Design and Analysis of Algorithms

Mohammad GANJTABESH

mgtabesh@ut.ac.ir

School of Mathematics, Statistics and Computer Science,
University of Tehran,
Tehran, Iran.

Analysis of Algorithms

Analysis of Algorithms

Different input requires different amount of resources. Instead of dealing with input itself, we prefer to deal with the Size of Input.

Analysis of Algorithms

Different input requires different amount of resources. Instead of dealing with input itself, we prefer to deal with the Size of Input.

Definition (Size of Input)

The amount of memory required for representing the input with respect to a fixed coding scheme.

Analysis of Algorithms

Different input requires different amount of resources. Instead of dealing with input itself, we prefer to deal with the Size of Input.

Definition (Size of Input)

The amount of memory required for representing the input with respect to a fixed coding scheme.

Examples:

- For an array $\Longrightarrow$ The size can be considered as the number of its elements...
- For an integer $\Longrightarrow$ The size can be considered as the number of its bit in binary representation...
- For a graph $\Longrightarrow$ The size can be considered as the number of vertices or edges...

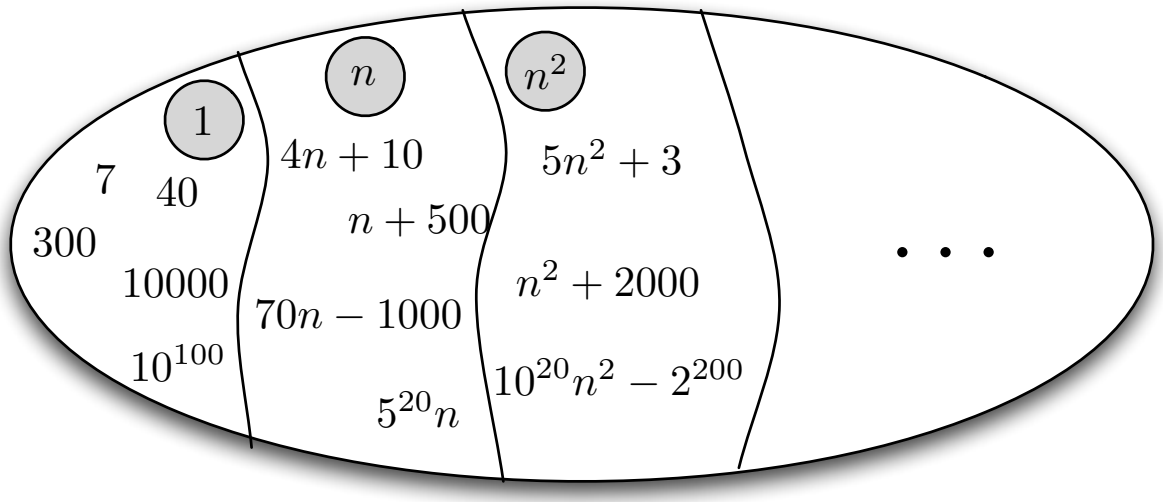
Analysis of Algorithms

- Predicting the amount of resources that the algorithm requires.
- We expect that the amount of resources used by any algorithm growth with respect to the input size.
- Measure the required time (time complexity) and memory (space complexity).
  - **Best case:** Measuring the required amount of resources for easy instances.
  - **Worse case:** Measuring the required amount of resources for hard instances.
  - **Average case:** Measuring the required amount of resources for all instances divided by the number of instances.
- Should be independent from the current technology (Hardware, Programming Languages, etc.).
- Should be independent from the way of implementing the algorithm.
- By analyzing several candidate algorithms for a specific problem, the most efficient one can be easily identified.

classifying the functions

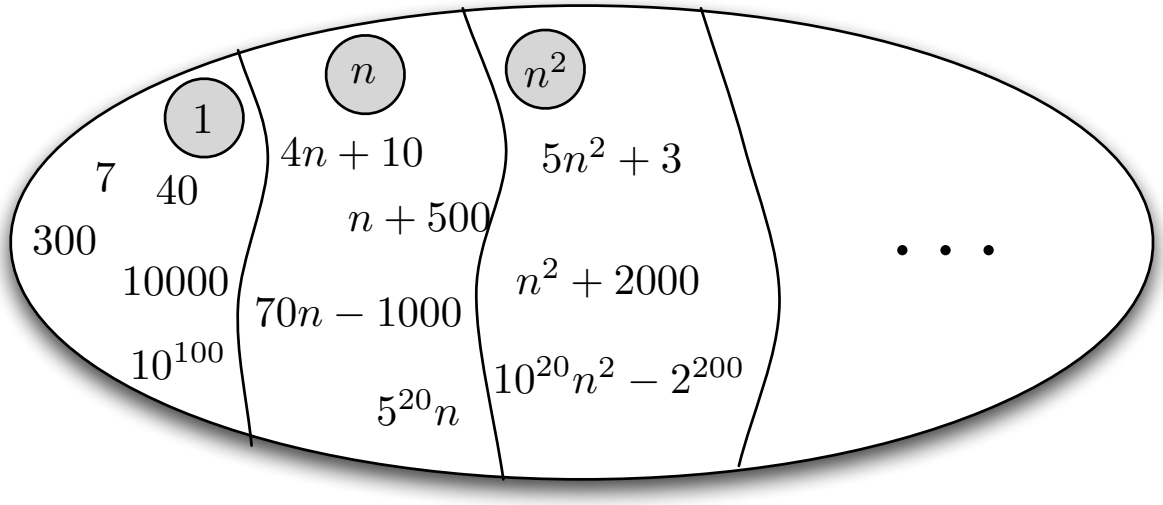
classifying the functions

There are infinitely many functions. In order to compare them with each other, we prefer to classify them as simple functions without loss of their properties. In fact we want to choose one simple function for each class of functions.



classifying the functions

There are infinitely many functions. In order to compare them with each other, we prefer to classify them as simple functions without loss of their properties. In fact we want to choose one simple function for each class of functions.



Suppose that $f(n)$ and $g(n)$ are two functions, defined for non-negative integers n and with values in the set of real numbers.

Θ Notation

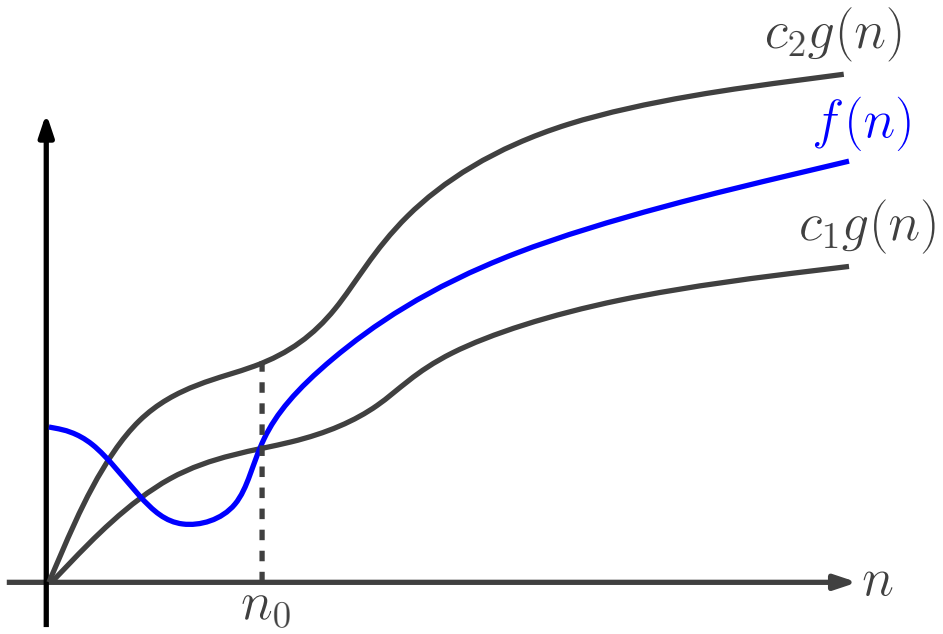
Definition

$f(n) = \Theta(g(n))$ if and only if
$\exists c_1, c_2 > 0, \exists n_0 > 0 \ni \forall n \geq n_0, 0 < c_1.g(n) \leq f(n) \leq c_2.g(n)$.

Θ Notation

Definition

$f(n) = \Theta(g(n))$ if and only if
$\exists c_1, c_2 > 0, \exists n_0 > 0 \ni \forall n \geq n_0, 0 < c_1.g(n) \leq f(n) \leq c_2.g(n)$.



Upper and Lower bound at the same time.

Big-O Notation

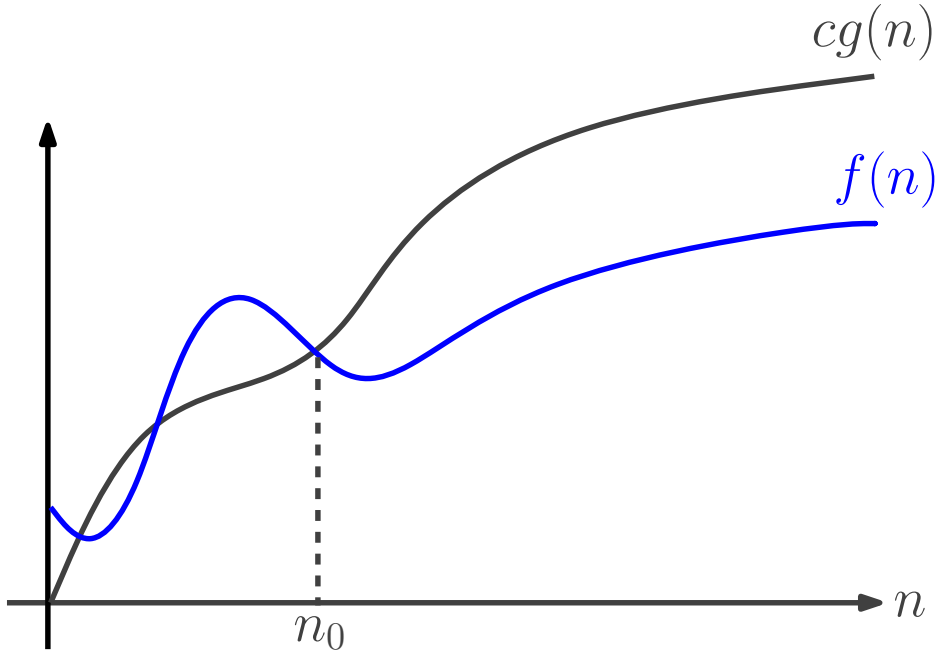
Definition

$f(n) = O(g(n))$ if and only if
$\exists c > 0, \exists n_0 > 0 \ni \forall n \geq n_0, 0 < f(n) \leq c.g(n)$.

Big-O Notation

Definition

$f(n) = O(g(n))$ if and only if
$\exists c > 0, \exists n_0 > 0 \ni \forall n \geq n_0, 0 < f(n) \leq c.g(n)$.



Smalest upper bound.

Ω Notation

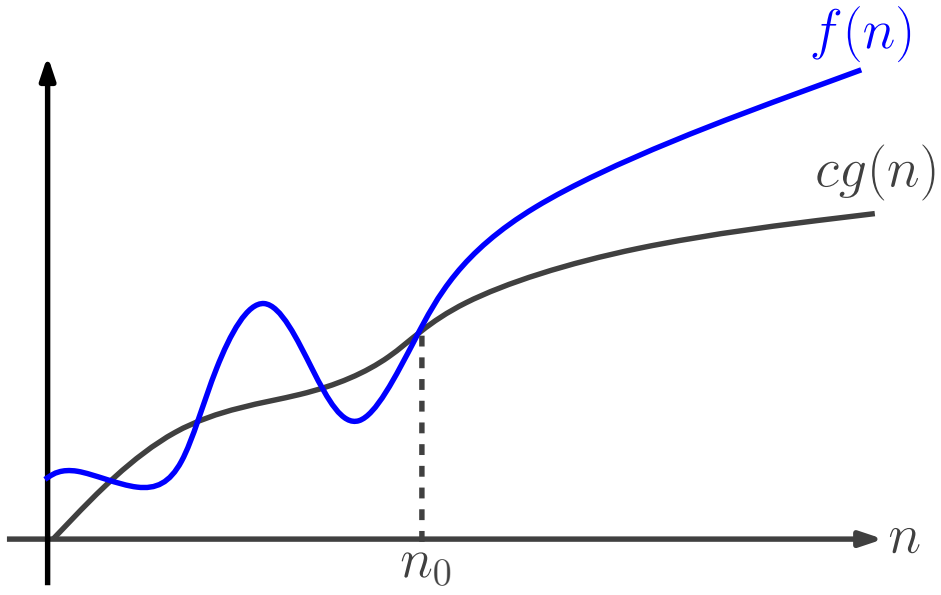
Definition

$f(n) = \Omega(g(n))$ if and only if
$\exists c > 0, \exists n_0 > 0 \ni \forall n \geq n_0, 0 < c.g(n) \leq f(n)$.

Ω Notation

Definition

$f(n) = \Omega(g(n))$ if and only if
$\exists c > 0, \exists n_0 > 0 \ni \forall n \geq n_0, 0 < c.g(n) \leq f(n)$.



Largest lower bound.

Small-o and ω Notations

Definition

$f(n) = o(g(n))$ if and only if
$\forall c > 0, \exists n_0 > 0 \ni \forall n \geq n_0, 0 < f(n) \leq c.g(n)$.

Definition

$f(n) = \omega(g(n))$ if and only if
$\forall c > 0, \exists n_0 > 0 \ni \forall n \geq n_0, 0 < c.g(n) \leq f(n)$.

Properties of these notations

Theorem

$f(n) = \Theta(g(n)) \iff f(n) = O(g(n))$ *and* $f(n) = \Omega(g(n))$.

Properties of these notations

Theorem

$f(n) = \Theta(g(n)) \iff f(n) = O(g(n))$ *and* $f(n) = \Omega(g(n))$.

Other Properties:

- Reflexivity: $f(n) = \Theta(f(n))$, $f(n) = O(f(n))$, and $f(n) = \Omega(f(n))$.
- Symmetry: $f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$.
- Transitivity:
  - $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \implies f(n) = \Theta(h(n))$.
  - $f(n) = O(g(n))$ and $g(n) = O(h(n)) \implies f(n) = O(h(n))$.
  - $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n)) \implies f(n) = \Omega(h(n))$.
- Transpose symmetry: $f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$.

Exercises

1. Let $p(n) = \sum_{i=0}^{d} a_i n^i$, where $a_d > 0$, be a degree-d polynomial in n, and let k be a constant. Use the definitions of the asymptotic notations to prove the following properties:
   a. If $k \geq d$,then $p(n) = O(n^k)$.
   b. If $k \leq d$,then $p(n) = \Omega(n^k)$.
   c. If $k = d$,then $p(n) = \Theta(n^k)$.
2. Let $f(n)$ and $g(n)$ be asymptotically positive functions. Prove or disprove each of the following conjectures:
   a. $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.
   b. $f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$.
   c. $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.
   d. Either $f(n) = O(g(n))$ or $f(n) = \Omega(g(n))$ holds.

Techniques for the design of Algorithms

The classical techniques are as follows:

1. Divide and Conquer
2. Dynamic Programming
3. Greedy Algorithms
4. Backtracking Algorithms
5. Branch and Bound Algorithms

Divide and Conquer

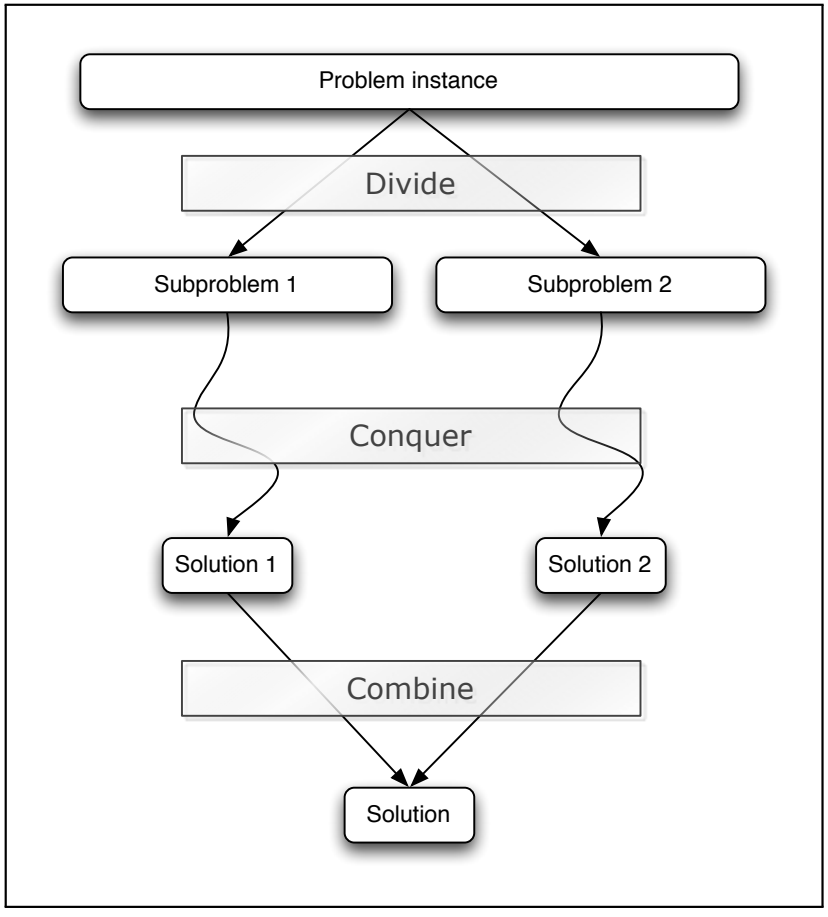
This approach involves three steps:

1. **Divide:** Break down the problem into two or more subproblems. These subproblems should be similar to the original problem, but smaller in size.
2. **Conquer:** Recursively solve the subproblems (If they are small enough, just solve them in a straightforward manner).
3. **Combine:** Combine the solutions to the subproblems into a solution for the original problem (optional).

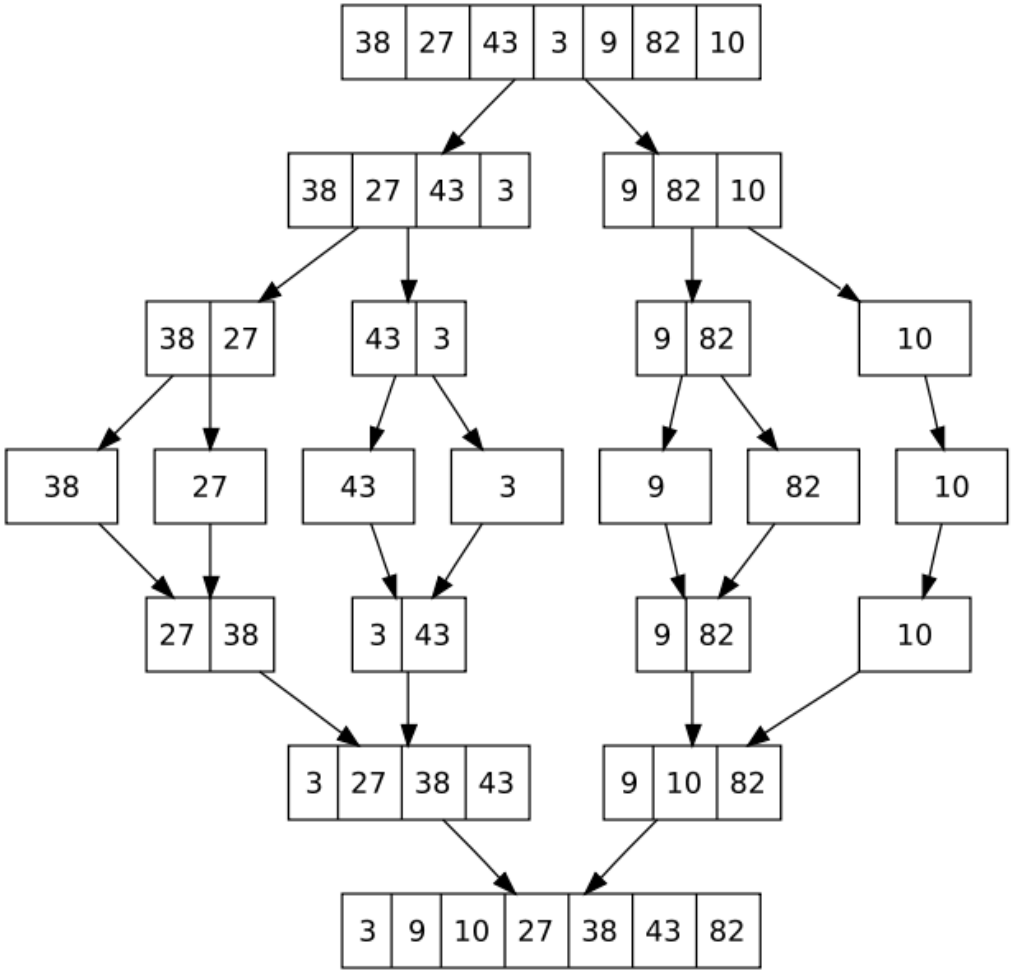
Divide and Conquer

This approach involves three steps:

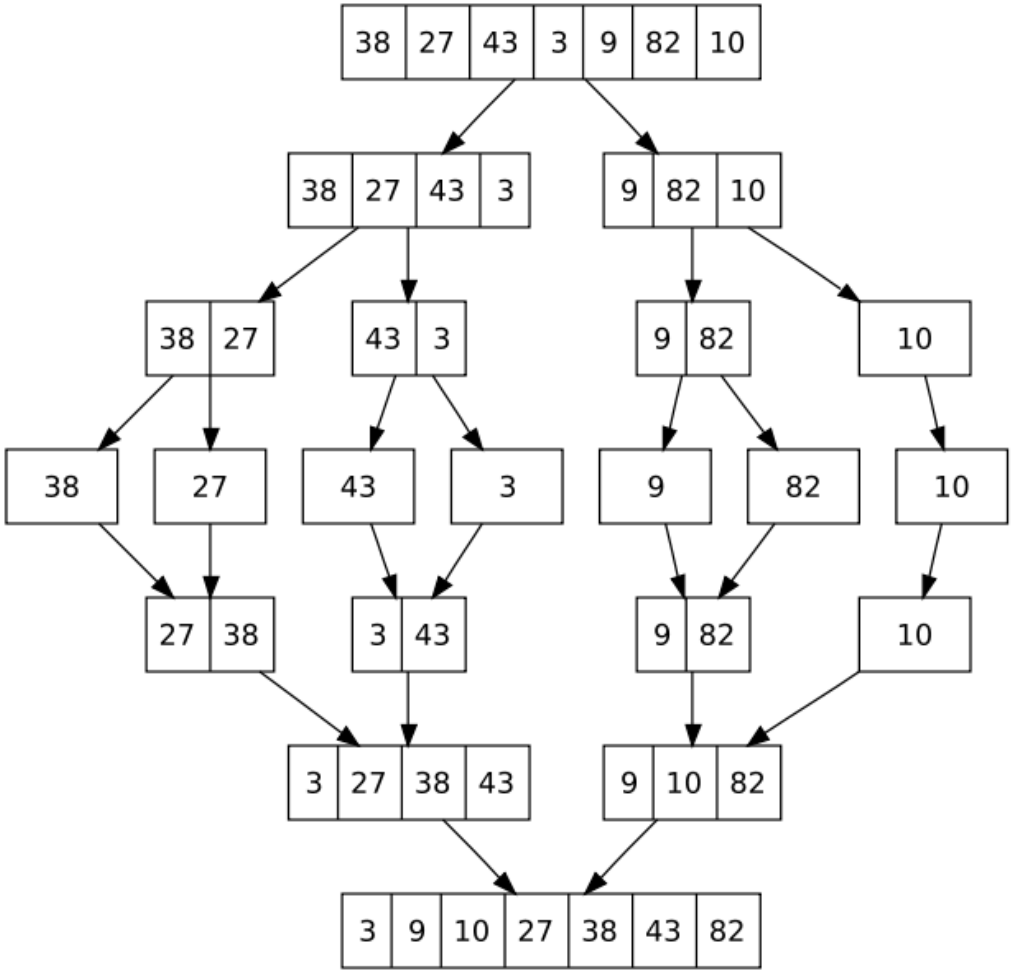
1. **Divide:** Break down the problem into two or more subproblems. These subproblems should be similar to the original problem, but smaller in size.
2. **Conquer:** Recursively solve the subproblems (If they are small enough, just solve them in a straightforward manner).
3. **Combine:** Combine the solutions to the subproblems into a solution for the original problem (optional).



Example: Merge Sort



Example: Merge Sort




```
MERGE-SORT(A, p, r)
1  if p < r
2     then q ← ⌊(p + r)/2⌋
3          MERGE-SORT(A, p, q)
4          MERGE-SORT(A, q + 1, r)
5          MERGE(A, p, q, r)
```

Example: Merge Sort

```
MERGE(A, p, q, r)
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
 4  for i ← 1 to n1
 5        do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7        do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13        do if L[i] ≤ R[j]
14              then A[k] ← L[i]
15                   i ← i + 1
16              else A[k] ← R[j]
17                   j ← j + 1
```

Analysing the Divide-and-Conquer Algorithms

Analysing the Divide-and-Conquer Algorithms

In general we have the following recurrence equation:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{Otherwise.} \end{cases}$$

where:

- $T(n)$: is the time requred for an input of size n
- n: is the size of problem
- c: is a constant number
- a: is the number of subproblems
- n/b: is the size of each subproblem
- $D(n)$: is the time needed for Divide
- $C(n)$: is the time needed for Combine

Example: Analysing Merge Sort

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ 2T(n/2) + O(1) + O(n) & \text{Otherwise.} \end{cases}$$

- What is the implicit formula of $T(n)$?
- How we can find it?

Solving the recurrence equations

There are different approaches to do this:

- Constructing Recursion Tree
- Performing Substitution
- Using Induction
- Master Theorem
- Generating Functions