Essential Search Algorithms: Navigating the Digital Maze

Muthukrishnan

Copyright

Cover Image Copyright 2023 Muthukrishnan. All Rights Reserved

The triangle featured on the cover page is known as the Penrose Triangle, also referred to as the impossible triangle. It is an optical illusion that depicts a three-dimensional figure in the form of a triangular loop, which, despite its appearance, cannot be constructed in a physical three-dimensional space. It was created by a Swedish artist Oscar Reutersvärd in the 1930s and was later brought to prominence by mathematician Roger Penrose in the 1950s. Although the Penrose Triangle cannot be realized as a physical object, it represents an intriguing example of how visual perception can be deceived to see an ostensibly impossible form. It has emerged as an iconic figure in the realm of optical illusions and is frequently employed to demonstrate ideas related to paradoxes and the visual perception in the fields of art and design.

Essential Search Algorithms: Navigating the Digital Maze
Copyright 2023 Muthukrishnan. All Rights Reserved

No part of this publication may be reproduced or transmitted in any form whatsoever, electronic, or mechanical, including photocopying, recording, or by any informational storage or retrieval system without express written, dated and signed permission from the author.

Preface

Ever wonder how Google finds your perfect search result in milliseconds? Or how your GPS maps the fastest route to your destination? It's all thanks to the clever search algorithms invented by great software engineers who needed to solve some complex search problems.

Search algorithms are everywhere! From finding your friend's profile on Facebook to suggesting the best route on a map, these algorithms are constantly sifting through mountains of data, searching for what you're looking for. Search is at the core of all these real-world technology problems.

This book is your guide to essential search methods used across different fields. I've carefully curated these algorithms from a vast collection, selecting those that form the building blocks. Understanding and applying these algorithms will provide you with the necessary baseline to explore more complex ones. Plus, you might even be able to create your own algorithm tailored to the problem you're solving. Each section explains the ideas clearly and includes Python code you can use in your projects. All the code is free and available on my GitHub page under the MIT license.

By the end, you'll have the skills to explore both simple and complex decision-making. Whether it's managing supply chains or designing smart robots, these essential search methods will guide you through tough challenges and help you discover solutions you didn't know were there.

So, join me on a journey to understand how search works. Let's get started!

Muthukrishnan
Bangalore, India
2023

For my parents, Amrita and Atharva

Table of contents

Introduction **17**
- Book Organization 18
- Code Samples 19

Setting up the python enviroment **20**
- Online resources 21

1. Time Complexity of Algorithms **22**
- 1.1. Big O Notation 22
- 1.2. Common Notations 23
- 1.3. Calculating Time complexity 24
- 1.4. Demystifying Logarithmic Time complexity 26
- 1.5. Common Time Complexities 26
- 1.6. NP and P 27
- 1.7. Bibliography 28

I. Basic Search Algorithms **29**

2. Linear search **30**
- 2.1. How it works 30
- 2.2. Implementation 31
- 2.3. Time Complexity Analysis 32
- 2.4. Advantages 32
- 2.5. Limitations 32
- 2.6. Applications 33
- 2.7. Bibliography 33

3. Binary Search **34**
- 3.1. How it works 35
- 3.2. Implementation 35
- 3.3. Time Complexity Analysis 37

3.4. Advantages . . . 37
3.5. Limitations . . . 37
3.6. Applications . . . 37
3.7. Bibliography . . . 38

4. Ternary Search 39
4.1. How it Works . . . 39
4.2. Implementation . . . 41
4.3. Time Complexity Analysis . . . 42
4.4. Applications . . . 43
4.5. Bibliography . . . 43

5. Jump Search 44
5.1. How it Works . . . 45
5.2. Implementation . . . 46
5.3. Time Complexity Analysis . . . 47
5.4. Advantages . . . 47
5.5. Limitations . . . 47
5.6. Applications . . . 47
5.7. Bibliography . . . 47

6. Indexed Sequential Search 48
6.1. How it works . . . 49
6.2. Implementation . . . 50
6.3. Time Complexity Analysis: . . . 52
6.4. Advantages . . . 52
6.5. Limitations . . . 52
6.6. Applications . . . 52
6.7. Bibliography . . . 53

7. Interpolation Search 54
7.1. How it works . . . 55
7.2. Implementation . . . 56
7.3. Time Complexity Analysis . . . 57
7.4. Advantages and Limitations . . . 57
7.5. Applications . . . 58
7.6. Bibliography . . . 59

8. Exponential Search 60
8.1. How it works 60
8.2. Implementation 62
8.3. Time Complexity Analysis 63
8.4. Advantages 63
8.5. Limitations 64
8.6. Applications 64
8.7. Bibliography 65

9. Fibonacci Search 66
9.1. How it works 66
9.2. Implementation 69
9.3. Time Complexity Analysis: 71
9.4. Advantages 72
9.5. Limitations 72
9.6. Applications 72
9.7. Bibliography 72

II. Hash-Based Search 73

10. Basics 74
10.1. Hash table 75
10.2. How Hashing Works 75
10.3. Types of Hash Functions 76
10.4. Hash Collisions 79
10.5. Bibliography 80

11. Hash Table Search 82
11.1. How it works 83
11.2. Implementation 83
11.3. Time Complexity Analysis 85
11.4. Advantages 85
11.5. Limitations 85
11.6. Applications 86
11.7. Bibliography 86

12. Bloom Filter 87
12.1. How it Works 88

12.2. Implementation 89
12.3. Visualization 91
12.4. Time complexity Analysis 93
12.5. Advantages 94
12.6. Limitations 94
12.7. Applications 94
12.8. Bibliography 95

13. Cuckoo Filter 96
13.1. Cuckoo hashing 97
13.2. Cuckoo Filter implementation 102
13.3. Time complexity Analysis 106
13.4. Cuckoo filters vs Bloom Filters 108
13.5. Advantages: 108
13.6. Limitations: 109
13.7. Bibliography 109

III. AI Search Algorithms 110

14. Hill Climbing Algorithm 111
14.1. How it works 112
14.2. Implementation 115
14.3. Variants of Hill Climbing Algorithm 117
14.4. Advantages 118
14.5. Disadvantages 118
14.6. Applications 119
14.7. Bibliography 119

15. Monte Carlo Tree Search (MCTS) 120
15.1. Playing Tic-Tac-Toe with Monte Carlo Tree Search 122
15.2. Implementation 125
15.3. Visualization 130
15.4. Advantages: 132
15.5. Limitations: 132
15.6. Applications 133
15.7. Bibliography 134

16. Simulated Annealing 135
16.1. How it works: . . . 135
16.2. The Traveling Salesman Problem (TSP) using Simulated Annealing . . . 137
16.3. Simulated Annealing for TSP Problem: Step-by-Step Guide 138
16.4. Implementation . . . 140
16.5. Visualization . . . 143
16.6. Advantages: . . . 146
16.7. Limitations: . . . 147
16.8. Applications . . . 148
16.9. Bibliography . . . 149

17. Tabu search 150
17.1. Basic principle . . . 151
17.2. Components of TS . . . 151
17.3. How it works . . . 154
17.4. Implementation . . . 156
17.5. Visualization . . . 160
17.6. Advantages . . . 164
17.7. Limitations . . . 165
17.8. Applications . . . 166
17.9. Bibliography . . . 166

18. Branch and Bound Algorithms 167
18.1. Branch and Bound algorithm . . . 168
18.2. How it works . . . 169
18.3. Implementation . . . 175
18.4. Advantages: . . . 181
18.5. Limitations: . . . 181
18.6. Applications . . . 182
18.7. Bibliography . . . 183

19. Beam search 184
19.1. How it works: . . . 185
19.2. Implementation . . . 185
19.3. Visualization . . . 188
19.4. Time Complexity Analysis . . . 191
19.5. Advantages . . . 193
19.6. Limitations . . . 193

19.7. Applications 193
19.8. Bibliography 194

20. Iterative Deepening Depth-First Search (IDDFS) 196
20.1. How it works: 197
20.2. Implementation 198
20.3. Time Complexity Analysis 200
20.4. Advantages 201
20.5. Limitations 202
20.6. Applications 202
20.7. Bibliography 203

21. Ant Colony Optimization (ACO) 204
21.1. Explanation of Ant Colony Optimization 205
21.2. Implementation 207
21.3. Visualization 211
21.4. Advantages 213
21.5. Limitations 214
21.6. Applications 214
21.7. Bibliography 215

22. Nearest Neighbor Search 216
22.1. Formal Definition 216
22.2. Approaches to Nearest Neighbor Search: 217
22.3. Curse of Dimensionality 227
22.4. Time complexity analysis 228
22.5. Advantages 229
22.6. Limitations 229
22.7. Applications 230
22.8. Bibliography 231

IV. Graph Search Algorithms 232

23. Basics 233
23.1. Graph Data structure 233
23.2. Common Implementations 234
23.3. Types of Graph-based algorithms 238
23.4. Search Strategies: Uninformed and Informed Search 238

23.5. Bibliography 240

24. Depth First Search (DFS) 241

24.1. How it Works 242
24.2. Implementation 244
24.3. Time Complexity Analysis: 246
24.4. Advantages 246
24.5. Limitations 247
24.6. Choosing between DFS and BFS 247
24.7. Applications 248
24.8. Bibliography 250

25. Breadth first search (BFS) 251

25.1. How it works 252
25.2. Implementation 253
25.3. Visualization 255
25.4. Time Complexity Analysis 257
25.5. Breadth-First Search vs. Depth-First Search: 257
25.6. Advantages 259
25.7. Limitations 259
25.8. Applications 260
25.9. Bibliography 262

26. Dijkstra's algorithm 263

26.1. A brief history 263
26.2. How it Works 264
26.3. Implementation 265
26.4. Visualization 268
26.5. Time Complexity Analysis 278
26.6. Advantages 279
26.7. Limitations 279
26.8. Applications 280
26.9. Bibliography 281

27. Uniform Cost Search 282

27.1. How it Works 282
27.2. Implementation 283
27.3. Visualization 287
27.4. Time Complexity Analysis 291

27.5. Advantages 292
27.6. Disadvantages 292
27.7. Applications 293
27.8. Bibliography 293

28. Greedy Best-First Search 294
28.1. How it works 295
28.2. Implementation 296
28.3. Visualization 298
28.4. Time Complexity Analysis 300
28.5. Advantages 301
28.6. Limitations 301
28.7. Applications 302
28.8. Bibliography 303

29. A* Search Algorithm 304
29.1. Heuristics and Cost function 304
29.2. How it works 306
29.3. Implementation 308
29.4. Visualization 312
29.5. Time Complexity Analysis: 313
29.6. Advantages 314
29.7. Limitations 314
29.8. Applications: 315
29.9. Bibliography 315

30. B* Search Algorithm 316
30.1. How it works: 316
30.2. Implementation 318
30.3. Bibliography 320

31. Bidirectional Search 321
31.1. How it Works 322
31.2. Implementation 322
31.3. Time Complexity Analysis 324
31.4. Advantages 324
31.5. Limitations 325
31.6. Applications 326
31.7. Bibliography 327

32. Kruskal's Algorithm **328**
32.1. Minimal Spanning Tree 328
32.2. Disjoint Set Data structure 329
32.3. The Kruskal Algorithm 330
32.4. Implementation 331
32.5. Visualization 332
32.6. Time complexity analysis 336
32.7. Advantages 336
32.8. Limitations 337
32.9. Algorithms 337
32.10.Bibliography 338

33. Prim's Algorithm **339**
33.1. How it works 339
33.2. Implementation 340
33.3. Visualization 342
33.4. Time Complexity Analysis 344
33.5. Advantages 345
33.6. Limitations 345
33.7. Applications 346
33.8. Bibliography 346

34. Floyd-Warshall algorithm **347**
34.1. How it works 347
34.2. Implementation 351
34.3. Time Complexity Analysis 352
34.4. Applications 353
34.5. Bibliography 354

35. The Bellman-Ford algorithm **355**
35.1. How it works: 356
35.2. Implementation 360
35.3. Time Complexity Analysis 362
35.4. Applications 362
35.5. Bibliography 362

V. Text and String Matching Algorithms 363

36. Suffix Array 364
36.1. How it works 364
36.2. Implementation 365
36.3. Visualization 366
36.4. Time complexity 369
36.5. Advantages 370
36.6. Limitations 370
36.7. Applications 371
36.8. Bibliography 371

37. Boyer-Moore Algorithm 373
37.1. How it works 373
37.2. Implementation 379
37.3. Visualization 381
37.4. Advantages 383
37.5. Limitations 384
37.6. Applications 384
37.7. Bibliography 384

38. Knuth-Morris-Pratt (KMP) Algorithm 385
38.1. How it works 385
38.2. Implementation 389
38.3. Visualization 391
38.4. Time Complexity Analysis 393
38.5. Advantages 394
38.6. Limitations 394
38.7. Applications 395
38.8. Bibliography 395

39. Rabin-Karp Algorithm 396
39.1. How it works 396
39.2. Implementation 399
39.3. Visualization 401
39.4. Time complexity 401
39.5. Advantages 402
39.6. Limitations 403
39.7. Bibliography 404

40. Levenshtein distance **405**
40.1. Calculating Levenshtein Distance 405
40.2. Representation of Levenshtein Distance 406
40.3. Implementation 406
40.4. Visualization 407
40.5. How is LD used in search? 409
40.6. Program for approximate string matching using levenshtein distance 410
40.7. Advantages 412
40.8. Limitations 412
40.9. Applications 413
40.10. Bibliography 414

VI. Data Structures for Search **415**

41. Arrays **416**
41.1. Arrays in Python 416
41.2. Types of Arrays 419
41.3. Applications 419

42. Linked Lists **421**
42.1. Implementing a linked list in Python 421
42.2. Operations on linked lists 422
42.3. Advantages of linked lists 422
42.4. Disadvantages 422

43. Queues **423**
43.1. Working with Queues 423
43.2. Applications 424

44. Heaps **426**
44.1. Types of heaps 426
44.2. Working with Heap 427
44.3. Heaps vs. arrays 428
44.4. Applications 428

45. Stack **429**
45.1. Working with Stack 429

45.2. Applications 430

46. Trie search **432**
46.1. How it works 433
46.2. Implementation 437
46.3. Time Complexity Analysis 439
46.4. Advantages 439
46.5. Limitations 440
46.6. Applications 440

47. Ternary Search Tree (TST) **442**
47.1. Structure and Representation of Ternary Search Tree . . . 442
47.2. Operations on Ternary Search Tree 443
47.3. Implementation 445
47.4. Visualization 446
47.5. How is a TST more space efficient than Trie? 451
47.6. Bibliography 453

VII. Appendix **454**

48. Python: Quick-Start Guide **455**
48.1. Overview 455
48.2. Language Basics 458
48.3. Further Reading 466

49. Acknowledgements **468**

Introduction

Welcome to *Essential Search Algorithms.* As a software engineer with over 15 years of writing code and building complex software, I've amassed a vast library of algorithms books. Many of these delve into algorithms in a textbook-theoretical style, making the concepts overly complex and challenging to grasp. This often required me to read even more books to understand a single algorithm. It was then that I began developing my own personal reference guide: a concise and visually structured collection of essential algorithm elements. These notes initially served as a personal reference, but as they grew, I realized the need wasn't mine alone. Many fellow software engineers needed the same clarity. So, I started writing this guidebook for a wider audience, focusing on the essentials and leaving the textbook jargon behind.

This book stands out from other search algorithm books in several key ways:

1. **Clear and concise explanations:** Complex concepts are explained in a clear and concise manner without complex mathematical notations, avoiding textbook-theoretical style, making the book accessible to a wide audience, from beginners to experienced programmers.

2. **Visualizations and Working code:** This book is packed with visualizations that make even complex algorithms easy to understand and remember. Additionally, all algorithms include a functional Python code, giving readers a firsthand understanding of their functionality and inner workings.

3. **Suitable for Self-Study or Classroom Use:** The book is suitable for both self-study and classroom use. It also serves as an excellent resource for instructors teaching courses on search algorithms.

4. **Real world Applications:** Knowing how search algorithms are used in real-world applications, when learning them, makes you

better equipped to pick the right one for solving problems. That's why each algorithm comes with a curated list of real-world uses.

5. **Bibliography:** This book wouldn't have been possible without the incredible work of researchers, writers, and engineers who generously shared their knowledge with the world. Each algorithm in this book has a dedicated bibliography, which I personally consulted while writing that chapter. This section not only includes relevant books and additional reading materials but also, in some chapters, the original papers authored by the algorithms' creators. This inclusion allows readers to delve deeper into the origins, intricacies, and advanced applications of each algorithm. It provides a comprehensive understanding and facilitates further exploration beyond the book's scope.

Whether you're an experienced programmer or a student seeking a deeper understanding of essential search algorithms, this book will serve as your go-to guide.

Book Organization

This book is organized into six parts:

1. Basic Search Algorithms

This introductory section delves into the fundamental search algorithms in computer science that are like the base for understanding more complicated ways of searching and moving through information. Once you understand these simple search algorithms well, you'll have a strong base to understand fancier ways of searching and working with data.

2. Hash-Based Search Algorithms

This part shows how hash-based search algorithms work using special functions called hash functions to link keys or IDs to exact spots in a data structure, usually a hash table.

3. Graph Search Algorithms

This part covers graph-based search algorithms, which are essential for navigating and analyzing interconnected data structures like graphs or networks.

4. Text and String Matching Algorithms

This part covers Text and String Matching Algorithms, which are fundamental techniques used to identify patterns or specific sequences within

text or data strings. These algorithms excel at detecting patterns, identifying specific substrings, and locating occurrences of target text within larger bodies of text, proving invaluable for tasks such as plagiarism detection, information retrieval, and natural language processing.

5. AI Search Algorithms

This part delves into AI Search Algorithms, fundamental tools used in artificial intelligence to find solutions or make decisions in various problems by exploring possible paths or states.

6. Data Structures for Search

This part looks at how data structures and search algorithms work together. Well-thought-out data structures can boost how well searches work. Whether it's arrays or tries, each type of data structure has special features that can make searches faster and more efficient.

Code Samples

Every code example in this book is written in Python. I've been writing code in Java, JavaScript, C#, and Python for many years, and I think Python is really versatile and easy to understand. Good Python code is clear and easy to read because it's written in an expressive way. Even if you're new to Python, you can learn the basics in a few hours and begin writing and understanding code.

The provided sample codes demonstrate the functionality of each technique, enabling readers to engage with and fully comprehend the methodology. The sample code is intentionally minimal to keep the book focused more on the concept than the implementation, providing ample room for extension and optimization. For those eager to dive deeper, the complete source code for all algorithms featured in this book is accessible on my GitHub repository at https://github.com/muthuspark/algorithms. It's worth noting that all algorithm implementations were rigorously tested with Python 3.

Setting up the python enviroment

To get started, follow these steps to install Python on Windows, macOS, and Linux operating systems.

Python Installation on macOS:

1. **Check Pre-installed Python:** macOS often comes with a pre-installed version of Python. Open Terminal and enter `python3 --version` to check if Python 3 is already available.

2. **Homebrew Installation (Optional):** If you prefer package managers, you can install Python using Homebrew. Run the command: `brew install python`.

3. **Official Python Installer:** Download the latest version of Python from the official Python website . Run the installer, follow the prompts, and Python will be ready for action.

Python Installation on Linux:

1. **Package Manager Installation:** Linux distributions usually offer Python in their package managers. For Debian-based systems like Ubuntu, use: `sudo apt-get install python3`. For Red Hat-based systems, try: `sudo yum install python3`.

2. **Python.org Installation:** Alternatively, you can install Python directly from the source. Visit the official website to download the source code. Extract it, navigate to the directory, and execute the commands: `./configure`, `make`, and `sudo make altinstall`.

Python Installation on Windows:

1. **Official Python Installer:** Download the Windows installer from the official Python website . Run the installer, ensure the “Add Python X.Y to PATH” option is checked, and click “Install Now.”

2. **Windows Store Installation (Optional):** If you're using Windows 10, Python is available through the Microsoft Store. Search for "Python" in the Store and install the version you need.

3. **Anaconda Distribution (Optional):** For data science and scientific computing, the Anaconda distribution Anaconda Distribution is an excellent choice. Download the installer, run it, and follow the installation instructions.

Verifying the Installation:

To verify that Python is successfully installed, open your terminal (or command prompt) and enter `python3 --version` (macOS and Linux) or `python --version` (Windows). You should see the installed Python version displayed.

Online resources

- Python Documentation

1. Time Complexity of Algorithms

Time complexity, in computer science, is a way of measuring how long it takes an algorithm to run as the size of the input increases. It's like a speedometer for algorithms, telling you how fast they can process information based on how much information they need to handle.

Imagine you have two different algorithms for searching a library for a specific book. One method, known as linear search, involves checking each book one by one from the first shelf to the last. In another method, knpws as Indexed Sequential Search, you quickly navigate to the section or shelf where the book is likely to be located and then find the book in that section. As you can imagine, the latter method, Indexed Sequential Search, would be significantly faster, especially as the library grows larger and the books increase.

Time complexity helps us understand this difference by analyzing how the running time of an algorithm changes with the input size. One of the most commonly used notation for denoting the time complexity of an algorithm or a part of code is the Big-O notation.

1.1. Big O Notation

Using Big O notation, we express the growth rate of an algorithm's runtime in terms of a function of the input size. The "O" in Big O stands for "order of," and the notation is followed by a mathematical function that represents the algorithm's behavior as the input size approaches infinity. Think of Big O as a way to describe the upper bound of an algorithm's growth rate. It provides a high-level view of how an algorithm's efficiency scales with input size, abstracting away constant factors and lower-order terms.

For example, an algorithm with a time complexity of $O(n)$ means that its runtime grows linearly with the input size, while $O(n^2)$ suggests a quadratic relationship. The "n" in these notations represents the input size.

1.2. Common Notations

Notation	Meaning	Interpretation
Big O (O)	Upper bound on the worst-case execution time	The algorithm's execution time will not grow faster than the specified function as the input size increases.
Theta (Θ)	Average-case execution time	The algorithm's execution time will be exactly proportional to the specified function as the input size increases.
Omega (Ω)	Lower bound on the worst-case execution time	The algorithm's execution time will not grow slower than the specified function as the input size increases.
Little Oh (o)	Asymptotic upper bound on the execution time	The algorithm's execution time will eventually be less than the specified function as the input size increases.
Little Omega (ω)	Asymptotic lower bound on the execution time	The algorithm's execution time will eventually be greater than the specified function as the input size increases.

1.3. Calculating Time complexity

Calculating time complexity involves identifying the dominant operations in the algorithm and determining how their execution time scales with the input size.

1.3.1. Calculating time complexity of Linear Search

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
```

1. **Identify the basic operations:** Break down the algorithm into basic operations, such as assignments, comparisons, function calls, and loop iterations. The basic operations are assignments ($i = 0$), comparisons ($arr[i] == target$), and function calls (`return i` or `return -1`).

2. **Count the operations:** Determine the number of times each basic operation is executed. This may involve using variables to represent the input size and the number of iterations. The `for` loop iterates `len(arr)` times, performing one assignment and one comparison per iteration. The return statement is executed either once (if the target is found) or not at all. The other operations are constant and can be ignored.

3. **Simplify expression:** $O(n)$

4. **Use Big O notation:** The time complexity of linear search is $O(n)$, where n is the input size ($len(arr)$).

1.3.2. Calculating time complexity of Binary Search

Binary search is a search algorithm that works by repeatedly dividing the input into half and checking which half contains the target value. This process is repeated until the target value is found or it is determined that the value is not present in the input.

```python
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1
```

Here is how to calculate the time complexity of binary search:

1. **Identify the basic operations:** The basic operations are assignments (`low = 0, high = len(arr) - 1, mid = (low + high) // 2, return mid`), comparisons (`arr[mid] == target, arr[mid] < target, arr[mid] > target`), and function calls (`len(arr)`).

2. **Count operations:** The number of comparisons is equal to the number of times the while loop iterates. In the worst case, the loop iterates until the target value is found, which means that the number of comparisons is $log_2(n)$. The other operations are constant and can be ignored.

3. **Simplify expression:** $O(log(n))$

4. **Use Big O notation:** $O(log(n))$

Therefore, the time complexity of binary search is $O(log(n))$. This means that the number of operations required to perform the search will grow logarithmically with the input size.

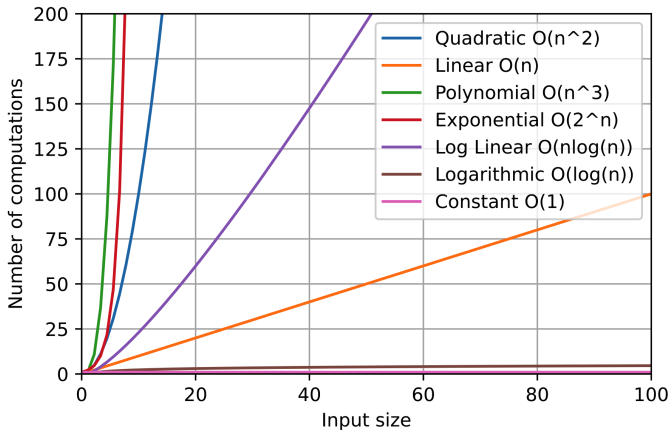
1.4. Demystifying Logarithmic Time complexity

The mathematical notation "log(n)" (where n is any positive real number) represents the logarithm of n to the base 2. In simpler terms, it indicates the power to which 2 must be raised to obtain the value n.

For instance, $log_2(16) = 4$ since 2 raised to the power of 4 equals 16. Similarly, $log_2(8) = 3$ as 2 raised to the power of 3 equals 8.

The logarithm of n to the base 2, denoted as $log_2(n)$, is commonly used in computer science to represent the number of times a value can be divided by 2 before reaching 1. As the input size increases, the number of operations required to execute the algorithm increases at a much slower rate compared to linear or exponential growth. This makes logarithmic time algorithms highly efficient for dealing with large datasets.

1.5. Common Time Complexities



1. $O(1)$ - **Constant Time**: This is the best-case scenario. The algorithm's runtime remains constant, regardless of the input size.

2. $O(log(n))$ - **Logarithmic Time**: Algorithms with logarithmic time complexity often divide the problem space in half with each step (iteration). These algorithms are incredibly efficient for large data sets.

3. $O(n)$ - **Linear Time**: The algorithm's runtime grows linearly with

the input size. For example, if you double the input, the algorithm takes roughly double the time to complete.

4. $O(n * log(n))$ - **Linearithmic Time**: This complexity is common in efficient sorting algorithms like Merge Sort and Quick Sort. It's faster than quadratic time but not as fast as linear time.

5. $O(n^2)$ - **Quadratic Time**: Algorithms with quadratic time complexity become slow as the input size increases. Nested loops are a common cause of quadratic time.

6. $O(2^n)$ - **Exponential Time**: The algorithm's runtime grows rapidly with the input size. It's common in brute-force approaches and should be avoided for large inputs.

1.6. NP and P

In computational complexity theory, the classes NP and P play a central role in understanding the difficulty of problems that can be solved by computers.

P (Polynomial Time)

The class P, which stands for polynomial time, includes all problems that can be solved by a deterministic algorithm in an amount of time that is polynomial in the size of the input. This means that the number of operations required to solve the problem grows at a reasonable rate as the input size increases. For instance, sorting a list of numbers is considered a problem in class P.

NP (Nondeterministic Polynomial Time)

The class NP, which stands for nondeterministic polynomial time, includes all problems for which a solution can be verified in polynomial time, given the solution itself. This means that while it may be difficult to find the solution, it can be quickly checked for correctness. An example of an $NP - complete$ problem is the Boolean satisfiability problem (SAT), where the goal is to determine whether a given Boolean formula has a satisfying assignment.

NP-Completeness

$NP - complete$ problems represent a subset of NP problems with a unique property: any problem in NP can be reduced to an $NP - complete$ problem in polynomial time. This means that if an efficient algorithm could be

found to solve any $NP - complete$ problem, it could be used to solve all other problems in NP efficiently as well.

1.7. Bibliography

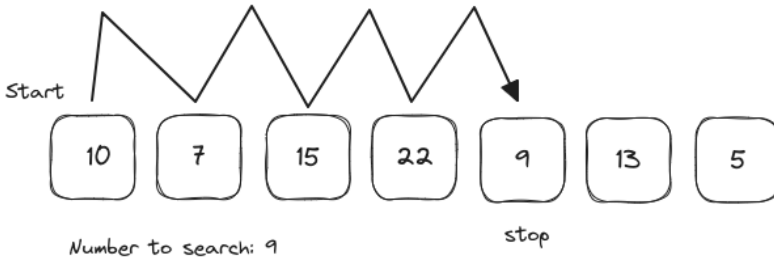
- Donald E. Knuth The art of computer programming, volume 1: Fundamental algorithms (Addison-Wesley 3rd) (1997)
- Donald E. Knuth The art of computer programming, volume 2: Semi-numerical algorithms (Addison-Wesley 3rd) (1997)
- Donald E. Knuth The art of computer programming, volume 3: Sorting and searching (Addison-Wesley 1st) (1973)
- Donald E. Knuth The art of computer programming, volume 4A: Combinatorial algorithms, part 1 (Addison-Wesley 1st) (2011)

Part I.

Basic Search Algorithms

2. Linear search

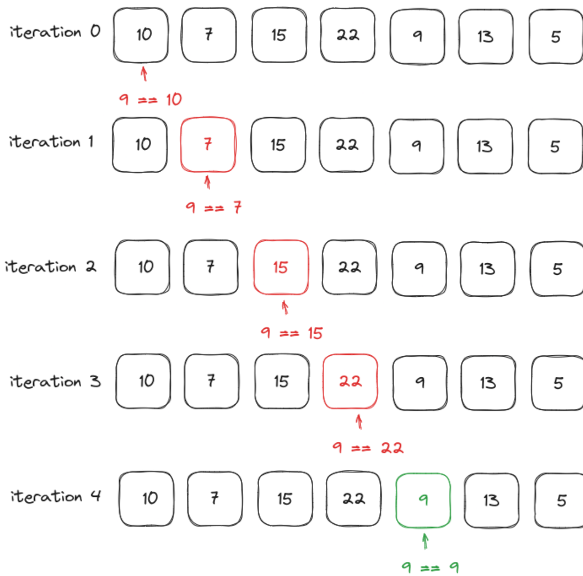
The linear search algorithm, also known as sequential search, is a straightforward method for finding a specific target value within a collection of elements, such as an array or a list.



It involves iterating through each element in the collection one by one and comparing it to the target value. If a match is found, the algorithm returns the index or position of the matching element; if no match is found after checking all elements, the algorithm indicates that the target value is not present in the collection. Linear search is simple to implement but may become inefficient for large datasets, as it has a time complexity of $O(n)$, where `n` is the number of elements in the collection.

2.1. How it works

To demonstrate the beauty of its simplicity, let's go through the steps of Linear Search:



- Start from the first element of the collection.
- Compare the target element with the current element.
- If the target element is found, return its index (or position).
- If not found, move to the next element in the collection and repeat the process.
- Continue until the target element is found, or the end of the collection is reached.

2.2. Implementation

```
def linear_search(target, lst):
    # Iterate over the elements in the list
    # and keep track of the index
    for index, element in enumerate(lst):
        # Check if the current element is
        # equal to the target element
        if element == target:
            # If a match is found, return
            # the index
            return index
```

```python
    # If the target element is not found,
    # return -1
    return -1

# Example usage:
my_list = [10, 7, 15, 22, 9, 13, 5]
target_element = 9

result = linear_search(target_element, my_list)
if result != -1:
    print(f"Element found at : {result}")
else:
    print(f"Element not found")
```

2.3. Time Complexity Analysis

The time complexity of Linear Search is $O(n)$, where n is the size of the collection. In the worst-case scenario, the algorithm may have to traverse the entire collection to find the target element. Therefore, as the collection grows, the search time increases linearly.

2.4. Advantages

- **Simplicity**: Pretty straightforward to implement.
- **Applicability to Small Lists**: In situations where the dataset is relatively small or unsorted, linear search can be efficient. Its simplicity makes it viable for small-scale searches without the need for a sorted dataset or additional complex data structures.

2.5. Limitations

- **Inefficiency with Large Datasets**: As you would have already guesses by looking at the time complexity, with a large number of elements, the time taken to search for a specific value grows linearly with the size of the list. For large datasets, more efficient algorithms like binary search or hash tables are preferred.

2.6. Applications

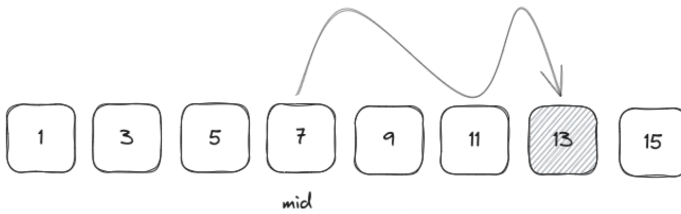
- **Checking for Availability in Databases:** Linear search is used in smaller databases or datasets to check for the existence or availability of specific items. For instance, in a library database, when a book's ISBN or title is entered, a linear search might be used to verify its availability by scanning through the database entries.
- **Searching for Files:** In file systems, especially when dealing with relatively small directories or file listings, linear search is employed to locate files. For instance, when a user searches for a file by name in a directory on a computer, a linear search scans through the directory entries until it finds a matching file name.

2.7. Bibliography

- Donald E. Knuth The art of computer programming, volume 3: (2nd ed.) sorting and searching (Addison Wesley Longman Publishing Co., Inc.) (1998)

3. Binary Search

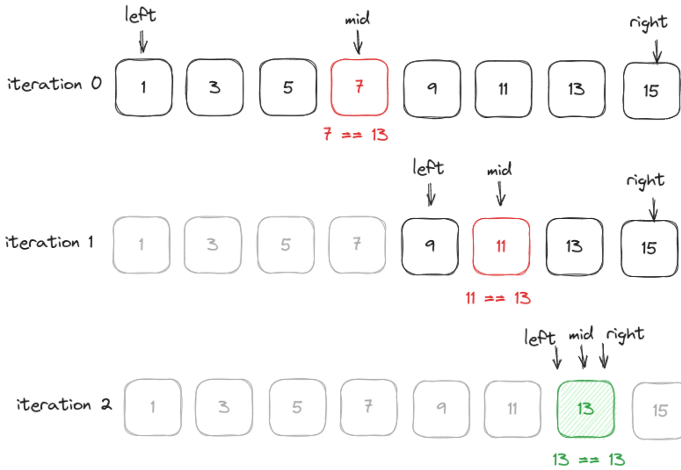
Binary search is a widely-used search algorithm that efficiently locates a specific target element within a sorted array or list. It works by repeatedly dividing the search space in half and comparing the middle element to the target. For example in a phone book, to find "John", you don't start from A, instead you'd begin by opening the book in the middle. If it is past J, you know to only search right half. This repeats, quickly narrowing on John's location by repeatedly halving search space.



If the middle element is equal to the target, the search is successful. If the target is less than the middle element, the search continues in the lower half of the remaining space; if the target is greater, the search continues in the upper half. This process eliminates half of the remaining elements with each step, leading to a significant reduction in search time compared to linear search algorithms. Binary search is particularly effective for large datasets and is characterized by its logarithmic time complexity of $O(log(n))$, where n is the number of elements in the array.

3.1. How it works

To perform a binary search, the list must be sorted in ascending or descending order. The algorithm works as follows:



- Set two pointers, `left` and `right`, to the first and last indices of the list, respectively.
- Calculate the `mid` index as the average of `left` and `right`.
- Compare the target element with the element at the `mid` index.
- If the target matches the element at `mid`, the search is successful, and we return the index.
- If the target is less than the element at `mid`, the target must be in the left half, so set `right` to `mid - 1`.
- If the target is greater than the element at `mid`, the target must be in the right half, so set `left` to `mid + 1`.
- Repeat the above steps until the target is found or `left` becomes greater than `right`, indicating the element is not present in the list.

3.2. Implementation

```
def binary_search(sorted_list, target):
    left = 0
    right = len(sorted_list) - 1
    while left <= right:
```

```
        mid = (left + right) // 2
        if sorted_list[mid] == target:
            # Check if the item at the middle
            # index of the search space is equal to the
            # target item. If it is, then the target item
            # has been found, and the function returns
            # the middle index.
            return mid
        elif sorted_list[mid] < target:
            # Checks if the item at the middle index of
            # the search space is less than the target item.
            # If it is, then the target item must be in
            # the 1right half of the search space, so the
            # left endpoint is updated to be the middle index plus
            ↪ one.
            left = mid + 1
        else:
            # checks if the item at the middle index of
            # the search space is greater than the target
            # item. If it is, then the target item must be in
            # the left half of the search space, so the
            # right endpoint is updated to be the middle
            # index minus one.
            right = mid - 1
    # If the loop exits without finding
    # the target item, then the function returns -1,
    # which indicates that the target item is not in the list.
    return -1

# Example usage:
sorted_list = [1, 3, 5, 7, 9, 11, 13, 15]
target = 13

result = binary_search(sorted_list, target)
if result != -1:
    print(f"Element found at : {result}")
else:
```

```python
    print(f"Element not found")
```

3.3. Time Complexity Analysis

Binary Search's time complexity is $O(log(n))$, where n is the number of elements in the list. Each step reduces the search space by half, leading to a logarithmic time complexity. This makes binary search incredibly efficient, especially for large datasets, as it can quickly locate the target element in just a few comparisons.

3.4. Advantages

- Very efficient for large datasets due to its logarithmic time complexity.
- It guarantees the target element's position will be found in a minimal number of comparisons.

3.5. Limitations

- Binary Search requires a sorted list, which can be a limitation if the data is not initially sorted.

3.6. Applications

- **Symbol Tables and Dictionaries**: Imagine you have a big dictionary with lots of words. If you're looking for a word like "Genesis," checking each word one by one takes a lot of time. Instead, you open the dictionary in the middle and see a word. If "Genesis" should come before that word, you ignore all the words after it. If it should come after, you ignore all the words before it. Then, you keep doing this until you find the word "Genesis." It's like a game of guessing where the word might be in the dictionary, and you get closer each time you check.
- **Finding specific data in databases**: Databases store massive amounts of data, and binary search is employed to efficiently retrieve specific information. Whether searching for a customer's address or

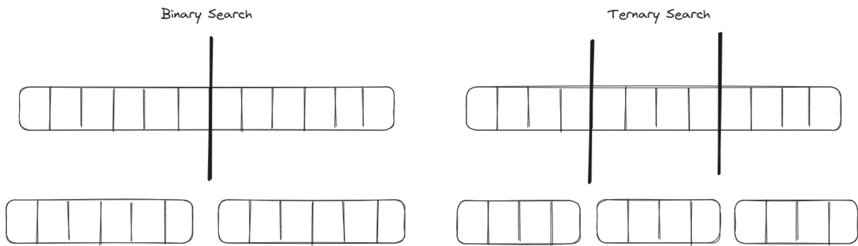
a product's price, binary search allows for rapid access to the desired data within large databases.

3.7. Bibliography

- Donald E. Knuth The art of computer programming, volume 3: (2nd ed.) sorting and searching (Addison Wesley Longman Publishing Co., Inc.) (1998)
- J. Bentley Programming pearls (Pearson Education) (2016)
- T. H. Cormen et al. Introduction to algorithms, third edition (MIT Press) (2009)

4. Ternary Search

Ternary search is a search algorithm that efficiently finds the position of a specific target value within a sorted dataset by repeatedly dividing the search range into three parts. The overall functioning of the algorithm is very similar to binary search, differing only in the number of divisions made in each iteration. In binary search, it is two, whereas in ternary search, it is three.



In Ternary search we compare the target value with elements at two positions within the dataset to determine if the target lies in the left or right portion of the current range. This process is repeated, narrowing down the search space to a smaller segment with each step. Ternary search is particularly effective when the dataset is known to be unimodal, meaning it has a single peak or trough. It has a time complexity of approximately $O(log_3 n)$, where n is the size of the dataset, making it more efficient than linear search but slightly less efficient than binary search.

4.1. How it Works

- Begin with an initial interval $[left, right]$, where $left$ and $right$ are the lower and upper bounds of the search space, respectively.
- While the interval $left$ is smaller than or equal to $right$:
  - Calculate two points, $mid1$ and $mid2$, dividing the interval into three segments.
  - Find the values at $mid1$ and $mid2$:

* If the value at $mid1$ is greater, move the $right$ boundary to $mid2$.
* If the value at $mid2$ is greater, move the $left$ boundary to $mid1$.
* Otherwise, the value likely lies between $mid1$ and $mid2$, so update the interval to $[mid1, mid2]$.

Let's take the following sorted array in which we have to find the number 55.

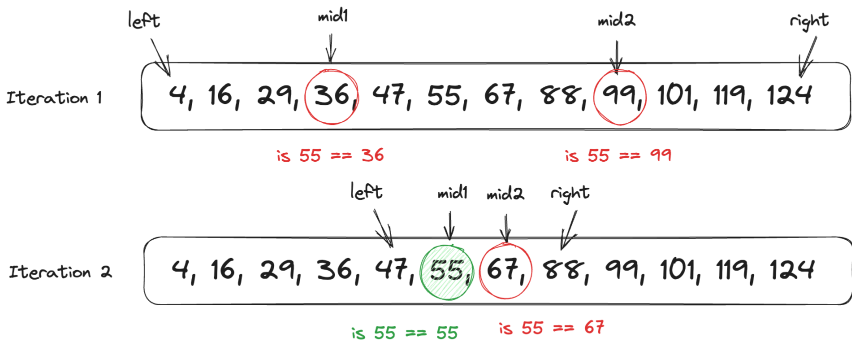
$$[4, 16, 29, 36, 47, 55, 67, 88, 99, 101, 119, 124]$$

Initially $left = 0, right = size - 1$, we calculate mid1 and mid2 using the below formula.

$$mid1 = left + \frac{right - left}{3}$$

$$mid2 = right - \frac{right - left}{3}$$

In each iteration we update the left, right, mid1 and mid2. We compare the value at index mid1 and mid2 till we either find the element or we exaust the list.



4.2. Implementation

```
def ternary_search(arr, target):
    left = 0
    right = len(arr) - 1

    while left <= right:
        # Calculate the mid1 and mid2 points
        mid1 = left + (right - left) // 3
        mid2 = right - (right - left) // 3

        if arr[mid1] == target:
            # Check if the target is at mid1
            return mid1
        if arr[mid2] == target:
            # Check if the target is at mid2
            return mid2

        if target < arr[mid1]:
            # If the target is smaller than
            # the element at mid1, update
            # the right pointer
            right = mid1 - 1
        elif target > arr[mid2]:
            # If the target is larger than
            # the element at mid2, update
            # the left pointer
            left = mid2 + 1
        else:
            # If the target is between mid1 and mid2
            # update the pointers
            left = mid1 + 1
            right = mid2 - 1

    return -1

# Example usage
```

```
arr = [4, 16, 29, 36, 47, 55, 67, 88, 99, 101, 119, 124]
target = 55
result = ternary_search(arr, target)
if result != -1:
    print("Element", target, "found at index", result)
else:
    print("Element", target, "not found in the array")
```

4.3. Time Complexity Analysis

The Ternary Search Algorithm's time complexity is $O(log_3 n)$, where n is the size of the array being searched. Here's how the time complexity is derived:

In each iteration of the algorithm, the search range is divided into three segments using the two midpoints. This means that with each iteration, the size of the remaining search range decreases by a factor of $\frac{1}{3}$. Therefore, the number of iterations required to narrow down the search range to a single element (or determine that the element is not present) is proportional to the number of times you can divide n by 3.

Let's express this in terms of iterations:

- In the first iteration, the size of the array is n.
- After the first iteration, the size becomes $\frac{n}{3}$.
- After the second iteration, the size becomes $\frac{\frac{n}{3}}{3} = \frac{n}{9}$
- After k iterations, the size becomes $\frac{n}{3^k}$

You can stop the algorithm when the size of the search range becomes 1 or less. In other words, when $\frac{n}{3^k}$ is less than or equal to 1:

$$\frac{n}{3^k} <= 1$$

$$n <= 3^k$$

$$log_3 n <= k$$

The number of iterations k required to narrow down the search range to a single element (or determine that the element is not present) is bounded by $log_3 n$. This is why the time complexity of the Ternary Search Algorithm

is $O(log_3 n)$, which is more efficient than binary search's time complexity of $O(log_2 n)$ for larger arrays.

4.4. Applications

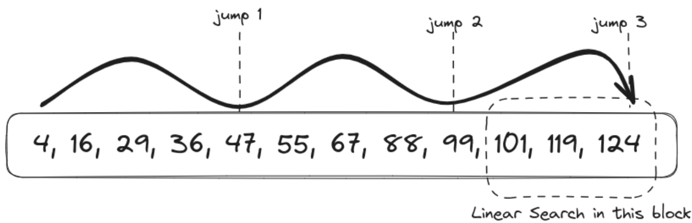
- **Peak Finding**: In signal processing or data analysis, ternary search can identify peaks or valleys in datasets efficiently. For instance, it might be used to find the peak intensity in signal processing applications or identifying the lowest or highest point in certain environmental or scientific datasets.
- **Minimizing Cost Functions**: In cost optimization problems like in industrial engineering, ternary search excels at converging on the input values that minimize manufacturing, transportation, or supply chain expenses expressed as a cost function.

4.5. Bibliography

- Donald E. Knuth The art of computer programming, volume 3: (2nd ed.) sorting and searching (Addison Wesley Longman Publishing Co., Inc.) (1998)
- Chen Xin-yi *Ternary search algorithm*, SCIENCE TECHNOLOGY AND ENGINEERING (2008), https://api.semanticscholar.org/CorpusID:124594637
- Dimitrios Ventzas & Nikos Petrellis *Peak searching algorithms and applications*, PROCEEDINGS OF THE IASTED INTERNATIONAL CONFERENCE ON SIGNAL AND IMAGE PROCESSING AND APPLICATIONS, SIPA 2011 (2011)
- T. H. CORMEN et al. INTRODUCTION TO ALGORITHMS, THIRD EDITION (MIT Press) (2009)
- Manpreet Singh Bajwa et al. *Ternary search algorithm: Improvement of binary search*, 2015 2ND INTERNATIONAL CONFERENCE ON COMPUTING FOR SUSTAINABLE GLOBAL DEVELOPMENT (INDIACOM) 1723–1725 (2015), https://api.semanticscholar.org/CorpusID:1259619
- MINIMIZING COST FUNCTION USING ITERATIVE SEARCH FOR A MINIMUM METHOD, https://scicomp.stackexchange.com/questions/34694/minimizing-cost-function-using-iterative-search-for-a-minimum-method (last visited Feb 26, 2023)

5. Jump Search

Jump Search is a search algorithm designed to efficiently find a target value within a sorted array by dividing the array into smaller blocks or “jumps,” and then performing linear search within those blocks to locate the target.



The algorithm starts by making jumps of a fixed step size through the array until it finds a block where the target value might be located. Once a block is identified, a linear search is performed within that block to find the exact position of the target.

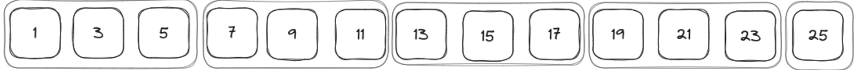
Determining the optimal jump distance in a jump search algorithm is crucial for maximizing its efficiency. While there is no universal formula for the ideal jump distance, it can be estimated based on the length of the sorted array (n) using the following formula:

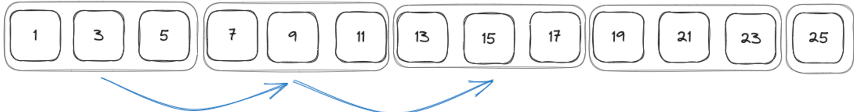
$$D = \sqrt{n}$$

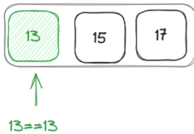
This formula is based on the concept that the jump distance should be large enough to quickly traverse the sorted array while avoiding excessive jumps that could lead to performance overhead.

5.1. How it Works

To perform a Jump Search, the list must be sorted in ascending or descending order. The algorithm works as follows:







- Set a jump distance or block size `step`, typically the square root of the list's length.
- Initialize two pointers, `left` and `right`, to mark the current block boundaries. Set `left` to 0 and `right` to the minimum between `step` and the list's length.
- While the target element is greater than the element at the right boundary or the end of the list is not reached, continue the search:

   a. If the target element is less than the element at the right boundary, perform a linear search within the current block from `left` to `right`.
   b. If the target element is found within the block, the search is successful.
   c. If the target element is not found, update `left` to `right + 1` and update `right` to the next block boundary (minimum between `right + step` and the list's length).

- If the end of the list is reached, the target element is not present in the list.

5.2. Implementation

```python
import math

def jump_search(sorted_list, target):
    n = len(sorted_list)
    # Calculate the jump distance
    step = int(math.sqrt(n))

    # Initialize the left pointer
    left = 0
    # Initialize the right pointer
    right = min(step, n)

    # While the target element is greater than the
    # element at the right boundary or the end of the
    # list is not reached, continue searching.
    while right < n and sorted_list[right] < target:
        # Move the left pointer to the right boundary
        left = right
        # Move the right pointer to the next block boundary
        right = min(right + step, n)
    # Perform a linear search within the current block
    for i in range(left, right):
        if sorted_list[i] == target:
            # Target element found at index `i`
            return i
    # Target element not found in the list
    return -1

# Example usage:
sorted_list = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25]
target = 13

result = jump_search(sorted_list, target)
if result != -1:
    print(f"Target element {target} found at index: {result}")
```

```python
else:
    print(f"Target element {target} not found in the list.")
```

5.3. Time Complexity Analysis

Jump Search's time complexity is $O(\sqrt{n})$, where n is the number of elements in the list. The algorithm's efficiency is derived from the reduced number of comparisons required to narrow down the search space by making "jumps" through the list.

5.4. Advantages

- **Efficiency for Large Arrays:** Jump search has a time complexity of $O(\sqrt{n})$, where n is the number of elements.

5.5. Limitations

- Jump Search requires a sorted list, which can be a limitation if the data is not initially sorted.

5.6. Applications

- **Searching in Large Datasets:** Jump Search is useful for searching within large datasets, especially when binary search might be overkill. You can make larger jumps for larger dataset and it can outperform binary search.
- **Searching in Logs:** Jump Search can help locate specific events or timestamps within log files, which are often sorted chronologically.

5.7. Bibliography

- Ben Shneiderman *Jump searching: A fast sequential search technique*, 21 COMMUN. ACM 831–834 (1978), https://doi.org/10.1145/359619.359623
- Thompson, Mark. Algorithm Handbook. N.p.: Lulu.com, 2018.

7. Interpolation Search

Imagine you're searching for a specific name like "Sarah" in a phone book. You wouldn't start at the beginning and check every name until you find the right one. Instead, you'd likely flip towards the end of the book and start scanning the names around there. Interpolation search works in a similar way. Instead of checking every item in the list, or even starting in the middle as in the case of binary search, interpolation search estimates where an item might be based on the values of the items at the beginning and end of the list. It then checks the item at that estimated position. If it's not the item you're looking for, interpolation search narrows down the list to either the part before or after the estimated position. This process continues until the item is found or it's clear that it's not in the list.

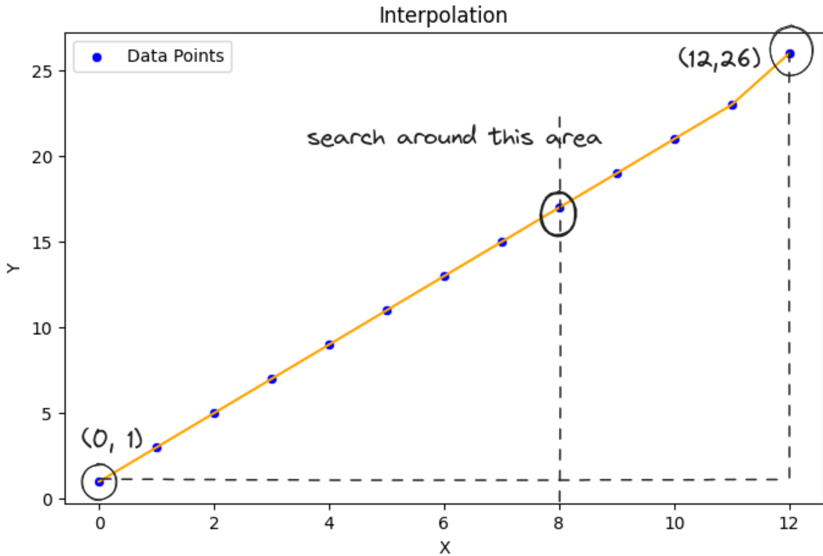
Binary search is taught in almost all introductory algorithm courses, but this lesser-known search algorithm can be surprisingly faster than binary search when data is sorted and uniformly distributed. It can be as fast as O(log(log(n))), where n is the size of the array.

Assume we have a dataset `[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 26]` and we have to find 17, Point A would be $(0, 1)$ and Point B would be $(12, 26)$ where 0 and 12 represent the indexes of 1 and 26 in our input array respectively. We find the position to search using the interpolation formula

$$x = x_1 + (x_2 - x_1) \times \frac{key - y_1}{y_2 - y_1}$$

$$7.8 = 0 + (12 - 0) \times \frac{17 - 1}{26 - 1}$$

`7.8` rounded to an integer comes around, so we look at the index `8`



7.1. How it works

To perform Interpolation Search, the list must be uniformly distributed and sorted in ascending or descending order. The algorithm works as follows:

- Calculate the estimated position of the target element using interpolation formula:

  $position = low + (target - arr[low]) \times \lfloor \frac{(high - low)}{arr[high] - arr[low]} \rfloor$

  Here, `low` and `high` are the indices representing the current search range.

- Compare the target element with the element at `pos`.

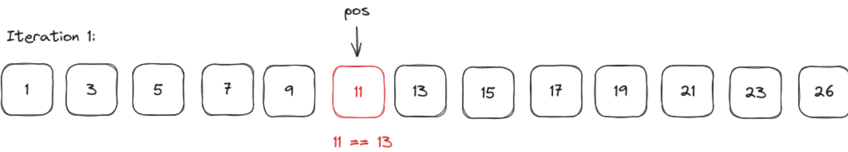
- If the target matches the element at `pos`, the search is successful, and we return `pos`.

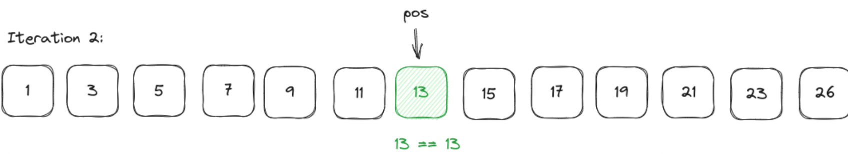
- If the target is less than the element at `pos`, the target must be in the left half, so update `high` to `pos - 1`.

- If the target is greater than the element at `pos`, the target must be in the right half, so update `low` to `pos + 1`.

- Repeat the above steps until the target is found or `low` becomes greater than `high`, indicating the element is not present in the list.

Figure 7.1.: Interpolation Search





7.2. Implementation

```
def interpolation_search(sorted_list, target):
    n = len(sorted_list)
    low = 0                                                              ①
    high = n - 1                                                         ②

    while low <= high and sorted_list[low] <= target <=
    ↪  sorted_list[high]:
        pos = low + ((target - sorted_list[low]) * (high - low) //
                     (sorted_list[high] - sorted_list[low]))             ③
        if sorted_list[pos] == target:
            return pos                                                   ④
        elif sorted_list[pos] < target:
            low = pos + 1                                                ⑤
        else:
            high = pos - 1                                               ⑥
```

```
    return -1

# Example usage:
sorted_list = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 26]
target = 13

result = interpolation_search(sorted_list, target)
if result != -1:
    print(f"Target element {target} found at index: {result}")
else:
    print(f"Target element {target} not found in the list.")
```

① Initialize the lower bound of the search range
② Initialize the upper bound of the search range
③ Estimate the position of the target using interpolation formula
④ Target element found at index pos
⑤ Target must be in the right half, update the lower bound
⑥ Target must be in the left half, update the upper bound

7.3. Time Complexity Analysis

Interpolation Search's time complexity can vary depending on the distribution of the data. In the best-case scenario, when the data is evenly distributed, the time complexity is close to $O(log(log(n)))$, making it one of the most efficient search algorithms for large datasets. However, in the worst-case scenario, the time complexity may be $O(n)$, especially when the data is unevenly distributed.

7.4. Advantages and Limitations

Advantages:

- **Efficient for Uniformly Distributed Data:** Interpolation Search can be highly efficient when the data is uniformly distributed. It leverages the sorted order and estimates the probable position of the target based on the data distribution.

- **Faster Convergence:** Compared to binary search, Interpolation Search can converge on the target faster in cases where the data is evenly spaced. This is because it adapts its search based on the estimated position.
- **Slightly Better Than Binary Search:** In situations where the data distribution is uniform, Interpolation Search can perform slightly better than binary search in terms of the number of comparisons made.
- **Suitable for Numerical Data:** Interpolation Search is particularly useful for searching numerical data sets, where values are distributed along a range.

Limitations:

- **Not Suitable for Non-Numerical Data:** Interpolation Search is designed for numerical data. It's not suitable for searching non-numerical data or categorical data.
- **Worst-Case Performance:** In some cases, Interpolation Search can have a worst-case time complexity of $O(n)$, where n is the size of the array. This can happen when the data distribution is not uniform.
- **Potential for Overflow or Underflow:** In cases where the data values are very large or very small, there's a risk of integer overflow or underflow during the calculation of the estimated position.
- **Depends on Data Characteristics:** Interpolation search is only effective if the differences between the values of the items in the list are consistent. If the differences are all over the place, interpolation search won't be able to make accurate estimates of where the item you're looking for might be.

7.5. Applications

- **Numerical Analysis:** Interpolation Search is used in numerical analysis to quickly locate intermediate values within sorted numerical datasets, making computations more efficient.

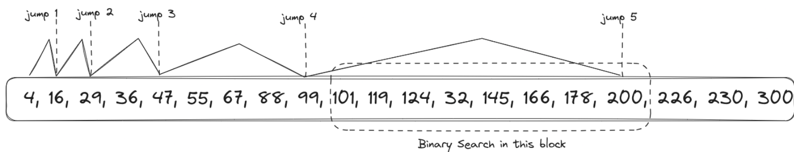
- **Financial Analysis:** Interpolation Search is applied in financial analysis to efficiently locate relevant data points within sorted datasets, such as historical stock prices or economic indicators.

7.6. Bibliography

- W. W. Peterson *Addressing for random-access storage*, 1 IBM Journal of Research and Development 130–146 (1957)
- Interpolation search, https://en.wikipedia.org/wiki/Interpolation/_search (last visited Mar 1, 2023)
- Skiena, Steven S.. The Algorithm Design Manual. Germany, TELOS–the Electronic Library of Science, 1998.

8. Exponential Search

Exponential search is an efficient search algorithm for sorted arrays based on the idea of exponentially jumping ahead early on before applying binary search.

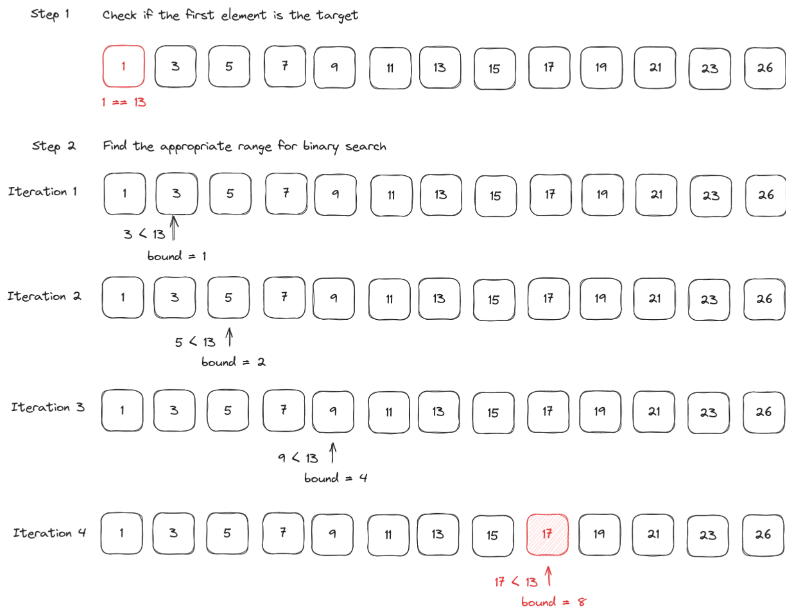


The idea behind exponential search is to start with a small search space and then exponentially increasing the size of the search space to find a range where the target value might exist and then running binary search within that range to find the target value.

Exponential Search can be particularly effective when the target value is close to the beginning of the array and the array size is large. Its time complexity is $O(log(n))$ in the worst case, making it more efficient than linear search $O(n)$.

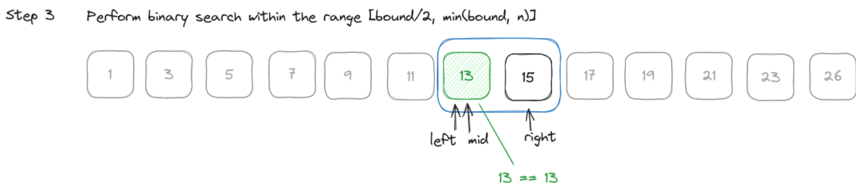
8.1. How it works

- Set an initial bound `bound` to 1, representing the first position.
- Double the `bound` until the element at `bound` is greater than or equal to the target element or the end of the list is reached.



- Perform a binary search within the range `[bound/2, min(bound, n)]`, where `n` is the number of elements in the list.
- If the target element is found, the search is successful, and we return the index.
- If the target is not found, double the `bound` again and repeat the process.

Figure 8.1.: Exponential Search Step 3



8.2. Implementation

```
def exponential_search(sorted_list, target):
    n = len(sorted_list)
    if sorted_list[0] == target:
        # Check if the first element is the target
        return 0

    bound = 1
    while bound < n and sorted_list[bound] < target:
        # Find the appropriate range for binary search
        bound *= 2

    # Perform binary search within the range [bound/2, min(bound, n)]
    left = bound // 2
    right = min(bound, n) - 1

    while left <= right:
        mid = (left + right) // 2

        if sorted_list[mid] == target:
            return mid
        elif sorted_list[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

# Example usage:
sorted_list = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 26]
target = 13

result = exponential_search(sorted_list, target)
if result != -1:
    print(f"Target element {target} found at index: {result}")
else:
    print(f"Target element {target} not found in the list.")
```

8.3. Time Complexity Analysis

The time complexity of exponential search can be broken down into two parts:

Exponential Search Phase: The exponential phase runs in $O(log(n))$ time, where 'n' is the size of the array. This phase involves finding the range where the target value might exist by doubling the index in each iteration.

Binary Search Phase: After determining the potential range using exponential search, a binary search is performed within that range. The binary search phase runs in $O(logm)$, where 'm' is the size of the identified range.

The overall time complexity of exponential search is the sum of the complexities of these two phases:

$O(log(n)) + O(logm)$ In the worst case, when the target value is at the beginning of the array or absent, the binary search phase would search through the entire range determined by the exponential phase, resulting in $O(log(n)) + O(log(n)) = O(log(n))$ time complexity.

However, in scenarios where the target value is closer to the start of the array, exponential search can be more efficient than performing a binary search throughout the entire array.

8.4. Advantages

- **Efficient for Early Elements:** Exponential Search is efficient when the target value is located near the beginning of the sorted array. It quickly narrows down the search space by exponentially increasing the search range.
- **Logarithmic Time Complexity:** In favorable scenarios, Exponential Search has a logarithmic time complexity of $O(log(i))$, where i is the index where the target is found. This can be more efficient than linear search, especially for larger arrays.
- **Simpler Implementation:** Exponential Search is easier to implement compared to more complex algorithms like binary search. It involves straightforward calculations and looping.
- **Adaptability to Distribution:** Exponential Search can be particularly effective when the data distribution is non-uniform. It can quickly skip over regions of the array that are unlikely to contain

the target.

8.5. Limitations

- **Limited to Sorted Arrays:** Exponential Search requires the input array to be sorted. If the array is not sorted, the algorithm will not work correctly.
- **Inefficient for Late Elements:** When the target value is located near the end of the array, Exponential Search can require a large number of iterations to reach the target, making it inefficient compared to binary search.
- **Array Size Impact:** Exponential Search's efficiency is significantly impacted by the array size. For larger arrays, the exponential growth of the search range can lead to a substantial number of iterations.
- **No Benefit from Uniform Distribution:** If the data distribution is uniform or lacks large gaps between elements, Exponential Search might not provide notable advantages over other search algorithms like binary search.

8.6. Applications

- **Searching in large sorted datasets**: Exponential search is particularly useful when dealing with large datasets that are sorted. It allows for efficient searching by reducing the number of comparisons required compared to linear search algorithms.
- **Searching in unbounded or unknown-sized arrays**: Exponential search is advantageous when the size of the array is unknown or unbounded. It can be used to quickly locate an element without needing to know the exact size of the array in advance.
- **Searching in distributed systems**: Exponential search can be applied in distributed systems where data is stored across multiple nodes or servers. It enables efficient searching by minimizing the number of network requests required to find the desired element.

8.7. Bibliography

- Thomas H. Cormen et al. Introduction to algorithms (The MIT Press 3rd) (2009)
- Robert Sedgewick & Kevin Wayne Algorithms (Addison-Wesley Professional 4th) (2011)
- Steven S. Skiena The algorithm design manual (Springer 2nd) (2008)
- Michael T. Goodrich et al. Data structures and algorithms in python (Wiley 1st) (2013)

9. Fibonacci Search

Fibonacci sequence is a series of numbers in which each number is the sum of the two preceding ones, typically starting with 0 and 1. 0, 1, 1, 2, 3, 5, 8, 13, 21, and so on.

```
0
1
1 = 1 + 0
2 = 1 + 1
3 = 2 + 1
5 = 3 + 2
8 = 5 + 3
13 = 8 + 5
21 = 13 + 8

..
..
..
```

Fibonacci search algorithm uses the Fibonacci sequence to narrow down the search space and find a specific value in a sorted array. It is a variant of binary search, but utilizes Fibonacci numbers to determine the partitions of the search space more effectively. By dividing the search space based on the Fibonacci sequence, the search algorithm can effectively determine the most likely location of the target value quickly. This leads to a time complexity of $O(log(n))$, where n is the size of the array, making it more efficient than linear search, which is $O(n)$.

9.1. How it works

Let's take the following sorted array in which we have to find the number 99.

$$[4, 16, 29, 36, 47, 55, 67, 88, 99, 101, 119, 124]$$

We start by finding the required Fibonacci numbers $F0$, $F1$ and $F2$ for $size = 12$. The smallest Fibonacci number ≥ 12 is 13, which means our $F_2 = 13$, $F_1 = 8$, and $F_0 = 5$.

Once we have our Fibonacci numbers, we keep narrowing down our search scope till we find our element or we run out of elements.

Iteration 1

Initially our offset will be equal to -1, and the end index can be calculated using the below formula.

$$index = \min\left(\text{offset} + F_0, \text{size} - 1\right) = \min((-1 + 5, 11) = 4$$

Since the element we are looking for (99) is greater than the element at index 4 (47), we move one Fibonacci number down

$$F2 = F1$$

$$F1 = F0$$

$$F0 = F2 - F1$$

The updated Fibonacci numbers are: $F_2 = 8, F_1 = 5, F_0 = 3$ and offset = 4

Iteration 2

We compute

$$index = \min\left(offset + F_0, n - 1\right) = \min((4 + 3, 11) = 7$$

The element at index 7 in arr is 88. Since $99 \geq 88$, we again move one Fibonacci number down,

$$F2 = F1$$

$$F1 = F0$$

$$F0 = F2 - F1$$

The updated Fibonacci numbers are: $F_2 = 5, F_1 = 3, F_0 = 2$ and offset = 7

Iteration 3

We compute

$$index = \min\left(offset + F_0, n - 1\right) = \min((7 + 2, 11) = 9$$

The element at index 9 in arr is 101. Since $99 \leq 101$, we move one Fibonacci number up,

$$F2 = F0$$

$$F1 = F1 - F0$$

$$F0 = F2 - F1$$

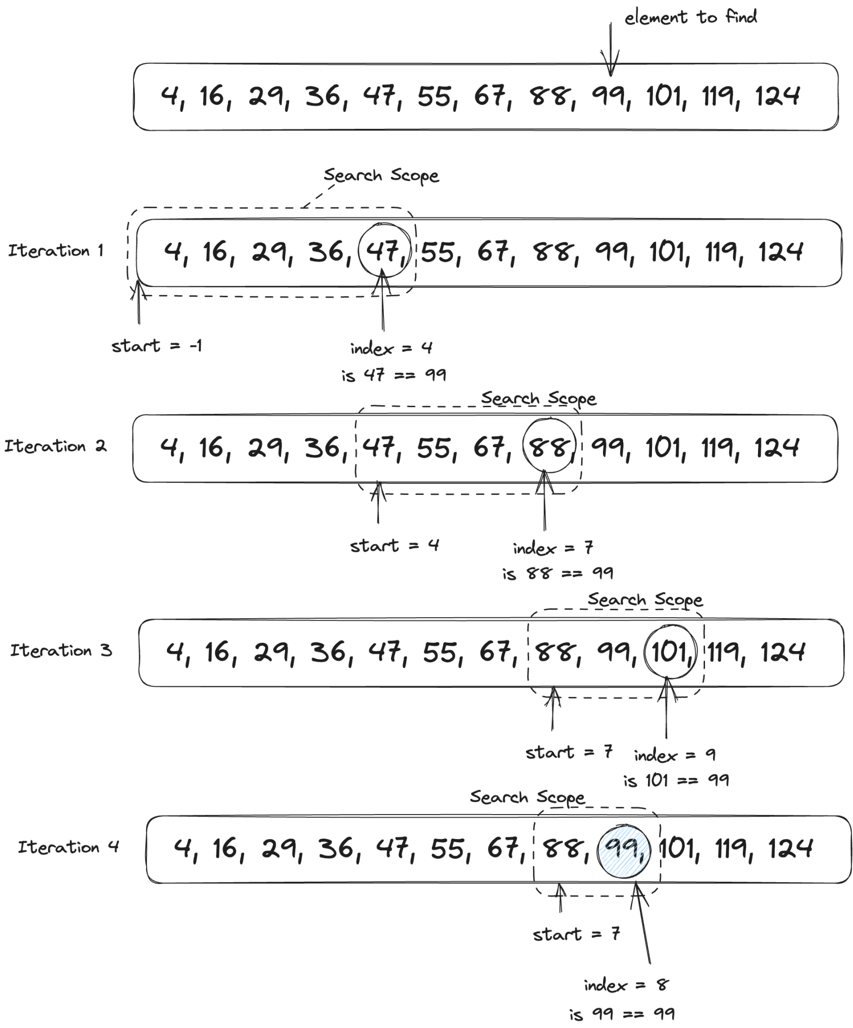
The updated Fibonacci numbers are: $F_2 = 2, F_1 = 1, F_0 = 1$ and offset = 7

Iteration 4

We compute

$$index = \min\left(offset + F_0, n - 1\right) = \min((7 + 1, 11) = 8$$

The element at index 8 is 99, which is what we are looking for. We return the index and stop.



9.2. Implementation

```
def fibonacci_search(lst, target):
    size = len(lst)
    start = -1
    f0 = 0
```

```
    f1 = 1
    f2 = 1
    while(f2 < size):
        # Generate Fibonacci Numbers
        f0 = f1
        f1 = f2
        f2 = f1 + f0

    while(f2 > 1):
        # Calculate the index to
        # compare with the target value
        index = min(start + f0, size - 1)
        if lst[index] < target:                                    ③
            f2 = f1
            f1 = f0
            f0 = f2 - f1
            start = index
        elif lst[index] > target:
            # Move two Fibonacci numbers up
            # and update the start index
            f2 = f0
            f1 = f1 - f0
            f0 = f2 - f1
        else:
            # Return the index if the target
            # value is found
            return index
    if (f1) and (lst[size - 1] == target):
        return size - 1
    return None

# Example usage
arr = [2, 3, 4, 6, 7, 8, 9, 12, 13, 15,
19, 20, 23, 25, 28, 32, 33, 37, 39, 41,
43, 46, 47, 49, 52, 55, 56, 58, 59, 60,
61, 63, 64, 68, 70, 71, 72, 74, 76, 78,
80, 81, 85, 89, 90, 91, 94, 96, 97, 99]
```

```python
x = 7
result = fibonacci_search(arr, x)
if result != -1:
    print("Element", x, "found at index", result)
else:
    print("Element", x, "not found in the array")
```

9.3. Time Complexity Analysis:

The time complexity of Fibonacci search is logarithmic, similar to binary search, but with a different constant factor.

Here's a breakdown of the time complexity:

1. **Calculating Fibonacci Numbers:** Generating the Fibonacci numbers up to a value greater than or equal to the array size takes $O(log(n))$ time. This is because the sequence of Fibonacci numbers grows exponentially, but the number of Fibonacci numbers needed for an array of size $'n'$ is limited (usually less than $log(n)$).

2. **Dividing the Array:** The search involves dividing the array using Fibonacci numbers, which takes $O(1)$ time per iteration. The number of iterations is limited by the sequence of Fibonacci numbers generated, which is at most log(n).

3. **Binary Search within Segments:** Once the array is divided using Fibonacci numbers, a binary search is performed in each segment. Binary search within each segment also takes $O(log(k))$ time, where $'k'$ is the size of the segment.

Therefore, the overall time complexity of Fibonacci search is $O(log(n))$ for calculating Fibonacci numbers plus $O(log(k))$ for performing binary searches within the identified segments, where $'k'$ is the size of the segment. In most practical scenarios, the total time complexity is still $O(log(n))$, but Fibonacci search might have a slightly higher constant factor compared to traditional binary search due to additional arithmetic operations involved in calculating Fibonacci numbers and determining the segments.

9.4. Advantages

- **Efficiency**: The algorithm narrows down the search space quickly, resulting in a time complexity of $O(log(n))$. This makes it more efficient than linear search $O(n)$ and comparable to binary search.

- **Simplicity**: The algorithm's logic is relatively simple, making it easy to implement.

9.5. Limitations

- **Requires a sorted array**: The array must be sorted for the Fibonacci Search Algorithm to work effectively. If the array is not sorted, a pre-processing step to sort it is needed.

9.6. Applications

- **Optimizing Machine Learning Hyperparameters** - When evaluating different hyperparameters like neural network layers or SVM cost values, Fibonacci search provides an efficient methodology.
- **Root Finding Algorithms** - Methods like Newton-Raphson leverage concepts similar to Fibonacci search for numerically approximating roots of equations.
- **Bioinformatics** - Fibonacci coding has been utilized in areas like DNA sequence analysis and alignment to improve computational efficiency

9.7. Bibliography

- Fibonacci search technique - Wikipedia — en.wikipedia.org, https://en.wikipedia.org/wiki/Fibonacci_search_technique
- David E. Ferguson *Fibonaccian searching*, 3 Commun. ACM 648 (1960), https://doi.org/10.1145/367487.367496
- Donald E. Knuth The art of computer programming, volume 3: (2nd ed.) sorting and searching (Addison Wesley Longman Publishing Co., Inc.) (1998)

Part II.

Hash-Based Search

10. Basics

Hashing is a process used to convert input data (such as a string or any other type of data) into a fixed-size value, typically a numeric value, using a hash function. This output value is often called a hash code or hash value.

A simple hash function could involve summing up the ASCII values of characters in a string and taking the modulo of a prime number to constrain the output within a specific range. Here's an example in Python:

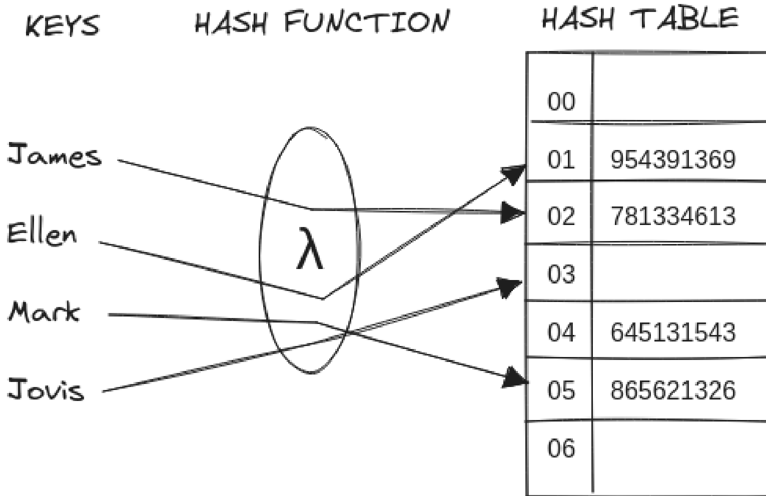
```python
def simple_hash(text):
    prime = 31  # Choosing a prime number for modulo
    hash_value = 0

    for char in text:
        hash_value += ord(char)  # Adding ASCII values of characters

    hash_value %= prime  # Taking modulo to limit the value

    return hash_value

# Example usage:
input_string = "Hello, Simple Hash Function!"
hashed_value = simple_hash(input_string)
print(f"Hashed value for '{input_string}': {hashed_value}")
```

```
Hashed value for 'Hello, Simple Hash Function!': 6
```

Keep in mind, this is a very basic hash function and not suitable for cryptographic purposes or extensive data integrity checks due to its simplicity and potential for collisions. In real-world applications, robust hash functions with lower collision probabilities and better distribution properties are used.

10.1. Hash table

The hash value is used to map the input to a specific location in a data structure called a hash table. Hash tables are a type of associative array that can be used to efficiently store and retrieve data.



Hashing is a very efficient way to implement data structures because it allows for constant-time lookup, insertion, and deletion operations. This is in contrast to other data structures, such as linked lists and trees, which can have lookup, insertion, and deletion times that are proportional to the size of the data structure.

10.2. How Hashing Works

The hashing process typically involves the following steps:

1. **Choose a hash function.** A hash function is a mathematical function that takes an input of variable size and produces a fixed-size output. The hash function should be designed so that it is difficult to find two different inputs that produce the same hash value. This is called a collision.

2. **Apply the hash function to the input.** This will generate a hash value.

3. **Use the hash value to map the input to a specific location in the hash table.** The hash table is an array of buckets, and the index of the bucket is determined by the hash value.
4. **Store the input-value pair in the appropriate bucket.**

10.3. Types of Hash Functions

There are many different types of hash functions, but some of the most common include:

10.3.1. Modulo arithmetic hash functions

These hash functions use the modulo arithmetic operator to generate a hash value. For example, the hash function $h(x) = (x) mod(m)$, where m when equal to 10 would generate a hash value between 0 and 9 for any input x.

Here is a simple program for a modulo arithmetic hash function:

```
def modulo_arithmetic_hash_function(key, table_size):
    """
    Calculates the hash value of a key using
    the modulo arithmetic hash function.

    Args:
        key (int): The key to hash.
        table_size (int): The size of the hash table.

    Returns:
        int: The hash value of the key.
    """
    return key % table_size
```

Here is an example of how to use the function:

```
key = 12345
table_size = 10000
hash_value = modulo_arithmetic_hash_function(key, table_size)
```

```
print("Hash value of", key, "using modulo arithmetic hash function:",
 ↪  \
                        hash_value)
```

This code will print the following output:

```
Hash value of 12345 using modulo arithmetic hash function: 2345
```

10.3.2. Multiplication hash functions

These hash functions multiply the input by a constant and then take the remainder after dividing by a prime number. For example, the hash function $h(x) = x * 53 mod 101$ would generate a hash value between 0 and 100 for any input x.

Here is a simple program that demonstrates the use of a multiplication hash function:

```
def multiplication_hash_function(key, table_size):
    """
    Calculates the hash value of a key
    using the multiplication hash function.

    Args:
        key (int): The key to hash.
        table_size (int): The size of the hash table.

    Returns:
        int: The hash value of the key.
    """

    constant = 0.53  # The constant used for multiplication
    result = key * constant
    fractional_part = result - int(result)                          ①
    hash_value = int(fractional_part * table_size)                  ②
    return hash_value
```

```
key = 12345
table_size = 10000
hash_value = multiplication_hash_function(key, table_size)
print("Hash value of", key, "using multiplication hash function:", \
    hash_value)
```

① Extract the fractional part of the result
② Multiply the fractional part by the table size and convert to an integer

Here is an example of how to use the function:

```
key = 12345
table_size = 10000
hash_value = multiplication_hash_function(key, table_size)
print("Hash value of", key, "using multiplication hash function:", \
     hash_value)
```

This code will print the following output:

```
Hash value of 12345 using multiplication hash function: 6475
```

10.3.3. Tabulation hash functions

It's a method that involves creating a table (or multiple tables) of precomputed values and using those values to compute the hash code of a given input. The idea is to break down the input into smaller components and use precomputed values for those components to create a final hash. The precomputed tables allow for quick computation of hash codes by directly looking up values associated with individual components of the input. This method also exhibits good properties in terms of randomness and distribution of hash codes.

Here's a very basic example in Python of a tabulation hash function for a string:

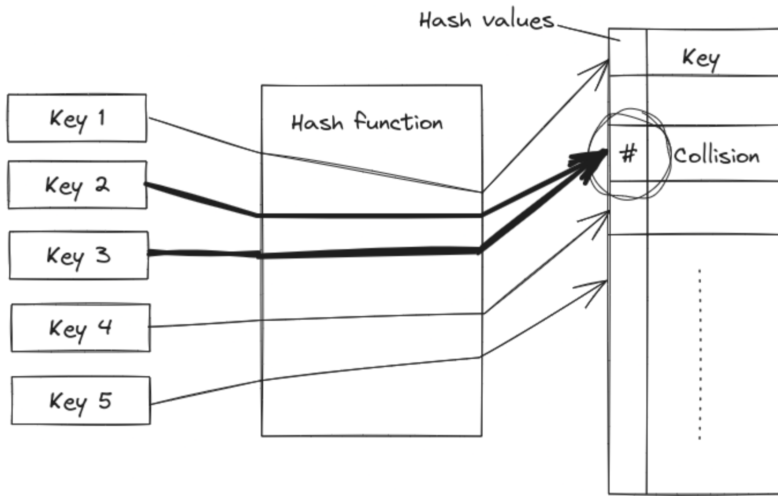
```
import random

class TabulationHash:
```

```python
    def __init__(self, num_tables, table_size):
        self.num_tables = num_tables
        self.table_size = table_size
        self.tables = []
        for _ in range(num_tables):
            table = []
            for _ in range(256):
                table.append(random.randint(0, 255))
            self.tables.append(table)

    def hash(self, input_string):
        result = 0
        for i, char in enumerate(input_string):
            table = self.tables[i % self.num_tables]
            result ^= table[ord(char)]
        return result

# Example usage
tabulation_hash = TabulationHash(num_tables=4, table_size=256)
hash_code = tabulation_hash.hash("example")
print("Hash Code:", hash_code)
```

This example demonstrates a simplified tabulation hash function for strings using XOR operations with precomputed random values. In practice, more sophisticated methods might be used for precomputation and combining values.

10.4. Hash Collisions

In computer science, a **hash collision** or **hash clash** occurs when two different pieces of data in a hash table share the same hash value. This means that the hashing algorithm has produced the same output for two distinct inputs.



Hash collisions are an inherent characteristic of hashing algorithms due to the fact that there are infinitely many possible inputs but a finite number of possible hash values. As a result, there is always a possibility of two or more inputs mapping to the same hash value, especially when the hash table is densely populated.

To mitigate the impact of hash collisions, several strategies can be employed:

1. **Choose a good hash function:** A well-designed hash function should distribute hash values evenly and minimize the likelihood of collisions.

2. **Use a large hash table:** Increasing the size of the hash table reduces the load factor, which in turn reduces the probability of collisions.

3. **Alternative techniques:** Techniques like linear probing, chaining, or quadratic probing can effectively handle collisions by exploring alternative locations in the hash table for the collided key.

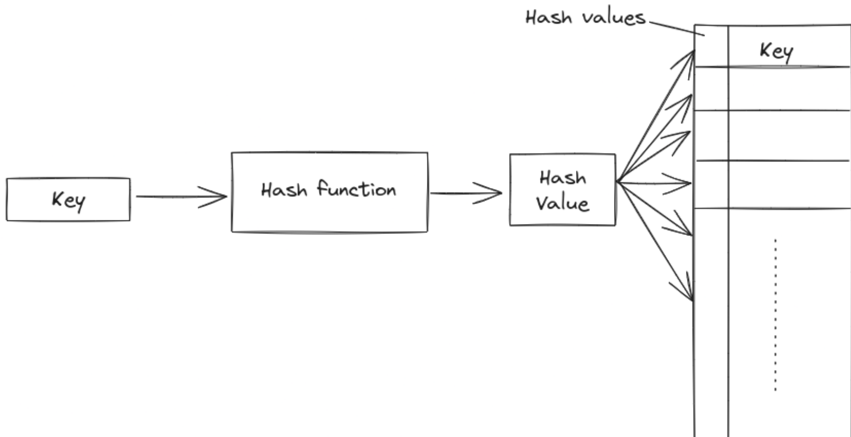
10.5. Bibliography

- A. G. Konheim Hashing in computer science: Fifty years of slicing and dicing (Wiley) (2010)
- T. Mailund The joys of hashing: Hash table programming with c (Apress) (2019)

- J. Pieprzyk & B. Sadeghiyan Design of hashing algorithms (Springer) (2014)

11. Hash Table Search

Hash Table Search is a searching technique that uses a hash function to efficiently retrieve data from a collection (usually an array) based on a unique key. It offers rapid data retrieval by converting the key into an index within the array where the desired data is stored.



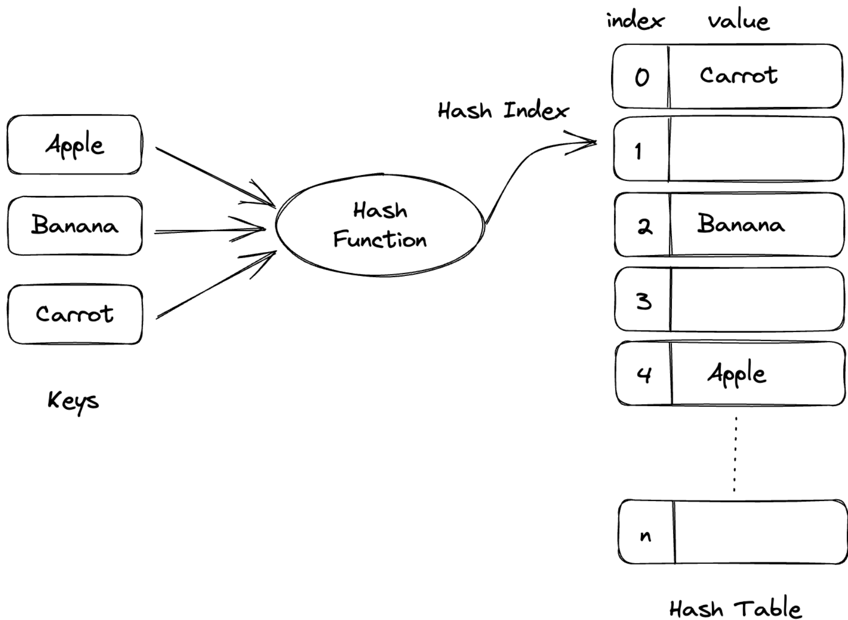
The hash function maps the key to an index, allowing for direct access to the corresponding data. This approach provides constant-time average case performance for retrieval, making Hashing Search highly efficient.

However, Hashing Search has some considerations:

- **Hash Collisions:** Different keys may hash to the same index, causing collisions. Techniques like chaining or open addressing are used to handle collisions and ensure accurate data retrieval.
- **Hash Function Quality:** The efficiency of Hashing Search relies on a good hash function that distributes keys evenly across the array. A poor hash function can lead to clustering and degrade performance.
- **Space Overhead:** Hashing Search may use extra space for hash table storage, especially in situations with frequent collisions.
- **No Sorting:** Hashing Search is primarily suited for retrieval and not for searching ranges or maintaining sorted data.

11.1. How it works

- Create a hash table data structure with an array and hash function.
- Insert key-value pairs into the hash table using the hash function to compute the index.
- To find a target element, apply the hash function to the target key to determine its index in the hash table.
- If the element is found at that index, the search is successful, and we retrieve the corresponding value.
- In case of hash collisions (when two elements map to the same index), use techniques like chaining or open addressing to handle them efficiently.



11.2. Implementation

```
class KeyValue:
    def __init__(self, key, value):
        self.key = key
```

```
        self.value = value

TABLE_SIZE = 10

def hash(key):
    # Basic hash function that calculates the sum
    # of ASCII values of characters in the key
    hash_value = 0
    for character in key:
        hash_value += ord(character)
    return hash_value % TABLE_SIZE

def hashSearch(hash_table, key):
    # Compute the hash value (index) for the given
    # key
    index = hash(key)
    # Check if the element exists at the computed
    # index in the hash table
    if hash_table[index] is not None:
        # If the key-value pair exists, compare the
        # key to ensure it is the target element
        if hash_table[index].key == key:
            # Target element found, return the
            # corresponding value
            return hash_table[index].value
    #  Target element not found in the hash table
    return None

# Example usage:

# Create a hash table and insert key-value pairs
hash_table = [None] * TABLE_SIZE
hash_table[hash("apple")] = KeyValue("apple", "fruit")
hash_table[hash("banana")] = KeyValue("banana", "fruit")
hash_table[hash("carrot")] = KeyValue("carrot", "vegetable")

# Perform the hash-based search for a target key
```

```
target_key = "banana"
result = hashSearch(hash_table, target_key)
if result is not None:
    print(f"The value for key '{target_key}' is '{result}'")
else:
    print(f"Key '{target_key}' not found in the hash table.")
```

11.3. Time Complexity Analysis

Hash-Based Search has an average-case time complexity of $O(1)$ for retrieval, making it highly efficient. In the best-case scenario, when the hash function distributes elements uniformly across the table, the search time is constant. However, in the worst-case scenario, hash collisions might cause the time complexity to degrade to $O(n)$, where n is the number of elements in the hash table.

11.4. Advantages

- **Fast Retrieval**: Offers constant-time retrieval, $O(1)$ on average for accessing elements based on keys. This efficiency remains consistent even with a large dataset.

- **Efficient Insertion and Deletion**: Adding or removing elements from a hash table is generally efficient with a well-implemented hashing function, providing $O(1)$ average-case complexity.
- **Ease of Implementation**: Implementing hash-based search is relatively straightforward, and modern programming languages often provide built-in hash table data structures.

11.5. Limitations

- **Collision Resolution**: Hash collisions occur when two different keys hash to the same location. Resolving collisions can affect performance, requiring additional techniques like chaining or open addressing.

- **Dependency on Hash Function**: The efficiency of hash-based search heavily relies on the quality of the hash function. A poor hash function might lead to more collisions, impacting overall performance.
- **Limited Range of Use**: It is less suitable for range queries or operations that require ordered traversal of elements compared to other data structures like trees.

11.6. Applications

- **Caching Mechanisms**: In computer systems, hash-based search is utilized in caching mechanisms. A hash table is employed to store frequently accessed data, enhancing access speed and reducing the need for expensive computations or I/O operations.
- **Compiler Symbol Tables**: Compilers use hash tables to store identifiers, variables, and functions, allowing quick access during parsing and compilation stages.
- **Password Verification**: Hash-based searching secures password storage by hashing and storing passwords in a hash table, enabling rapid authentication without directly storing plaintext passwords.
- **Deduplication**: Hashing is used to identify and eliminate duplicate data in storage systems, aiding in deduplication processes to optimize storage space.
- **Load Balancing and Distributed Systems**: Hashing is employed in load balancing algorithms to distribute requests and data across multiple servers in a distributed system. By using hash functions to assign requests to specific servers, the system can achieve efficient resource utilization and improve overall performance.

11.7. Bibliography

- B. J. McKenzie et al. *Selecting a hashing algorithm*, 20 SOFTWARE: PRACTICE AND EXPERIENCE 209–224 (1990), https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380200207
- G. T. HEINEMAN et al. ALGORITHMS IN A NUTSHELL: A PRACTICAL GUIDE (O'Reilly Media) (2016)

12. Bloom Filter

Imagine you have a large library with thousands of books, and you want to quickly find a specific book without having to check every single shelf. A Bloom filter can be like a librarian's assistant who helps you narrow down your search.

The librarian's assistant has a set of special cards, each with a unique code. When you ask for a book, the assistant takes the book's title and uses it to generate a code. Then, the assistant checks three different cards using that code. If all three cards are marked with a special symbol, then the book is probably in the library.

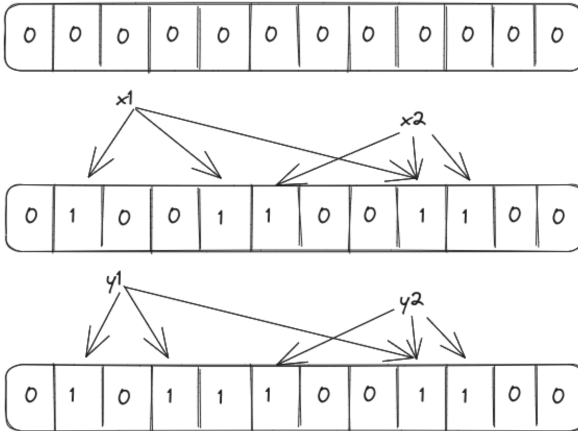
However, just like the librarian's assistant might sometimes make a mistake, the Bloom filter can also make a false positive. This means that it might tell you that a book is in the library when it actually isn't. But the probability of this happening is very small, and it's much faster than checking every single shelf.

A Bloom filter is like a quick and dirty way to check if something is in a set. It's not perfect, but it's very efficient and can be used in many different situations.

Bloom filters are often used in applications where false positives are acceptable, but false negatives are not. For example, they are used in spam filters to quickly determine whether an email is spam, even if there is a small chance of incorrectly identifying a legitimate email as spam.

12.1. How it Works

12.1.1. A high level view



The Bloom filter is initialized with an array of all 0s. Each element in the set x_1 is hashed k times, and each hash function generates a bit location. These locations are then set to 1. To check if an element y is present in the database, we first check its likelihood of being in the database using the Bloom filter. We hash this element using k hash functions. If any of the corresponding bit locations in the Bloom filter is 0, then we can definitively say that the element y is not in the database. Otherwise, if all of the corresponding bit locations are 1, then the element y is likely in the database, but there is a possibility of a false positive.

12.1.2. Detailed view

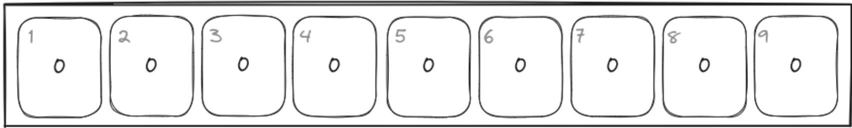
A Bloom Filter consists of the following components:

- **Bit Array**: This is the heart of the Bloom Filter, an array of bits, typically initialized to all zeros.
- **Hash Functions**: A set of k hash functions, each of which maps an element to one of the bits in the bit array.

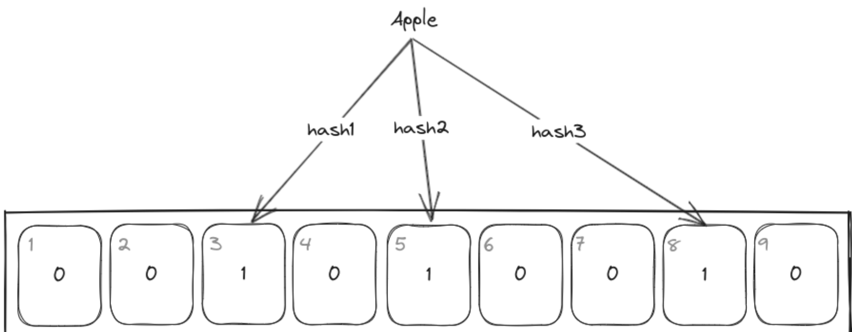
Here's how a Bloom Filter operates:

- **Initialization**: Create a bit array of size m, initialized with all zeros. Choose k hash functions. We need k number of hash functions to

calculate the hashes for a given input. When we want to add an item in the filter, the bits at k indices $h1(x), h2(x), ...hk(x)$ are set, where indices are calculated using hash functions.



- **Insertion**: To add an element to a Bloom filter, the element is hashed using each of the hash functions. The bit corresponding to each hash value is then set to 1.



- **Membership Test**: To check if an element is in the set, apply the same k hash functions to the element. If all the corresponding bits in the bit array are 1, it's likely that the element is in the set. If any bit is 0, the element is definitely not in the set.

Bloom filters are probabilistic data structures, which means that there is a small chance of false positives. A false positive occurs when a Bloom filter returns a positive result for an element that is not a member of the set. The probability of a false positive depends on the size of the bit array and the number of hash functions used.

12.2. Implementation

This is a simplified implementation for demonstration purposes, and a complete production-ready Bloom Filter would require careful consideration of hash functions and optimal parameters.

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-
import mmh3
import bitarray

class BloomFilter:
    def __init__(self, size, hash_count):
        # Initialize a bit array of the given size
        self.bit_array = bitarray.bitarray(size)
        # Set all bits in the bit array to 0
        self.bit_array.setall(0)
        # Store the size of the bit array
        self.size = size
        # Store the number of hash functions to use
        self.hash_count = hash_count

    def add(self, item):
        for i in range(self.hash_count):
            index = mmh3.hash(item, i) % self.size
            # Set the bit at the calculated index to 1
            self.bit_array[index] = 1

    def lookup(self, item):
        # Check each hash function
        for i in range(self.hash_count):
            # Calculate the index using the hash function
            index = mmh3.hash(item, i) % self.size
            # If the bit at the calculated index is 0,
            # the item is not in the filter
            if self.bit_array[index] == 0:
                return False
        # If all hash functions return 1, the item
        # may be in the filter
        return True

bf = BloomFilter(100, 3)
bf.add("key1")
```

```
print(bf.lookup("key1")) # True
print(bf.lookup("key2")) # Could be False Positive!
```

In this example, the `BloomFilter` class represents a Bloom Filter with a given size and number of hash functions. The `add` method adds items to the Bloom Filter by setting corresponding bits in the bit array. The `contains` method checks if items are present in the Bloom Filter by checking the corresponding bits in the bit array using multiple hash functions.

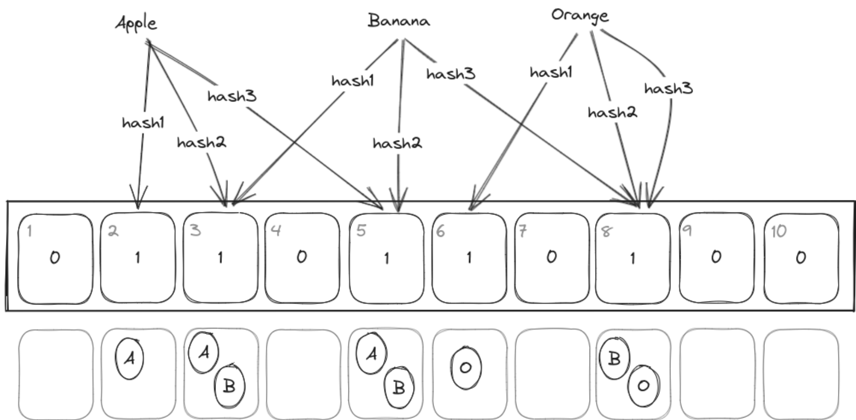
Please note that this is a simplified implementation, and a complete Bloom Filter would require choosing appropriate hash functions, determining the optimal number of hash functions, and considering the acceptable false positive rate.

12.3. Visualization

Here is what the data may look like when we add multiple words to our bit array by running multiple hash functions.

12.3.1. Adding an item to the Bloom Filter

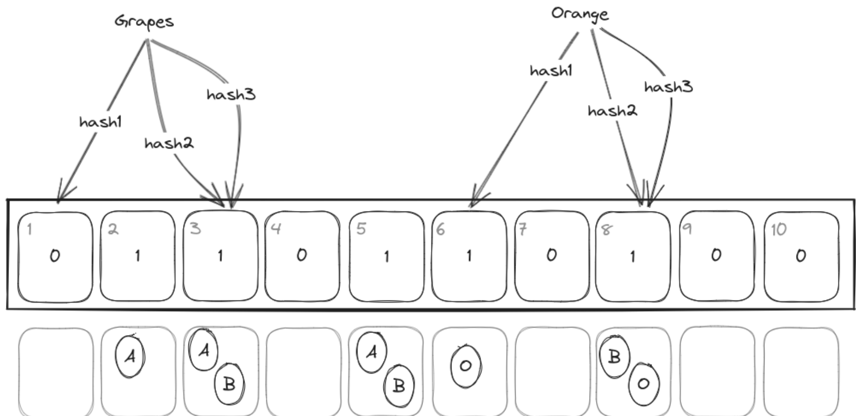
- The element is hashed through k hash functions
- The modulo n (length of bit array) operation is executed on the output of the hash function to identify the k array positions
- The bits at all identified blocks are set to one There is a probability that some bits on the array are set to one multiple times due to hash collisions.



12.3.2. Check the membership of an item

The following operations are executed to check if an item is a member of the bloom filter:

- The item is hashed through the same *k* hash functions
- The modulo n (length of bit array) operation is executed on the output of the hash functions to identify the k array positions
- Verify if all the bits at identified blocks are set to one



In the above figure, you can see that Grapes points to a position in the bit array which is 0. If any of the identified bits are set to zero, the item is not a member of the bloom filter. If all the bits are set to one, the item might be a member of the bloom filter. The uncertainty about the membership of

an item is due to the possibility of some bits being set to one by different items or due to hash function collisions.

12.4. Time complexity Analysis

The time complexity analysis of Bloom Filter operations involves considering the hash calculations and bit manipulations for adding elements and querying membership. Here's a breakdown of the time complexity for various operations in the Bloom Filter:

1. **Adding Elements (Insertion):**
    - Adding an element involves applying hash functions and setting corresponding bits in the bit array.
    - Since the number of hash functions is constant (denoted as "k"), the time complexity for adding an element is $O(k)$.
2. **Querying Membership (Contains Operation):**
    - Querying membership also involves applying hash functions and checking corresponding bits in the bit array.
    - Since the number of hash functions is constant (denoted as "k"), the time complexity for querying membership is $O(k)$.
3. **False Positive Rate and Hash Collisions:**
    - The primary trade-off of Bloom Filters is their probabilistic nature, which can lead to false positives due to hash collisions.
    - The number of hash collisions is determined by factors such as the number of hash functions, size of the bit array, and the number of elements added.
    - The probability of a false positive can be approximated as $(1 - e^{-\frac{kn}{m}})^k$, where "k" is the number of hash functions, "n" is the number of elements added, and "m" is the size of the bit array.

Overall, the time complexity of Bloom Filter operations (insertion and querying) is primarily determined by the constant number of hash functions used, which is denoted as "k". This makes Bloom Filters efficient for both insertion and querying, as these operations are independent of the number of elements in the filter.

It's important to note that the primary trade-off of Bloom Filters is their probabilistic nature, which introduces the possibility of false positives. The choice of hash functions, number of hash functions, and the size of the bit array affects both the false positive rate and the time complexity of operations.

12.5. Advantages

- **Memory-Efficient:** Bloom Filters use significantly less memory than traditional data structures like hash tables.
- **Constant-Time Lookups:** Regardless of the size of the dataset, Bloom Filters provide constant-time membership testing.
- **Parallel Processing:** Bloom Filters are highly parallelizable, making them suitable for distributed systems and multi-threaded applications.

12.6. Limitations

While Bloom Filters offer many advantages, they also have limitations:

- **False Positives:** Bloom Filters may produce false positives, meaning they can mistakenly indicate that an element is in the set when it's not. The probability of false positives can be controlled by adjusting the size of the bit array and the number of hash functions.
- **No Deletion:** Once an element is inserted into a Bloom Filter, it cannot be efficiently removed.
- **Limited to Set Membership Testing:** Bloom Filters are not suitable for tasks that require retrieval of the stored data.

12.7. Applications

- **Spelling Correction:** Spell checkers can use Bloom Filters to quickly check if a word is in the dictionary.
- **Network Routers:** Bloom Filters help routers determine whether an IP address is in a blacklist.
- **Duplicate Detection:** In distributed systems, Bloom Filters assist in detecting duplicate data blocks.

12.8. Bibliography

- R. Patgiri et al. Bloom filter: A data structure for computer networking, big data, cloud computing, internet of things, bioinformatics and beyond (Elsevier Science) (2023)
- Burton H. Bloom *Space/time trade-offs in hash coding with allowable errors*, 13 Communication. ACM 422–426 (1970), https://doi.org/10.1145/362686.362692
- O'Sullivan, Bryan, et al. Real World Haskell. United States, O'Reilly Media, 2008.

Part III.

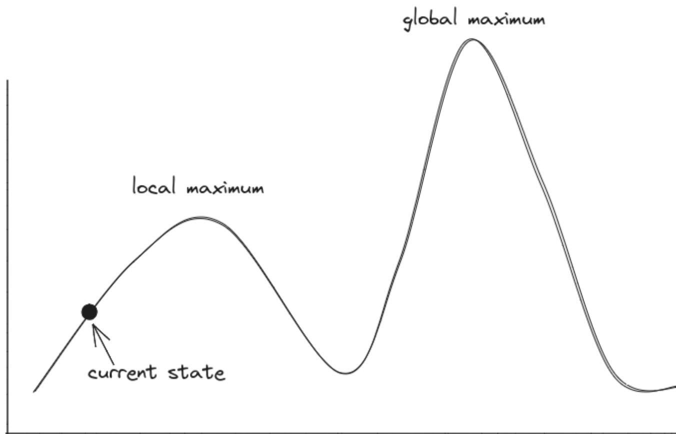
AI Search Algorithms

14. Hill Climbing Algorithm

Hill climbing is a simple optimization algorithm used in artificial intelligence, particularly for solving search and optimization problems. It's inspired by the metaphor of climbing a hill to reach the highest point, where the goal is to find the peak (the maximum or minimum value) of a function.

Here is a simple analogy to help understand hill climbing:

Imagine you are standing on a mountain top. You want to find the highest point on the mountain, but you can only move up or down. If you are on a convex mountain, then hill climbing will eventually lead you to the highest point. However, if you are on a non-convex mountain, then hill climbing could lead you to a local peak, which is not the highest point on the mountain.



To avoid getting stuck on a local peak, you could restart your climb from a different starting point, or you could use a more complex algorithm that takes into account the shape of the mountain.

The basic idea of the hill climbing algorithm is straightforward:

1. Start from a random or given initial solution.

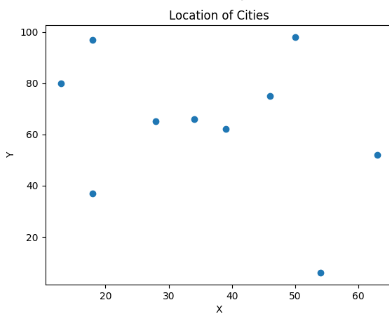
2. Repeatedly make small improvements by changing the current solution slightly to reach a better solution.

3. Continue this process until no better solutions can be found or a stopping criterion is met.

The algorithm operates under the assumption that moving in the direction of steepest ascent or descent (depending on whether you're maximizing or minimizing) will lead to an optimal solution.

14.1. How it works

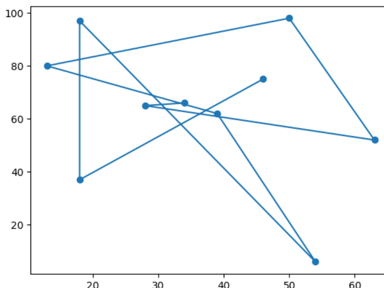
Let's take the Traveling Salesman Problem (TSP) as an example.



Using a hill climbing approach, it is easy to find an initial solution that visits all the cities, but it is likely to be poor compared to the optimal solution. The algorithm starts with such a solution and makes small improvements to it, such as switching the order in which two cities are visited. Eventually, the algorithm is likely to obtain a much shorter route.

Here is an overview:

1. **Initialize**: Start from a random initial solution.



2. **Generate Neighbors**: Generate neighboring solutions by making small, incremental changes to the current solution. We can achieve this by following a specific procedure: first, duplicate the current solution, and then perform a swap between two cities. This process results in a slightly altered yet valid solution.

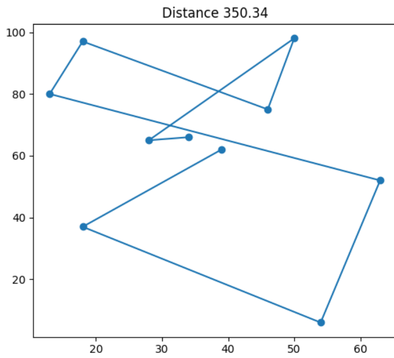
generated neighbours	dist
[0 6 1 7 8 3 4 2 5 9]	397.20
[1 0 6 7 8 3 4 2 5 9]	450.49
[7 0 1 6 8 3 4 2 5 9]	446.68
[8 0 1 7 6 3 4 2 5 9]	337.17
[3 0 1 7 8 6 4 2 5 9]	404.41
[4 0 1 7 8 3 6 2 5 9]	424.29
[2 0 1 7 8 3 4 6 5 9]	415.62
....	
....	
....	
[6 0 8 7 1 3 4 2 5 9]	328.04
[6 0 3 7 8 1 4 2 5 9]	426.72
[6 0 4 7 8 3 1 2 5 9]	413.30
[6 0 2 7 8 3 4 1 5 9]	440.22
[6 0 5 7 8 3 4 2 1 9]	435.30
[6 0 9 7 8 3 4 2 5 1]	429.07
[6 0 1 8 7 3 4 2 5 9]	439.03
[6 0 1 3 8 7 4 2 5 9]	410.90

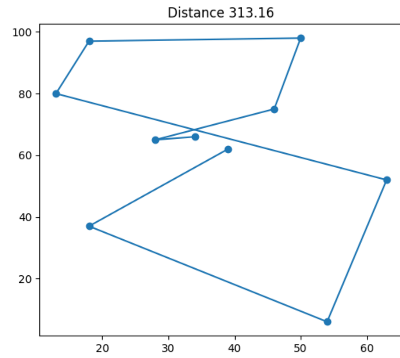
3. **Evaluate**: Calculate the objective function value (the value to be optimized) which in our current case is the total distance of the currently selected neighbors.

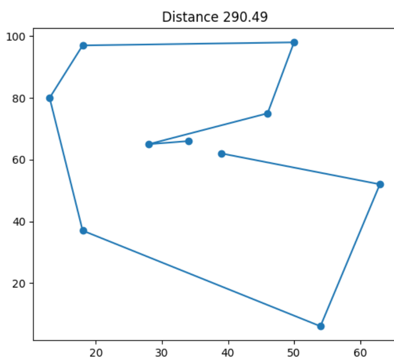
4. **Select**: Choose the neighboring solution with the best objective function value. If it's better than the current solution, move to the selected neighbor. Otherwise, terminate the algorithm.

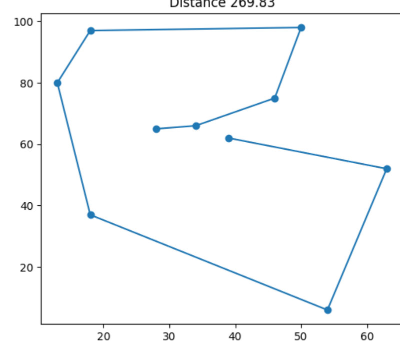
5. **Repeat**: Repeat steps 2 to 4 until a termination condition is met (e.g., a maximum number of iterations or no improvement for a specified number of iterations).

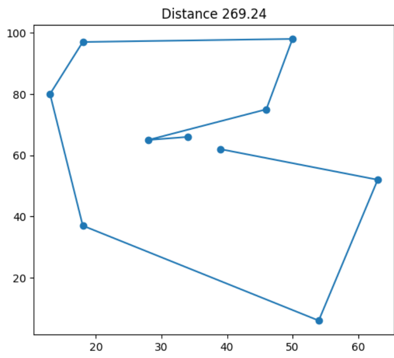
Here is the plot depicting the overall optimization of the distance.

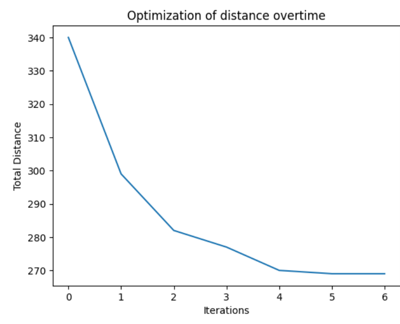

Distance 350.34


Distance 313.16


Distance 290.49


Distance 269.83


Distance 269.24


Optimization of distance overtime
Total Distance
Iterations

14.2. Implementation

```
import numpy as np

# Function to create an adjacency matrix based
# on Euclidean distances between points
def adjacency_matrix(coordinate):
    matrix = np.zeros((len(coordinate), len(coordinate)))

    # Loop through each point
    for i in range(len(coordinate)):
        for j in range(i + 1, len(coordinate)):
            # Calculate Euclidean distance between
            # points i and j
            p = np.linalg.norm(
                coordinate[i] - coordinate[j])
            # Set distance in both positions of the matrix
            # (symmetric for undirected graph)
            matrix[i][j] = p
            matrix[j][i] = p
    return matrix

# Function to generate a random solution
# (permutation of indices)
def solution(matrix):
    return np.random.permutation(len(matrix))

# Calculate total distance of the path in a solution
def total_distance_of_path(matrix, solution):
    return sum(matrix[solution[i]][solution[i - 1]]
               for i in range(len(solution)))

# Generate neighboring solutions by swapping cities
def get_neighbors(solution):
    neighbors = []
    for i in range(len(solution)):
        for j in range(i + 1, len(solution)):
```

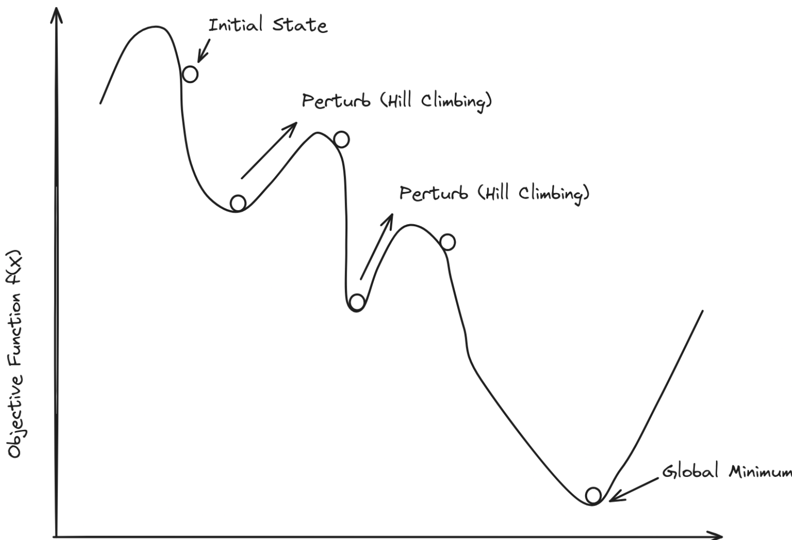
```
            neighbor = solution.copy()
            # Swap positions i and j
            neighbor[i] = solution[j]
            neighbor[j] = solution[i]
            neighbors.append(neighbor)
    return neighbors

# Find the best neighbor based on the total
# distance of the path
def best_neighbors(matrix, solution):
    neighbors = get_neighbors(solution)
    best_neighbour = neighbors[0]
    best_route_length = total_distance_of_path(matrix,
                                              best_neighbour)
    for neighbour in neighbors:
        current_route_length = total_distance_of_path(matrix,
                                                     neighbour)
        if current_route_length < best_route_length:
            best_route_length = current_route_length
            best_neighbour = neighbour
    return best_neighbour, best_route_length

# Hill Climbing algorithm to find the shortest path
def hill_climbing(coordinate):
    distance_matrix = adjacency_matrix(coordinate)
    current_solution = solution(distance_matrix)
    current_distance = total_distance_of_path(distance_matrix,
                                             current_solution)
    best_neighbor, best_neighbor_path = best_neighbors(
        distance_matrix, current_solution)

    # Continue until the best neighbor has a
    # longer path than the current solution
    while best_neighbor_path < current_distance:
        current_solution = best_neighbor
        current_distance = best_neighbor_path
        print(current_distance)
```

16. Simulated Annealing

Imagine sculpting a perfect statue from a block of metal. You heat the metal (representing high energy in the system) to a molten state, allowing it to cool gradually (annealing). As it cools, the metal's particles arrange themselves in a low-energy state, minimizing defects and forming the desired shape. Simulated Annealing operates similarly by exploring a solution space. It accepts less optimal solutions early on (high energy) and gradually refines them to reach an optimal or near-optimal solution (low energy).

Simulated Annealing (SA) is a heuristic optimization technique inspired by the annealing process in metallurgy, where metals are slowly cooled to reach a state of minimum energy and reduce defects. SA mimics this process to find optimal or near-optimal solutions in complex search spaces, especially in combinatorial optimization problems[]

16.1. How it works:



1. **Initialization**: Start with an initial solution and set an initial temperature and cooling rate. The initial solution can be generated randomly or using a heuristic method.

2. **Temperature Annealing**: The algorithm operates in iterations, each associated with a specific temperature. The temperature decreases over time according to a cooling schedule, which controls the rate of exploration. Common cooling schedules include linear and exponential cooling.

3. **Perturbation**: In each iteration, perturb the current solution to generate a neighboring solution. The perturbation can involve small changes to the current solution or more extensive modifications, depending on the problem domain.

4. **Evaluation**: Calculate the cost or objective function value of the current solution and the neighboring solution.

5. **Acceptance**: Decide whether to accept or reject the neighboring solution based on the cost difference and the current temperature. The Metropolis criterion is often used for acceptance:

    - If the neighboring solution is better (lower cost), accept it.
    - If the neighboring solution is worse (higher cost), accept it with a certain probability, which decreases as the temperature decreases. This probability is controlled by the Boltzmann distribution formula.

6. **Iteration**: Repeat steps 3-5 for a predetermined number of iterations or until a termination condition is met (e.g., reaching a target temperature).

7. **Termination**: When the cooling schedule completes, return the best solution encountered during the entire process as the output.

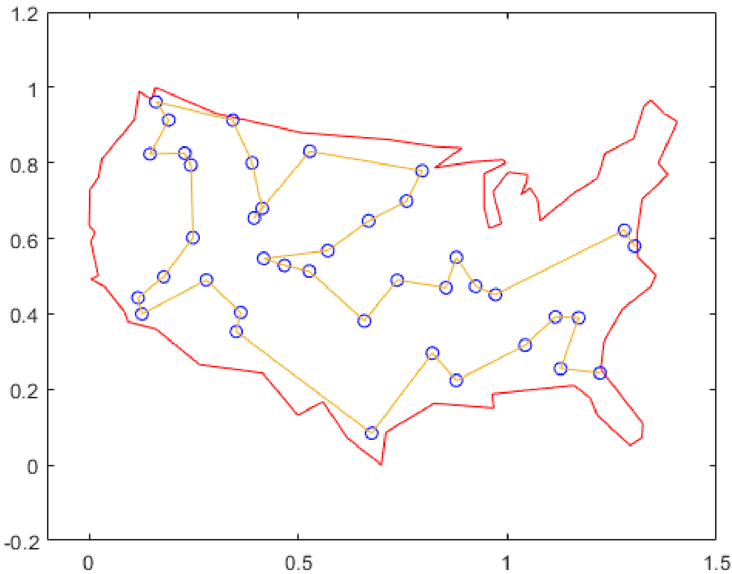
Key characteristics and considerations of Simulated Annealing include:

- **Exploration vs. Exploitation**: The algorithm balances exploration (searching for new solutions) and exploitation (improving the current solution).

- **Global Optimization**: Simulated Annealing can find global optima, making it suitable for problems with complex, multi-modal solution spaces.
- **Stochastic Nature**: The acceptance of worse solutions is controlled by randomness, allowing the algorithm to escape local optima.
- **Parameter Tuning**: The choice of initial temperature, cooling schedule, and acceptance probability function can significantly impact the algorithm's performance and convergence.
- **Convergence**: Simulated Annealing typically converges to an approximate solution as the temperature decreases.

16.2. The Traveling Salesman Problem (TSP) using Simulated Annealing

One example problem that can be effectively solved using Simulated Annealing is the Traveling Salesman Problem (TSP). The TSP is a classic optimization problem in which a salesperson is given a list of cities and must find the shortest possible route that visits each city exactly once before returning to the starting city, aiming to minimize the total distance traveled.



Simulated Annealing can be applied to find a near-optimal solution for the TSP by iteratively exploring different routes and gradually improving the solution over time.

16.3. Simulated Annealing for TSP Problem: Step-by-Step Guide

Step 1: Define Functions

- **Distance function:** `distance(city1, city2)`: Calculates the Euclidean distance between two cities.

$$\text{distance}(city1, city2) = \sqrt{(city1.x - city2.x)^2 + (city1.y - city2.y)^2}$$

This formula computes the Euclidean distance between two points (cities) in a two-dimensional space using their coordinates.

- **Objective function:** `total_distance(tour, cities)`: Calculates the total tour length.

$$\text{distance}(tour, cities) = \sum_{i=1}^{n-1} \text{distance}(cities[tour[i]], cities[tour[i+1]])$$

This formula calculates the total distance of a tour by summing the distances between consecutive cities in the tour, where (n) represents the number of cities in the tour.

Step 2: Initialize Parameters

- **Initial temperature:** `initial_temperature` (high value)
- **Cooling rate:** `cooling_rate` (between 0 and 1, controls temperature decrease rate)
- **Number of iterations:** `num_iterations` (controls the number of times the algorithm attempts to improve the solution)
- **Number of cities:** `num_cities` (determined by the problem)
- **Current tour:** `current_tour` (randomly chosen initial tour)
- **Best tour:** `best_tour` (initialized same as current_tour)
- **Current temperature:** `current_temperature` (set to initial_temperature)

Step 3: Loop for Iterations

```
for iteration in range(num_iterations):
```

Step 4: Perturb the Current Tour

- Randomly select two cities using `random.sample(range(num_cities), 2)`
- Swap their positions to create a new neighbor tour `new_tour`

Step 5: Calculate Cost Difference

- Calculate the total distance of the current tour: `current_cost = total_distance(current_tour, cities)`
- Calculate the total distance of the new tour: `new_cost = total_distance(new_tour, cities)`
- Calculate the difference in cost: `delta_cost = new_cost - current_cost`

Step 6: Accept or Reject New Tour

- If the new tour has a lower cost (improvement): `delta_cost < 0`
  - Accept the new tour: `current_tour = new_tour.copy()`
  - If the new tour has the best cost so far:

    * Update the best tour: `best_tour = new_tour.copy()`

- Else (new tour has higher cost):
    - Accept the new tour with probability `P = math.exp(-delta_cost / current_temperature)`
    - If a random number between 0 and 1 is less than P, accept the new tour, else keep the current tour.

Step 7: Update Temperature

- Reduce the temperature by multiplying it with the cooling rate: `current_temperature *= cooling_rate`

Step 8: Repeat
Continue iterating through steps 4 to 7 until the maximum number of iterations is reached or another stopping criterion is met.
Step 9: Output Results

- The best tour found during the iterations is stored in `best_tour`.
- The total distance of the best tour can be calculated using `total_distance(best_tour, cities)`.

Simulated Annealing allows the algorithm to explore a wide range of possible tours, including suboptimal ones, while gradually reducing the likelihood of accepting worse tours as the temperature decreases. This exploration-exploitation trade-off helps in finding a good approximation of the optimal TSP tour.

The TSP is just one example of a combinatorial optimization problem that can be solved using Simulated Annealing. Simulated Annealing has also been applied to various other optimization problems, including job scheduling, parameter optimization in machine learning, and layout optimization in integrated circuits, among others.

16.4. Implementation

Solving the Traveling Salesman Problem (TSP) using Simulated Annealing involves implementing the algorithm and creating a cost function to evaluate tour lengths. Below is a Python program to solve the TSP using Simulated Annealing:

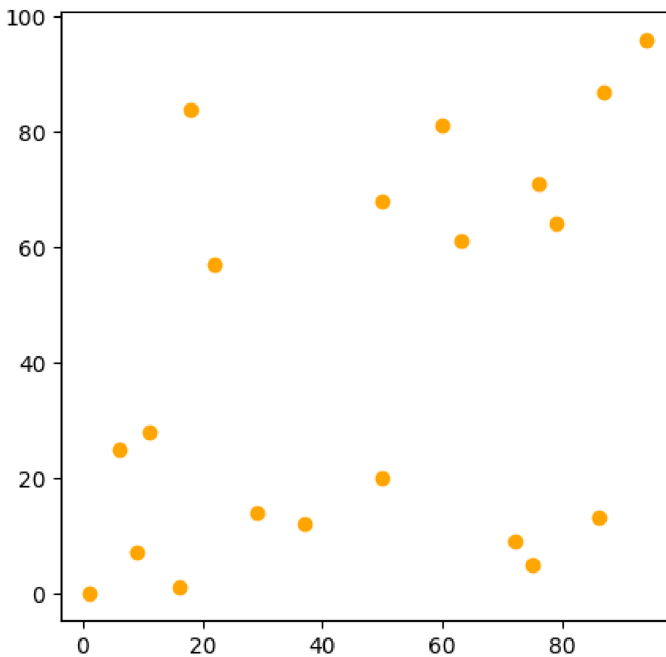
```python
import random
import math
import numpy as np

# Define a class to represent a city with x and y coordinates
class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y

# Define a function to calculate the Euclidean distance between two
↪  cities
def distance(city1, city2):
    return math.sqrt((city1.x - city2.x) ** 2 + (city1.y - city2.y) **
    ↪  2)

# Define the objective function to calculate the total tour length
def total_distance(tour, cities):
    return sum(distance(cities[tour[i]], cities[tour[i + 1]])
               for i in range(len(tour) - 1))

# Simulated Annealing function to find the best tour
def simulated_annealing(cities,
                        initial_temperature,
                        cooling_rate,
                        num_iterations):
    num_cities = len(cities)

    # Initialize a random tour
    current_tour = random.sample(range(num_cities), num_cities)
    best_tour = current_tour.copy()

    current_temperature = initial_temperature

    for iteration in range(num_iterations):
        # Perturb the current tour by swapping two random cities
        new_tour = current_tour.copy()
```

```
        i, j = random.sample(range(num_cities), 2)
        new_tour[i], new_tour[j] = new_tour[j], new_tour[i]

        # Calculate the cost of the current and new tours
        current_cost = total_distance(current_tour, cities)
        new_cost = total_distance(new_tour, cities)

        # Decide whether to accept the new tour based on
        # cost and temperature
        if (new_cost <
            current_cost or random.random() <
            math.exp((current_cost - new_cost) /
 ↪  current_temperature)):
            current_tour = new_tour.copy()
            if new_cost < total_distance(best_tour, cities):
                best_tour = new_tour.copy()

        # Reduce the temperature
        current_temperature *= cooling_rate

    return best_tour, total_distance(best_tour, cities)

# Example usage
np.random.seed(1)
# Generate array of random coodinates for city locations
coordinates = np.random.randint(0, 100, (20, 2))
cities = []
# Build Cities coordinates with their coordinates (x, y)
for c in coordinates:
    cities.append(City(c[0], c[1]))

# Set Simulated Annealing parameters
initial_temperature = 1000.0
cooling_rate = 0.995
num_iterations = 10000

# Solve the TSP using Simulated Annealing
```

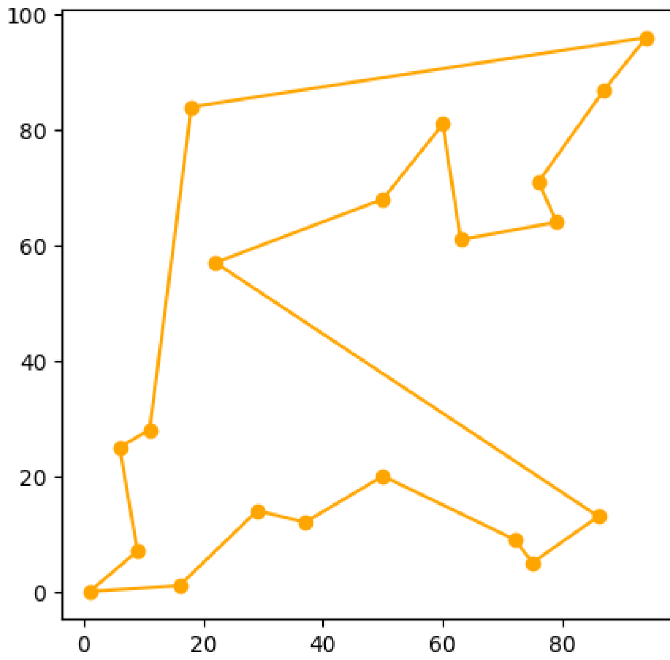
```
best_tour, shortest_distance = simulated_annealing(cities,

↪  initial_temperature,
                                                    cooling_rate,
                                                    num_iterations)

# Print the best tour and its total distance
print("Best Tour:", best_tour)
print("Shortest Distance:", shortest_distance)
```

16.5. Visualization

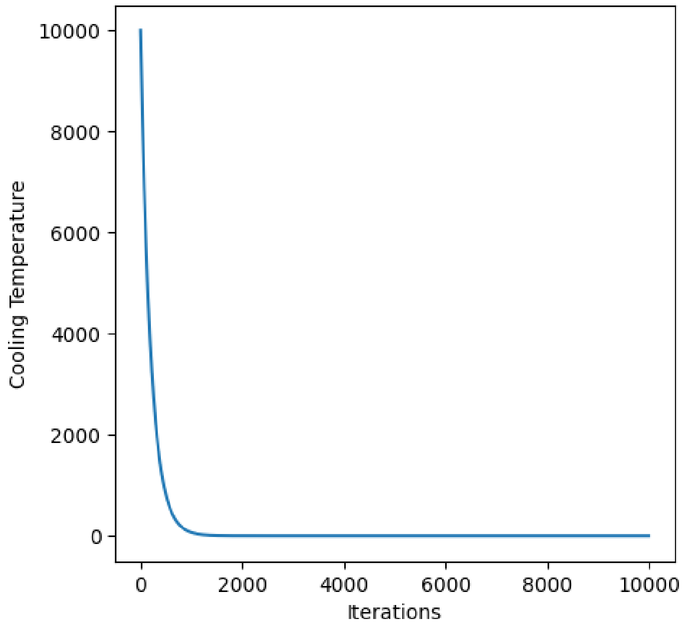
- A random dataset of city coordinates



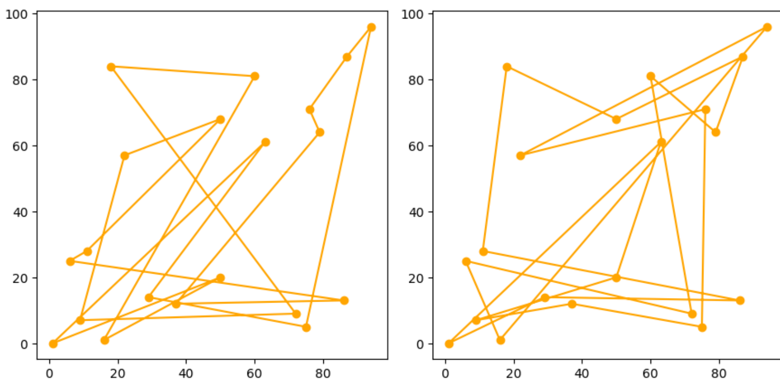
- The path found after simulated annealing

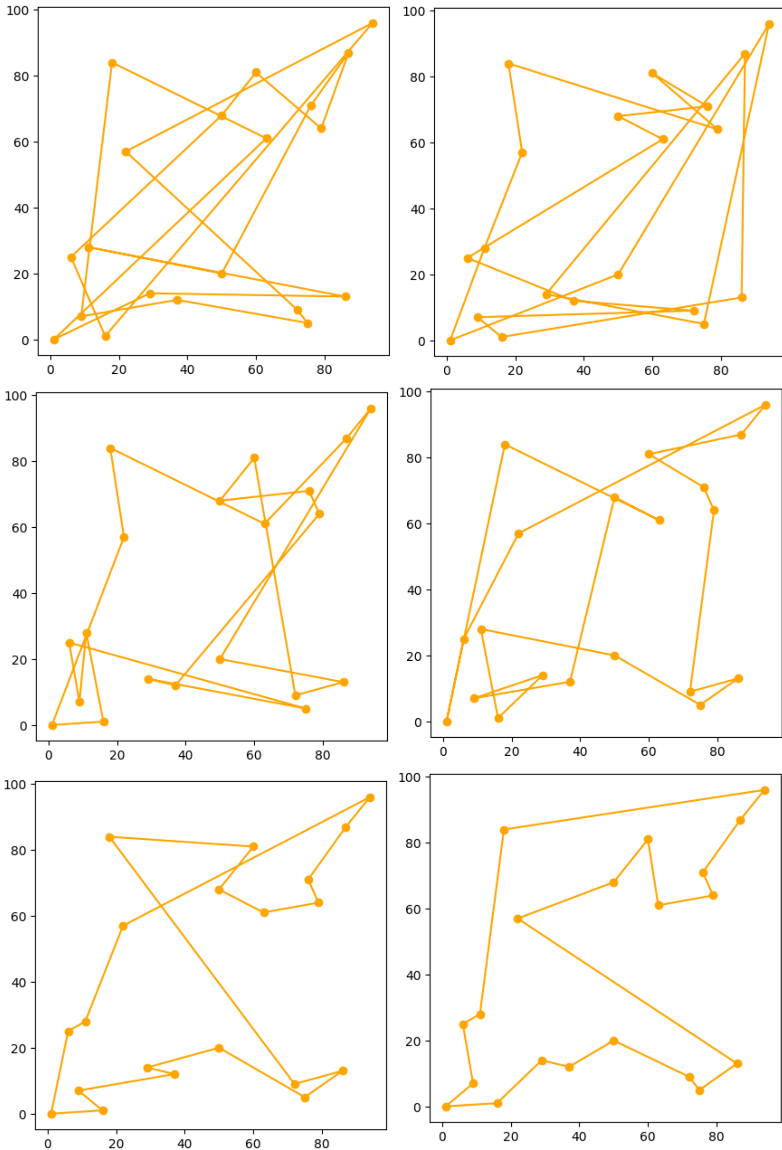


- Optimization over the iterations with cooling temperature approaching 0 with each iteration



- View of the Optimizations over the iterations (Only picking a small set)





16.6. Advantages:

- **Global Optimization:** Simulated Annealing is capable of finding global optima in complex, multi-dimensional search spaces, making it suitable for a wide range of optimization problems where finding

the exact solution is difficult.

- **Robustness:** It is robust and versatile, applicable to problems with various objective functions, constraints, and irregular search spaces, such as combinatorial optimization, scheduling, and parameter optimization.
- **Escape Local Optima:** Unlike some local search algorithms, Simulated Annealing has the ability to escape local optima by probabilistically accepting worse solutions during the early stages of the search, allowing for exploration of the solution space.
- **Parallelizable**: SA can be easily implemented in a parallel manner, allowing for faster computation on problems with large solution spaces. This can significantly reduce the time required to find the optimal solution.
- **Controlled Exploration-Exploitation Tradeoff:** The algorithm's temperature parameter controls the balance between exploration and exploitation, allowing controlled exploration of the search space in the early stages and gradual exploitation of better solutions as the temperature decreases.

16.7. Limitations:

- **Parameter Sensitivity:** Simulated Annealing's performance can be sensitive to the choice of parameters, such as initial temperature, cooling schedule, and the number of iterations, requiring careful tuning for optimal results.
- **Computationally Intensive:** For complex problems or large search spaces, Simulated Annealing might require a high number of iterations to converge, making it computationally expensive compared to some other optimization methods.
- **Slow Convergence:** It may converge slowly, especially in situations where the temperature needs to decrease gradually to explore the solution space adequately, resulting in longer execution times.
- **Heuristic Nature:** Being a heuristic method, there's no guarantee that Simulated Annealing will always find the optimal solution, and the quality of the obtained solution might vary depending on the problem instance and parameters chosen.
- **Difficulty in Handling Constraints:** Incorporating constraints into Simulated Annealing can be challenging, especially in problems

with complex constraints, requiring specialized handling methods.

16.8. Applications

Simulated Annealing (SA) is a versatile optimization algorithm that has gained widespread popularity due to its ability to effectively solve complex problems with large state spaces. Its applications span across a wide range of domains, including:

1. Combinatorial Optimization: SA is particularly well-suited for solving combinatorial optimization problems, where the number of possible solutions is vast and exhaustive search is impractical. Examples include:

- **Traveling Salesman Problem (TSP):** Finding the shortest route for a salesman to visit all cities and return to the starting point.
- **Graph Coloring:** Assigning colors to vertices of a graph so that no two adjacent vertices share the same color.
- **Scheduling:** Optimizing resource allocation and task scheduling in production planning and project management.

2. Parameter Optimization: SA can be used to tune parameters in various models and algorithms to improve their performance. For instance:

- **Neural Network Training:** Adjusting the weights and biases of a neural network to minimize the error in prediction or classification tasks.
- **Control System Optimization:** Tuning the parameters of a control system to achieve desired performance and stability.
- **Machine Learning Hyperparameter Optimization:** Finding the optimal values of hyperparameters in machine learning algorithms to maximize their effectiveness.

3. Circuit Design: SA plays a crucial role in circuit design, particularly in:

- **Placement and Routing:** Optimizing the placement of components on a circuit board and routing connections between them to minimize wirelength and power consumption.

- **Analog Circuit Optimization:** Tuning the parameters of analog circuits to achieve desired performance characteristics, such as gain, bandwidth, and power consumption.

4. Image Processing: SA has been successfully applied in image processing tasks, including:

- **Image Enhancement:** Improving the quality of images by reducing noise, sharpening details, and enhancing contrast.
- **Image Restoration:** Recovering degraded or corrupted images by removing artifacts, distortions, or noise.
- **Image Segmentation:** Segmenting images into meaningful regions or objects based on intensity, texture, or other image features.

5. Scientific Computing: SA finds applications in various scientific computing areas, such as:

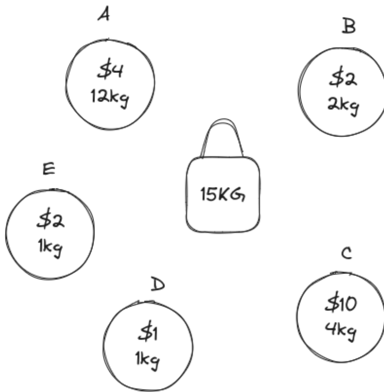
- **Protein Folding:** Predicting the three-dimensional structure of proteins from their amino acid sequences, which is crucial for understanding protein function.
- **Molecular Dynamics Simulations:** Optimizing the parameters of molecular dynamics simulations to accurately model the behavior of molecules and materials.
- **Computational Physics:** Solving optimization problems arising in physics, such as determining the optimal configuration of particles in a system.

16.9. Bibliography

- P. Venkataraman Applied optimization with MATLAB programming (Wiley) (2009)
- P. J. van Laarhoven & E. H. Aarts Simulated annealing: Theory and applications (Springer Netherlands) (2013)
- S. Kirkpatrick et al. *Optimization by simulated annealing*, 220 Science 671–680 (1983), https://www.science.org/doi/abs/10.1126/science.220.4598.671

18. Branch and Bound Algorithms

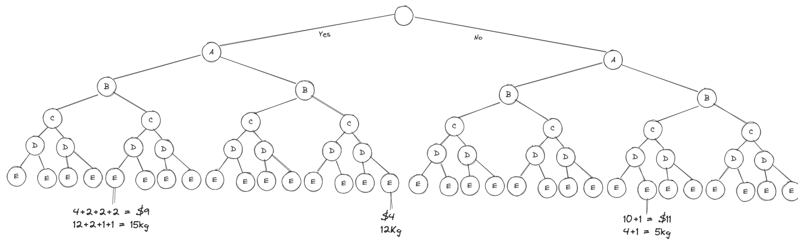
Suppose you have a set of items, each with its own specific weight and value. You also have a knapsack with a maximum weight capacity of 15 kg. Your goal is to determine which combination of these items will provide the highest total value without exceeding the weight limit of your knapsack.



A brute force approach for finding the optimal solution involves generating all possible combinations, which in this case is 2^n for n items. Each combination's total value is calculated, and the one selected as optimal and feasible must satisfy two conditions:

- The total weight must be less than or equal to the knapsack's capacity.
- Among all feasible combinations, it must have the highest total weight.

In this context, for 5 items, there are a total of $2^5 = 32$ possible combinations. The below constructed tree represents the decision tree for the brute force approach. The left node of any node denotes that we picked the item, and the right node denotes that we dropped the item.



As we can see, the search space for combinatorial optimization problems (COPs) grows exponentially with higher input. This is why COPs are also called NP Hard problems, meaning that they cannot be solved by a polynomial-time algorithm. There are several non-brute force approaches to solving COPs, including the greedy approach, dynamic programming, and branch and bound.

18.1. Branch and Bound algorithm

The Branch and Bound algorithm is an optimization technique used to solve combinatorial problems. Combinatorial problems involve making selections or decisions from a finite set of possibilities, and the goal is to find the best combination that optimizes a certain objective function. The Branch and Bound algorithm systematically explores the search space to find the best solution while avoiding inefficient or redundant evaluations.

Here's how the Branch and Bound algorithm works:

1. **Initialization**: The algorithm starts with an initial solution, typically set to an empty or partial solution. It also initializes a lower bound (usually set to negative infinity) and an upper bound (usually set to positive infinity).

2. **Branching**: The algorithm divides the problem into smaller subproblems, often by considering different choices or options. This creates a branching structure similar to a tree. Each branch represents a different possible decision.

3. **Bounding**: At each node of the tree, the algorithm computes an upper and lower bound for the objective function. These bounds are used to evaluate whether the node is worth further exploration. A bound is an optimistic estimate of how good a partial solution (or node) may be once completed; if it is not better than current best solution then there is no need to evaluate the children of that node.

4. **Pruning**: The algorithm prunes (eliminates) nodes from the search space if their bounds indicate that they cannot lead to a better solution than the current best-known solution. This pruning helps reduce the number of evaluations.
5. **Selection**: The algorithm selects the next node to explore based on a specific strategy. Common strategies include choosing nodes with the highest potential or nodes that are most promising according to the bounds.
6. **Exploration**: The selected node is explored further by branching into smaller subproblems. This process continues recursively until the search space is fully explored or until a termination condition is met.
7. **Updating Bounds**: As the algorithm explores the search space, it updates the lower and upper bounds for the objective function based on the solutions found so far.
8. **Termination**: The algorithm terminates when it has fully explored the search space, and no further nodes can lead to a better solution, or when a specific termination condition is met.
9. **Optimal Solution**: Once the algorithm completes, it returns the best solution found during the search, along with its objective value.

Branch and Bound is widely used for solving various optimization problems, including the Traveling Salesman Problem, the Knapsack Problem, and many others. Its key advantage is its ability to guarantee optimality, ensuring that the solution found is indeed the best possible solution within the search space. However, it can be computationally expensive for large problem instances due to the need to explore an exponential number of nodes.

18.2. How it works

Consider the knapsack problem with $n = 5$, $v = [4, 2, 10, 1, 2]$, $w = [12, 2, 4, 1, 1]$ and $W = 15$, where

- n denotes the number of items

- v is an array of values
- w is an array of weights
- W denotes the knapsack capacity

Item	Value	Weight	Value/Weight
P3	10	4	2.5
P5	2	1	2.0
P2	2	2	1.0
P4	1	1	1 .0
P1	4	12	0.33

Here is a tabular representation of our problem dataset. The last column denotes value-to-weight ratio, an important ratio which allows for a greedy approach, where you consider adding items to the knapsack in descending order of their value-to-weight ratio. This means you prioritize items that give you the most value for the least weight. So, as a pre-processing step, we first sort all items by the value-to-weight ratio.

In the knapsack problem above, we are trying to maximize the value given the constraint of maximum weight. This is a maximization problem, so we need to find the upper bound at each step and prune the search tree.

$$ub = v_i + (W - w_i) * \frac{v_{i+1}}{w_{i+1}}$$

where,

- v_i represents the value of the item at index i

- w_i represents the weight of the item at index i

- W represents the knapsack capacity, and

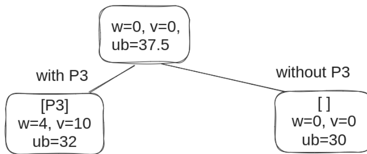
- $\frac{v_{i+1}}{w_{i+1}}$ represents the value-to-weight ratio of the next item.

w=0, v=0,
ub=37.5

We start with an empty knapsack. The Upper bound at the root node is

$$ub = 0 + (15 - 0) * 2.5 = 37.5$$

The value and weight at the root node is 0 since we haven't picked any item yet.

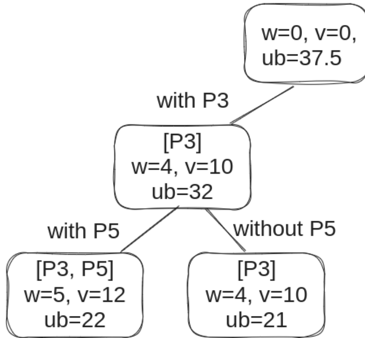


Next we branch into two nodes, the left one with the next item being picked into the knapsack and the right node without the next item. For the left node, we pick $P3$. We add it to our root and calculate the upper bound.

$$ub = 10 + (15 - 4) * 2.0 = 32$$

For the right node, we do not pick $P3$ and calculate the upper bound using the v/w ratio of the next item, which is $P5$.

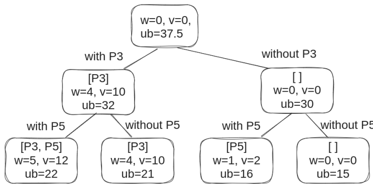
$$ub = 0 + (15 - 0) * 2.0 = 30$$



We have two upper bounds, we pick the node with higher $ub = 32$ to continue the process. Now, we branch again into two nodes, left with $P5$ and right without $P5$

$$ub(left) = 12 + (15 - 5) * 1.0 = 22$$

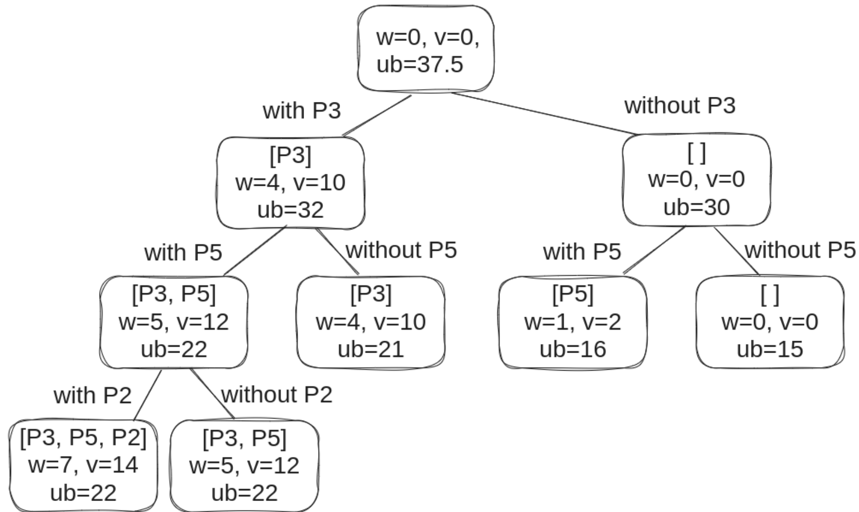
$$ub(right) = 10 + (15 - 4) * 1.0 = 22$$



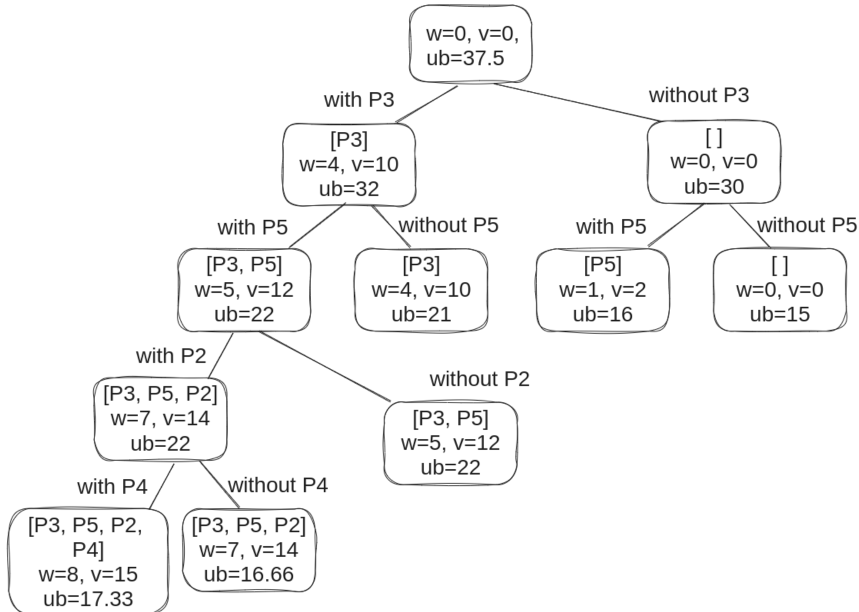
Next we pick the node whose upper bound is 30 as it is the highest upper bound in the current tree and expand from there.

$$ub(left) = 2 + (15 - 1) * 1.0 = 16.0$$

$$ub(right) = 0 + (15 - 0) * 1.0 = 15.0$$

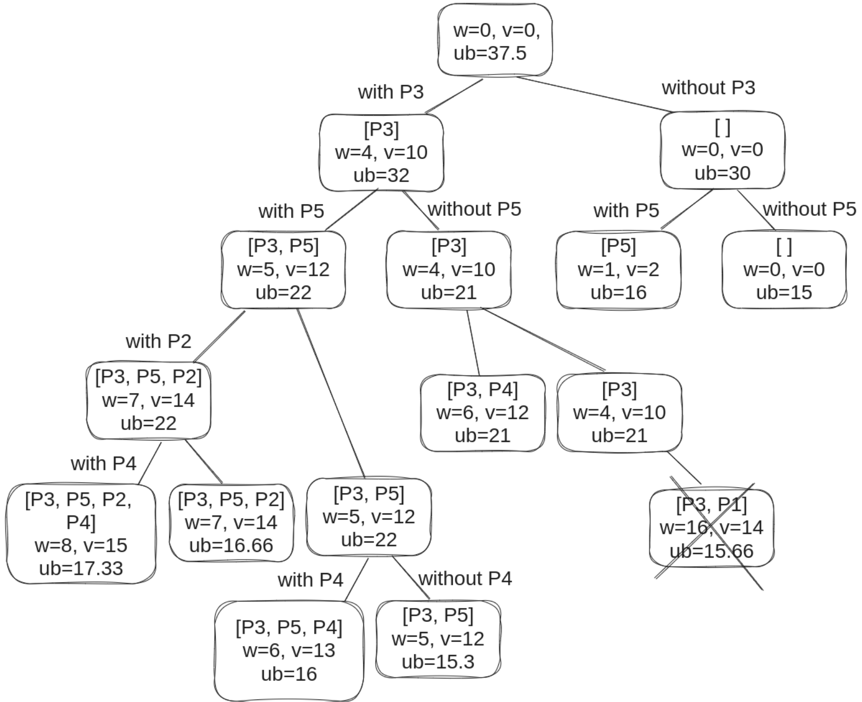


The highest upper bound now is 22, so we expand that node.
$ub(left) = 14 + (15 - 7) * 1.0 = 22$ $ub(right) = 12 + (15 - 5) * 1.0 = 22$
—–



$ub(left) = 15 + (15 - 8) * 0.33 = 17.33$
$ub(right) = 14 + (15 - 7) * 0.33 = 16.66$

Post few iterations we reach a node whose capacity exceeds the knapsack capacity, so we discard that node.



We keep running this process till we reach the end of the tree which is *Count*(*items*) and once we have reached the last item, we stop the processing and return the best combination

w=0, v=0, ub=37.5

with P3

without P3

[P3] w=4, v=10 ub=32

[ ] w=0, v=0 ub=30

with P5

without P5

with P5

without P5

[P3, P5] w=5, v=12 ub=22

[P3] w=4, v=10 ub=21

[P5] w=1, v=2 ub=16

[ ] w=0, v=0 ub=15

with P2

[P3, P5, P2] w=7, v=14 ub=22

[P3, P4] w=6, v=12 ub=21

[P3] w=4, v=10 ub=21

with P4

[P3, P5, P2, P4] w=8, v=15 ub=17.33

[P3, P5, P2] w=7, v=14 ub=16.66

[P3, P5] w=5, v=12 ub=22

[P3, P1] w=16, v=14

with P4

without P4

[P3, P5, P4] w=6, v=13 ub=16

[P3, P5] w=5, v=12 ub=15.3

[P3, P5, P2, P4, P1] w=20, v=19

[P3, P5, P2,P4] w=9, v=15

18.3. Implementation

```python
# ****************************************
# Solve the knapsack problem using the Branch and Bound
# algorithm.
# ****************************************
import heapq
```

```python
class Item:
    def __init__(self, name, value, weight):
        self.name = name
        self.value = value
        self.weight = weight
        # if weight is 0, value_to_weight is also 0
        self.value_to_weight = value / weight if weight != 0 else 0


class TreeNode:
    def __init__(self, level, included_indexes):
        self.level = level
        self.included_indexes = included_indexes
        self.priority = 0
        self.upper_bound = 0
        self.total_weight = 0
        self.total_value = 0

    def __lt__(self, other):
        return self.priority < other.priority


class BranchAndBoundSolver:
    def __init__(self, knapsack_capacity, items):
        """
        Initialize the BranchAndBoundSolver.

        Args:
            max_weight (int): The maximum weight the knapsack can
↪   hold.
            items (list): The list of items to consider for the
            knapsack problem.
        """
        self.knapsack_capacity = knapsack_capacity
        # Sort the items based on their value-to-weight ratio in
        # descending order
        # Sorting items by value-to-weight ratio allows for a greedy
```

```
        # approach, where you consider adding items to the knapsack in
        # descending order of their value-to-weight ratio. This means
         ↪  you
        # prioritize items that give you the most value for
        # the least weight.
        self.items = sorted(
            items, key=lambda el: el.value_to_weight, reverse=True)

        self.items.insert(0, Item("0", 0, 0))
        self.best_profit = 0
        self.best_combination = []

    def build_tree_node(self, level, included_indexes):
        node = TreeNode(level, list(set(included_indexes)))
        node.total_weight = self.total_weight_of(included_indexes)
        node.total_value = self.total_value_of(included_indexes)
        node.upper_bound = self.get_upper_bound(node)
        # Priority of a node to be picked is decided by the
        # upper bound of the current set of element in this
        # branch. -1 is multiplied here is to pick the largest from
         ↪  the
        # priority queue
        node.priority = -1 * node.upper_bound
        return node

    def total_weight_of(self, included_indexes):
        return sum(
            self.items[index].weight
            for index in included_indexes
            if index < len(self.items) - 1
        )

    def total_value_of(self, included_indexes):
        return sum(
            self.items[index].value
            for index in included_indexes
            if index < len(self.items) - 1
```

```python
        )

    def solve(self):
        """
        Solve the knapsack problem using the Branch and Bound
        algorithm.

        Returns:
            list: The best solution found for the knapsack problem.
        """
        priority_queue = []
        # First an empty/dummy node
        start_node = self.build_tree_node(0, [])
        heapq.heappush(priority_queue, start_node)

        while priority_queue:
            current_node = heapq.heappop(priority_queue)

            # Get out of this loop if reached limit of
            # elements in terms of tree levels.
            if current_node.level == len(self.items) - 1:
                break

            if self.is_infeasible(current_node):
                continue

            print(f"upper_bound:{current_node.upper_bound}")
            print(f"items:{current_node.included_indexes}")
            print(f"profit:{current_node.total_value}")
            print(f"weight:{current_node.total_weight}")
            print(f"best profit:{self.best_profit}")

            # if the total value of items added till now exceeds
            # the tracked best profit then update the optimal best
            # solution
            if current_node.total_value > self.best_profit:
                self.best_profit = current_node.total_value
```

```
                self.best_combination = current_node

            print("\n")
            next_level = current_node.level + 1

            left_node = self.build_tree_node(
                next_level, current_node.included_indexes +
↪   [next_level]
            )
            heapq.heappush(priority_queue, left_node)

            right_node = self.build_tree_node(
                next_level, current_node.included_indexes)
            heapq.heappush(priority_queue, right_node)

        return self.get_best_solution()

    def get_upper_bound(self, node):
        """
        Calculate the upper bound of a given node.

        Args:
            node (TreeNode): The node for which to
            calculate the upper bound.

        Returns:
            int: The upper bound of the node.
        """
        value = self.total_value_of(node.included_indexes)
        weight = self.total_weight_of(node.included_indexes)

        if node.level == len(self.items) - 1:
            # this means we just encountered an end node, so we add
            # a zero value item to stop further deep processing.
            next_element = Item("END", 0, 0)
        else:
            next_element = self.items[node.level + 1]
```

```
        bound = value + (self.knapsack_capacity - weight) * \
            next_element.value_to_weight
        return bound

    def is_infeasible(self, node):
        # If the upper bound of the current node is less than our
        # best profit or if the total weight included till now
        # is greater than the allowed maximum weight then discard
        # this branch
        return (
            node.upper_bound < self.best_profit
            or node.total_weight > self.knapsack_capacity
        )

    def get_best_solution(self):
        return self.best_profit

    def included_items(self):
        return [self.items[i] for i in
        ↪  self.best_combination.included_indexes]

capacity = 15
items = [
    Item("P1", 4, 12),
    Item("P2", 2, 2),
    Item("P3", 10, 4),
    Item("P4", 1, 1),
    Item("P5", 2, 1),
]

solver = BranchAndBoundSolver(knapsack_capacity=capacity, items=items)
print("Maximum Profit:", solver.solve())
print("Included Items")
for item in solver.included_items():
    print(f"Product {item.name}: profit= {item.value},
    ↪  weight={item.weight}")
```

```
"""

Maximum Profit: 15
Included Items
item P3: profit= 10, weight=4
item P5: profit= 2, weight=1
item P2: profit= 2, weight=2
item P4: profit= 1, weight=1

"""
```

Branch and Bound (B&B) algorithms offer several advantages for solving optimization problems, but they also come with certain limitations:

18.4. Advantages:

- **Optimality Assurance:** B&B algorithms guarantee finding the optimal solution (if it exists) for problems where a finite set of potential solutions can be enumerated and pruned.
- **Efficient for Discrete Optimization:** Particularly effective for discrete or combinatorial optimization problems, including integer programming, TSP, and other NP-hard problems, by systematically exploring the solution space.
- **Reduction of Search Space:** They intelligently prune branches of the search tree by bounding and eliminating regions that cannot contain an optimal solution, reducing the search space.

18.5. Limitations:

- **Exponential Complexity:** The time complexity can grow exponentially with problem size, making B&B algorithms impractical for large-scale problems due to exhaustive search requirements.
- **Memory Intensive:** For problems with large search spaces, storing and managing nodes in the search tree can consume significant memory resources.
- **Difficulty in Tight Bounds:** Obtaining tight upper and lower bounds for pruning branches might be challenging, leading to less

effective pruning and increased computation.

- **Inaccuracy with Heuristics:** The use of heuristics might affect optimality, as heuristic-driven pruning could skip potential optimal solutions.

18.6. Applications

Branch and Bound (B&B) algorithms are widely applied in various domains due to their ability to solve optimization problems efficiently by systematically searching through the solution space. Some notable applications include:

- **Combinatorial Optimization:** B&B methods are extensively used in solving combinatorial optimization problems like the Traveling Salesman Problem (TSP), Knapsack Problem, Graph Coloring, and Job Scheduling, aiming to find the best arrangement or selection from a discrete set of options.
- **Operations Research:** In operations research, B&B algorithms aid in solving linear programming, integer programming, and mixed-integer programming problems. They efficiently explore feasible solutions while pruning branches that cannot lead to better solutions.
- **Resource Allocation and Scheduling:** B&B techniques are utilized in resource allocation tasks, such as task scheduling in project management, workforce scheduling, and resource allocation in manufacturing or production processes.
- **Network Design and Routing:** B&B algorithms play a crucial role in network design problems like facility location, network routing, and optimal pathfinding in transportation and logistics networks.
- **Optimal Control and Robotics:** B&B techniques are used in optimal control problems, trajectory optimization, motion planning in robotics, and autonomous systems to find the most efficient paths or actions.
- **Circuit Design:** B&B algorithms contribute to circuit design optimization by optimizing layouts, placement of components, and routing in integrated circuits, leading to more efficient and compact designs.
- **Machine Learning and AI:** In some cases, B&B approaches are integrated into machine learning algorithms for solving optimiza-

tion tasks, feature selection, hyperparameter tuning, and model optimization.

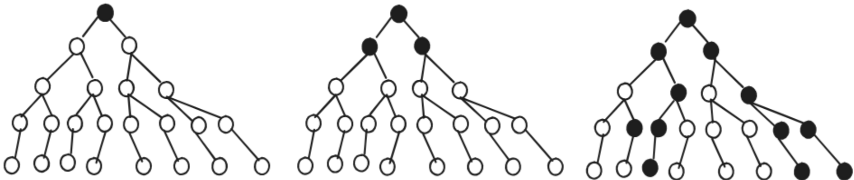
- **Game Theory:** B&B methods are employed in game theory for solving games with imperfect information, like solving game trees in AI for games like chess, checkers, and Go.

18.7. Bibliography

- J. D. C. Little Branch and bound methods for combinatorial problems (LEGARE STREET Press) (2023)
- S. Martello & P. Toth Knapsack problems: Algorithms and computer implementations (Wiley) (1990)

19. Beam search

Beam search is a heuristic search algorithm, a variant of breadth first search designed in such a way that it only explores a limited set of promising paths or solutions in a search space instead of all possible paths, which is often computationally expensive. It is used in the field of artificial intelligence, particularly in the context of search and optimization problems. The main difference between BEAM search and breadth-first search is that at every level of the search tree, only the top β candidates are chosen for further exploration. Here, β is known as the beam width. The reasoning behind this is that a path from source to destination is likely to pass through some top number of most promising nodes. This leads to an algorithm that is fast and memory-efficient because it can disregard many nodes that may appear to be too far from the goal. An evaluation function is used to evaluate the candidacy of nodes for further exploration.



Here is an analogy that may help you better understand beam search:

Imagine trying to find the shortest path from your house to a grocery store. You start by walking in the direction you think will lead you there, keeping track of different paths as you walk. At each intersection, you consider the available options and choose the one most likely to lead you to the store. Beam search is a search algorithm that works similarly. It only keeps track of a limited number of paths at each iteration.

If the beam width is too small, the algorithm may not find the best path. Conversely, if the beam width is too large, the algorithm may use too much memory. This renders the algorithm incomplete, meaning that finding the shortest path is not guaranteed. Beam search is suitable for problems where an exact solution is not required, and a good approximation is sufficient.

The trade-off between beam width and solution quality is crucial in beam search. A narrower beam focuses on the best candidates but may miss global optima. A wider beam explores a larger portion of the search space but may spend more time on unpromising paths. The choice of beam width depends on the specific problem and the desired trade-off between speed and solution quality.

19.1. How it works:

Beam search uses an open set prioritized based on the total cost and a closed set to keep track of visited nodes. The algorithm begins with a start node S, which also serves as the root node for the search tree.

1. Initialize a search tree with the root node being the start node S.
2. Add S to the closed set and evaluate all successors of node S.
3. Select the top β (beam width) nodes and add them as children to S in the tree. Ignore all other nodes.
4. Add the selected nodes to the open set for further exploration.
5. Remove all nodes in the open set. Once a node is removed for exploration, add it to the closed set.
6. Evaluate all adjacent nodes and sort them according to the evaluation function.
7. Select the top β nodes and add them to the tree, adding them as children of their respective parent nodes.
8. Repeat this process until the goal node G is found in the open set, indicating that a path has been found.

19.2. Implementation

Below is a Python program that demonstrates the Beam Search algorithm to find the shortest path in a weighted graph from a start node to a goal node with a specified beam width.

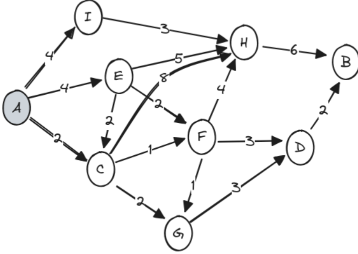
```
import queue


def beam_search(graph, start, goal, beam_width):
    open_set = queue.PriorityQueue()
```

```python
    open_set.put((0, start))
    closed_set = set()
    # A dictionary to represent the search tree
    search_tree = {start: None}

    while not open_set.empty():
        print(open_set.queue, closed_set)
        current_cost, current_node = open_set.get()

        if current_node == goal:
            # Goal reached, reconstruct and return the path
            path = []
            while current_node is not None:
                path.insert(0, current_node)
                current_node = search_tree[current_node]
            return path

        closed_set.add(current_node)
        successors = graph[current_node]
        successors.sort(key=lambda x: x[1])  # Sort successors by cost

        for successor, cost in successors[:beam_width]:
            if successor not in closed_set:
                open_set.put((current_cost + cost, successor))
                search_tree[successor] = current_node

    # If goal not reached
    return None

# Example graph represented as an adjacency list with costs
graph = {
    'A': [('E', 4), ('C', 2), ('I', 4)],
    'B': [],
    'C': [('F', 1), ('G', 2), ('H', 8)],
    'D': [('B', 2)],
    'E': [('F', 2), ('C', 2), ('H', 5)],
    'F': [('H', 4), ('G', 1)],
```

```
    'G': [('D', 3)],
    'H': [('B', 6)],
    'I': [('H', 3)]
}
start_node = 'A'
goal_node = 'B'
beam_width = 2  # Adjust the beam width as needed

path = beam_search(graph, start_node, goal_node, beam_width)
if path:
    print("Shortest Path:", ' -> '.join(path))
else:
    print("No path found.")
```

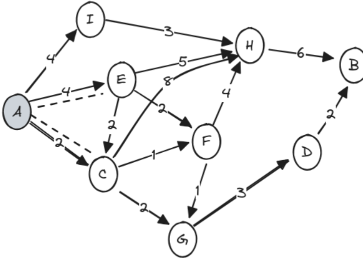
19.3. Visualization

We start with an empty closed set and add the starting node to the priority queue.



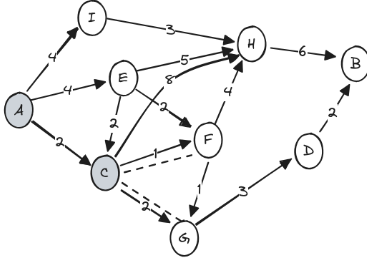
closed: [ ]	
cost	node
0	A

From A, node I, C and E are reachable with cost 4, 4, 2 respectively. Since our beam width is 2, we pick the top 2 nodes C and E and add them to our priority queue. We mark A as visited by addind it into the closed set.



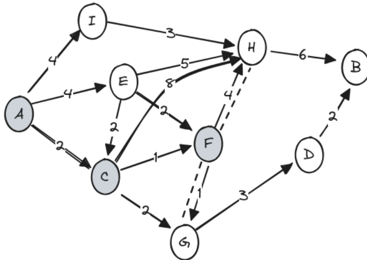
closed: [A]	
cost	node
2	C
4	E

Top of the queue we have C, so we add C to the closed set and continue to look for the next 2 top nodes. For the next nodes F and G, the costs are 3 and 4 respectively.

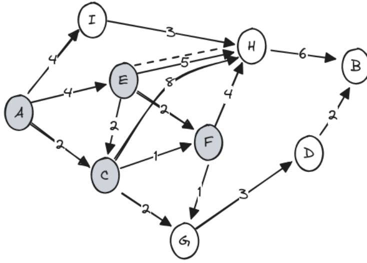


closed: [C, A]	
cost	node
3	F
4	E
4	G

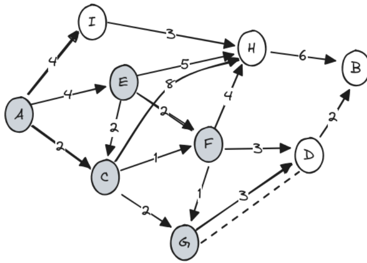
Top of the queue we have F, so we add F to the closed set and continue to look for the next 2 top nodes. For the next nodes H and G, the costs are 7 and 4 respectively.



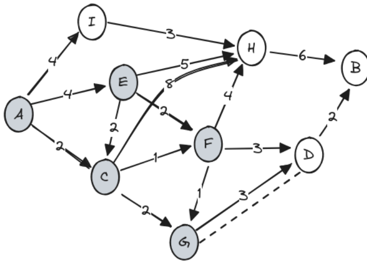
closed: [F, C, A]	
cost	node
4	E
4	E
4	G
7	H



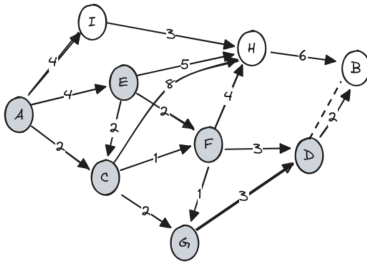
closed: [E, F, C, A]	
cost	node
4	E
4	G
7	H



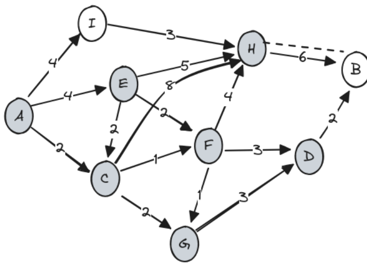
closed: [E, C, A, G, F]	
cost	node
4	G
7	H
7	D



closed: [E, C, A, G, F]	
cost	node
7	D
7	H
7	D

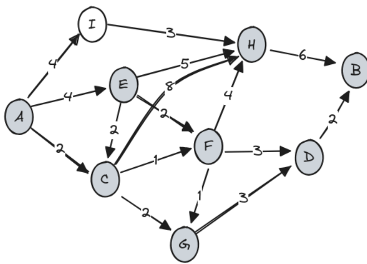


closed: [D, E, C, A, G, F]	
cost	node
7	D
7	H
9	B



closed: [D, E, C, A, G, F]	
cost	node
7	D
9	H
9	B

We stop when we reach the goal and print the path. A -> C -> F -> H -> B



closed: [D, E, C, A, H, G, F]	
cost	node
9	B
9	B
13	B

19.4. Time Complexity Analysis

The time complexity of Beam Search depends on several factors, including the size of the search space, the characteristics of the problem, and the chosen beam width. Let's break down the time complexity analysis:

1. **Search Space Size N:**
    - Beam Search explores a limited set of candidates at each level, controlled by the beam width β. The number of nodes at each level that need to be considered is $O(\beta)$.
2. **Depth of the Search Tree D:**
    - The depth of the search tree depends on the distance from the start node to the goal node. Let's denote the depth of the optimal solution as $D*$.
3. **Overall Time Complexity:**
    - The overall time complexity of Beam Search is approximately $O(\beta^D*)$.
4. **Comparison with Other Algorithms:**
    - Compared to uninformed search algorithms like breadth-first search, which has a time complexity of $O(b^D)$, where b is the branching factor, Beam Search can be more efficient when β « b. However, it may be less efficient when β approaches or exceeds b.
5. **Influence of Heuristic Evaluation:**
    - If a heuristic function is used to guide the search, the time complexity is influenced by the efficiency and accuracy of the heuristic. A more effective heuristic can significantly reduce the number of nodes explored.
6. **Best, Worst, and Average Case:**
    - The best-case scenario occurs when the goal is found early in the search, leading to a time complexity close to $O(\beta)$. The worst-case occurs when the goal is at the maximum possible depth, resulting in a time complexity of $O(\beta^D)$. The average-case time complexity is challenging to determine precisely and depends on the characteristics of the problem.
7. **Time Complexity in Practice:**

- In practice, the efficiency of Beam Search often depends on the ability to find good solutions quickly with a limited beam width. The algorithm's performance can be sensitive to the choice of the beam width and the characteristics of the problem.

19.5. Advantages

- **Efficient:** Beam search is a very efficient search algorithm, especially for large and complex problems. This is because beam search only explores a limited number of paths at each iteration.
- **Flexible:** Beam search can be used to find a variety of different types of solutions, including optimal solutions, suboptimal solutions, and constrained solutions.

19.6. Limitations

- **Incompleteness:** Beam search is not a complete search algorithm, which means that it may not always find a solution to a problem, even if a solution exists. This is because beam search only explores a limited number of paths at each iteration.
- **Sensitivity to beam width:** The performance of beam search is sensitive to the choice of beam width. A too small beam width may cause the algorithm to miss the optimal solution, while a too large beam width may make the algorithm inefficient.

19.7. Applications

Beam search is a powerful algorithm with a wide range of applications across various domains. Here are some of the most common applications:

Natural language processing (NLP):

- **Machine Translation:** Beam search is widely used in machine translation to decode the output sequence word by word. It allows the model to explore different translation options and choose the most likely one based on the context and the overall translation quality.
- **Text Summarization:** Beam search can be used to summarize text by generating a shorter version that captures the main points of the

original text. The algorithm can be used to find the most concise and informative summary that retains the key ideas.

- **Dialogue Systems:** Beam search is employed in dialogue systems to generate responses that are relevant to the conversation history and user intent. It allows the system to explore different response options and choose the one that best fits the context and user expectations.

Computer Vision:

- **Image Captioning:** Beam search can be used to generate captions for images. It allows the model to consider different interpretations of the image and generate captions that are relevant and descriptive.
- **Object Detection and Recognition:** Beam search can be used to detect and recognize objects in images. It helps the model identify multiple objects and their positions within the image by exploring different potential detections and selecting the most likely ones.
- **Video Captioning:** Beam search can be used to generate captions for videos by analyzing the video content and generating summaries that capture the key events and actions.

Speech Recognition:

- **Automatic Speech Recognition (ASR):** Beam search is applied in ASR systems to translate speech signals into text. It allows the system to consider different possible interpretations of the speech and select the one that best matches the acoustic signal and the linguistic context.
- **Speaker Diarization:** Beam search can be used to identify and segment speech from different speakers in a recording. It helps the system distinguish between multiple voices and assign each speaker their corresponding segments.

19.8. Bibliography

- Markus Freitag & Yaser Al-Onaizan *Beam search strategies for neural machine translation*, abs/1702.01806 CoRR (2017), http://arxiv.org/abs/1702.01806
- Sina Zarrieß et al. *Decoding methods in neural language generation: A survey*, 12 Information 355 (2021), http://dx.doi.org/10.3390/info12090355

- Oriol Vinyals et al. *Show and tell: A neural image caption generator*, abs/1411.4555 CoRR (2014), http://arxiv.org/abs/1411.4555
- *Improvements in beam search for 10000-word continuous speech recognition*, 1 *in* [Proceedings] ICASSP-92: 1992 IEEE international conference on acoustics, speech, and signal processing 9–12 vol.1
- Stefan Ortmanns & Hermann Ney *Look-ahead techniques for fast beam search*, 14 Computer Speech & Language 15–32 (2000), https://www.sciencedirect.com/science/article/pii/S0885230899901316
- Clara Meister et al. *Best-first beam search*, 8 Transactions of the Association for Computational Linguistics 795–809 (2020), https://aclanthology.org/2020.tacl-1.51

Index

$O(1+\alpha)$, 106
$O(1)$, 85, 107
$O(E * log(V))$, 344
$O(V+E)$, 257
$O(V^2)$, 344
$O(b)$, 52
$O(b^D)$, 192
$O(log(n))$, 60, 66, 71
$O(log_3 n)$, 39
$O(m)$, 439
$O(\sqrt{n})$, 47

A* Search, 304
ACO, 204
Anomaly Detection, 230
Ant Colony Optimization, 204
Approximate String Matching, 410, 414
Arrays, 416
Auto-completion, 371
Autocompletion, 440
Autocorrection, 414

Backpropagation, 121
Bad Character rule, 373
Ball Trees, 226
Beam search, 184
Big O Notation, 22
Binary search, 34
Bloom Filter, 87
Boyer-Moore, 373
Branch and Bound, 167
Burrows-Wheeler Transform (BWT), 371

Cluster Analysis, 346
Collaborative Filtering, 230
Common Notations, 23
Compiler Design, 395
Constant Time, 26
Constant-Time, 94
Cost function, 304
Cuckoo Filter, 96
Cuckoo hashing, 97
Curse of Dimensionality, 227

Data Compression, 371
Data Deduplication, 414
Decay Rate, 206
Depth First Search, 241
Depth-First Search, 197
Detecting Negative Cycles, 353, 362
Dictionaries, 37
digital tree, 432
Dijkstra's algorithm heuristic, 305
distributed systems, 64

Euclidean distance, 305
Exponential Search, 60
Exponential Time, 27

Fibonacci Search, 66
Fingerprint, 107
Fraud Prevention, 230
Fuzzy String Matching, 414

Game playing, 202
Genome Sequencing, 371
Good Suffix rule, 373

Hash Collisions, 82
Hash Table Search, 82
Hashing, 396
Heaps, 426
Heuristics Function, 304
Hill Climbing, 111

IDDFS, 196
Image Segmentation, 346
Indexed Sequential Search, 48
Interpolation Search, 54
ISAM, 48
Iterative Deepening, 197
Iterative Deepening Depth-First Search, 196

Jump Search, 44

K-D Trees, 219
KMP, 385
Knuth-Morris-Pratt, 385

Large sorted datasets, 64
Levenshtein distance, 405
Linear search, 30
Linear Time, 26
Linearithmic Time, 27
Linked Lists, 421
Load Factor, 107
Locality-Sensitive Hashing, 227
Log File Analysis, 52
Logarithmic Time, 26
Longest Common Substring, 371
Longest Prefix Suffix, 385

Manhattan distance, 305
Maze solving, 202
MCTS, 120
Minimizing Cost Functions, 43
minimum spanning tree, 339
Monte Carlo Tree Search, 120
MST, 339

Natural language processing, 134, 193, 315
Nearest Neighbor Search, 216
Network Design, 346
NLP, 134, 193, 315
NP (Nondeterministic Polynomial Time), 27
NP-Completeness, 27

O(log(n)), 37
O(n), 32
OCR Error Correction, 414

P (Polynomial Time), 27
Parallel Processing, 94
Pattern Matching, 371
Peak Finding, 43
Pheromone Matrix, 205
Phone Number Lookup, 441
Plagiarism Detection, 413
Prefix Matching, 433
prefix tree, 432
Prim's Algorithm, 339
Python, 455
Python Arrays, 460
Python Conditional statements, 458
Python Data types, 458

Python Dictionary, 462
Python Functions, 464
Python Language Basics, 458
Python Lists, 461
Python Loops, 459
Python Math Operators, 463
Python Sets, 461
Python Variables, 458

Quadratic Time, 27
Queues, 423

Rabin-Karp Algorithm, 396
Recommendation Systems, 230
Robotics, 133, 281, 302
robotics, 315
Rolling Hash, 396

Searching Database Archives, 52
Simulated Annealing, 135, 150
Spell checkers, 94
Spell Checking, 409, 413, 440
Stack, 429
String Matching, 395
Substring Search, 371
Suffix Array, 364
Symbol Tables, 37

Tabu search, 150
Ternary search, 39
Ternary Search Tree, 442
Text Search, 395, 441
Tic-Tac-Toe, 122
Time Complexity, 22
Traveling Salesman Problem, 112, 205
Trie, 432
Trie search, 432
TSP, 205
TST, 442

Unbalanced Trees, 228
unbounded, 64
unknown-sized arrays, 64