Karolinska Institutet

Algorithms part II

Billy Langlet & Alen Lovric

Introductory Knowledge

Basic

- What is an algorithm?
- Where are algorithms used?

Advanced

- What should you keep in mind when designing algorithms?
- How do you know if an algorithm is effective?

<Think about it>

Searcing Algorithms

Search algorithms are methods and techniques used to find specific items or information within a dataset or data structure.

Common search algorithms include:

- **Linear search, binary search**, hash tables, depth-first search (DFS), breadth-first search (BFS), A search*, K-means clustering

The choice of a search algorithm depends on the specific problem, data structure, and performance requirements.

Search Algorithms

Linear Search

Binary Search

Hash Tables

Depth-First Search (DFS)

Breadth-First Search (BFS)

A Search*

K-Means Clustering

Binary Search Trees (BSTs)

Trie

Linear search, also known as sequential search, is the simplest search algorithm. It sequentially checks each element in a list or array until a match is found or the entire list is searched. Linear search has a time complexity of O(n), where 'n' is the number of elements in the list.

Search Algorithms

Linear Search

Binary Search

Hash Tables

Depth-First Search (DFS)

Breadth-First Search (BFS)

A Search*

K-Means Clustering

Binary Search Trees (BSTs)

Trie

Binary search is an efficient search algorithm that works on sorted arrays or lists. It repeatedly divides the search interval in half and compares the middle element with the target value. Depending on the comparison, it narrows the search to the left or right half of the array. Binary search has a time complexity of O(log n).

Search Algorithms

Linear Search
Binary Search
Hash Tables
Depth-First Search (DFS)
Breadth-First Search (BFS)
A Search*
K-Means Clustering
Binary Search Trees (BSTs)
Trie

Hash tables (or dictionaries) use a hash function to map keys to specific locations (buckets) in a data structure. Hash tables provide constant-time (O(1)) average-case access to values associated with keys.

Search Algorithms

Linear Search
Binary Search
Hash Tables
Depth-First Search (DFS)
Breadth-First Search (BFS)
A Search*
K-Means Clustering
Binary Search Trees (BSTs)
Trie

DFS is a graph traversal algorithm that explores as far as possible along a branch before backtracking. It is used for searching paths and connectivity in graphs and trees.

Search Algorithms

Linear Search
Binary Search
Hash Tables
Depth-First Search (DFS)
Breadth-First Search (BFS)
A Search*
K-Means Clustering
Binary Search Trees (BSTs)
Trie

BFS is another graph traversal algorithm that explores all neighbor nodes at the current level before moving to the next level. It is often used to find the shortest path in unweighted graphs.

Search Algorithms

Linear Search
Binary Search
Hash Tables
Depth-First Search (DFS)
Breadth-First Search (BFS)
A Search*
K-Means Clustering
Binary Search Trees (BSTs)
Trie

A* is a popular pathfinding and search algorithm used in artificial intelligence and robotics. It combines elements of BFS and heuristic functions to find the shortest path in a graph.

Search Algorithms

Linear Search
Binary Search
Hash Tables
Depth-First Search (DFS)
Breadth-First Search (BFS)
A Search*
K-Means Clustering
Binary Search Trees (BSTs)
Trie

K-Means is an iterative clustering algorithm used to partition a dataset into clusters. It searches for centroids that minimize the distance between data points and cluster centers.

Search Algorithms

Linear Search
Binary Search
Hash Tables
Depth-First Search (DFS)
Breadth-First Search (BFS)
A Search*:
K-Means Clustering
Binary Search Trees (BSTs)
Trie

BSTs are a type of data structure where each node has at most two children. They provide efficient searching and insertion operations when balanced. The time complexity for search operations is O(log n) on average.

Search Algorithms

Linear Search
Binary Search
Hash Tables
Depth-First Search (DFS)
Breadth-First Search (BFS)
A Search*:
K-Means Clustering
Binary Search Trees (BSTs)
Trie

A trie is a tree-like data structure used for efficient string searching and prefix matching. It's commonly used in autocomplete and dictionary applications.

Linear Search Algorithm

<Example “Algorithm” in R>

Binary Search Algorithm

<Example “Algorithm” in R>

Linear Search vs Binary Search

- Are there instances where one algorithm performs better than the other.
- How would you express this in a mathematical formula and with Big O annotations?

<Discuss>

More Complex Algorithms

Divide and conquer

Divide and Conquer

The divide-and-conquer paradigm is a fundamental technique used in algorithm design. It involves breaking down a complex problem into smaller, more manageable subproblems, solving those subproblems independently, and then combining their solutions to solve the original problem.

Key steps of the divide-and-conquer paradigm:

- Divide
- Conquer
- Combine

Divide and Conquer

- Divide
- Conquer
- Combine

Divide and Conquer

- **Divide** – Break the problem into smaller, often equal-sized, subproblems. This step involves decomposing the original problem into simpler, more manageable parts.
- Conquer
- Combine

Divide and Conquer

- Divide
- **Conquer** – Solve each subproblem independently. Typically, this is done recursively by applying the same algorithm to each subproblem.
- Combine

Divide and Conquer

- Divide
- Conquer
- **Combine**: Combine the solutions of the subproblems to obtain the solution for the original problem. This step is crucial because it ties together the results of the subproblems to provide a solution that addresses the entire problem.

Divide and Conquer

The divide-and-conquer paradigm is used to design algorithms that take advantage of **subproblem reusability** and can lead to more efficient solutions for problems that exhibit specific characteristics. Common examples of algorithms and problems that utilize the divide-and-conquer approach include:

- Quick Sort
- Karatsuba Algorithm

Divide and Conquer

- **Quick Sort** - A sorting algorithm that selects a "pivot" element, partitions the array into elements less than and greater than the pivot, and then recursively sorts the subarrays. It has an average time complexity of O(n log n).
- Karatsuba Algorithm

Divide and Conquer

- Quick Sort
- **Karatsuba Algorithm** - A multiplication algorithm that uses a divide-and-conquer approach to multiply two large numbers more efficiently than the traditional long multiplication method.

Quick Sort Algorithm

<Example “Algorithm” in R>

Karatsuba Algorithm

<Example “Algorithm” in R>

Quick Sort and Karatsuba Algorithm

- How does the quick sort and Karatsuba algorithm differ from simpler algorithms?

<Discuss>

Advanced Algorithms

Dynamic programming and greedy algorithms

Dynamic Programming

Dynamic programming is a powerful technique used to solve problems by breaking them down into smaller subproblems and solving each subproblem only once, storing the results to avoid redundant calculations.

Key characteristics in dynamic programming are:

- Optimal substructure
- Overlapping subproblems
- Memorization (top-down approach)
- Tabulation (bottom-up approach)

Dynamic Programming

- Optimal substructure
- Overlapping subproblems
- Memorization (top-down approach)
- Tabulation (bottom-up approach)

Dynamic Programming

- **Optimal substructure** – Identify if an optimal solution to the problem can be constructed from optimal solutions to its subproblems. In other words, can you break down the problem into smaller parts, solve those parts optimally, and then combine their solutions to get the optimal solution for the original problem.
- Overlapping subproblems
- Memorization (top-down approach)
- Tabulation (bottom-up approach)

Dynamic Programming

- Optimal substructure
- **Overlapping subproblems** - Dynamic programming problems often involve overlapping subproblems, which means that the same subproblem is solved multiple times. Dynamic programming algorithms avoid redundant calculations by storing the results of subproblems in a data structure like an array or a table.
- Memorization (top-down approach)
- Tabulation (bottom-up approach)

Dynamic Programming

- Optimal substructure
- Overlapping subproblems
- **Memorization (top-down approach)** - In this approach, you start from the original problem and work your way down to smaller subproblems. Whenever you solve a subproblem, you store its solution in a data structure (usually a memorization table) so that you can reuse it.
- Tabulation (bottom-up approach)

Dynamic Programming

- Optimal substructure
- Overlapping subproblems
- Memorization (top-down approach)
- **Tabulation (bottom-up approach)** - In this approach, you start by solving the smallest subproblems first and then use their solutions to build up to the original problem. This approach is often more efficient in terms of both time and space compared to memorization.

Dynamic Programming

Dynamic programming can be applied to a wide range of problems, including:

- Fibonacci sequence
- Shortest path problems
- Longest common subsequence
- Knapsack problem
- Matrix chain multiplication

Dynamic Programming

Dynamic programming can be applied to a wide range of problems, including:

- **Fibonacci sequence** - Calculating the nth Fibonacci number efficiently.
- Shortest path problems
- Longest common subsequence
- Knapsack problem
- Matrix chain multiplication

Dynamic Programming

Dynamic programming can be applied to a wide range of problems, including:

- Fibonacci sequence
- **Shortest path problems** - Finding the shortest path in a graph (e.g., Dijkstra's algorithm).
- Longest common subsequence
- Knapsack problem
- Matrix chain multiplication

Dynamic Programming

Dynamic programming can be applied to a wide range of problems, including:

- Fibonacci sequence
- Shortest path problems
- **Longest common subsequence** - Finding the longest common subsequence between two sequences.
- Knapsack problem
- Matrix chain multiplication

Dynamic Programming

Dynamic programming can be applied to a wide range of problems, including:

- Fibonacci sequence
- Shortest path problems
- Longest common subsequence
- **Knapsack problem** – Optimizing the selection of items to maximize a value within a weight constraint.
- Matrix chain multiplication

Dynamic Programming

Dynamic programming can be applied to a wide range of problems, including:

- Fibonacci sequence
- Shortest path problems
- Longest common subsequence
- Knapsack problem
- **Matrix chain multiplication** - Optimally parenthesizing matrix chain multiplication to minimize the number of scalar multiplications.

Dynamic Programming

The key idea behind dynamic programming is to **avoid redundant** work by breaking down **complex problems into simpler subproblems** and storing their solutions. This leads to more efficient algorithms for solving a wide variety of optimization problems.

Fibonacci Sequence Algorithm

<Example “Algorithm” in R>

Knapsack Problem Algorithm

<Example “Algorithm” in R>

Dynamic Programming

- Have you used, or do you see use of dynamic programming in your own field of research?

<Discuss>

Greedy Algorithms

A greedy algorithm is an approach to problem-solving that makes locally optimal choices at each step in the hope of finding a globally optimal solution. In other words, it makes the best choice at each stage of the algorithm without considering the consequences of that choice on future steps.

Here are the key characteristics of greedy algorithms:

- Greedy choice property
- Optimal substructure
- No backtracking
- May or may not always provide an optimal solution

Greedy Algorithms

- Greedy choice property
- Optimal substructure
- No backtracking
- May or may not always provide an optimal solution

Greedy Algorithms

- **Greedy choice property** – At each step, the greedy algorithm makes the choice that appears to be the best option without considering the larger context. This choice is often determined by a specific criterion or heuristic.
- Optimal substructure
- No backtracking
- May or may not always provide an optimal solution

Greedy Algorithms

- Greedy choice property
- **Optimal substructure** – Greedy algorithms often work well when the problem has an optimal substructure, meaning that a globally optimal solution can be constructed from locally optimal solutions.
- No backtracking
- May or may not always provide an optimal solution

Greedy Algorithms

- Greedy choice property
- Optimal substructure
- **No backtracking** - Unlike dynamic programming, where solutions to subproblems are stored and reused, greedy algorithms do not backtrack. Once a decision is made, it's final, and the algorithm moves forward without reconsidering past choices.
- May or may not always provide an optimal solution

Greedy Algorithms

- Greedy choice property
- Optimal substructure
- No backtracking
- **May or may not always provide an optimal solution** - Greedy algorithms are not guaranteed to produce the globally optimal solution for all problems. Whether a greedy algorithm works depends on the problem's characteristics and the specific greedy strategy employed.

Greedy Algorithms

Greedy algorithms are often used for optimization problems where you're trying to find the best solution among a set of choices. However, their success depends on the problem's nature and whether the greedy choice property holds.

Greedy Algorithms

Examples of problems that can be solved using greedy algorithms include:

- Minimum spanning tree
- Dijkstra's shortest path algorithm
- Fractional knapsack problem
- Huffman coding

Greedy Algorithms

Examples of problems that can be solved using greedy algorithms include:

- **Minimum spanning tree** – Finding the minimum spanning tree in a graph (e.g., Kruskal's or Prim's algorithm).
- Dijkstra’s shortest path algorithm
- Fractional knapsack problem
- Huffman coding

Greedy Algorithms

Examples of problems that can be solved using greedy algorithms include:

- Minimum spanning tree
- **Dijkstra's shortest path algorithm** – Finding the shortest path in a weighted graph.
- Fractional knapsack problem
- Huffman coding

Greedy Algorithms

Examples of problems that can be solved using greedy algorithms include:

- Minimum spanning tree
- Dijkstra's shortest path algorithm
- **Fractional knapsack problem** - Selecting items with fractional weights to maximize a value within a weight constraint.
- Huffman coding

Greedy Algorithms

Examples of problems that can be solved using greedy algorithms include:

- Minimum spanning tree
- Dijkstra's shortest path algorithm
- Fractional knapsack problem
- **Huffman coding** - Constructing a variable-length prefix coding to minimize the total encoding length of symbols in data compression.

Greedy Algorithms

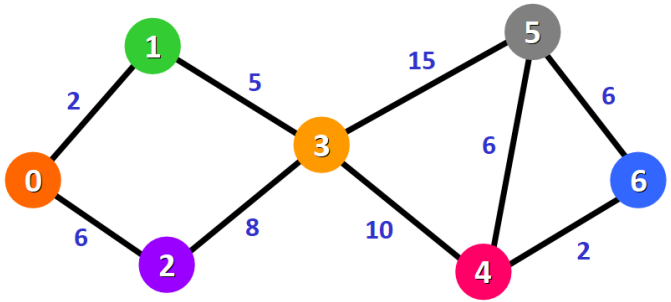
It is essential to be cautious when applying a greedy algorithm, as it may not always produce the optimal solution. In some cases, a more comprehensive search approach, like dynamic programming, may be necessary to guarantee the best possible result.

Fractional Knapsack Algorithm

<Example “Algorithm” in R>

Dijkstra's Shortest Path Algorithm

<Example “Algorithm” in R>



Greedy Algorithms

- Have you used, or do you see use of greedy algorithms in your own field of research?

<Discuss>

Algorithm design

Problem Solving Strategies

Algorithm development involves a variety of problem-solving strategies and techniques. These strategies help algorithm designers systematically approach problems, devise efficient solutions, and optimize algorithms.

Common problem-solving strategies:

- Brute Force
- Greedy Algorithms
- Dynamic Programming
- Divide and Conquer
- Binary Search
- And more...

Problem Solving Strategies

Brute force

Greedy algorithms

Dynamic programming

Divide and conquer

Binary search

And more...

This is the simplest approach, where you exhaustively examine all possible solutions. While it may not be efficient for large problem instances, it is a straightforward way to solve problems when the solution space is small.

Problem Solving Strategies

Brute force

Greedy algorithms

Dynamic programming

Divide and conquer

Binary search

And more...

Is an approach to problem-solving that makes locally optimal choices at each step in the hope of finding a globally optimal solution

Problem Solving Strategies

Brute force
Greedy algorithms
Dynamic programming
Divide and conquer
Binary search
And more...

Is a technique used to solve problems by breaking them down into smaller subproblems and solving each subproblem only once, storing the results to avoid redundant calculations

Problem Solving Strategies

Brute force
Greedy algorithms
Dynamic programming
Divide and conquer
Binary search
And more...

The divide-and-conquer paradigm is used to design algorithms that take advantage of subproblem reusability and can lead to more efficient solutions for problems that exhibit specific characteristics.

Problem Solving Strategies

Brute force
Greedy algorithms
Dynamic programming
Divide and conquer
Binary search
And more...

Repeated division of search interval comparing the middle element with the target value. Depending on the comparison, it narrows the search to the left or right half of the array. Binary search has a time complexity of O(log n).

Problem Solving Strategies

Brute force
Greedy algorithms
Dynamic programming
Divide and conquer
Binary search
And more…

Backtracking, branch and bound, heuristic algorithms, graph algorithms, randomized algorithms, network flow algorithms, string matching algorithms, computational geometry algorithms, and machine learning and AI techniques.

Problem Solving Strategies

The choice of problem-solving strategy depends on the nature of the problem, the problem's constraints, the available resources, and the desired level of optimization. Algorithm designers often analyze the problem, select an appropriate strategy, and refine the algorithm through iterative testing and optimization to achieve the desired results efficiently.

Ethical considerations

Ethical Considerations

Due to the non-deterministic nature of certain algorithms and potential problems with results many different approaches have been discussed regarding ethics.

- Discuss in groups what ethical problems may occur because of algorithm development and use.

<Discuss groups>

Recap

What are Algorithms?

Algorithms are step-by-step procedures or sets of instructions for solving a specific problem or accomplishing a particular task.

Algorithm Anatomy

Algorithms require input and produces output, based on a set of instructions. For algorithms to be effective the steps should be finite and clearly defined.

Why are Algorithms Used?

Algorithms are the backbone of essentially all programming languages and are used for Problem solving, efficiency, reusability, correctness, scalability, and maintainability

Where are Algorithms Used?

Algorithms are used in all sciences, for example: simulation and modelling, genomics and bioinformatics, medical research, social sciences, and neuroscience.

Creating Algorithms

To create an effective algorithm you, at least, need an understanding of the problem, problem-solving and algorithmic thinking, programming knowledge, data structures.

Analysing Algorithms

When analyzing an algorithm, it is common to describe its time and space complexity.

The time complexity describes how the algorithm's runtime grows as the size of the input increases. It is often expressed using Big O notation (e.g., O(1), O(log n), O(n), O(n^2)).

The space complexity measures the amount of memory or space an algorithm requires to execute.

Complex Algorithms

More complex algorithms are usually a combination of simpler algorithms.

The problem-solving strategy for the algorithm depends on the nature of the problem, with problem solving examples such as: brute force, greedy algorithms, dynamic programming, divide and conquer and binary search.

Example of Algorithms

A few algorithms described during the lecture: linear search, binary search, bubble sort, insertion sort, quick sort, Karatsuba algorithm, minimum spanning tree, and Dijkstra's shortest path algorithm.

What Do I Now Know About Algorithms?

Basic

- What is an algorithm?
- Where are algorithms used?

Advanced

- What should you keep in mind when designing algorithms?
- How do you know if an algorithm is effective?

Karolinska
Institutet