Algorithm Design Strategies II

Joaquim Madeira

Version 0.4 – September 2024

Overview

- Counting basic operations – Recap
- Deterministic vs Non-Deterministic Algorithms
- Problem Types and Design Strategies
- Algorithm Efficiency and Complexity Analysis
- Brute-Force
- Divide-and-Conquer
- Decrease-and-Conquer
- Example: computing powers

RECAP
– COUNTING OPERATIONS

Running Time vs. Operations Count

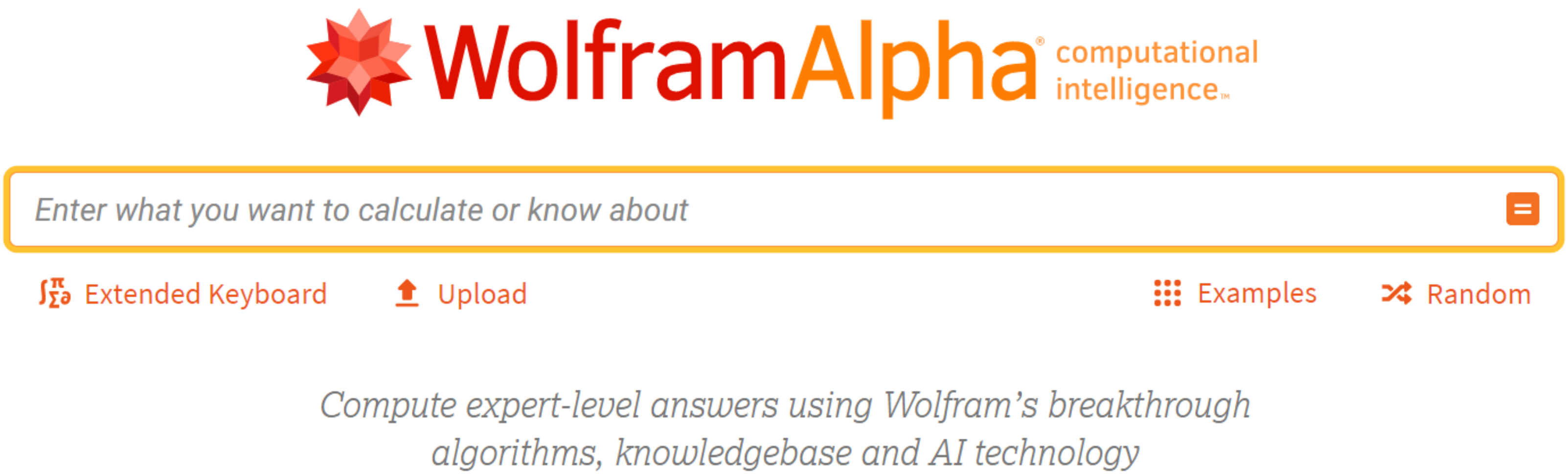
- Running time is not (very) useful for comparing algorithms
  - Speed of particular computers
  - Chosen computer language
  - Quality of programming implementation
  - Compiler optimizations

- Evaluate efficiency in an independent way
  - Count the "basic operations" !!
    - Contribute the most to overall running time

Formal Analysis – Pencil and paper

- Understand algorithm behavior
  - Count arithmetic operations / comparisons
  - Find a closed formula !!
  - Identify best, worst and average case situations, if that is the case
- Iterative algorithms
  - Loops : how many iterations ?
  - Set a sum for the basic operation counts
- Recursive algorithms
  - How many recursive calls ?
  - Establish and solve appropriate recurrences

WolframAlpha – Did you use it?



https://www.wolframalpha.com/

Return value? – Number of iterations?

```
int f1(int n) {
  int i,r=0;
  for(i = 1; i <= n; i++)
    r += i;
  return r;
}
```

```
int f2(int n) {
  int i,j,r=0;
  for(i = 1; i <= n; i++)
    for(j = 1; j <= n; j++)
      r += 1;
  return r;
}
```

```
int f3(int n) {
  int i,j,r=0;
  for(i = 1; i <= n; i++)
    for(j = i; j <= n; j++)
      r += 1;
  return r;
}
```

```
int f4(int n) {
  int i,j,r=0;
  for(i = 1; i <= n; i++)
    for(j = 1; j <= i; j++)
      r += j;
  return r;
}
```

Closed formulas? – Comput. tests?

- f1(n) = n (n + 1) / 2 n_iters1(n) = n
- f2(n) = n^2 n_iters2(n) = f2(n)
- f3(n) = n (n + 1) / 2 n_iters3(n) = f3(n)
- f4(n) = n (n + 1) (n + 2) / 6
- n_iters4(n) = n (n + 1) / 2
- Use WolframAlpha to get / check results !

Return value? – Number of calls?

```
unsigned int
r1(unsigned int n) {
   if(n == 0) return 0;
   return 1 + r1(n – 1);
}
```

```
unsigned int
r2(unsigned int n) {
   if(n == 0) return 0;
   if(n == 1) return 1;
   return n + r2(n – 2);
}
```

```
unsigned int
r3(unsigned int n) {
  if(n == 0) return 0;
   return 1 + 2 * r3(n – 1);
}
```

```
unsigned int
r4(unsigned int n) {
  if(n == 0) return 0;
   return 1 + r4(n – 1) + r4(n – 1);
}
```

Closed formulas? – Comput. tests?

- r1(n) = n   n_calls1(n) = r1(n)

- r2(n) = n (n + 2) / 4, if n is even
- r2(n) = 1 + (n − 1) (n + 3) / 4, if n is odd

- n_calls2(n) = *floor*(n / 2)

- Use WolframAlpha to get / check results !

Closed formulas? – Comput. tests?

- $r3(n) = 2^n - 1$ $\quad\quad$ $n\_calls3(n) = n\_calls1(n)$
- $r4(n) = r3(n) = 2^n - 1$
- $n\_calls4(n) = 2 \times (2^n - 1) = 2 \times r4(n)$

- r3 and r4 compute the same result
- BUT, r4 will take much more time…
  - How far can you go with your computer?

DETERMINISTIC VS NON-DETERMINISTIC

Algorithms

- Algorithm
  - Sequence of non-ambiguous instructions
  - Finite amount of time

- Input to an algorithm
  - An instance of the problem the algorithm solves

- How to classify / group algorithms?
  - Type of problems solved
  - Design techniques
  - Deterministic vs non-deterministic

Deterministic Algorithms

- A deterministic algorithm
  - Returns the same answer no matter how many times it is called on the same data.
  - Always takes the same steps to complete the task when applied to the same data.

- The most familiar kind of algorithm !

- There is a more formal definition in terms of state machines…

Non-Deterministic Algorithms

- A non-deterministic algorithm
  - Can exhibit different behavior, for the same input data, on different runs.
  - As opposed to a deterministic algorithm !

- Often used to obtain approximate solutions to given problem instances
  - When it is too costly to find exact solutions using a deterministic algorithm

Non-Deterministic Algorithms

- How to behave differently from run to run ?
- Factors of non-deterministic behavior
  - External state other than the input data
    - User input / timer values / random values
  - Timing-sensitive operations on multiple processor machines
  - Hardware errors might force state to change in unexpected ways

PROBLEM TYPES

Problem Types

- Searching
- Sorting
- String Processing
- Graph / Network problems
- Combinatorial problems
- Bioinformatics
- …

- Examples of algorithms ?

Searching

- Which items?
  - Numbers, strings, records (key?), etc.

- Possible representations?
  - Arrays, lists, trees, etc.

- Ordered vs. non-ordered items
- Dynamically changing set?

- Sequential vs. binary search
- Others?

Sorting

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Tim Sort	O(n)	O(nlogn)	O(nlogn)	O(n)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)

- Which items?
  - Numbers, strings, records (key?), etc.
- Possible representations?
  - Arrays, lists, trees, etc.
- Use an indexing array?
- Which ordering? Repeated items?
- Stable? In-place?
- How many algorithms do you know?
- Which ones are the "most efficient"? When?

String Processing

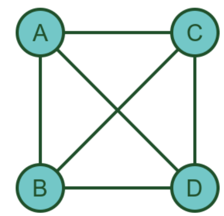
- Text strings, bit strings, gene sequences, etc.

- String matching?
- Longest-common substring?
- String-edit distance?
- Other problems / algorithms?

Graph / Network Problems

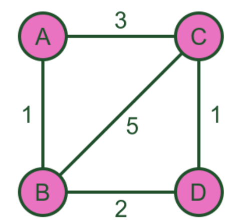
- Modeling the real-world!
- Dense vs. sparse graphs / networks
- Representations
  - Adjacency matrices vs. lists
  - Forward-star and reverse-star forms

    ???????????????
- Depth vs. breadth traversals
- Shortest path? K-shortest paths?
- Minimum spanning tree?
- Traveling salesman !
- Other problems?



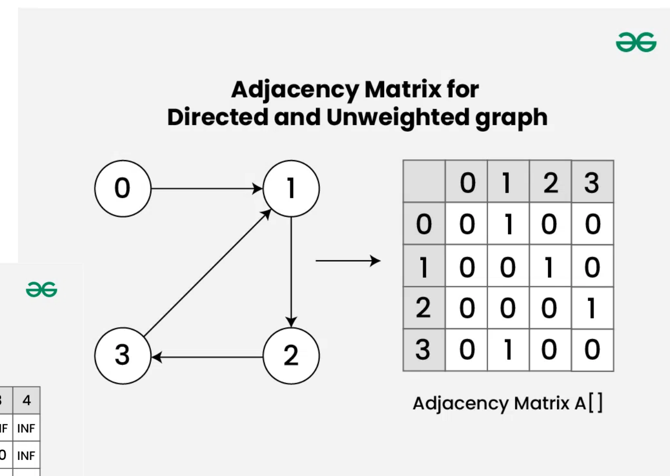
Complete graph

	A	B	C	D
A	0	1	1	1
B	1	0	1	1
C	1	1	0	1
D	1	1	1	0

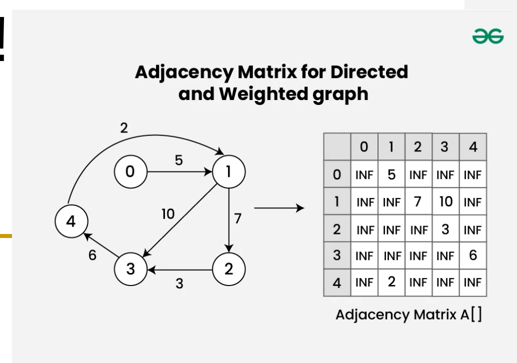


Weighted graph

	A	B	C	D
A	0	1	3	0
B	1	0	5	2
C	3	5	0	1
D	0	2	1	0



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



	0	1	2	3	4
0	INF	5	INF	INF	INF
1	INF	INF	7	10	INF
2	INF	INF	INF	3	INF
3	INF	INF	INF	INF	6
4	INF	2	INF	INF	INF

Combinatorial Problems

- Find a permutation, combination or subset !!

- What are the constraints?
- Are we optimizing some property?
  - Max value, min cost, etc.

- The most difficult problems in computing !!
- No (known?) polynomial algorithms for some problems !!

- Instance size vs. execution time
  - Exhaustive search?
- Optimal solutions vs. approximations

- Examples
  - N-Queens / Knapsack / Traveling salesman

Bioinformatics

- Applications in molecular biology

- Dealing with sequences (DNA or proteins)
  - Storing
  - Mapping and analyzing
  - Aligning

ALGORITHM DESIGN TECHNIQUES

Algorithm Design Techniques

- Design techniques / strategies / paradigms
- General approaches to problem solving
- Apply to
  - Various problem types
  - Different application areas

Algorithm Design Techniques

- Brute-Force
- Divide-and-Conquer
- Decrease-and-Conquer
- Transform-and-Conquer
- Dynamic Programming
- Greedy Algorithms

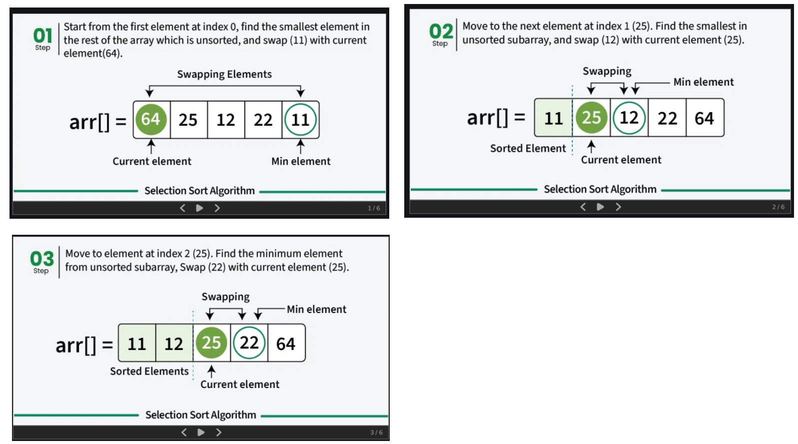
- Examples of algorithms ?

- What about problems / instances that cannot be solved within a reasonable amount of time ?

Brute-Force

Selection Sort is a comparison-based sorting algorithm. It sorts an array by repeatedly selecting the **smallest (or largest)** element from the unsorted portion and swapping it with the first unsorted element. This process continues until the entire array is sorted.

1. First we find the smallest element and swap it with the first element. This way we get the smallest element at its correct position.
2. Then we find the smallest among remaining elements (or second smallest) and swap it with the second element.
3. We keep doing this until we get all elements moved to correct position.

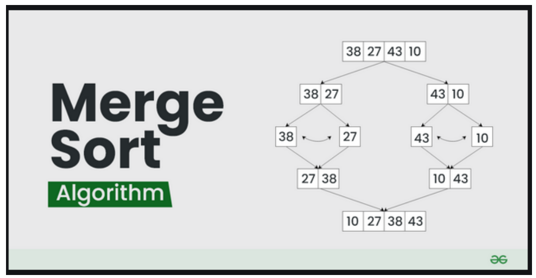


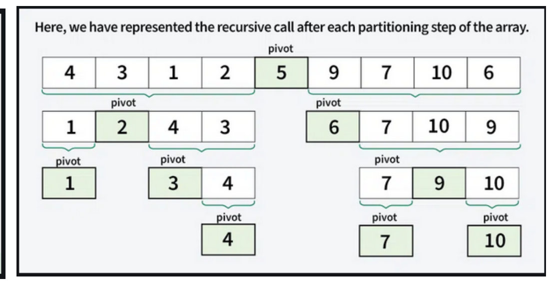
- Direct approaches
  - Selection sort
  - Sequential search
  - …
- Exhaustive search
  - Problem instances of small (?!) size
  - Traveling salesman
  - Knapsack
  - …

Divide-and-Conquer

- Recursive decomposition into “smaller” prob. instances
- Solve them all !

- Sorting
  - Mergesort
  - Quicksort

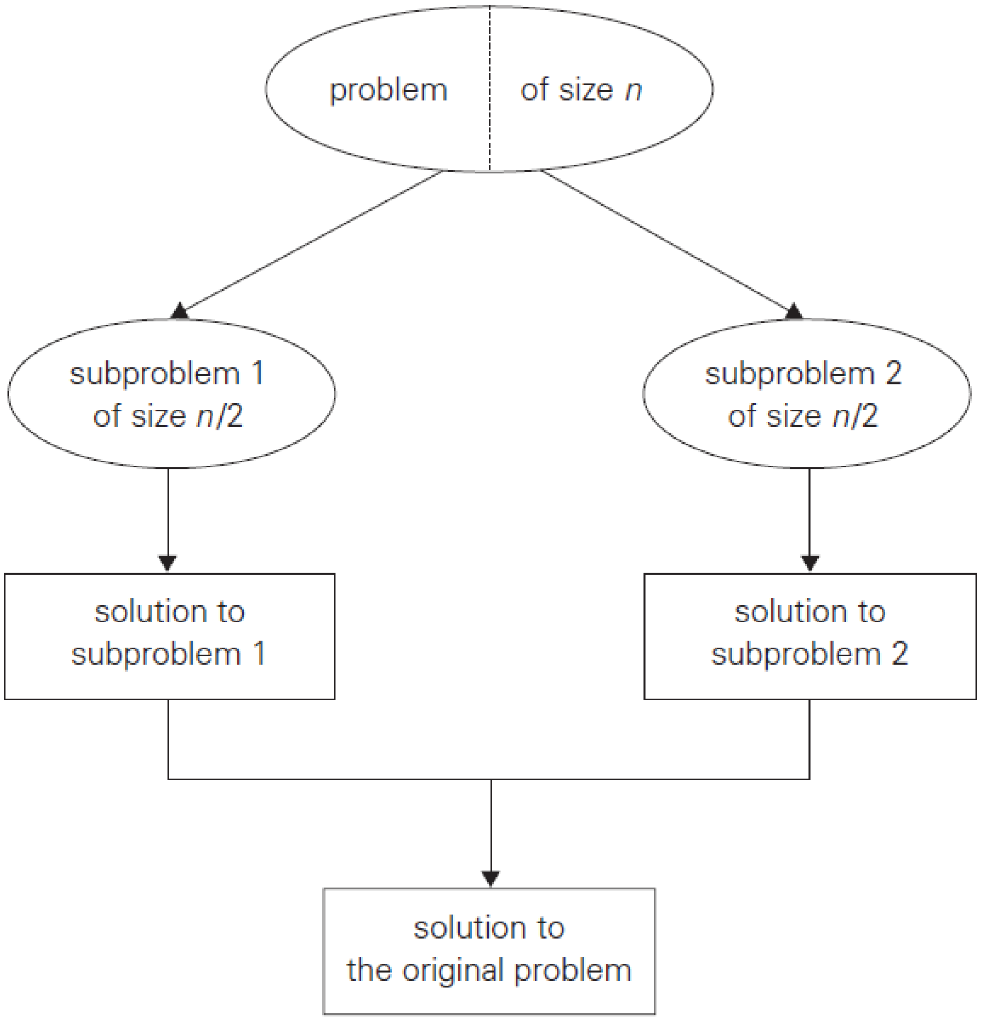




- Multiplication
  - Multiplying large integers
  - Strassen matrix multiplication

- …

Divide-and-Conquer



[Levitin]

FIGURE 5.1 Divide-and-conquer technique (typical case).

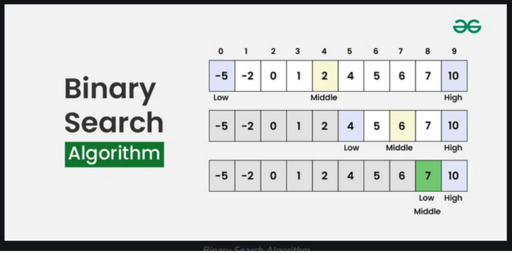
Decrease-and-Conquer

- Successive decomposition into a “smaller” problem instance

- How small is it?
  - Decrease-by-one
  - Decrease by a constant factor
  - Variable-size decrease

- Examples
  - Binary search
  - Interpolation search
  - Fake-coin problem

Binary Search Algorithm is a searching algorithm used in a sorted array by **repeatedly dividing the search interval in half**. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).



Interpolation Search Algorithm

As it is an improvisation of the existing BST algorithm, we are mentioning the steps to search the 'target' data value index, using position probing –

```
1. Start searching data from middle of the list.
2. If it is a match, return the index of the item, and exit.
3. If it is not a match, probe position.
4. Divide the list using probing formula and find the new middle.
5. If data is greater than middle, search in higher sub-list.
6. If data is smaller than middle, search in lower sub-list.
7. Repeat until match.
```

Let us solve the classic “fake coin” puzzle using decision trees. There are the two different variants of the puzzle given below. I am providing description of both the puzzles below, try to solve on your own, assume N = 8.

Easy: *Given a two pan fair balance and N identically looking coins, out of which only one coin is* ***lighter (or heavier)****. To figure out the odd coin, how many minimum number of weighing are required in the worst case?*

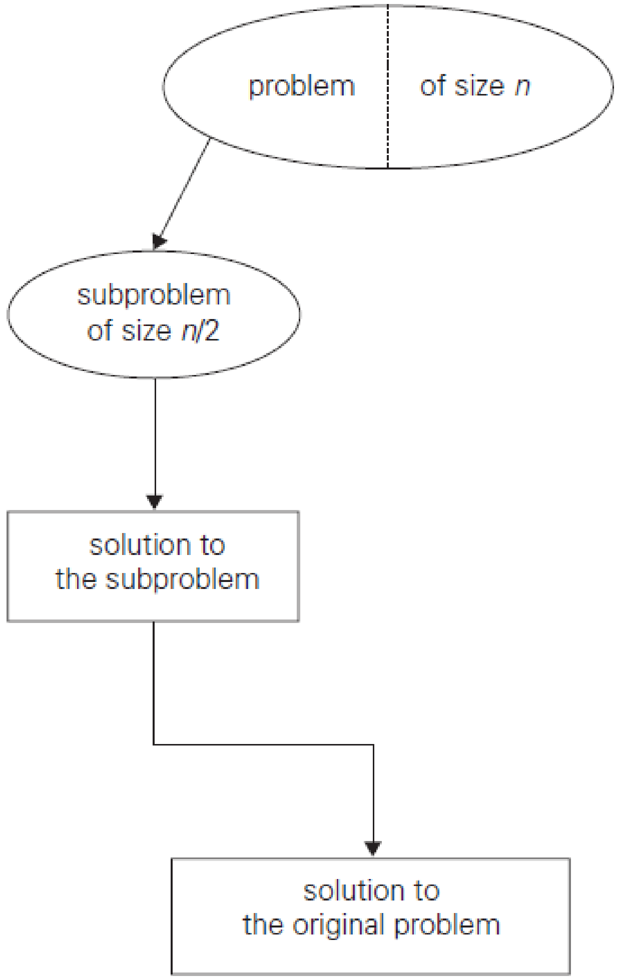
Difficult: *Given a two pan fair balance and N identically looking coins out of which only one coin* ***may be*** *defective. How can we trace which coin, if any, is odd one and also determine whether it is lighter or heavier in minimum number of trials in the worst case?*

The main idea here is to use more knowledge of the problem in setting up your test: If you separate into 3 instead of two stacks and do a weighing with two of those stacks (each containing the same number of coins), you can have only two cases given that the single fake coin can be in only one of these three stacks:

1.) Both sides have *identical* weight: the fake coin cannot be in the two stacks weighed, so must be in the 3rd: you reduced the problem space to 1/3

2.) One side weighs more than the other: since there is only one fake coin it must be on the side that weighs less: again you reduced the problem space to 1/3

Decrease-and-Conquer



[Levitin]

FIGURE 4.2 Decrease-(by half)-and-conquer technique.

Transform-and-Conquer

- Solve a different problem and get the desired result
  - Problem reduction

- Sometimes, perform some kind of pre-processing on the data

- Examples
  - Searching on ordered and balanced trees
    - AVL and 2-3 trees

(Adelson-Velsky and Landis Tree)

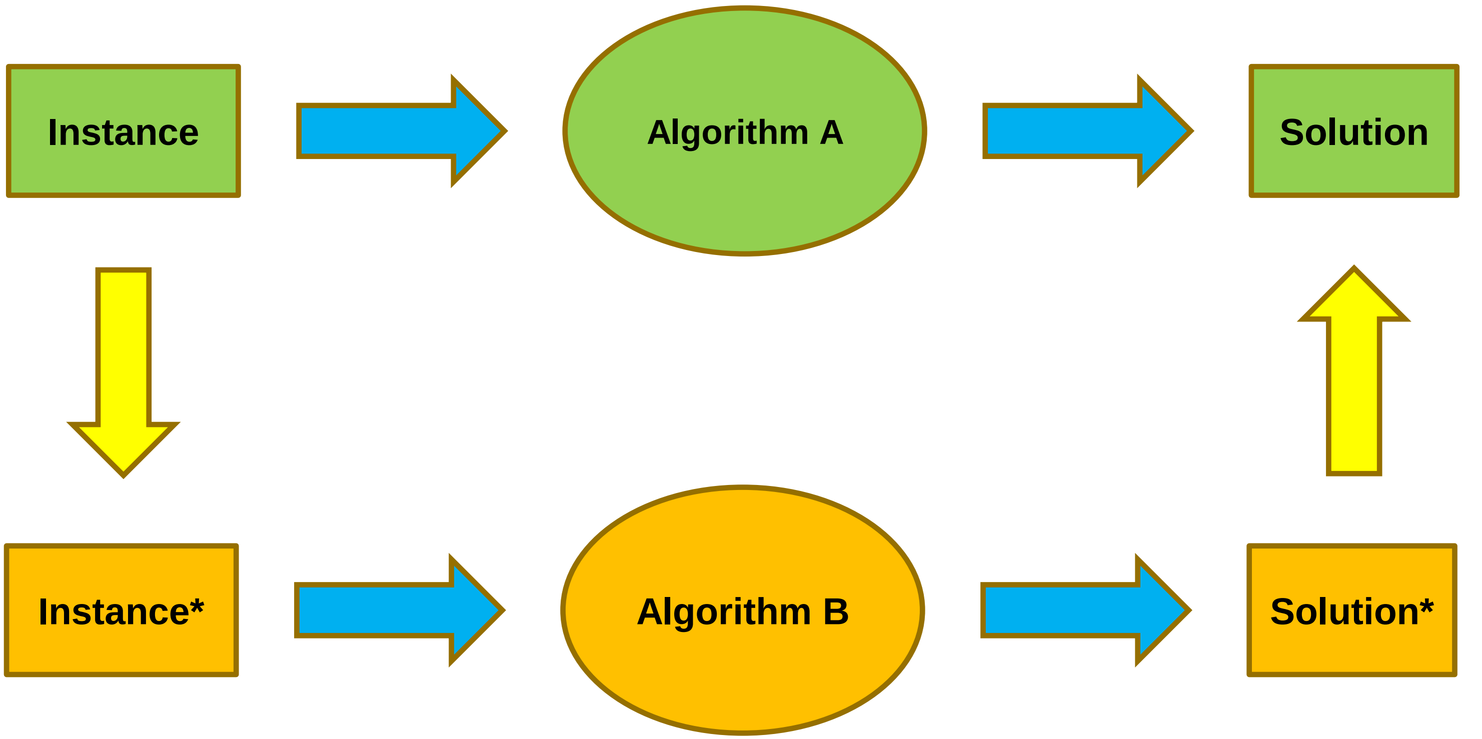
  - Heapsort

https://www.geeksforgeeks.org/heap-sort/#heap-sort-algorithm

2. Heapsort:

- **What it means**: Heapsort is a sorting algorithm that uses a binary heap (a complete binary tree satisfying the heap property) to sort elements.
  - **Heap Property**: For a max-heap, every parent node is greater than or equal to its children; for a min-heap, every parent node is smaller than or equal to its children.
  - The binary heap allows for efficient extraction of the maximum or minimum element in $O(\log n)$ time.
- **Process**:
  1. **Heapify**: The input array is transformed into a heap.
  2. **Sorting**: The root element (maximum/minimum) is repeatedly removed, and the heap is restructured.
- **Why it's "Transform-and-Conquer"**: The preprocessing step transforms the input array into a heap structure, which then allows sorting to be performed systematically and efficiently.

Transform-and-Conquer



Dynamic Programming

- Decomposition into overlapping (smaller !) sub-problems
  - Avoid solving them all !!
  - Proceed bottom-up
  - Store results and use them !!
- Simple examples
  - Computing Fibonacci numbers

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233

  - Computing binomial coefficients

Triângulo de Pascal

  - …
- Other
  - Graphs: Warshall alg.; Floyd alg; etc.
  - Knapsack

Greedy Algorithms

- Construct a solution through a sequence of steps
  - Expand a partially constructed solution

- The choice made at each step is
  - Feasible : satisfies constraints
  - Locally optimal : best choice at each step
  - Irrevocable (there is no backtracking)

- Examples
  - Coin-changing problem
  - Graphs
    - Dijkstra’s shortest-path algorithm
    - Prim’s minimum-spanning tree algorithm
    - Kruskal’s minimum-spanning tree algorithm

Introduction to Coin Change Problem. You are given an array of coins with varying denominations and an integer sum representing the total amount of money; you must return the fewest coins required to make up that sum; if that sum cannot be constructed, return -1. 23/07/2024

		0	1	2	3	4
No Coin	0	0	0	0	0	0
Only Coin 1	1	1	1	1	1	1
Coin 1 and Coin 2	2	1	1	2	2	3
All 1, 2 and 3 Coin	3	1	1	2	3	4

Limitations of Algorithmic Power

- How to cope?
- Backtracking
  - N-Queens problem
  - …
- Branch-and-Bound
  - Assignment problem
  - Knapsack problem
  - TSP (Travelling Salesman Problem)
  - …
- Approximation algorithms for NP-hard problems
  - Knapsack problem
  - TSP
  - ...

The **assignment problem** is a fundamental combinatorial optimization problem. In its most general form, the problem is as follows:

> The problem instance has a number of *agents* and a number of *tasks*. Any agent can be assigned to perform any task, incurring some *cost* that may vary depending on the agent-task assignment. It is required to perform as many tasks as possible by assigning at most one agent to each task and at most one task to each agent, in such a way that the *total cost* of the assignment is minimized.

Alternatively, describing the problem using graph theory:

> The assignment problem consists of finding, in a weighted bipartite graph, a matching of a given size, in which the sum of weights of the edges is minimum.

Branch and bound (BB, B&B, or BnB) is a method for solving optimization problems by breaking them down into smaller sub-problems and using a bounding function to eliminate sub-problems that cannot contain the optimal solution.

DATA STRUCTURES & ABSTRACT DATA TYPES

Fundamental Data Structures

- Algorithms operate on data !

- How to organize and store related data items?
  - Data structures (DS)

- Which operations should be provided?
  - Abstract data types (ADT) or classes (in OO languages)

- How to choose?
  - Identify the most common operations on the data
  - Identify the needs of particular algorithms

- Different algorithms for the same problem often require different data structures
  - Efficiency !!

Fundamental Data Structures

- Arrays
  - 1D, 2D, …

- Linked Lists
  - Single pointer vs. two pointers per node
  - List of lists
  - …

- Trees
  - Binary tree
  - Quaternary tree
  - …

Common Abstract Data Types

- Stack
- Queue
- Priority Queue
- Ordered List
- Binary Search Tree
- …
- Graph / Network
- …

ALGORITHM EFFICIENCY ANALYSIS

Algorithm Efficiency

- Analyze algorithm efficiency
  - Running time ?
  - Memory space ?

- Time
  - How fast does an algorithm run?

- Space
  - Does an algorithm require additional memory?

Efficiency Analysis

- How fast does an algorithm run ?
  - Most algorithms run longer on larger inputs !

- How to relate running time to input size ?

- How to rank / compare algorithms ?
  - If there is more than one available…

- How to estimate running time for larger problem instances ?

Running Time vs. Operations Count

- Running time is not (very) useful for comparing algorithms
  - Speed of particular computers
  - Chosen computer language
  - Quality of programming implementation
  - Compiler optimizations
- Evaluate efficiency in an independent way
  - Count the "basic operations" !!
    - Contribute the most to overall running time

Input Size

- Relate operations count / running time to input size !!
  - Number of array / matrix / list elements
  - …

- Relate size metric to the main operations of an algorithm
  - Working with individual chars vs. with words
  - Number of bits in binary rep., when checking if n is prime
  - …

Formal Analysis – Pencil and paper

- Understand algorithm behavior
  - Count arithmetic operations / comparisons
  - Find a closed formula !!
  - Identify best, worst and average case situations, if that is the case
- Iterative algorithms
  - Loops : how many iterations ?
  - Set a sum for the basic operation counts
- Recursive algorithms
  - How many recursive calls ?
  - Establish and solve appropriate recurrences

Worst, Best and Average Cases

- Running time depends on input size
- BUT, for some algorithms, it might also depend on particular data configurations !!

- Sequential search on a n-element array
  - Non-ordered array ?
  - Ordered array ?
  - Increasing vs. decreasing order
  - Probability of a successful search ?

Worst, Best and Average Cases

- Worst case : W(n)
  - Input(s) of size n for which an algorithm runs longest
  - Upper bound for operations count
- Best case : B(n)
  - Input(s) of size n for which an algorithm runs fastest
  - Lower bound for operations count
  - Not very useful…
- Average case : A(n)
  - Behavior for "typical" or "random" inputs
  - Establish assumptions about possible inputs of size n
  - For some algorithms, much better than worst case !!

Growth Rate

- Identify algorithm efficiency for large input sizes

- How fast does the running time (i.e., number of operations) of an algorithm grow, when input size becomes (much) larger ?

- What happens when the input size
  - doubles ?
  - increases ten-fold ?
  - …
- How to represent such growth rate?

Orders of Growth

- Approximate values for some common functions

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10	3.3×10^1	10^2	10^3	10^3	3.6×10^6
10^2	6.6	10^2	6.6×10^2	10^4	10^6	1.3×10^{30}	9.3×10^{157}
10^3	10	10^3	10^4	10^6	10^9	?	?
10^4	13	10^4	1.3×10^5	10^8	10^{12}	?	?
10^5	17	10^5	1.7×10^6	10^{10}	10^{15}	?	?
10^6	20	10^6	2.0×10^7	10^{12}	10^{18}	?	?

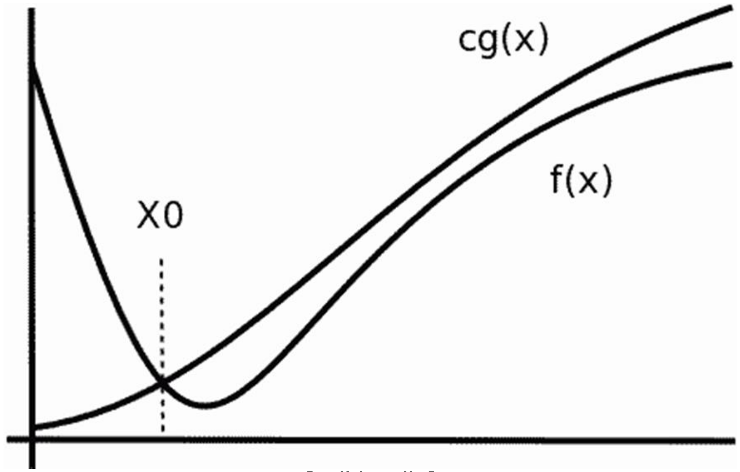
Asymptotic Notations

- Order of growth of operations count indicates efficiency

- How to compare / rank algorithms for the same problem?
  - Compare their orders of growth !!

- Useful notations: $O(n)$, $\Omega(n)$, $\Theta(n)$

Big-Oh Notation

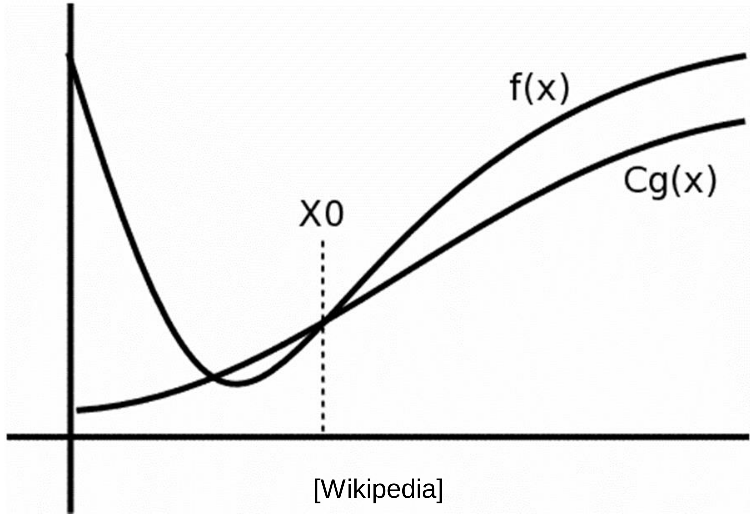


[Wikipedia]

- Asymptotic upper bound

- O(g(n)) : set of all functions with smaller or same order of growth as g(n)
  - $t(n) \leq c\, g(n)$, for all $n \geq n_0$, positive constant c
  - t(n), g(n) : non-negative functions on the set of natural numbers

Big-Omega Notation

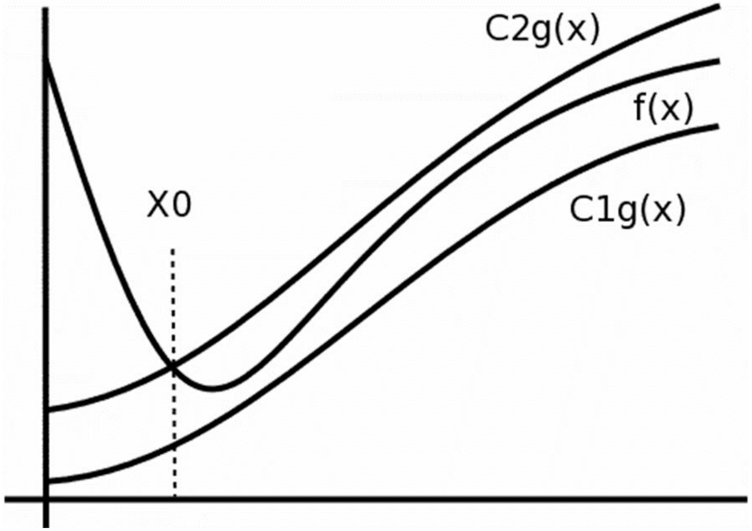


- Asymptotic lower bound

- Ω(g(n)) : set of all functions with larger or same order of growth as g(n)

  - $t(n) \geq c\, g(n)$, for all $n \geq n_0$, positive constant c

Big-Theta Notation



[Wikipedia]

- Asymptotic tight bound

- Θ(g(n)) : set of all functions with the same order of growth as g(n)

  - c_1 g(n) ≤ t(n) ≤ c_2 g(n), for all n ≥ n_0, positive constants c_1 , c_2

  - t(n) in O(g(n)) and t(n) in Ω(g(n))

Asymptotic Notation

- Hide unimportant details about how fast a function grows
  - Forget constants and lower-order terms

- $T_1(n) = 2\, n^2 + 3000\, n + 5$
- $T_2(n) = 10\, n^2 + 100\, n - 23$
- For large values of n, $T_2(n)$ grows faster than $T_1(n)$
- BUT both grow quadratically : $\Theta(n^2)$

Asymptotic Notation – Example

- $T(n) = 10\ n^2 + 100\ n - 23$

$T(n) = O(n^2)$ $T(n) = O(n^3)$ $T(n) \neq O(n)$

$T(n) = \Omega(n^2)$ $T(n) \neq \Omega(n^3)$ $T(n) = \Omega(n)$

$T(n) = \Theta(n^2)$ $T(n) \neq \Theta(n^3)$ $T(n) \neq \Theta(n)$

Efficiency Classes

- O(1) : constant
  - Which algorithms?
- O(log n) : logarithmic
  - E.g., decrease-and-conquer
- O(n) : linear
  - Processing all elements of an array, list, etc.
- O(n log n) : n-log-n
  - E.g., divide-and-conquer

Efficiency Classes

- $O(n^k)$: polynomial (quadratic, cubic, etc.)
  - k nested loops
- $O(2^n)$: exponential
  - Generating all subsets of an n-element set
- $O(n!)$: factorial
  - Generating all permutations of an n-element set

Empirical Analysis

- Run the algorithm on a sample of test inputs
  - Input data should represent all possible cases
  - Input data should encompass large (set) sizes
  - Pseudo-random data

- Record and analyze – Tables
  - operation counts
  - running times (?)

- Identify best, worst and average case behavior
  - If that is the case…

- Identify complexity classes

Example – Table of operations count

*2

n	1	2	4	8	16	32	64	128	256
$M(n)$	1	3	10	36	136	528	2080	8256	32896

*4

- M(n) : the number of operations carried out
- Complexity order ? O(n^2) ----- Ω(n^2)
- Closed formula for the number of operations ?

n(n+1)/2

Another table of operations count

+ 1

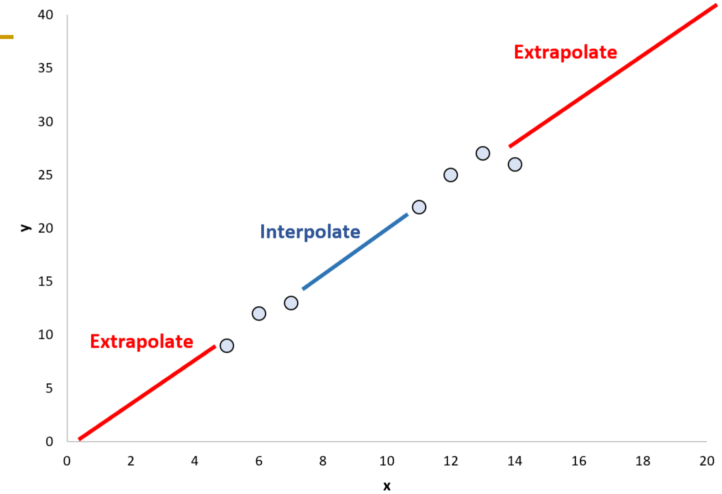
n	1	2	3	4	5	6	7	8	9	10
$M(n)$	1	3	7	15	31	63	127	255	511	1023

close to 2

- M(n) : the number of operations carried out
- Complexity order ? O(n^2) ----- Ω(n^2)
- Closed formula for the number of operations ?

n^2 -1

Empirical Analysis



Mathematically speaking, interpolation is the process of determining an unknown value within a sequence based on other points in that set, while extrapolation is the process of determining an unknown value outside of a set based on the existing "curve." 13/09/2021

interpolation vs. extrapolation

- Problems
  - Inadequate sample input data
    - Size? Configurations?
  - Dependence of running times
- Advantages
  - Avoid difficult formal analysis
  - Allow predicting the running time for different input data sets
    - Interpolation and extrapolation (?)
- BUT, some problems / instances cannot be solved quickly enough…

BRUTE-FORCE

Brute-Force

- The (most) straightforward approach to solving a problem

- Directly based on
  - The problem statement
  - The definitions involved

- Strengths
  - Simplicity
  - Applicable to different kinds of problems

- Weaknesses
  - (Very!) Low efficiency in some cases
  - Useful only for instances of (relatively) small size !!

Brute-Force

- Where to apply?

- Numerical problems, searching, sorting, etc.
  - Acceptable efficiency
  - Can be used for large problem instances

- Combinatorial problems
  - Exhaustive search
  - Set of candidate solutions grows very fast
  - Used only for reduced size instances

Brute-Force

- How many examples do you know?

- Add n numbers
- Direct matrix multiplication
- Sequential search
- Selection sort
- Bubble sort
- …

TASK 1 – DIRECT ALGORITHM ITERATIVE VS RECURSIVE

Brute-Force – Tasks

- Compute a^b, with $b \geq 0$, using

$$a^b = a \times a \times \ldots \times a$$

(iterative)

$$a^b = a \times a^{b-1}$$

(recursive)

- Base cases for the recursion ?

One of, prefer last
a^0 = 1, para b=0
a^1 = a, para b=1

- Number of multiplications ?
  - Formal + Empirical analysis

- Any gains from the recursive approach ?

DIVIDE-AND-CONQUER

Divide-And-Conquer

- The best-known algorithm design technique

- General framework
  - Divide a problem instance into (two or more) similar, smaller instances
  - The smaller instances are solved recursively
  - Solutions for smaller instances are combined to get the solution of the original problem, if needed

Divide-And-Conquer

- In each subdivision step, the smaller instances should have approx. the same size !
  - This might not happen, for some particular instances

- All smaller problem instances have to be solved !!
  - Usually two new smaller instances, at each step

- When do we stop the subdivision process ?
  - Base cases ? Just one or more ?
  - Smaller instances might be solved by another algorithm

Divide-And-Conquer

- This recursive strategy can be implemented
  - Using recursive functions / procedures (obvious solution !)
  - Iteratively, using a stack, queue, etc.
    - Choose which sub-problem to solve next !!

- Problems ?
  - Recursion is slow !
    - Identify all possible base cases
    - Solve small instances using other algorithms
  - Not the best approach for simple problems !
    - E.g., adding N numbers
  - Sub-problems might overlap !
    - Reuse previous results / solutions !

TASK 2 – DIVIDE & CONQUER RECURSIVE FUNCTION

Divide-And-Conquer – Tasks

- Compute a^b, with $b \geq 0$, using

$$a^b = a^{b \text{ div } 2} \times a^{(b+1) \text{ div } 2}$$

- Base cases ? Need both
  a^0 = 1, para b=0
  a^1 = a, para b=1
- Always use two recursive calls !!

- Number of multiplications ?
  - Formal + Empirical analysis

DECREASE-AND-CONQUER

Decrease-And-Conquer

- Exploit the relationship between
  - A solution to a given problem instance
  - A solution to a smaller instance of the same problem

- General framework (Top-Down)
  - Identify ONE similar and smaller problem instance
  - The smaller instance is solved recursively
  - Solutions for smaller instances are processed to get the solution of the original problem, if needed

- Compare with Divide-and-Conquer !!

TASK 3 – DEC. & CONQUER RECURSIVE FUNCTION

Decrease-And-Conquer – Tasks

- Compute a^b, with $b \geq 0$, using

$$a^b = a^{b \text{ div } 2} \times a^{b \text{ div } 2} \text{ , if } b \text{ is even}$$

$$a^b = a \times a^{(b-1) \text{ div } 2} \times a^{(b-1) \text{ div } 2} \text{ , if } b \text{ is odd}$$

- Base cases ? Can stop at 0 or 1 depending on how you do it
- Use just ONE recursive call !!
- Number of multiplications ?
  - Formal + Empirical analysis

EXTRA TASK – D & C ITERATIVE FUNCTION

Decrease-And-Conquer – Extra-Task

- Compute a^b, with $b \geq 0$, using

$$a^b = a^{b \text{ div } 2} \times a^{b \text{ div } 2} \text{, if } b \text{ is even}$$

$$a^b = a \times a^{(b-1) \text{ div } 2} \times a^{(b-1) \text{ div } 2} \text{, if } b \text{ is odd}$$

- Develop an iterative version !!

- It should have the same behavior as the recursive version
  - Same algorithm, but a different implementation

ADDITIONAL TASKS
- TRY DOING IT AT HOME

New Task – Counting

- Given an array with non-negative integer values

- Count the number of even-valued elements

- Implement the 3 strategies:
  - Brute-Force / Div & C / Dec & C

- Formal + Empirical analysis : Comparisons

New Task – Sequential Search

- Given an array with non-negative integer values
- Use the iterative Sequential Search algorithm to look for a given value
- Formal + Empirical analysis : Comparisons
- Best / Worst / Average Cases ?

REFERENCES

References

- A. Levitin, *Introduction to the Design and Analysis of Algorithms,* 3rd Ed., Pearson, 2012
  - Chapter 1 + Chapter 2
- D. Vrajitoru and W. Knight, *Practical Analysis of Algorithms*, Springer, 2014
  - Chapter 1 + Chapter 3 + Chapter 5
- T. H. Cormen et al., *Introduction to Algorithms*, 3rd Ed., MIT Press, 2009
  - Chapter 1 + Chapter 2 + Chapter 3