Chapter 2- Analysis of Algorithms

2021

Prepared by: Beimnet G.

Introduction

- Analyzing an algorithm has come to mean predicting the resources that the algorithm requires.
- Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure.
- By analyzing several possible correct algorithms, we can identify the most efficient one.

Introduction

- If computers were infinitely fast and computer memory was free, would you have any reason to study algorithms?
- At the very least you'd still need to demonstrate that your solution method terminates and does so with the correct answer.
- Computers may be fast, but they are not infinitely fast. And memory may be inexpensive, but it is not free. Computing time is therefore a bounded resource, and so is space in memory.

Introduction

- If computers were infinitely fast and computer memory was free, would you have any reason to study algorithms?
- At the very least you'd still need to demonstrate that your solution method terminates and does so with the correct answer.
- Computers may be fast, but they are not infinitely fast. And memory may be inexpensive, but it is not free. Computing time is therefore a bounded resource, and so is space in memory.
- For this course, we will assume the resource that we are most concerned with is computational time. All further analysis will be in relation to time.

Model of Computation

- Before we start analyzing algorithms, we should set a model of implementation to set a standard for evaluation.
- Cost models:
  - Uniform cost model- every machine operation has a constant cost, regardless of the size of the numbers involved.
  - Logarithms cost model- the cost of an operation is dependent on the number of bits involved.
- In this course, to model the technology implementation, we will use the **RAM model**.
- Other models: external memory model, cache oblivious mode ...

Pseudocode Conventions

- Indentation indicates block structure.
- The looping and control statements for, while, if-else have similar interpretation to their usage in C, C++, Java …
- Variables are assumed to be local unless stated otherwise
- Comments are specified by using “//”

RAM Model

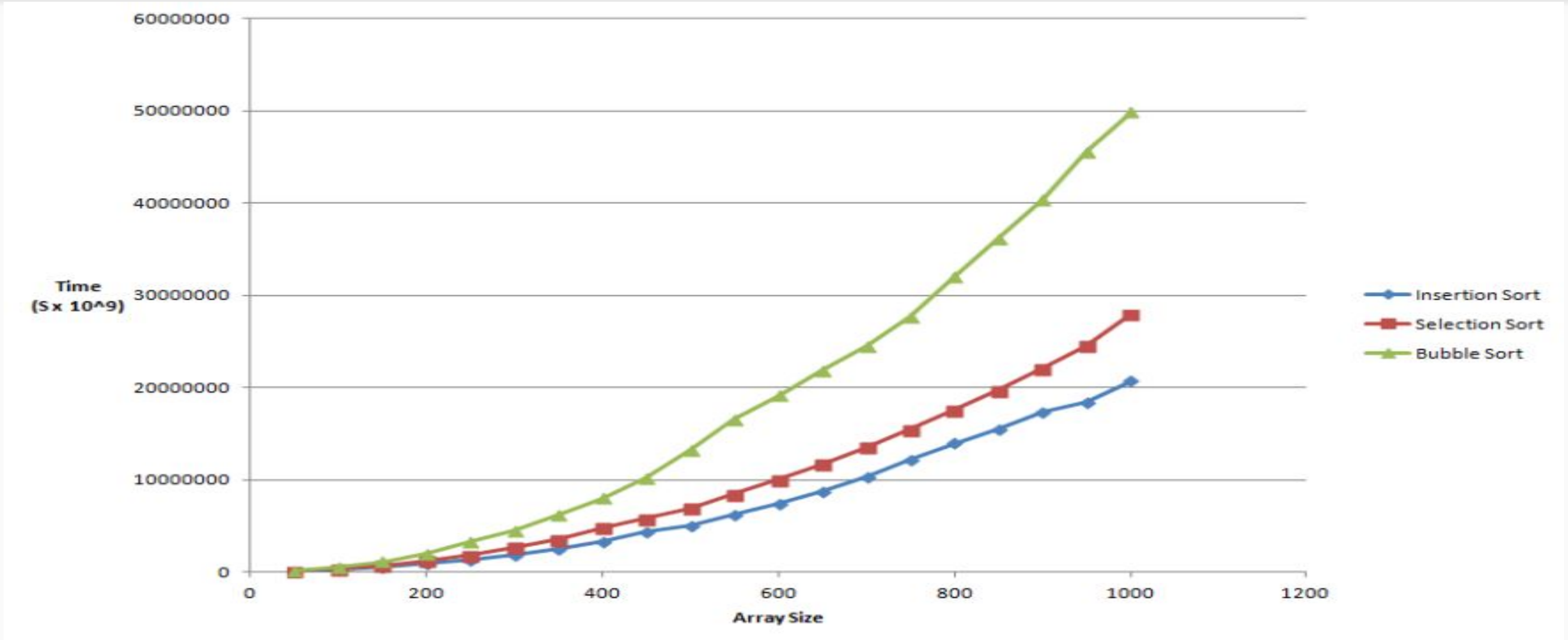
- This model defines some basic assumptions we're going to make about how our algorithms get executed.
- In the RAM model, instructions are executed one after the other.
- Assume operations (instructions) in real machines are available and each instruction takes a constant amount of time
- Example: data movement, addition, multiplication, shift, control instructions….
- Don't abuse this model by assuming unrealistic operations.
- The RAM model assumes no memory hierarchy.

Analysis

- Let's design a sorting algorithm.
- Problem Statement :

  Input: A sequence of n numbers {a1, a2, a3...an}
  Output: A permutation (reordering) {ai, aii, aiii, ..., am} of the input sequence such that {ai ≤ aii ≤ aiii ≤ ... ≤ am}
- There are a number of known sorting algorithms: Insertion sort, selection sort, bubble sort, merge sort...

Sorting Algorithms



Time Complexity

- The time it takes to execute an algorithm.
- The sum of the time it take to execute every operation.
- Elementary operations are assumed to take a single unit of time, these operations are summed to give the total time it takes to run an algorithm.
- The running time for an algorithm is dependent on the input that is provided.
- As the input size grows larger and larger so might the run time of an algorithm.

Time Complexity

- The running time of an algorithm on a particular input is the number of primitive operations or “steps” executed.
- The analysis is supposed to be as machine-independent as possible.
- Assume a constant amount of time is required to execute each line in the algorithm.
- One line may take a different amount of time than another line, but we assume that each execution of the ith line takes time ci, where ci is a constant.

Sorting Algorithm: Insertion Sort

```
INSERTION-SORT(A)
    for j=2 to A.length
        key= A[j]
        // Insert A[j]  into the sorted sequence A[1….j-1]
        i= j-1
        while i>0 and A[i] > key
            A[i+1]=A[i]
            i=i-1
        A[i+1]=key
```

Sorting Algorithm: Insertion Sort

- Given the previous algorithm to sort an array, let us analyze the time complexity of insertion sort.
- The time taken to execute the insertion sort depends on:
  - Input size
  - Input sort status: how sorted is the input array already?
- The run time of a program is represented as a function of the input size.
- What exactly we mean by input size depends on the problem area under study. It could be the number of items in the input, the total number of bits needed to represent the input or two numbers that describe the input rather than one.

Insertion Sort: Running Time

	INSERTION-SORT(A)	cost	times
1	for j=2 to A.length	C1	n
2	key= A[j]	C2	n-1
3	// Insert A[j] into the sorted sequence A[1….j-1]	C3=0	~~n-1~~
4	i= j-1	C4	n-1
5	while i>0 and A[i] > key	C5	$\Sigma^{n}_{i=2} t_j$
6	A[i+1]=A[i]	C6	$\Sigma^{n}_{i=2}(t_j-1)$
7	i=i-1	C7	$\Sigma^{n}_{i=2}(t_j-1)$
8	A[i+1]=key	C8	n-1

Insertion Sort: Running Time

$T(n) = C1n + C2(n - 1) + C4(n - 1) + C5 \Sigma^{n}_{i=2} t_j + C6 \Sigma^{n}_{i=2}(tj - 1) + C7 \Sigma^{n}_{i=2}(tj - 1) + C8(n - 1)$

Insertion Sort: Running Time

Best Case: array is already sorted

T(n)=C1n+C2(n-1)+C4(n-1)+C5(n-1)+C8(n-1)

T(n) = (C1 + C2 + C4 + C5 + C8)n − (C2 + C4 + C5 + C8)

Worst Case: array is in a reverse order

$$T(n) = c1n + c2(n-1) + c4(n-1) + c5\left(\frac{n(n+1)}{2} - 1\right) + c6\left(\frac{n(n-1)}{2}\right) + c7\left(\frac{n(n-1)}{2}\right) + c8(n-1)$$

T(n) = (C5/2 + C6/2 + C7/2)n^2 + (C1 + C2 + C4 + C5/2 − C6/2 + C7/2 + C8)n − (C2 + C4 + C5 + C8)

Insertion Sort: Order of Growth

These formulas could be simplified to represent an abstraction of the running time. This is done by ignoring any constants.

For further simplification, we only consider the **rate of growth** or **order of growth**. i.e. how is the runtime affected as the input size gets larger and larger?

This would be the leading term in the running time equation. The leading term is chosen because as the input size gets very large, the other terms quickly become insignificant.

This expressions is known as the theta notation(Θ). (more on this later)

One algorithm is said to be more efficient than another if its worst case running time has a lower order of growth.

Insertion Sort: Order of Growth

Best Case: array is already sorted

$T(n) = (C1 + C2 + C4 + C5 + C8)n - (C2 + C4 + C5 + C8)$

$T(n) = \Theta(n)$

Worst Case: array is in a reverse order

$T(n) = (C5\ 2 + C6\ 2 + C7\ 2)n^2 + (C1 + C2 + C4 + C5\ 2 - C6\ 2 + C7\ 2 + C8)n - (C2 + C4 + C5 + C8)$

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

Best Case vs Worst Case vs Average Case

The worst case is the longest running time for an algorithm.

For the duration of this course, we will always use the worst case of a problem to analyze the time complexity of an algorithm. Why?

- The worst-case running time of an algorithm gives us an upper bound on the running time for any input.
- For some algorithms, the worst case occurs fairly often.
- The “average case” is often roughly as bad as the worst case. What we consider to be “average” is also hard to define. And it is also expensive to calculate.

Sorting Algorithm: Bubble Sort

```
BUBBLE-SORT(A)
    for i=n downto 2
        for j=1 to i-1
            if A[j]> A[j+1]
                temp= A[j+1]
                A[j+1]=A[j]
                A[j]=temp
```

Bubble Sort: Running Time

```
    BUBBLE-SORT(A)                         cost   time
1       for i=n downto 2                   c1     n
2           for j=1 to i-1                 c2     $\sum^{n}_{i=2} i$
3               if A[j]> A[j+1]            c3     $\sum^{n}_{i=2} i-1$
4                   temp= A[j+1]           c4     $\sum^{n}_{i=2} t_i$
5                   A[j+1]=A[j]            c5     $\sum^{n}_{i=2} t_i$
6                   A[j]=temp              c6     $\sum^{n}_{i=2} t_i$
```

Correctness of an Algorithm

- After designing algorithms, one would be interested in:
    1. **Proving its correctness**
    2. Proving its efficiency
- Proving the correctness means that showing that the algorithm halts and halts with correct answer for all valid inputs.
- The primary technique we will use for proving correctness of an algorithm is mathematical induction; which is used to make statements about well-ordered sets.

Correctness of an Algorithm

- When trying to prove a given statement for a set of natural numbers, the first step, known as the **base case**, is to prove the given statement for the first natural number.
- The second step, known as the **inductive step**, is to prove that, if the statement is assumed to be true for any one natural number, then it must be true for the next natural number as well.
- Having proved these two steps, the rule of inference establishes the statement to be true for all natural numbers.
- In addition to these two steps in the conventional mathematical induction, while proving correctness of algorithm we add a third step to check the **termination** of the algorithm.

Correctness of an Algorithm: Example

```
max(a,b)
    If a>= b
        max=a
    Else   max=b
    Return max
```

```
max(A)
    max=A[0]
    for i=1 to A.length-1
        if A[i] >max
            max=A[i]
    return max
```

Correctness of an Algorithm: Loop Invariant

- A loop invariant is a property of a program loop that is true before (and after) each iteration.
- Knowing the loop invariant(s) is essential in understanding the effect of a loop.
- It is what we will use to prove in the base case and the inductive step.

Correctness Proof

To show that an algorithm is correct, we must proof 3 things about the loop invariant.

- **Initialization**: It is true prior to the first iteration of the loop.
- **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination**: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Correctness Proof: Example

```
max(A)
    max=A[0]
    for i=1 to A.length-1
        if A[i] >max
            max=A[i]
    return max
```

Loop Invariant

after the ith iteration max will contain the maximum value from the subarray A[0:i]

Correctness Proof: Example

```
max(A)
    max=A[0]
    for i=1 to A.length-1
        if A[i] >max
            max=A[i]
    return max
```

Initialization

When i=1, max will contain the maximum value of the subarray A[0:1] => A[0] => trivially true

Correctness Proof: Example

```
max(A)
    max=A[0]
    for i=1 to A.length-1
        if A[i] >max
            max=A[i]
    return max
```

Maintenance

After the ith iteration max will contain the maximum value from the subarray A[0:i]; *proof for i+1*

Case 1: A[i]> max; max will be replaced with the new maximum value thus, max is the maximum value in the subarray A[0:i+1]

Case 2: A[i]<= max; max will retain its previous value, max is the maximum value in the subarray A[0:i+1]

Correctness Proof: Example

```
max(A)
    max=A[0]
    for i=1 to A.length-1
        if A[i] >max
            max=A[i]
    return max
```

Termination

i=n

When i=1, max will contain the maximum value of the subarray A[0:n] => A[0],A[1],A[2]....A[n-1] => the whole array.

Finding a Loop Invariant

- A loop invariant often makes a statement depending on i and about the data seen so far.
- Ask: what do you want to know at the end?
- What do you know? What information do you gain after each iteration?

Checklist for your loop invariant

1. Have you stated your loop invariant explicitly when beginning?
2. Does you loop variable occur in your loop invariant statement ?
3. Does the loop invariant hold before the first iteration of the loop?
4. Is your invariant strong enough to conclude the right answer?
5. If you have multiple loops, is it clear for which loop you have defined the invariant?
6. Did you use the loop invariant in the maintenance and termination step?
7. Does your argument line up with what the algorithm is trying to do?

Correctness Proof: Example- Linear Search

```
improved-linear-search(A)
    for i=0 to A.length-1
        if A[i]=x
            return i
    return -1
```

Loop Invariant

at the ith iteration, if x is present in the array, it is present in the subarray A[i:n]

or

at the ith iteration, x is not present in the subarray A[0:i]

Correctness Proof: Example- Linear Search

```
improved-linear-search(A)
    for i=0 to A.length-1
        if A[i]=x
            return i
    return -1
```

Initialization

When i=0, if x is present in the array it is present in the subarray A[0:n]=> the whole array

Correctness Proof: Example- Linear Search

```
improved-linear-search(A)
    for i=0 to A.length-1
        if A[i]=x
            return i
    return -1
```

Maintenance

At the ith iteration, if x is in A, then it is present in the subarray A[i:n]

If A[i]≠ x, if x in in A it is present in the subarray A[i+1:n]

Correctness Proof: Example- Linear Search

```
improved-linear-search(A)
    for i=0 to A.length-1
        if A[i]=x
            return i
    return -1
```

Termination

Case 1: A[i]=x, x is in A[i:n]

Case 2: i=n

When i=n

If x is present in A it is present in A[n:n] => empty => therefore x is not in A

Correctness Proof: Exercise - Linear Search

```
improved-linear-search(A,x)
    for i=0 to A.length-1
        if A[i]=x
            return i
    return -1

linear-search(A)
    answer=-1
    for i=0 to A.length-1
        if A[i]=x
            answer=i
    return answer
```

Exercises

1. Use the alternative loop invariant stated earlier to prove the correctness of the algorithm
2. Modify this algorithm to a traditional linear search algorithm (one that stores the index of the matched value instead of returning right away) and prove the correctness of that algorithm.

Correctness Proof: Exercise - Insertion Sort

```
INSERTION-SORT(A)
    for j=2 to A.length
        key= A[j]
        i= j-1
        while i>0 and A[i] > key
            A[i+1]=A[i]
            i=i-1
        A[i+1]=key
```

Loop Invariant

At the jth iteration the subarray A[1:j] is sorted in a non-decreasing order.

Correctness Proof: Exercise - Insertion Sort

```
INSERTION-SORT(A)
    for j=2 to A.length
        key= A[j]
        i= j-1
        while i>0 and A[i] > key
            A[i+1]=A[i]
            i=i-1
        A[i+1]=key
```

Initialization

j=2

When j=2, A[1:2] is already sorted.

A[1:2] => A[1] => sorted relative to itself

Correctness Proof: Exercise - Insertion Sort

```
INSERTION-SORT(A)
    for j=2 to A.length
        key= A[j]
        i= j-1
        while i>0 and A[i] > key
            A[i+1]=A[i]
            i=i-1
        A[i+1]=key
```

Maintenance

At the jth iteration the subarray A[1:j] is sorted in a non-decreasing order.

For the jth iteration, A[j] is placed in its correct position in the subarray A[1:j+1]. Thus, the subarray A[1:j+1] is relatively sorted.

Correctness Proof: Exercise - Insertion Sort

```
INSERTION-SORT(A)
    for j=2 to A.length
        key= A[j]
        i= j-1
        while i>0 and A[i] > key
            A[i+1]=A[i]
            i=i-1
        A[i+1]=key
```

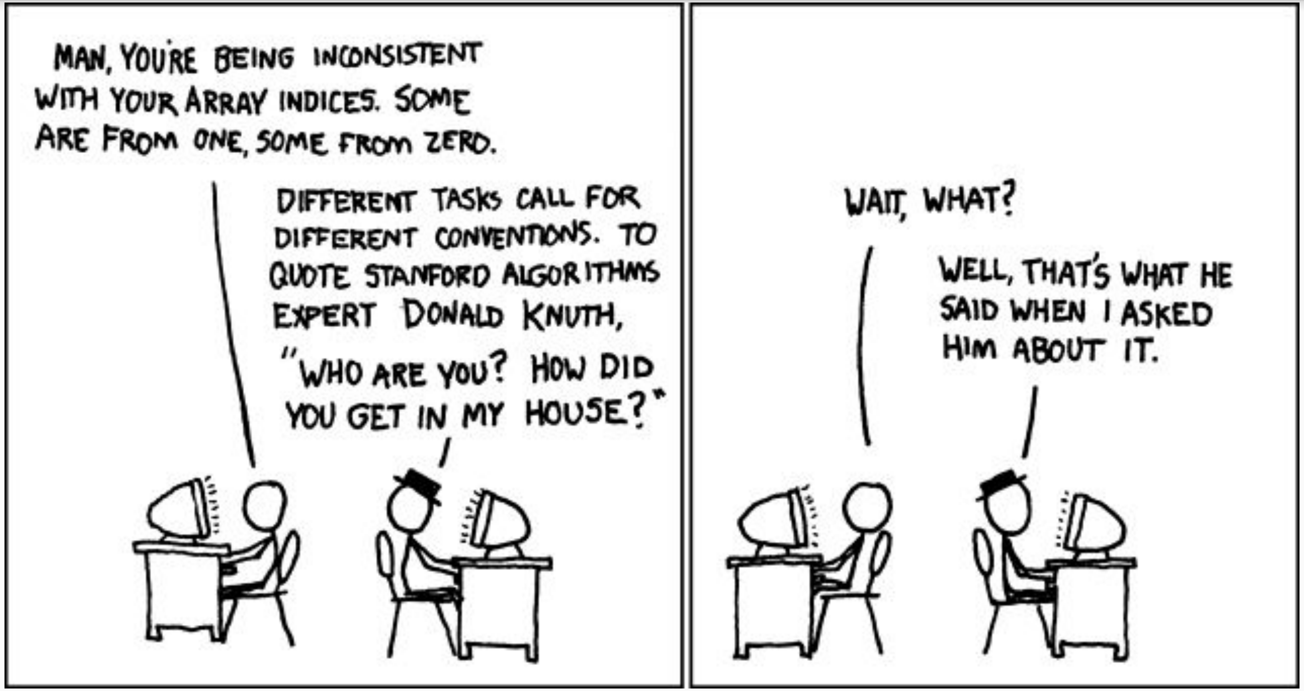
Termination

j=n+1

A[1:n] is already sorted => A[1], A[2], A[3]... A[n]

=> the whole array

arr[0]!=arr[1]: which is it?


MAN, YOU'RE BEING INCONSISTENT WITH YOUR ARRAY INDICES. SOME ARE FROM ONE, SOME FROM ZERO.
DIFFERENT TASKS CALL FOR DIFFERENT CONVENTIONS. TO QUOTE STANFORD ALGORITHMS EXPERT DONALD KNUTH, "WHO ARE YOU? HOW DID YOU GET IN MY HOUSE?"
WAIT, WHAT?
WELL, THAT'S WHAT HE SAID WHEN I ASKED HIM ABOUT IT.

Growth of Functions

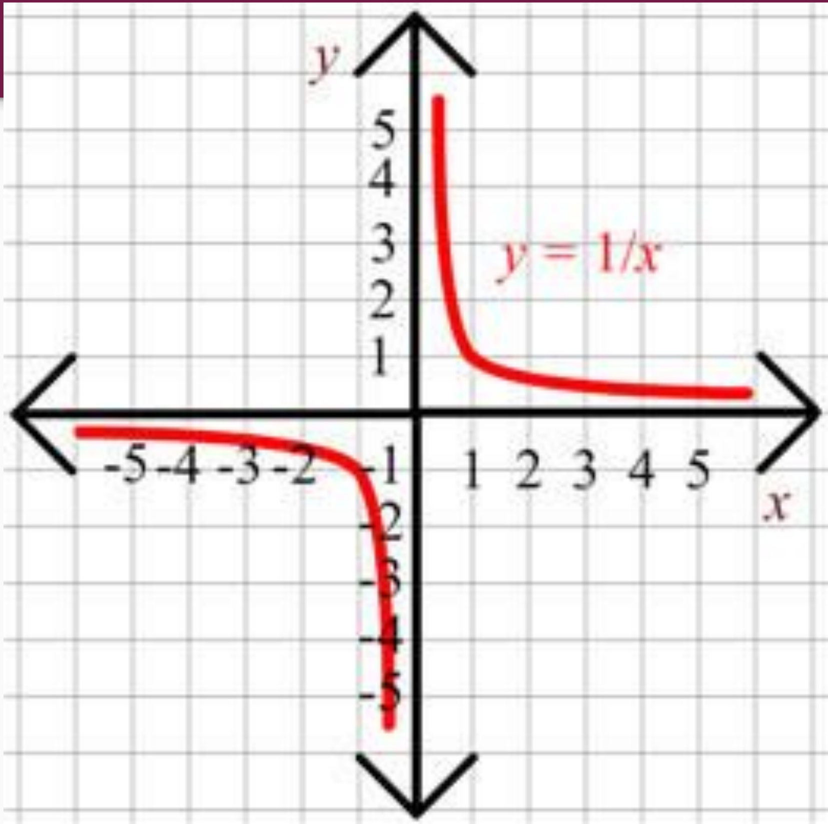
- Although we can sometimes determine the exact running time of an algorithm the extra precision is not usually worth the effort of computing it. We are instead interested in studying how the running time changes as the input gets very large.
- For large enough inputs, the effects of the input size itself dominate the multiplicative constants and lower-order terms of an exact running time. The order of growth is indicated by the leading term in the running time equation.
- When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the **asymptotic efficiency**.
- That is, we are concerned with how the running time of an algorithm increases with the size of the input as it increases without bound.

Asymptotic Notations

- Standard notations for expressing the asymptotic analysis of algorithms.
- A family of notations that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.
- In this course we will use these notations to describe the time complexity of algorithms. However these notations could be used to characterize some other aspects of algorithms or even functions that have nothing to do with algorithms.

Asymptotic Behaviour

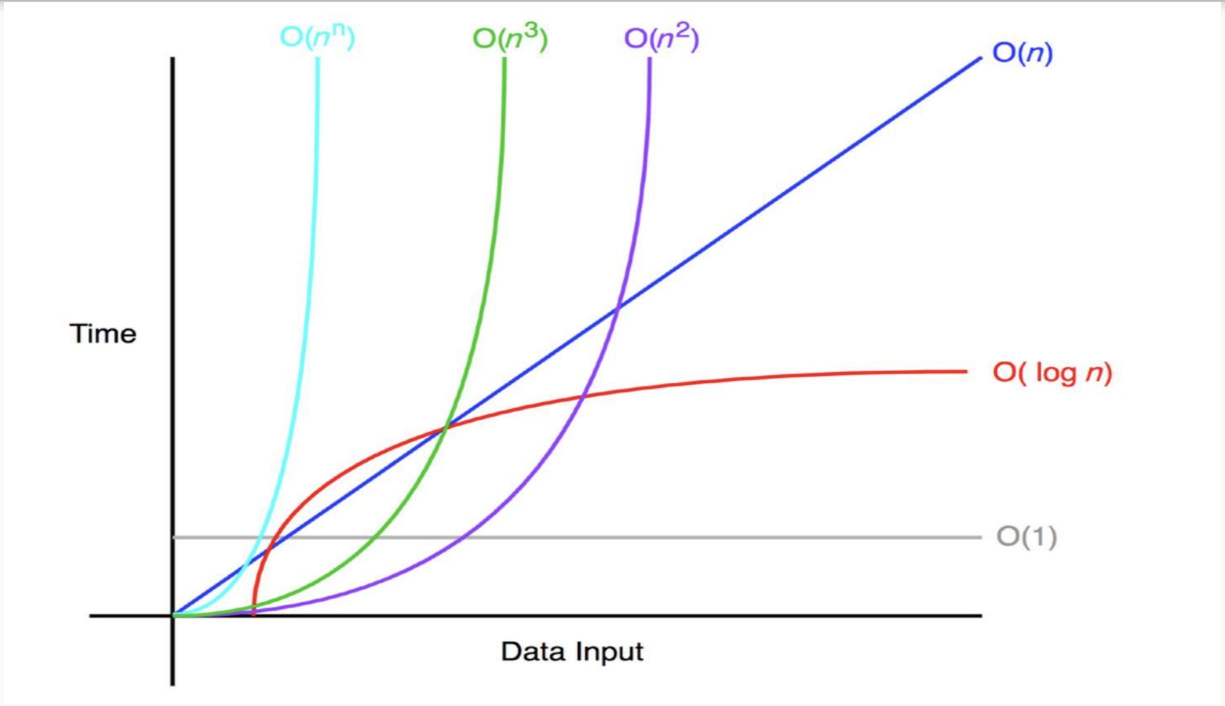
- Point of interest: tail behaviour of the graph.
- What is $\lim_{x \to \infty} \frac{1}{x}$
- How does the graph behave s x gets to very large values?
- Why the need for asymptotic analysis? Why not just measure the elapsed real time it took to run your algorithm?



Asymptotic Bounding

- Give a general characterization of different algorithms; a general description of how the algorithm performs as input grows large.
- There are several classes of bounding functions: constant time, linear time, logarithmic time, quadratic, polynomial, exponential …
- Start with T(n)- a function describing the time an algorithm takes.
- As n changes the value of T(n) changes as well.
- We want to bound this change. I.e. predict or bound the possibility of how this function changes as n changes (gets very large).

Graph Behaviours



Asymptotic Notations

- Big Oh (O) Notation
- Big Omega(Ω) Notation
- Theta (Θ) Notation
- Small Oh (o) Notation
- Small Omega (ω) Notation

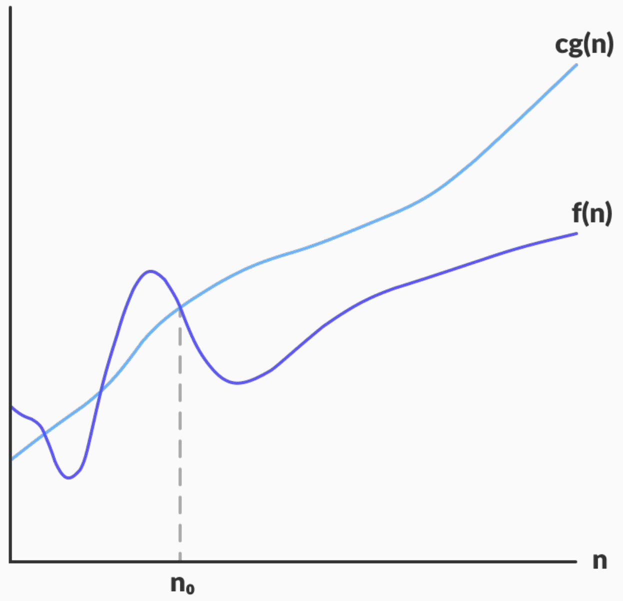
Boundes: Upper Bound

- Big-Oh (O)
- Bounds a function from above.
- Mathematically:

  $f(n) = O(g(n))$ iff

  for every $n \geq n_o$ where $n_o \geq 1$ and $c>0$

  $f(n) \leq c\, g(n)$



$f(n) = O(g(n))$

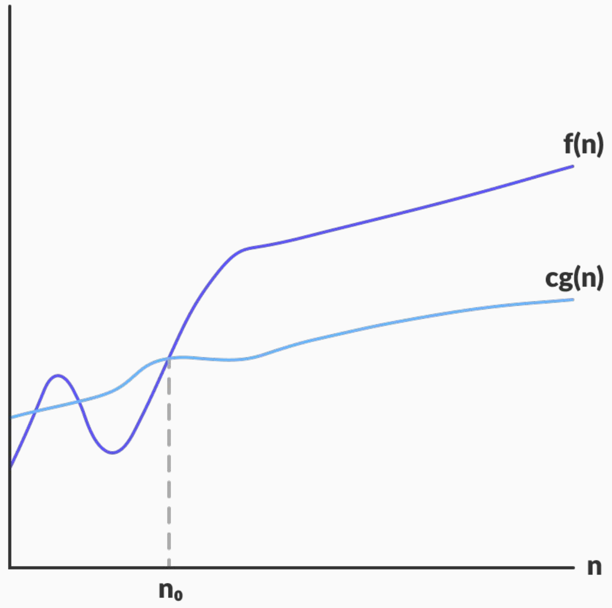
Boundes: Lower Bound

- Big-Omega (Ω)
- Bounds a function from below.
- Mathematically:

  $f(n) = \Omega(g(n))$ iff

  for every $n \geq n_o$ where $n_o \geq 1$ and $c>0$

  $f(n) \geq c\, g(n)$



f(n) = Ω(g(n))

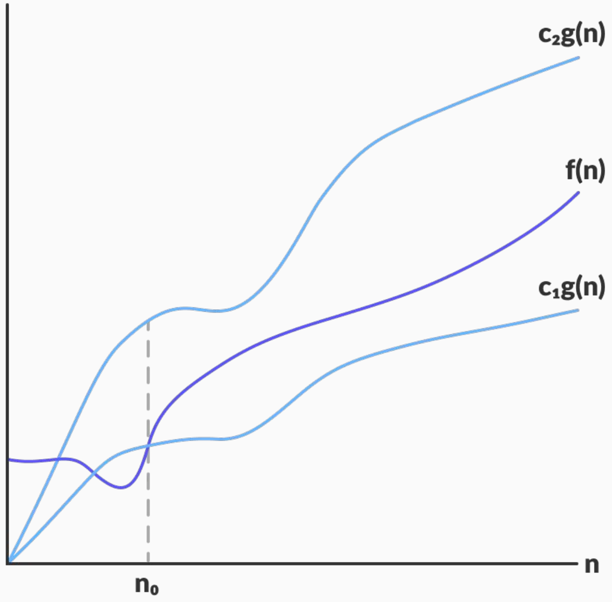
Boundes: Exact Bound

- Theta (Θ)
- Bounds a function from both above and below.
- Mathematically:

  $f(n) = \Theta(g(n))$ iff

  for every $n \geq n_o$ where $n_o \geq 1$ and $c1,c2>0$

  $C1\ g(n) \leq f(n) \leq c2\ g(n)$



$f(n) = \Theta(g(n))$