r) = \text{CLOSESTPAIR2D}(R)$

$\delta = \min\{\delta_\ell, \delta_r\}$

$D =$  the set of points whose  $x$ -coordinates are within  $\delta$  of  $m$ , sorted by  $y$ -coordinate

**for all**  $p$  in  $D$  **do**

        consider the triple  $(p, p', \|p - p'\|)$  for the (up to) 7 points  $p'$  preceding  $p$  in  $D$

**return** one of the triples among  $(\ell, \ell', \delta_\ell)$ ,  $(r, r', \delta_r)$ , and those above that has smallest distance

We have already seen that we can presort by  $x$ -coordinate, so that finding the median and constructing  $L, R$  in each run of the algorithm takes just  $O(n)$  time. We can also separately presort by  $y$ -coordinate (into a different array) so that we do not have to sort the  $\delta$ -strip in each run. Instead, we merely filter the points from the presorted array according to whether they lie within the  $\delta$ -strip, which also takes  $O(n)$  time. Finally, we consider at most  $7n = \Theta(n)$  pairs in the  $\delta$ -strip. So, the non-recursive work takes  $\Theta(n)$  time, resulting in the runtime recurrence and solution

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n) .$$

This matches the asymptotic efficiency of the one-dimensional algorithm. The algorithm can be further generalized to higher dimensions, retaining the  $\Theta(n \log n)$  runtime for any fixed dimension.

**Exercise 25** Prove that any  $\delta$ -by- $2\delta$  rectangle of the  $\delta$ -strip can have at most 8 of the given points, where  $\delta$  is as defined in [Lemma 20](#).

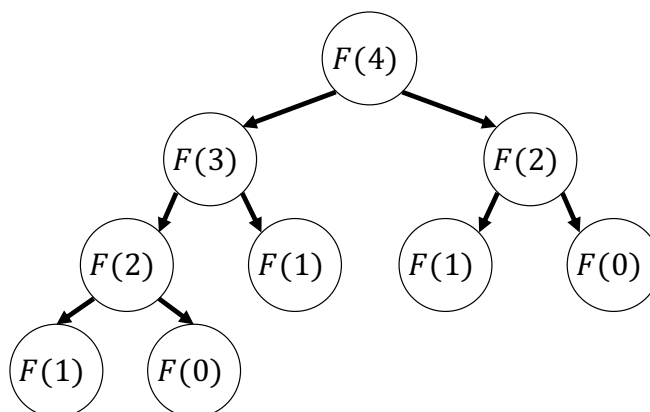
Specifically, prove that the left  $\delta$ -by- $\delta$  square of the rectangle can have at most four of the points (all from left subset), and similarly for the right square.

**Hint:** Partition each square into four congruent sub-squares, and show that each sub-square can have at most one point (from the relevant subset).

## DYNAMIC PROGRAMMING

The idea of subdividing a large problem instance into smaller instances of the same problem lies at the core of the divide-and-conquer paradigm. However, for some problems, this recursive subdivision may result in many recurrences of the exact same subinstance. Such situations are amenable to the paradigm of *dynamic programming*, which is applicable to problems that have the following features:

1. The *principle of optimality*, also known as an *optimal substructure*. This means that an optimal solution to a larger instance is made up of optimal solutions to smaller subinstances. For example, shortest-path problems on graphs generally obey the principle of optimality. If a shortest path between vertices  $a$  and  $b$  in a graph goes through some other vertex  $c$ , so that the path has the form  $a, u_1, \dots, u_j, c, v_1, \dots, v_k, b$ , then the subpath  $a, u_1, \dots, u_j, c$  is a shortest path from  $a$  to  $c$ , and similarly for the subpath from  $c$  to  $b$ .
2. *Overlapping subproblems/subinputs*. This means that the same input occurs many times when recursively decomposing the original instance down to the base cases. A classic example of this is a recursive computation of the Fibonacci sequence, which follows the recurrence  $F(n) = F(n-1) + F(n-2)$ . The same subinstances appear over and over again, making a naïve computation takes time exponential in  $n$ .



The latter characteristic of overlapping subinputs is what distinguishes dynamic programming from divide and conquer.

### 4.1 Implementation Strategies

The first, and typically most challenging and creative, step in applying dynamic programming to a problem is to determine a recurrence relation that solutions adhere to (along with the base case(s)). In the case of the Fibonacci sequence, for example, the recurrence and bases cases are simply given as:

$$F(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

Once we have established the recurrence relation and base cases, we can turn to an implementation strategy for *computing* the desired value(s) of the recurrence. There are three typical patterns:

1. *Top-down recursive*. The naïve implementation directly translates the recurrence relation into a recursive algorithm, as in the following:

**Algorithm 26 (Top-down Fibonacci)**

**Input:** an integer  $n \geq 0$   
**Output:** the  $n$ th Fibonacci number  
**function** FIBRECURSIVE( $n$ )  
    **if**  $n \leq 1$  **then return** 1  
    **return** FIBRECURSIVE( $n - 1$ ) + FIBRECURSIVE( $n - 2$ )

As alluded to previously, the problem with this strategy is that it repeats the same computations many times, to the extent that the overall number of recursive calls is exponential in  $n$ . More generally, naïve top-down implementations are wasteful when there are overlapping subinputs. (They do not use any auxiliary storage<sup>9</sup>, so they are space efficient, but this is usually outweighed by their poor running times.)

2. *Top-down memoized*, or simply *memoization*. This approach also translates the recurrence relation into a recursive algorithm, but it saves all its results in a lookup table and queries that table before doing any computation. The following is an example:

**Algorithm 27 (Memoized Fibonacci)**

**Input:** an integer  $n \geq 0$   
**Output:** the  $n$ th Fibonacci number  
    memo = an empty table (e.g., an array or dictionary)  
**function** FIBMEMOIZED( $n$ )  
    **if**  $n \leq 1$  **then return** 1  
    **if** memo( $n$ ) is not defined **then**  
        memo( $n$ ) = FIBMEMOIZED( $n - 1$ ) + FIBMEMOIZED( $n - 2$ )  
    **return** memo( $n$ )

This memoized algorithm avoids recomputing the answer for any previously encountered subinput. Each call to Fib( $n$ ), where  $n$  is not a base case, first checks the *memo* table to see if Fib( $n$ ) has been computed before. If not, it computes it recursively as above, saving the result in *memo*. If the subinput was previously encountered, the algorithm just returns the previously computed result.

Memoization trades space for time. The computation of Fib( $n$ ) requires  $O(n)$  auxiliary space to store the results for each subinput. On the other hand, since the answer to each subinput is computed only once, the overall number of operations required is  $O(n)$ , a significant improvement over the exponential naïve algorithm. However, for more complicated algorithms, it can be harder to analyze the running time of the associated algorithm.

3. *Bottom-up table*.<sup>10</sup> Rather than starting with the desired input and working our way down to the base cases, we can invert the computation to start with the base case(s), and then work our way up to the desired input. Typically, we need a table to store the results for the subinputs we have handled so far, since those results will be needed to compute answers for larger inputs.

The following is a bottom-up implementation for computing the Fibonacci sequence:

<sup>9</sup> Space is required to represent the recursion stack; for the Fibonacci computation, there are as many as  $O(n)$  active calls at any point. For other algorithms, however, the naïve implementation can incur lower space costs than the table-based approaches.

<sup>10</sup> In many contexts, the term “dynamic programming” is used to refer specifically to the bottom-up approach. In this text, however, we use the term to refer to either the top-down-memoized or the bottom-up-table strategies, and we will be explicit when we refer to a specific approach.

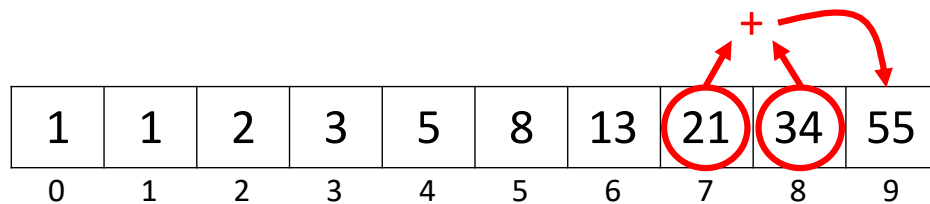
**Algorithm 28 (Bottom-up Fibonacci)****Input:** an integer  $n \geq 0$ **Output:** the  $n$ th Fibonacci number

```

function FIBBOTTOMUP( $n$ )
    allocate table[0, . . . ,  $n$ ]
    table[0] = table[1] = 1
    for  $i = 2$  to  $n$  do
        table[ $i$ ] = table[ $i - 1$ ] + table[ $i - 2$ ]
    return table[ $n$ ]

```

We start with an empty table and populate it with the results for the base cases. Then we work our way forward, computing the result of each larger input from the previously computed results for the smaller inputs. We stop when we reach the desired input, and return the result. The following is an illustration of the table for Fib(9).



1	1	2	3	5	8	13	21	34	55
0	1	2	3	4	5	6	7	8	9

When we get to computing Fib( $i$ ), we have already computed Fib( $i - 1$ ) and Fib( $i - 2$ ), so we can just look up those results from the table and add them to get the result for Fib( $i$ ).

Like memoization, the bottom-up approach trades space for time. In the case of Fib( $n$ ), it too uses  $O(n)$  auxiliary space to store the result of each subproblem, and the overall number of additions required is  $O(n)$ .

In the specific case of computing the Fibonacci sequence, we don't actually need to keep the entire table for the entire computation – once we are computing Fib( $i$ ), we will no longer need the results for Fib( $i - 3$ ) or lower. So, at any moment we only need to keep the two previously computed results. This lowers the storage overhead to  $O(1)$ . However, this isn't the case in general for dynamic programming. Other problems require maintaining a larger subset or all of the table throughout the computation.

The three implementation strategies have different tradeoffs. The naïve top-down strategy often takes the least implementation effort, as it is a direct translation of the recurrence relation to code. It is also usually the most space efficient, but it often is very *inefficient* in time, due to the many redundant computations. Memoization adds some implementation effort in working with a lookup table, and can require special care to implement correctly in practice. (In particular, it might require a global variable for the “memo table,” which is not considered a good programming technique due to its risks. For example, completely unrelated inputs might end up using the same global variable, resulting in incorrect results. Fortunately, some programming languages provide built-in facilities for converting a naïve recursive function into a memoized one, such as `@functools.lru_cache` in Python.) Its main advantages over the bottom-up approach are:

- It maintains the same structure as the recurrence relation, so it typically is simpler to reason about and implement.
- It computes answers for only the subinputs that are actually needed. If the recurrence induces a *sparse* computation, meaning that it requires answers for only a small subset of the smaller subinputs, then the top-down memoization approach can be more time and space efficient than the bottom-up strategy.

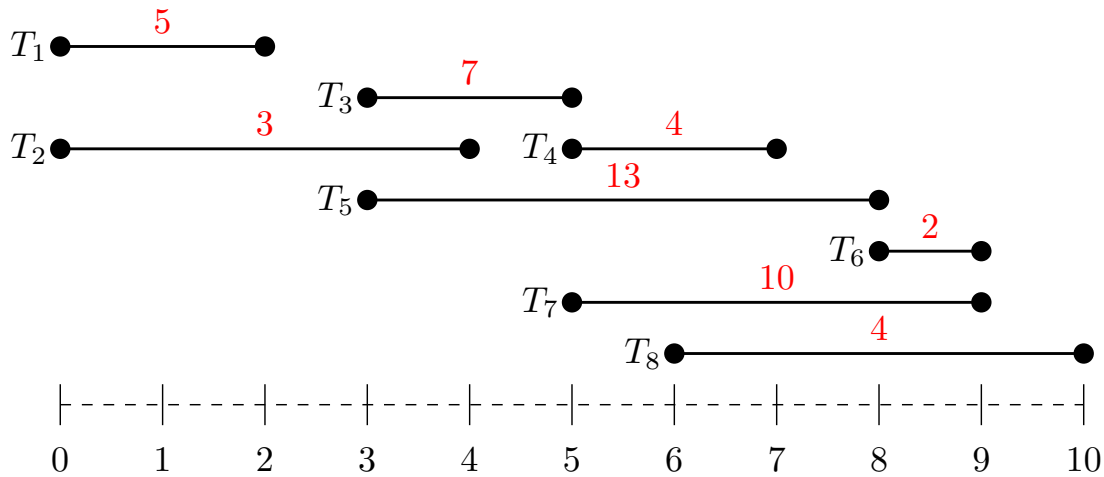
However, top-down memoization often suffers from higher constant-factor overheads than the bottom-up approach (e.g., function-call overheads and working with sparse lookup structures that are less time efficient than dense data structures). Thus, the bottom-up approach is preferable when a large fraction of the subproblems are needed for computing the desired result, which is usually the case for dynamic programming problems.



## 4.2 Weighted Task Selection

As a first nontrivial example of dynamic programming, let us consider a problem called *weighted task selection*. We are given a list of  $n$  tasks  $T_1, \dots, T_n$ . Each task  $T_i$  is a triple of numbers  $(s_i, f_i, v_i)$ , where  $s_i$  denotes the task's starting time,  $f_i$  denotes its finishing time, and  $v_i$  denotes its value. We assume that the tasks are sorted according to their finish time  $f_i$  (in particular, we can sort them in  $O(n \log n)$  time), and we always have  $s_i < f_i$ . The goal is to select a set of (pairwise) non-overlapping tasks that maximizes the total value of the selected tasks. A pair of tasks  $T_i, T_j$  overlap if  $s_i \leq s_j < f_i$  or  $s_j \leq s_i < f_j$ , i.e., if the intersection of their time intervals  $[s_i, f_i) \cap [s_j, f_j)$  is nonempty.

The figure below shows an example instance of the problem with eight tasks. The set  $\{T_2, T_3\}$  is overlapping. The set  $\{T_1, T_5, T_6\}$  is not overlapping, and has a total value of  $5 + 13 + 2 = 20$ , which is not optimal. An optimal set of tasks is  $\{T_1, T_3, T_7\}$ , and has a total value of  $5 + 7 + 10 = 22$ .



This problem models various situations including scheduling of jobs on a single machine, choosing courses to take, etc. So it is indeed useful, but can we solve it efficiently?

The design and analysis of a dynamic programming algorithm usually follows these main steps:

1. Focus on the “**value version**” of the problem. For an optimization problem, temporarily put aside the goal of finding an optimal solution itself, and simply aim to find the *value* of an optimal solution (e.g., the lowest cost, the largest total value, etc.). In some cases, the original problem is already stated in a value version—e.g., count the *number* of objects meeting certain constraints—so there is nothing to do in this step.
2. Devise a **recurrence for the value in question**, including base case(s), by understanding how a solution is made up of solutions to appropriate subinputs. This step usually requires some creativity and insight, both to discover a good set of relevant subinputs, and to see how optimal solutions are related across subinputs.
3. By understanding the dependencies between subinputs as given by the recurrence, **implement the recurrence in (pseudo)code** that fills a table in a bottom-up fashion. Given the recurrence, this step is usually fairly “mechanical” and does not require a great deal of creativity.
4. If applicable, extend the pseudocode to **solve the original problem using “backtracking”**. Given the recurrence and an understanding of why it is correct, this is also usually quite mechanical.
5. Perform a **runtime analysis** of the algorithm. This is also usually fairly mechanical, and follows from analyzing the number of table entries that are filled, and the amount of time it takes to fill each entry (or in some cases, all the entries in total).

Notice that all steps but the second one are fairly mechanical, but they rely crucially on the recurrence derived in that step.

For example, for the above example instance of our task-selection problem, we first want to focus on designing an

algorithm that simply computes the optimal *value* 22, instead of directly computing an optimal *set* of non-overlapping tasks  $\{T_1, T_3, T_7\}$ . As we will see, a small enhancement of the value-optimizing algorithm will also give us a selection of tasks that achieves the optimal value, so we actually lose nothing by initially focusing on the value alone. To do this, we first need to come up with a recurrence that yields the optimal value for a given set of tasks.

For  $0 \leq i \leq n$ , let  $\text{OPT}(i)$  denote the optimal value that can be obtained by selecting from among just the tasks  $T_1, \dots, T_i$ . The base case is  $i = 0$ , for which the optimal value is  $\text{OPT}(0) = 0$ , because there are no tasks available to select. Now suppose that  $i \geq 1$ , and consider task  $T_i$  (which has the latest finish time among those we are considering). There are two possibilities: either  $T_i$  is in an optimal selection of tasks (from just  $T_1, \dots, T_i$ ), or it is not.

- If  $T_i$  is not in an optimal selection, then  $\text{OPT}(i) = \text{OPT}(i - 1)$ . This is because there is an optimal selection for all  $i$  tasks that selects from just the first  $i - 1$  tasks, so the availability of  $T_i$  does not affect the optimal value.
- If  $T_i$  is in an optimal solution, then  $\text{OPT}(i) = v_i + \text{OPT}(j)$ , where  $j < i$  is the maximum index such that  $T_j$  and  $T_i$  do not overlap, i.e.,  $f_j \leq s_i$  (with  $j = 0$  if no such  $T_j$  exists). This is because in any optimal selection  $S$  that has  $T_i$ , the other selected tasks must come from  $T_1, \dots, T_j$  (because these are the tasks that don't overlap with  $T_i$ ), and the selected ones must have optimal value. For if they did not, we could replace them with a higher-value selection from  $T_1, \dots, T_j$ , then include  $T_i$  to get a valid selection of tasks with a higher value than that of  $S$ , contradicting the assumed optimality of  $S$ .

Overall, the optimal value is the maximum of what can be obtained from these two possibilities. Therefore, we obtain the final recurrence as follows:

$$\text{OPT}(i) = \max\{\text{OPT}(i - 1), v_i + \text{OPT}(j)\} \text{ for the maximum } j < i \text{ s.t. } f_j \leq s_i.$$

To ensure that some such  $j$  exists, we consider  $f_0 = -\infty$ , so that  $j = 0$  at minimum (in which case the second value in the considered set is just  $v_i + \text{OPT}(0) = v_i$ ).

Now that we have a recurrence, we can write pseudocode that implements it by filling a table in a bottom-up manner.

#### Algorithm 29 (Weighted Task Selection, Value Version)

**Input:** array of tasks  $T[i] = (s_i, f_i, v_i)$

**Output:** maximum achievable value for (pairwise) non-overlapping tasks

**function** OPTIMALTASKSVALUE( $T[1, \dots, n]$ )

    allocate table[0, ..., n]

    table[0] = 0

**for**  $i = 1$  to  $n$  **do**

        find the maximum  $j < i$  such that  $f_j \leq s_i$

        table[i] =  $\max\{\text{table}[i - 1], v_i + \text{table}[j]\}$

**return** table[n]

A naïve implementation of this algorithm, which just does a linear scan inside the loop to determine  $j$ , takes  $O(n^2)$  time because there are two nested loops that each take at most  $n$  iterations. Since the tasks are sorted by finish time, a more sophisticated implementation would use a binary search, which would take just  $O(\log n)$  time for the inner (search) loop, and hence  $O(n \log n)$  time overall. (The initial time to sort the tasks by finish time is also  $O(n \log n)$ .)

For a better understanding of this algorithm, let us see the filled table for the example instance given above. We see that indeed  $\text{OPT}(n) = 22$ , which is the correct answer.

Table 4.1: Example Table of Weighted Task Selection DP Algorithm

Index $i$	0	1	2	3	4	5	6	7	8
$\text{OPT}(i)$	0	5	5	12	16	18	20	22	22

Now that we have an efficient algorithm for the *value* version of the problem, let us see how to solve the problem we actually care about, which is to obtain an optimal *selection of tasks*. The idea is to keep some additional information showing “why” each entry of the table has the value it does, and to construct an optimal selection by “backtracking” through the table. That is, we enhance the algorithm to add “pointers” (sometimes called “breadcrumbs”) that store how each cell in the table was filled, based on the recurrence and the values of the previous cells.

To carry out this approach for our problem, we will add pointers, denoted by  $\text{backtrack}[i]$ , into our algorithm. When we fill in  $\text{table}[i]$ , we also set  $\text{backtrack}[i] = i - 1$  if we set  $\text{table}[i] = \text{table}[i - 1]$ , otherwise we set  $\text{backtrack}[i] = j$  where  $j$  is as in the algorithm. Recalling the reasoning for why the recurrence is valid, these two possibilities respectively correspond to task  $T_i$  not being in, or being in, an optimal subset of tasks from  $T_1, \dots, T_i$ , and the value of  $\text{backtrack}[i]$  indicates the prefix of tasks from which the rest of that optimal subset is drawn.

Given the two arrays (table, backtrack), we can now construct an optimal set of tasks, not just the optimal value. Start with index  $i = n$ . Recall that  $\text{table}[i] > \text{table}[i - 1]$  if and only if  $T_i$  is in an optimal selection of tasks. So, if  $\text{table}[i] > \text{table}[i - 1]$ , then we include  $T_i$  in the output selection, otherwise we skip it. We then “backtrack” by setting  $i = \text{backtrack}[i]$  and repeat this process until we have backtracked to the beginning.

The modified full pseudocode, including the backtracking, can be seen below.

#### Algorithm 30 (Weighted Task Selection, with Backtracking)

**Input:** array of tasks  $T[i] = (s_i, f_i, v_i)$ , sorted by  $f_i$

**Output:** values and backtracking information

**function** OPTIMALTASKSINFO( $T[1, \dots, n]$ )

    allocate  $\text{table}[0, \dots, n]$ ,  $\text{backtrack}[1, \dots, n]$

$\text{table}[0] = 0$

**for**  $i = 1$  to  $n$  **do**

        find the maximum  $j < i$  such that  $f_j \leq s_i$  (where  $f_0 = -\infty$ )

$\text{table}[i] = \max\{\text{table}[i - 1], v_i + \text{table}[j]\}$

**if**  $\text{table}[i] = \text{table}[i - 1]$  **then**

$\text{backtrack}[i] = i - 1$

**else**

$\text{backtrack}[i] = j$

**return** (table, backtrack)

**Input:** array of tasks  $T[i] = (s_i, f_i, v_i)$ , sorted by  $f_i$

**Output:** a set of non-overlapping tasks having maximum total value

**function** OPTIMALTASKS( $T[1, \dots, n]$ )

    (table, backtrack) = OPTIMALTASKSINFO( $T$ )

$i = n$

$S = \emptyset$

**while**  $i > 0$  **do**

**if**  $\text{table}[i] > \text{table}[i - 1]$  **then**

$S = S \cup \{T_i\}$

$i = \text{backtrack}[i]$

**return**  $S$

### 4.3 Longest Increasing Subsequence

Given a sequence of numbers  $S$ , an *increasing subsequence* of  $S$  is a subsequence whose elements are in strictly increasing order. As with a subsequence of a string, the elements need not be contiguous in the original sequence, but they must appear in the same relative order as in the original sequence.

Now suppose we want to find a *longest increasing subsequence (LIS)* of  $S$ . For example, the sequence  $S = (0, 8, 7, 12, 5, 10, 4, 14, 3, 6)$  has several longest increasing subsequences:

- (0, 8, 12, 14)
- (0, 8, 10, 14)
- (0, 7, 12, 14)
- (0, 7, 10, 14)
- (0, 5, 10, 14)

As in the previous problem, we first focus on finding the length of an LIS before concerning ourselves with finding an LIS itself. Let  $N$  be the length of  $S$ . For our first attempt, we look at the subproblems of computing the length of the LIS for each sequence  $S[1..i]$  for  $i \in [1, N]$ . In the case of  $S = (0, 8, 7, 12, 5, 10, 4, 14, 3, 6)$ , we can determine these lengths by hand.

1	2	2	3	3	3	3	4	4	4
1	2	3	4	5	6	7	8	9	10

However, it is not clear how we can relate the length of the LIS for a sequence  $S$  of  $N$  elements to a smaller sequence. In the example above,  $S[1..9]$  and  $S[1..10]$  both have the same length LIS, but that is not the case for  $S[1..7]$  and  $S[1..8]$ . Yet, in both cases, the additional element is larger than the previous one (i.e.  $S[10] > S[9]$  and  $S[8] > S[7]$ ). Without knowing the contents of the LIS itself, it is not obvious how the result for the larger problem is related to that of smaller ones.

For dynamic programming to be applicable, we must identify a problem formulation that satisfies the principle of optimality. So rather than tackling the original problem directly, let's constrain ourselves to subsequences of  $S[1..N]$  that actually contain the last element  $S[N]$ . Define  $CLIS(S[1..N])$  to be the longest of the increasing subsequences of  $S[1..N]$  that contain  $S[N]$ . Then  $CLIS(S[1..N])$  has the form  $(S[i_1], S[i_2], \dots, S[i_k], S[N])$ . Observe that by definition of increasing subsequence, we must have  $i_k < N$  and  $S[i_k] < S[N]$ . In addition,  $(S[i_1], S[i_2], \dots, S[i_k])$  is itself an increasing subsequence that contains  $S[i_k]$ . Moreover, we can show via contradiction that  $CLIS(S[1..i_k]) = (S[i_1], S[i_2], \dots, S[i_k])$ . Thus,  $CLIS(S[1..i_k])$  is a subproblem of  $CLIS(S[1..N])$  when  $S[i_k] < S[N]$ .

We do not know a priori which index  $i_k$  produces a result that is a subsequence of  $CLIS(S[1..N])$ . But only the indices  $i_k$  such that  $i_k < N$  and  $S[i_k] < S[N]$  may do so, so we can just try all such indices and take the maximum. Let  $L(i)$  be the length of  $CLIS(S[1..i])$ . Then we have:

$$L(i) = 1 + \max\{L(j) : 0 < j < i \text{ and } S[j] < S[i]\}$$

The following illustrates this computation for the concrete sequence  $S$  above, specifically showing which values need to be compared when computing  $L(10)$ .

$S$	0	8	7	12	5	10	4	14	3	6
	1	2	3	4	5	6	7	8	9	10
$L$	1	2	2	3	2	3	2	4	2	3
	1	2	3	4	5	6	7	8	9	10

We also have to account for the base cases, where no such  $j$  exists. In that case,  $L(i) = 1$ , since  $CLIS(S[1..i])$  consists of just  $S[i]$  itself. The full recurrence is as follows:

$$L(i) = \begin{cases} 1 & \text{if } i = 1 \text{ or } S[j] \geq S[i] \text{ for all } 0 < j < i \\ 1 + \max\{L(j) : 0 < j < i \text{ and } S[j] < S[i]\} & \text{otherwise.} \end{cases}$$

We can then construct an algorithm for computing  $L(i)$ . Once we have  $L(i)$  for all  $i \in [1, N]$ , we just take the maximum value of  $L(i)$  as the result for the length of the unconstrained LIS. As with weighted task selection, we can also backtrack through the table of values for  $L(i)$  to find the actual elements belonging to a longest increasing subsequence.

How efficient is the algorithm, assuming a bottom-up approach? We must compute  $N$  values for  $L(i)$ , and each value requires scanning over all the previous elements of  $S$ , as well as all the previous values of  $L(i)$  in the worst case. Thus, it takes  $O(N)$  time to compute  $L(i)$  for a single  $i$ , and  $O(N^2)$  to compute them all. Finding the maximum  $L(i)$  takes linear time, as does backtracking. So the algorithm as a whole takes  $O(N^2)$  time, and it uses  $O(N)$  space to store the values of  $L(i)$ .

## 4.4 Longest Common Subsequence

As a more complex example of dynamic programming, we take a look at the problem of finding the *longest common subsequence* (LCS) of two strings. A *subsequence* is an ordered subset of the characters in a string, preserving the order in which the characters appear in the original string. However, the characters in the subsequence need not be adjacent in the original string<sup>11</sup>. The following are all subsequences of the string "Fibonacci sequence":

- "Fun"
- "seen"
- "cse"

The longest common subsequence  $LCS(S_1, S_2)$  of two strings  $S_1$  and  $S_2$  is the longest string that is both a subsequence of  $S_1$  and of  $S_2$ . For instance, the longest common subsequence of "Go blue!" and "Wolverines" is "ole".

Finding the longest common subsequence is useful in many applications, including DNA sequencing and computing the diff of two files. As such, we wish to devise an efficient algorithm for determining the LCS of two arbitrary strings.

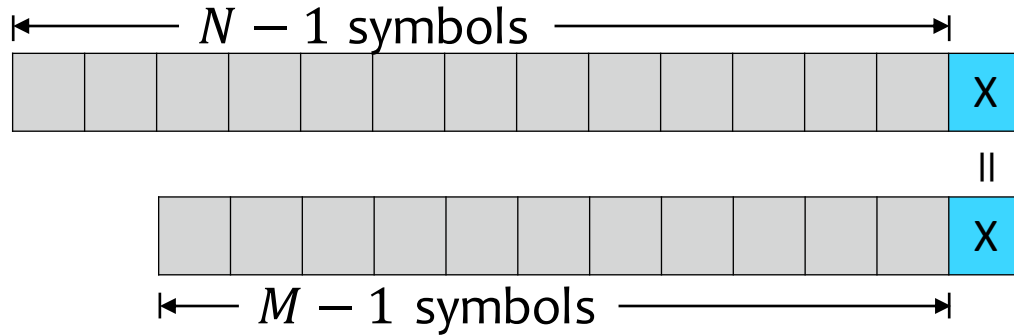
Let's assume that we are given two strings  $S_1$  and  $S_2$  as input, and let their lengths be  $N$  and  $M$ , respectively. For now, let's set aside the problem of finding the actual longest common subsequence and focus on just determining the **length** of this subsequence. To apply dynamic programming, we first need to come up with a recurrence relation that relates the length of the LCS of  $S_1$  and  $S_2$  to smaller subproblems. Let's consider just the last<sup>12</sup> character in each string. There are two possibilities:

<sup>11</sup> Contrast this with a *substring*, where the characters in the substring must all be adjacent in the original string.

<sup>12</sup> It is equally valid to start with the first character in each string, but we have chosen to start with the last character as it simplifies some implementation details when working with strings in real programming languages.

- The last character is the same in both strings, i.e.  $S_1[N] = S_2[M]$ ; call this  $x$ . Then  $x$  must be the last character of the LCS of  $S_1$  and  $S_2$ . We can prove this by contradiction. Let  $C = LCS(S_1, S_2)$ , and suppose that  $C$  does not end with  $x$ . Then it must consist of characters that appear in the first  $N - 1$  characters of  $S_1$  and the first  $M - 1$  characters of  $S_2$ . But  $C + x$  is a valid subsequence of both  $S_1$  and  $S_2$ , since  $x$  appears in  $S_1$  and  $S_2$  after all the characters in  $C$ . Since  $C + x$  is a longer common subsequence than  $C$ , this contradicts the assumption that the longest common subsequence does not end with  $x$ .

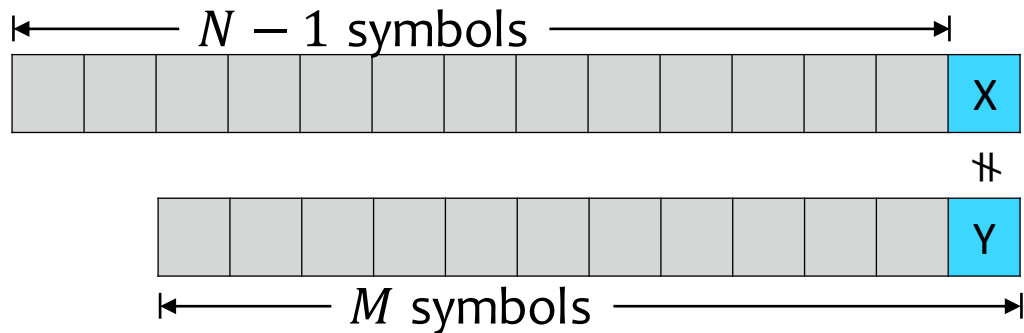
Let  $k$  be the length of  $C = LCS(S_1, S_2)$ . By similar reasoning to the above, we can demonstrate that  $C[1, \dots, k - 1] = LCS(S_1[1, \dots, N - 1], S_2[1, \dots, M - 1])$  – that is,  $C$  with its last character removed is the LCS of the strings consisting of the first  $N - 1$  characters of  $S_1$  and the first  $M - 1$  characters of  $S_2$ . This is an example of the principle of optimality, that we can construct an optimal solution to the larger problem from the optimal solutions to the smaller subproblems.



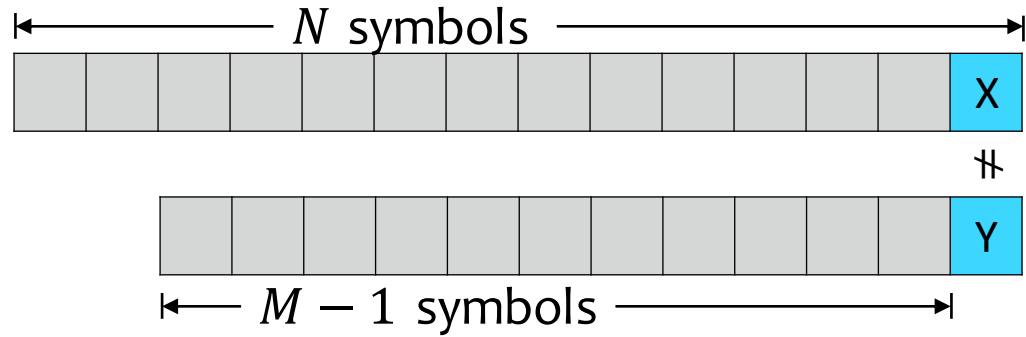
Let  $L(S_1, S_2)$  be the length of the LCS of  $S_1$  and  $S_2$ . We have demonstrated that when  $S_1[N] = S_2[M]$ , we have:

$$L(S_1, S_2) = L(S_1[1, \dots, N - 1], S_2[1, \dots, M - 1]) + 1.$$

- The last character differs between the two strings, i.e.  $S_1[N] \neq S_2[M]$ . Then at least one of these characters must not be in  $LCS(S_1, S_2)$ . We do not know a priori which of the two (or both) are not in the LCS, but we observe the following:
  - If  $S_1[N]$  is not in  $LCS(S_1, S_2)$ , then all the characters in  $LCS(S_1, S_2)$  must appear in the first  $N - 1$  characters of  $S_1$ . Then  $LCS(S_1, S_2) = LCS(S_1[1, \dots, N - 1], S_2)$ .



- If  $S_2[M]$  is not in  $LCS(S_1, S_2)$ , then all the characters in  $LCS(S_1, S_2)$  must appear in the first  $M - 1$  characters of  $S_2$ . Then  $LCS(S_1, S_2) = LCS(S_1, S_2[1, \dots, M - 1])$ .



Since at least one of these cases must hold, we just compute both  $LCS(S_1[1, \dots, N-1], S_2)$  and  $LCS(S_1, S_2[1, \dots, M-1])$  and see which one is longer. In terms of length alone, we have demonstrated that when  $S_1[N] \neq S_2[M]$ , we have that:

$$L(S_1, S_2) = \max(L(S_1[1, \dots, N-1], S_2), L(S_1, S_2[1, \dots, M-1]))$$

Finally, we note that when either  $S_1$  or  $S_2$  is empty (i.e.  $N = 0$  or  $M = 0$ , respectively), then  $LCS(S_1, S_2)$  is also empty. Putting everything together, we have:

$$L(S_1[1, \dots, N], S_2[1, \dots, M]) = \begin{cases} 0 & \text{if } N = 0 \text{ or } M = 0 \\ 1 + L(S_1[1, \dots, N-1], S_2[1, \dots, M-1]) & \text{if } S_1[N] = S_2[M] \\ \max(L(S_1[1, \dots, N-1], S_2), L(S_1, S_2[1, \dots, M-1])) & \text{if } S_1[N] \neq S_2[M] \end{cases}$$

Now that we have a recurrence relation, we can proceed to construct an algorithm. Let's take the bottom-up approach. We need a table to store the solutions to the subproblems, which consist of  $L(S_1[1, \dots, i], S_2[1, \dots, j])$  for all  $(i, j) \in [0, N] \times [0, M]$  (we consider  $S_1[1, \dots, 0]$  to denote an empty string). Thus, we need a table of size  $(N+1) \times (M+1)$ . The following is the table for  $S_1 = \text{"Go blue!"}$  and  $S_2 = \text{"Wolverines"}$ .

		W	o	l	v	e	r	i	n	e	s
G	0	0	0	0	0	0	0	0	0	0	0
o	0	0	0	0	0	0	0	0	0	0	0
b	0	0	0	1	1	1	1	1	1	1	1
l	0	0	0	1	1	1	1	1	1	1	1
u	0	0	0	1	2	2	2	2	2	2	2
e	0	0	0	1	2	2	3	3	3	3	3
!	0	0	0	1	2	2	3	3	3	3	3

The  $(i, j)$ th entry in the table denotes the length of the longest common subsequence for  $S_1[1, \dots, i]$  and  $S_2[1, \dots, j]$ . Using the recurrence relation, we can compute the value of the  $(i, j)$ th entry from the entries at locations  $(i-1, j-1)$ ,

$(i-1, j)$ , and  $(i, j-1)$ , with the latter two required when  $S_1[i] \neq S_2[j]$  and the former when  $S_1[i] = S_2[j]$ . The entry at  $(N, M)$  is the length of the LCS of the full strings  $S_1[1, \dots, N]$  and  $S_2[1, \dots, M]$ .

The last thing we need before proceeding to the algorithm is to determine an order in which to compute the table entries. There are multiple valid orders, but we will compute the entries row by row from top to bottom, moving left to right within a row. With this order, the entries required for  $(i, j)$  have already been computed by the time we get to  $(i, j)$ .

We can now write the algorithm for computing the table:



**Algorithm 31 (LCS Table)****Input:** strings  $S_1, S_2$ **Output:** table of LCS lengths for all prefixes  $S_1[1, \dots, i], S_2[1, \dots, j]$ **function** LCSTABLE( $S_1[1, \dots, N], S_2[1, \dots, M]$ )

allocate table[0, ..., N][0, ..., M]

**for**  $i = 0$  to  $N$  **do**        table[ $i$ ][0] = 0    **for**  $j = 0$  to  $M$  **do**        table[0][ $j$ ] = 0    **for**  $i = 1$  to  $N$  **do**        **for**  $j = 1$  to  $M$  **do**            **if**  $S_1[i] = S_2[j]$  **then**                table[ $i$ ][ $j$ ] = 1 + table[ $i - 1$ ][ $j - 1$ ]            **else**                table[ $i$ ][ $j$ ] = max{table[ $i - 1$ ][ $j$ ], table[ $i$ ][ $j - 1$ ]}    **return** table

As stated before, the length of  $LCS(S_1, S_2)$  is in the table entry at position  $(N, M)$ .

Now that we have determined how to compute the length of a longest common subsequence, let's return to the problem of computing such a subsequence itself. It turns out that we can *backtrack* through the table to recover the actual characters. We start with the last entry and determine the path to that entry from a base case. For each entry on the path, we check to see if the characters corresponding to that location match. If so, the matching character contributes to the LCS. If the characters do not match, we check to see if we came from the left or from above – the larger of the two neighboring entries is the source, since we took the max of the two in computing the current entry. If both neighbors have the same value, we can arbitrarily choose one or the other. The following demonstrates the path that results from backtracking. The solid path arbitrarily chooses to go up when both neighbors have the same value, while the dashed path chooses to go left. Both paths result in a valid LCS (and in this case, the same set of characters, though that isn't necessarily always the case).

		W	o	l	v	e	r	i	n	e	s
	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	0	0
o	0	0	1	1	1	1	1	1	1	1	1
	0	0	1	1	1	1	1	1	1	1	1
b	0	0	1	1	1	1	1	1	1	1	1
l	0	0	1	2	2	2	2	2	2	2	2
u	0	0	1	2	2	2	2	2	2	2	2
e	0	0	1	2	2	3	3	3	3	3	3
!	0	0	1	2	2	3	3	3	3	3	3

The algorithm for backtracking is as follows:

**Algorithm 32 (Longest Common Subsequence, via Backtracking)**

**Input:** strings  $S_1, S_2$

**Output:** a longest common subsequence of the strings

**function**  $\text{LCS}(S_1[1, \dots, N], S_2[1, \dots, M])$

$\text{table} = \text{LCSTABLE}(S_1, S_2)$

$s = \varepsilon$

▷ the empty string

$i = N, j = M$

**while**  $i > 0$  and  $j > 0$  **do**

**if**  $S_1[i] = S_2[j]$  **then**

$s = S_1[i] + s$

$i = i - 1, j = j - 1$

**else if**  $\text{table}[i][j - 1] > \text{table}[i - 1][j]$  **then**

$j = j - 1$

**else**

$i = i - 1$

**return**  $s$

How efficient is this algorithm? Computing a single table entry requires a constant number of operations. Since there are  $(N+1) \cdot (M+1)$  entries, constructing the table takes  $O(NM)$  time and requires  $O(NM)$  space. Backtracking also does a constant number of operations per entry on the path, and the path length is at most  $N + M + 1$ , so backtracking takes  $O(N + M)$  time. Thus, the total complexity of this algorithm is  $O(NM)$  in both time and space.

## 4.5 All-Pairs Shortest Paths

Suppose you are building a flight-aggregator website. Each day, you receive a list of flights from several airlines with their associated costs, and some flight segments may actually have negative cost if the airline wants to incentivize a particular route (see [hidden-city ticketing](https://en.wikipedia.org/wiki/Airline_booking_ploys#Hidden-city_ticketing)<sup>13</sup> for an example of how this can be exploited in practice, and what the perils of doing so are). You'd like your users to be able to find the cheapest itinerary from point A to point B. To provide this service efficiently, you determine in advance the cheapest itineraries between all possible origin and destination locations, so that you need only look up an already computed result when the user puts in a query.

This situation is an example of the *all-pairs shortest path* problem. The set of cities and flights can be represented as a graph  $G = (V, E)$ , with the cities represented as vertices and the flight segments as edges in the graph. There is also a weight function  $\text{weight} : E \rightarrow \mathbb{R}$  that maps each edge to a cost. While an individual edge may have a negative cost, no negative cycles are allowed. (Otherwise a traveler could just fly around that cycle to make as much money as they want, which would be very bad for the airlines!) Our task is to find the lowest-cost path between all pairs of vertices in the graph.

How can we apply dynamic programming to this problem? We need to formulate it such that there are self-similar subproblems. To gain some insight, we observe that many aggregators allow the selection of *layover* airports, which are intermediate stops between the origin and destination, when searching for flights. The following is an example from [kayak.com](https://kayak.com)<sup>14</sup>.

<sup>13</sup> [https://en.wikipedia.org/wiki/Airline\\_booking\\_ploys#Hidden-city\\_ticketing](https://en.wikipedia.org/wiki/Airline_booking_ploys#Hidden-city_ticketing)

<sup>14</sup> <https://kayak.com>

**Layover airports**

- ☒ Atlanta (ATL)
- ☐ Baltimore (DD8)
- ☒ Boston (BOS)
- ☒ Buffalo (BUF)
- ☒ Charlotte (CLT)
- ☒ Chicago (ORD)
- ☒ Cincinnati (CVG)
- ☒ Cleveland (CLE)
- ☒ Columbus (CMH)
- ☐ Denver (DEN)
- ☒ Detroit (DTW)
- ☒ Fort Lauderdale (FLL)
- ☒ Miami (MIA)
- ☒ Myrtle Beach (MYR)
- ☒ Norfolk (ORF)
- ☒ Orlando (MCO)

**Layover airports**

- ☒ Atlanta (ATL)
- ☐ Baltimore (DD8)
- ☐ Boston (BOS)
- ☐ Buffalo (BUF)
- ☐ Charlotte (CLT)
- ☐ Chicago (ORD)
- ☐ Cincinnati (CVG)
- ☐ Cleveland (CLE)
- ☐ Columbus (CMH)
- ☐ Denver (DEN)
- ☐ Detroit (DTW)
- ☐ Fort Lauderdale (FLL)
- ☐ Miami (MIA)
- ☐ Myrtle Beach (MYR)
- ☐ Norfolk (ORF)
- ☐ Orlando (MCO)

We take the set of allowed layover airports as one of the key characteristics of a subproblem – computing shortest paths with a smaller set of allowed layover airports is a subproblem of computing shortest paths with a larger set of allowed layover airports. Then the base case is allowing only direct flights, with no layover airports.

Coming back to the graph representation of this problem, we formalize the notion of a layover airport as an *intermediate vertex* of a simple path, which is a path without cycles. Let  $p = \{v_1, v_2, \dots, v_m\}$  be a path from origin  $v_1$  to destination  $v_m$ . Then  $v_2, \dots, v_{m-1}$  are intermediate vertices.

Assume that the vertices are labeled as numbers in the set  $\{1, 2, \dots, |V|\}$ . We parameterize a subproblem by  $k$ , which signifies that the allowed set of intermediate vertices (layover airports) is restricted to  $\{1, 2, \dots, k\}$ . Then we define  $d^k(i, j)$  to be the length of the shortest path between vertices  $i$  and  $j$ , where the path is only allowed to go through intermediate vertices in the set  $\{1, 2, \dots, k\}$ .

We have already determined that when no intermediate vertices are allowed, which is when  $k = 0$ , the shortest path between  $i$  and  $j$  is just the direct edge (flight) between them. Thus, our base case is

$$d^0(i, j) = \text{weight}(i, j)$$

where  $\text{weight}(i, j)$  is the weight of the edge between  $i$  and  $j$ .

We proceed to the recursive case. We have at our disposal the value of  $d^{k-1}(i', j')$  for all  $i', j' \in V$ , and we want to somehow relate  $d^k(i, j)$  to those values. The latter represents adding vertex  $k$  to our set of permitted intermediate vertices. There are two possible cases for the shortest path between  $i$  and  $j$  that is allowed to use any of the intermediate vertices  $1, 2, \dots, k$ :

- Case 1: The path does not go through  $k$ . Then the length of the shortest path that is allowed to go through  $1, 2, \dots, k$  is the same as that of the shortest path that is only allowed to go through  $1, 2, \dots, k-1$ , so  $d^k(i, j) = d^{k-1}(i, j)$ .
- Case 2: The path does go through  $k$ . Then this path is composed of two segments, one that goes from  $i$  to  $k$

and another that goes from  $k$  to  $j$ . We minimize the cost of the total path by minimizing the cost of each of the segments – the costs respect the principle of optimality.

Neither of the two segments may have  $k$  as an intermediate vertex – otherwise we would have a cycle. The only way for a path with a cycle to have lower cost than one without is for the cycle as a whole to have negative weight, which was explicitly prohibited in our problem statement. Since  $k$  is not an intermediate vertex in the segment between  $i$  and  $k$ , the shortest path between them that is allowed to go through intermediate vertices  $1, 2, \dots, k$  is the same as the shortest path that is only permitted to go through  $1, 2, \dots, k-1$ . In other words, the length of this segment is  $d^{k-1}(i, k)$ . By the same reasoning, the length of the segment between  $k$  and  $j$  is  $d^{k-1}(k, j)$ .

Thus, we have that in this case,  $d^k(i, j) = d^{k-1}(i, k) + d^{k-1}(k, j)$ .

We don't know a priori which of these two cases holds, but we can just compute them both and take the minimum. This gives us the recursive case:

$$d^k(i, j) = \min(d^{k-1}(i, j), d^{k-1}(i, k) + d^{k-1}(k, j))$$

Combining this with the base case, we have our complete recurrence relation:

$$d^k(i, j) = \begin{cases} \text{weight}(i, j) & \text{if } k = 0 \\ \min(d^{k-1}(i, j), d^{k-1}(i, k) + d^{k-1}(k, j)) & \text{if } k \neq 0 \end{cases}$$

We can now construct a bottom-up algorithm to compute the shortest paths:

#### Algorithm 33 (Floyd-Warshall)

**Input:** a weighted directed graph

**Output:** all-pairs (shortest-path) distances in the graph

**function** FLOYDWARSHALL( $G = (V, E)$ )

**for all**  $u, v \in V$  **do**

$d_0(u, v) = \text{weight}(u, v)$

**for**  $k = 1$  to  $|V|$  **do**

**for all**  $u, v \in V$  **do**

$d_k(u, v) = \min\{d_{k-1}(u, v), d_{k-1}(u, k) + d_{k-1}(k, v)\}$

**return**  $d_{|V|}$

This is known as the *Floyd-Warshall* algorithm, and it runs in time  $O(|V|^3)$ . The space usage is  $O(|V|^2)$  if we only keep around the computed values of  $d^m(i, j)$  for iterations  $m$  and  $m+1$ . Once we have computed these shortest paths, we need only look up the already computed result to find the shortest path between a particular origin and destination.

## GREEDY ALGORITHMS

A *greedy algorithm* computes a solution to an optimization problem by making (and committing to) a sequence of locally optimal choices. In general, there is no guarantee that such a sequence of locally optimal choices produces a global optimum. However, for some specific problems and greedy algorithms, we can prove that the result is indeed a global optimum.

As an example, consider the problem of finding a *minimum spanning tree (MST)* of a weighted, connected, undirected graph. Given such a graph, we would like to find a subset of the edges so that the subgraph induced by those edges touches every vertex, is connected, and has minimum total edge cost. This is an important problem in designing networks, including transportation and communication networks, where we want to ensure that there is a path between any two vertices while minimizing the overall cost of the network.

Before we proceed, let's review the definition of a *tree*. There are three equivalent definitions:

**Definition 34 (Tree #1)** An undirected graph  $G$  is a *tree* if it is connected and acyclic (i.e., has no cycle).

A graph is *connected* if for any two vertices there is a path between them. A *cycle* is a nonempty sequence of adjacent edges that starts and ends at the same vertex.

**Definition 35 (Tree #2)** An undirected graph  $G$  is a tree if it is *minimally connected*, i.e., if it is connected, and removing any single edge causes it to become disconnected.

**Definition 36 (Tree #3)** An undirected graph  $G$  is a tree if it is *maximally acyclic*, i.e., if it has no cycle, and adding any single edge causes it to have a cycle.

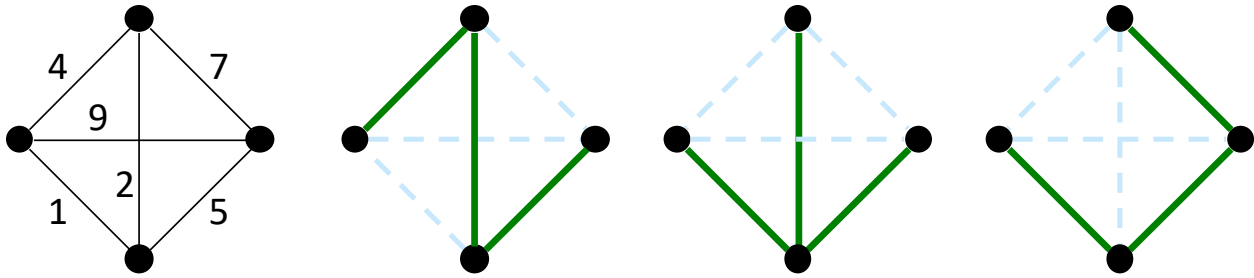
**Exercise 37** Show that the three definitions of a tree are equivalent.

**Definition 38 (Minimum spanning tree)** A *minimum spanning tree (MST)* of a connected graph is a subset of its edges that:

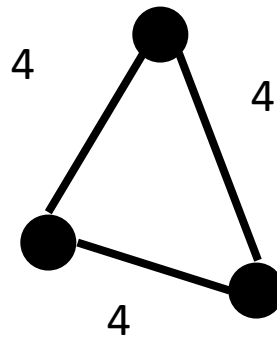
- connects all the vertices,
- is acyclic,
- and has minimum total edge weight over all subsets that meet the first two requirements.

The first two requirements imply that an MST (together with all the graph's vertices) is indeed a tree, by [Definition 34](#). Since the tree spans all the vertices of the graph, we call it a *spanning tree*. A *minimum spanning tree* is then a spanning tree that has the minimum weight over all spanning trees of the original graph.

The following illustrates three spanning trees of an example graph. The middle one is an MST, since its total weight of 8 is no larger than that of any other spanning tree.



A graph may have multiple minimum spanning trees (so we say “an MST,” not “the MST,” unless we have some specific MST in mind, or have a guarantee that there is a unique MST in the graph). In the following graph, any two edges form an MST.



Now that we understand what a minimum spanning tree is, let’s consider *Kruskal’s algorithm*, a greedy algorithm for computing an MST in a given graph. The algorithm simply examines the edges in sorted order by weight (from smallest to largest), selecting any edge that does not induce a cycle when added to the set of already-selected edges. It is “greedy” because it repeatedly selects an edge of minimum weight that does not induce a cycle (a locally optimal choice), and once it selects an edge, it never “un-selects” it (each choice is committed).

#### Algorithm 39 (Kruskal)

**Input:** a weighted, connected, undirected graph

**Output:** a minimum spanning tree of the graph

**function** KRUSKALMST( $G = (V, E)$ )

$S = \emptyset$

▷ empty set of edges

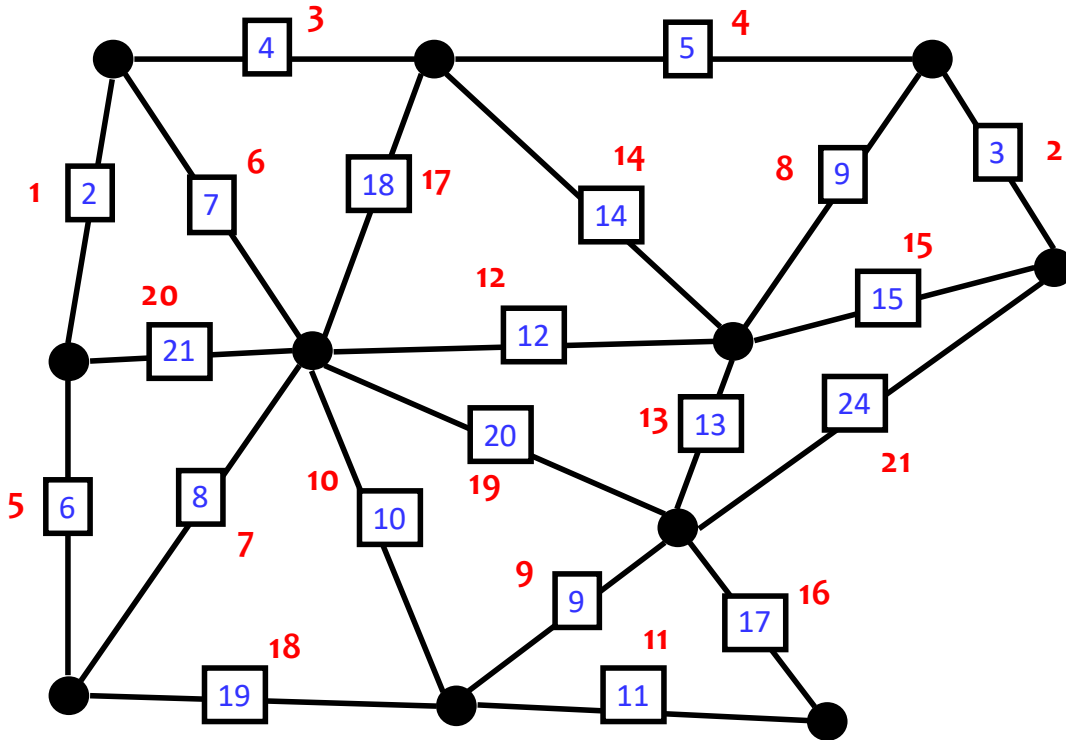
**for all** edges  $e \in E$ , in increasing order by weight **do**

**if**  $S \cup \{e\}$  does not have a cycle **then**

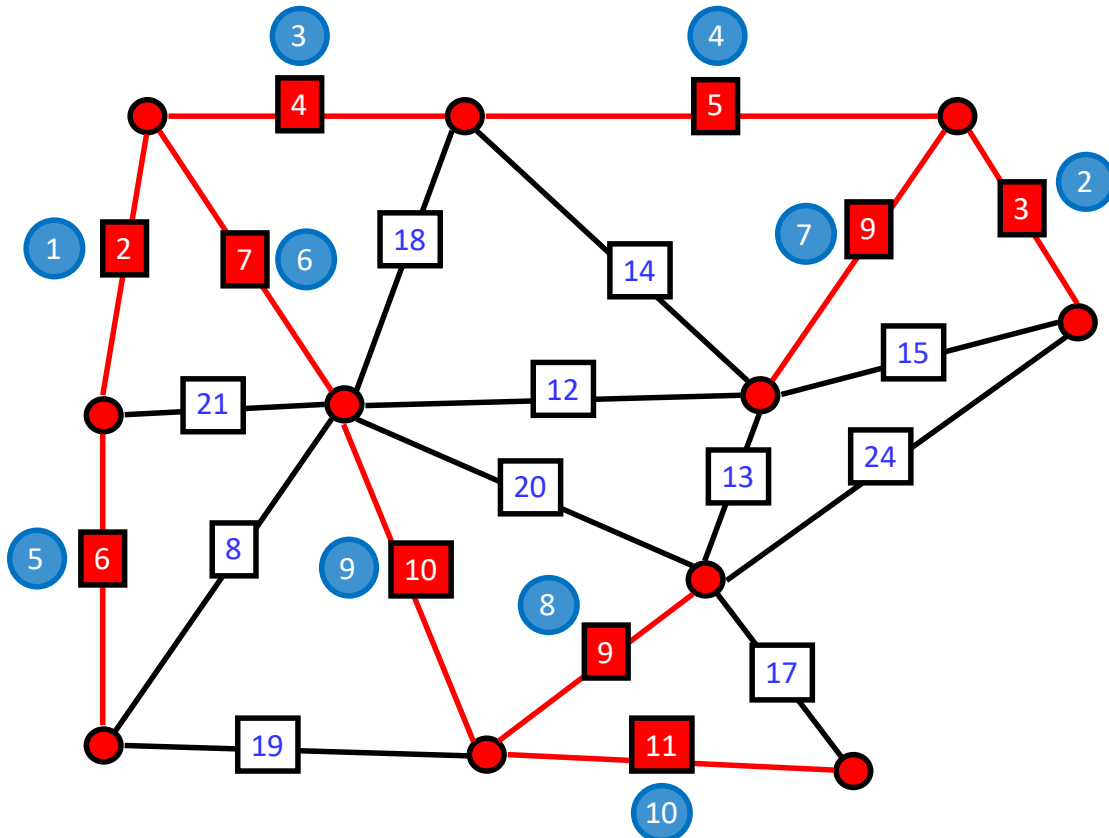
$S = S \cup \{e\}$

**return**  $S$

As an example, let’s see how Kruskal’s algorithm computes an MST of the following graph. We start by sorting the edges by their weights.



Then we consider the edges in order, including each edge in our partial result as long as adding it does not introduce a cycle among our selected edges.



Observe that when the algorithm considers the edge with weight 8, the two incident vertices are already connected by the already-selected edges, so adding that edge would introduce a cycle. Thus, the algorithm skips it and continues on to the next edge. In this graph, there are two edges of weight 9, so the algorithm arbitrarily picks one of them to consider first. The algorithm terminates when all edges have been examined. For this graph, the resulting spanning tree has a total weight of 66.

As stated previously, for many problems it is not the case that a sequence of locally optimal choices yields a global optimum. But as we will now show, for the MST problem, it turns out that Kruskal's algorithm does indeed produce a minimum spanning tree.

**Claim 40** *The output of Kruskal's algorithm is a tree.*

To prove this, we will assume for convenience that the input graph  $G$  is *complete*, meaning that there is an edge between every pair of vertices. We can make any graph complete by adding the missing edges with infinite weight, so that the new edges do not change the minimum spanning trees.

**Proof 41** Let  $T$  be the output of Kruskal's algorithm, which is the final value of  $S$ , on a complete input graph  $G$ . Recall from [Definition 36](#) that a tree is a maximally acyclic graph. Clearly  $T$  is acyclic, since Kruskal's algorithm initializes  $S$  to be the empty set, which is acyclic, and it adds an edge to  $S$  only if it does not induce a cycle.

So, it just remains to show that  $T$  is maximal. By definition, we need to show that for any potential edge  $e \notin T$  between two vertices of  $T$ , adding  $e$  to  $T$  would introduce a cycle. The input graph is complete, and the algorithm examined each of its edges, so it must have considered  $e$  at some point. Because  $T$  does not contain  $e$ , and no edge was ever removed from  $S$ , the algorithm did not add  $e$  to  $S$  when it considered  $e$ . Recall that when the algorithm considers an edge, it leaves it out of  $S$  only if it would create a cycle in  $S$  (at that time). So, since it did not add  $e$  to  $S$ , doing so would have induced a cycle at the time. And since no edges are ever removed from  $S$ , adding  $e$  to the final output  $T$  also would induce a cycle, which is what we set out to show. Thus,  $T$  is maximal, as desired.  $\square$

We can similarly demonstrate that the output of Kruskal's algorithm is a *spanning* tree, but we leave that as an exercise.

**Exercise 42** Show that if the input graph  $G$  is connected, then the output of Kruskal's algorithm spans all the vertices of  $G$ .

Next, we show that the result of Kruskal's algorithm is a minimum spanning tree. To do so, we actually prove a stronger claim:

**Claim 43** *At any point in Kruskal's algorithm on an input graph  $G$ , let  $S$  be the set of edges that have been selected so far. Then there is some minimum spanning tree of  $G$  that contains  $S$ .*

Since this claim holds at any point in Kruskal's algorithm, in particular it holds for the final output set of edges  $T$ , i.e., there is an MST that contains  $T$ . Since we have already seen above that  $T$  is a spanning tree, no edge of the graph can be added to  $T$  without introducing a cycle, so we can conclude that the MST containing  $T$  as guaranteed by [Claim 43](#) must be  $T$  itself, and hence  $T$  is an MST.

**Proof 44** We prove [Claim 43](#) by induction over the size of  $S$ , i.e., the sequence of edges added to it by the algorithm.

- **Base case:**  $S = \emptyset$  is the empty set. Every MST trivially contains  $\emptyset$  as a subset, so the claim holds.
- **Inductive step:** Let  $T$  be an MST that contains  $S$ . Suppose the algorithm would next add edge  $e$  to  $S$ . We need to show that there is some MST  $T'$  that contains  $S \cup \{e\}$ .

There are two possibilities: either  $e \in T$ , or not.

- Case 1:  $e \in T$ . The claim follows immediately: since  $T$  is an MST that contains  $S$ , and also  $e \in T$ , then  $T$  is an MST that contains  $S \cup \{e\}$ . (In other words, we can take  $T' = T$  in this case.)
- Case 2:  $e \notin T$ . Then by [Definition 36](#),  $T \cup \{e\}$  contains some cycle  $C$ , and  $e \in C$  because  $T$  alone



is acyclic.

By the code of Kruskal’s algorithm, we know that  $S \cup \{e\}$  does not contain a cycle. Thus, there must be some edge  $f \in C$  for which  $f \notin S \cup \{e\}$ , and in particular,  $f \neq e$ . Since  $f \in C \subseteq T \cup \{e\}$ , we have that  $f \in T$ .

Observe that  $S \cup \{f\} \subseteq T$ , since  $S \subseteq T$  by the inductive hypothesis. Since  $T$  does not contain a cycle, neither does  $S \cup \{f\}$ .

Since adding  $f$  to  $S$  would not induce a cycle, the algorithm *must not have considered  $f$  yet* (at the time it considers  $e$  and adds it to  $S$ ), or else it would have added  $f$  to  $S$ . Because the algorithm considers edges in sorted order by weight, and it has considered  $e$  but has not considered  $f$  yet, it must be that  $w(e) \leq w(f)$ .

Now define  $T' = T \cup \{e\} \setminus \{f\}$ , which has weight  $w(T') = w(T) + w(e) - w(f) \leq w(T)$ . Moreover,  $T'$  is a spanning tree of  $G$ : for any two vertices, there is a path between them that follows such a path in  $T$ , but instead of using edge  $f$  it instead goes the “other way around” the cycle  $C$  using the edges of  $C \setminus \{f\} \subseteq T'$ .

Since  $T'$  is a spanning tree whose weight is no larger than that of  $T$ , and  $T$  is an MST,  $T'$  is also an MST.<sup>15</sup> And since  $S \subseteq T$  and  $f \notin S$ , we have that  $S \cup \{e\} \subseteq T \cup \{e\} \setminus \{f\} = T'$ . Thus,  $T'$  is an MST that contains  $S \cup \{e\}$ , as needed.

<sup>15</sup> This actually shows that  $w(T') = w(T)$  and hence  $w(e) = w(f)$ , but these facts are not needed for this proof.

We have proved that Kruskal’s algorithm does indeed output an MST of its input graph. We also state without proof that its running time on an input graph  $G = (E, V)$  is  $O(|E| \log |E|)$ , by using an appropriate choice of supporting data structure to detect for cycles.

Observe that the analysis of Kruskal’s algorithm is nontrivial. This is often the case for greedy algorithms. Again, it is usually not the case that locally optimal choices lead to a global optimum, so we typically need to do significant work to demonstrate that this is actually the case for a particular problem and algorithm.

A standard strategy for proving that a greedy algorithm correctly solves a particular optimization problem proceeds by establishing a “greedy choice” property, often by means of an exchange-style argument like the one we gave above for MSTs. A “greedy choice” property consists of two parts:

- Base case: the algorithm’s initial state is contained in *some* optimal solution. Since a typical greedy algorithm’s initial state is the empty set, this property is usually easy to show.
- Inductive step: for any (greedy) choice the algorithm makes, there *remains* an optimal solution that contains the algorithm’s set of choices so far. (Observe that [Claim 43](#) has this form.)

With a greedy-choice property established, by induction, the algorithm’s set of choices is at all times contained in *some* optimal solution. So, it just remains to show that the final output consists of a full solution (not just a partial one); it therefore must be an optimal solution.

To show the inductive step of the greedy-choice property, we have the inductive hypothesis that the previous choices  $S$  are contained in some optimal solution  $\text{OPT}$ , and we need to show that the updated choices  $S \cup \{s\}$  are contained in some optimal solution  $\text{OPT}'$ . If  $s \in \text{OPT}$ , then the claim holds trivially. But if  $s \notin \text{OPT}$ , then we aim to invoke some exchange argument, which modifies  $\text{OPT}$  to some *other* optimal solution  $\text{OPT}'$  that contains  $S \cup \{s\}$ . Indeed, this is exactly what we did in the correctness proof for Kruskal’s argument, by changing  $\text{OPT} = T$  to  $\text{OPT}' = T' = T \cup \{e\} \setminus \{f\}$ , for some carefully identified edge  $f \in T$  whose weight is no smaller than that of  $e$ . This means that  $\text{OPT}'$  is also an optimal solution, as needed.