Algorithms and Data Structures (ALG)

Indhold

[Concepts 2](#_Toc41430453)

[Computational Problem 2](#_Toc41430454)

[Algorithm and Pseudocode 2](#_Toc41430455)

[Worst-case and best-case analysis 2](#_Toc41430456)

[Asymptotic Notation (big-O, big-Omega, and big-Theta) 3](#_Toc41430457)

[Properties of Asymptotic Notations : 5](#_Toc41430458)

[Proving the correctness of an algorithm (using e.g., induction or loop invariants) 6](#_Toc41430459)

[Solving Recurrences (using, substitution, recursion-tree, and Master methods) 6](#_Toc41430460)

[max/min-heap property 6](#_Toc41430461)

[Hash functions and their properties 6](#_Toc41430462)

[Binary search tree property 6](#_Toc41430463)

[Red-black-tree property 6](#_Toc41430464)

[Optimal Substructure Property 6](#_Toc41430465)

[Graph of dependencies 6](#_Toc41430466)

[Graphs, cycles on graphs, negative cycles, directed acyclic graphs 6](#_Toc41430467)

[Topological ordering on an acyclic graph 6](#_Toc41430468)

[shortest paths problems on graphs and weighted graphs 6](#_Toc41430469)

[Algorithms 6](#_Toc41430470)

[Insertion Sort 6](#_Toc41430471)

[Merge Sort 6](#_Toc41430472)

[Heapsort 6](#_Toc41430473)

[Quicksort 6](#_Toc41430474)

[Counting Sort 6](#_Toc41430475)

[Radix Sort 6](#_Toc41430476)

[Breadth-First Search 6](#_Toc41430477)

[Depth-first Search 6](#_Toc41430478)

[Topological Sorting 6](#_Toc41430479)

[Computing Strongly Connected Components 6](#_Toc41430480)

[Bellman-Ford Algorithm 6](#_Toc41430481)

[Dijkstra’s algorithm 6](#_Toc41430482)

[Floyd-Warshall algorithm 6](#_Toc41430483)

[Transitive closure of a directed graph 6](#_Toc41430484)

[Data Structures 7](#_Toc41430485)

[Max-Heaps & Min-Heaps 7](#_Toc41430486)

[Stacks 7](#_Toc41430487)

[Queues 7](#_Toc41430488)

[Linked Lists (both doubly-linked and singly-linked) 7](#_Toc41430489)

[Rooted Trees 7](#_Toc41430490)

[Hash Tables (direct-address, open addressing) 7](#_Toc41430491)

[Binary Search Trees 7](#_Toc41430492)

[Red-Black Trees 7](#_Toc41430493)

[Directed and Undirected Graphs (adjacency-list & adjacency-matrix representations) 7](#_Toc41430494)

[Weighted Graphs (adjacency-list & adjacency-matrix representations) 7](#_Toc41430495)

[Algorithm Design 7](#_Toc41430496)

[Divide and Conquer 7](#_Toc41430497)

[Dynamic Programming 7](#_Toc41430498)

## Concepts

### Computational Problem

A computational problem can be viewed as an infinite collection of instances together with a solution for every instance.

### Algorithm and Pseudocode

### Worst-case and best-case analysis

Worst case:  
In the worst case analysis, we calculate upper bound - O(f(n)) on running time of an algorithm.

We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched for is not present in the array. The function will then compare it with all the elements of arr[] one by one. And find nothing going through the whole array.

#### Best case:

In the best case analysis, we calculate lower bound - Ω(f(n)) on running time of an algorithm.  
We must know the case that causes minimum number of operations to be executed.   
**Example:** In the linear search problem, the best case occurs when x is present at the first location.   
The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Θ(1)

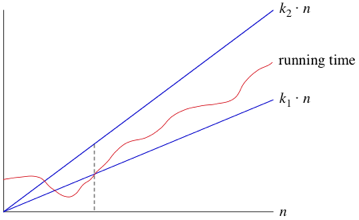
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorithm | Data structure | Time complexity:Best | Time complexity:Average | Time complexity:Worst | Space complexity:Worst |
| Quick sort | Array | O(*n* log(*n*)) | O(*n* log(*n*)) | O(*n*2) | O(*n*) |
| Merge sort | Array | O(*n* log(*n*)) | O(*n* log(*n*)) | O(*n* log(*n*)) | O(*n*) |
| Heap sort | Array | O(*n* log(*n*)) | O(*n* log(*n*)) | O(*n* log(*n*)) | O(1) |
| Smooth sort | Array | O(*n*) | O(*n* log(*n*)) | O(*n* log(*n*)) | O(1) |
| Bubble sort | Array | O(*n*) | O(*n*2) | O(*n*2) | O(1) |
| Insertion sort | Array | O(*n*) | O(*n*2) | O(*n*2) | O(1) |
| Selection sort | Array | O(*n*2) | O(*n*2) | O(*n*2) | O(1) |
| Bogo sort | Array | O(*n*) | O(*n* *n*!) | O(∞) | O(1) |

### Asymptotic Notation (big-O, big-Omega, and big-Theta)

For each little computation it takes a constant amount of time each time it executes. If the for-loop iterates  times, then the time for all  iterations is c1 ​⋅, is the sum of the times for the computations in one loop iteration.

#### Big-Theta - Θ(n) (asymptotically tight bound)

When we say that a particular running time is Θ(n) , we're saying that once ​ gets large enough, the running time is at least c1 ​⋅ and at most c2 ​⋅ for some constants c1 and c2:

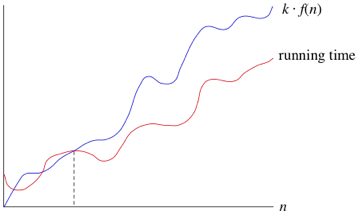


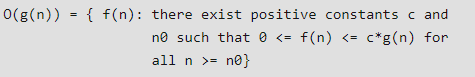
As long as these constants c1 and c2 exist, we say that the running time is Θ(n).

When you use big-Θ notation, you remove the constants and can therefore just say Θ(n).



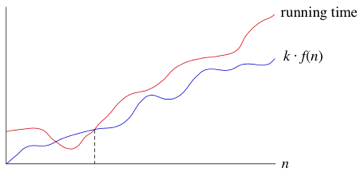
big-O - O(f(n)) (asymptotic upper bounds)  
  
“the running time grows at most this much, but it could grow more slowly."

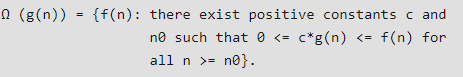
If a running time is O(f(n)), then for large enough  , the running time is at most , for some constant c:  




#### big-Omega - Ω(f(n)) (asymptotic lower bounds)

If a running time is Ω(f(n)) then for large enough , the running time is at least  for some constant c:





Properties of Asymptotic Notations :  
As we have gone through the definition of this three notations let’s now discuss some important properties of those notations.

#### General Properties :

If f(n) is O(g(n)) then a\*f(n) is also O(g(n)) ; where a is a constant.

**Example**: f(n) = 2n²+5 is O(n²)  
**then** 7\*f(n) = 7(2n²+5)  
= 14n²+35 is also O(n²)

Similarly this property satisfies for both Θ and Ω notation.  
We can say  
If f(n) is Θ(g(n)) then a\*f(n) is also Θ(g(n)) ; where a is a constant.  
If f(n) is Ω (g(n)) then a\*f(n) is also Ω (g(n)) ; where a is a constant.

#### Reflexive Properties :

If f(n) is given then f(n) is O(f(n)).

**Example**: f(n) = n² ; O(n²) i.e O(f(n))

Similarly this property satisfies for both Θ and Ω notation.  
We can say  
If f(n) is given then f(n) is Θ(f(n)).  
If f(n) is given then f(n) is Ω (f(n)).

#### Transitive Properties :

If f(n) is O(g(n)) and g(n) is O(h(n)) then f(n) = O(h(n)) .

**Example**: if f(n) = n , g(n) = n² and h(n)=n³  
n is O(n²) and n² is O(n³)  
**then** n is O(n³)

Similarly this property satisfies for both Θ and Ω notation.  
We can say  
If f(n) is Θ(g(n)) and g(n) is Θ(h(n)) then f(n) = Θ(h(n)) .  
If f(n) is Ω (g(n)) and g(n) is Ω (h(n)) then f(n) = Ω (h(n))

#### Symmetric Properties :

If f(n) is Θ(g(n)) then g(n) is Θ(f(n)) .

**Example**: f(n) = n² and g(n) = n²  
**then** f(n) = Θ(n²) and g(n) = Θ(n²)

**This property only satisfies for Θ notation.**

Transpose Symmetric Properties :

If f(n) is O(g(n)) then g(n) is Ω (f(n)).

**Example**: f(n) = n , g(n) = n²  
**then** n is O(n²) and n² is Ω (n)

**This property only satisfies for O and Ω notations**.

#### Some More Properties :

If f(n) = O(g(n)) and f(n) = Ω(g(n)) then f(n) = Θ(g(n))

If f(n) = O(g(n)) and d(n)=O(e(n))  
**then** f(n) + d(n) = O( max( g(n), e(n) ))  
**Example**: f(n) = n i.e O(n)  
d(n) = n² i.e O(n²)  
**then** f(n) + d(n) = n + n² i.e O(n²)

If f(n)=O(g(n)) and d(n)=O(e(n))  
**then** f(n) \* d(n) = O( g(n) \* e(n) )  
**Example**: f(n) = n i.e O(n)  
d(n) = n² i.e O(n²)  
then f(n) \* d(n) = n \* n² = n³ i.e O(n³)

### Proving the correctness of an algorithm (using e.g., induction or loop invariants)

Induction  
mathematical demonstration of the validity of a law concerning all the positive integers by proving that it holds for the integer 1 and that if it holds for an arbitrarily chosen positive integer k, it must hold for the integer k + 1.

Step 1. Show it is true for the **first one**

Step 2. Show that if **any one** is true then the **next one** is true

**Example:** is 3n−1 a multiple of 2?

**1.** Show it is true for **n=1**

Yes 2 is a multiple of 2. That was easy.

**2.** Assume it is true for **n=k**

is true

(Not necessarly true but, it is an assumption ... that we treat as a fact for the rest of this example)

Now, prove that  is a multiple of 2

 is also

is also +

And each of these are multiples of 2

Because:

 is a multiple of 2 (we are multiplying by 2)

is true (we said that in the assumption above)

**Example:** Show that 2n < (n + 1)!, for n ≥ 2.

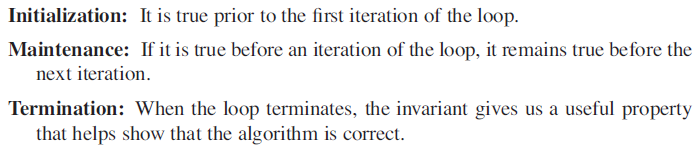
**Proof**: The base step is as follows:  
**Base Step**: n = 2: we have to check that 22 < 3!, but 4 < 6, so this is true.   
**Inductive Step:** Assume 2k < (k + 1)!, for k ≥ 2. We have  
2k+1 = 2 · 2k < 2 · (k + 1)!   
Since k ≥ 2, we have (k + 2) > 2. Therefore:  
2 · (k + 1)! < (k + 2) · (k + 1)! = (k + 2)!  
This completes the inductive step.

#### Loop invariants

An inductive statement which says something which is always true about a program loop.  
Typically written as: S(n) where n is some changing element of the loop:

* The loop counter / Index
* A loop variable which changes on each iteration

To prove a loop invariant you to prove 3 things:



**Example:** Pre-condition: m nonnegative integer, x is a real number, i = 0, and exp = 1.  
**Program:**   
While (I ≠ m)

1. Exp = exp\*x
2. I = I +1

End while

Post-condition: exp = xm

Loop-invariant: S(n) is i = n and exp = xn

**Proof:**

**Basis property**:   
n = 0, then i = 0, exp = x0 = 1, cause exp started on 1.

**Inductive property:**  
I ≠ m and S(k), then:

By 1: expnew = expold \* x = xk \* x = xk +1

By 2: inew = iold + 1 = k + 1

**Eventaul falsity of the Guard:**   
At each iteration, i = i + 1, and i = 0 at the start, so after m iterations we have i = m.

Correctness of the Post-Condition: Guard false implies i = m after m iterations and S(m) = xm

### Solving Recurrences (using, substitution, recursion-tree, and Master methods)

### max/min-heap property

### Hash functions and their properties

### Binary search tree property

### Red-black-tree property

### Optimal Substructure Property

### Graph of dependencies

### Graphs, cycles on graphs, negative cycles, directed acyclic graphs

### Topological ordering on an acyclic graph

### shortest paths problems on graphs and weighted graphs

## Algorithms

### Insertion Sort

### Merge Sort

### Heapsort

### Quicksort

### Counting Sort

### Radix Sort

### Breadth-First Search

### Depth-first Search

### Topological Sorting

### Computing Strongly Connected Components

### Bellman-Ford Algorithm

### Dijkstra’s algorithm

### Floyd-Warshall algorithm

### Transitive closure of a directed graph

## Data Structures

### Max-Heaps & Min-Heaps

### Stacks

### Queues

### Linked Lists (both doubly-linked and singly-linked)

### Rooted Trees

### Hash Tables (direct-address, open addressing)

### Binary Search Trees

### Red-Black Trees

### Directed and Undirected Graphs (adjacency-list & adjacency-matrix representations)

### Weighted Graphs (adjacency-list & adjacency-matrix representations)

## Algorithm Design

### Divide and Conquer

### Dynamic Programming