Asymptotic Analysis

Apurba Sarkar

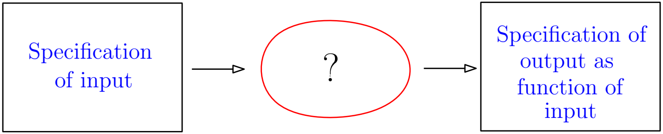
IIEST Shibpur

July 27, 2018

Data Structures and Algorithms

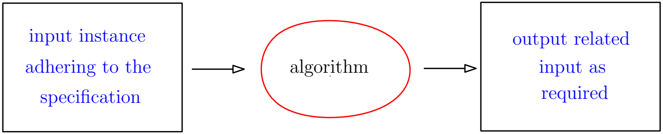
- Algorithm: Outline, the essence of a computational procedure, step-by-step instructions
- Program: an implementation of an algorithm in some programming language
- Data structure: Organization of data needed to solve the problem

Algorithmic problem



- Infinite number of input instances satisfying the specification. For eg: A sorted, non-decreasing sequence of natural numbers of non-zero, finite length
  - $1, 20, 908, 909, 100000, 1000000000$.
  - 3.

Algorithmic Solution



- Algorithm describes actions on the input instance
- Infinitely many correct algorithms for the same algorithmic problem

What is a Good Algorithm?

- Efficient:
  - Running time
  - Space used
- Efficiency as a function of input size:
  - The number of bits in an input number
  - Number of data elements (numbers, points)

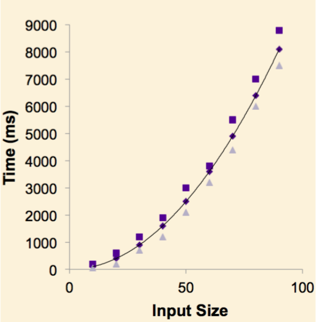
Measuring the Running time/efficiency

Two approaches

- Experimental study.
- Formal/Theoretical Analysis

Experimental study

- Write a program implementing the algorithm.
- Run the program with inputs of varying size and composition.
- Use a method like `System.currentTimeMillis()` to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis

Needs a general methodology for for analyzing running time of algorithms. This approach

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Pseudo Code

- A mixture of natural language and high-level programming concepts that describes the main ideas behind a generic implementation of a data structure or algorithm
- Eg:

Algorithm arrayMax(A, n)

Input: An array A storing n integers.
Output: The maximum element in A.

currentMax $\leftarrow$ A[0]
for $i \leftarrow 1$ to $n - 1$ do
    if currentMax $<$ A[i] then currentMax $\leftarrow$ A[i]
return currentMax

Pseudo Code

It is more structured but less formal than a programming language

- Expression
  - use standard mathematical symbols to describe numeric and boolean expressions
  - use $\leftarrow$ for assignment (" $=$ "in java)
  - use $=$ for the equality relationship (" $==$ " in Java)
- Method Declarations:
  - Algorithm name(param1, param2)

Pseudo Code

- Programming Constructs:
  - decision structures $if \ldots then \ldots [else] \ldots$
  - while-loops: $while \ldots do$
  - repeat-loops: $repeat \ldots until \ldots$
  - for-loop: $for \ldots do$
  - array indexing: $A[i], A[i, j]$
- Methods:
  - calls: object method(args)
  - returns: return value

Analysis of Algorithm

- Primitive Operation: Low-level operation independent of programming language. Can be identified in pseudo-code. For e.g.:
  - Data movement (assign)
  - Control (branch, subroutine call, return)
  - Arithmetic and and logical operations (e.g. addition, comparison)
- By inspecting the pseudo-code, we can count the number of primitive operations executed by an algorithm.

Primitive Operation

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Assumed to take a constant amount of time

Primitive Operation

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Assumed to take a constant amount of time

Examples

- Evaluating an expression
- Assigning a value to a variable
- Indexing into an array
- Calling a method
- Returning from a method

Example 1: ArrayMax

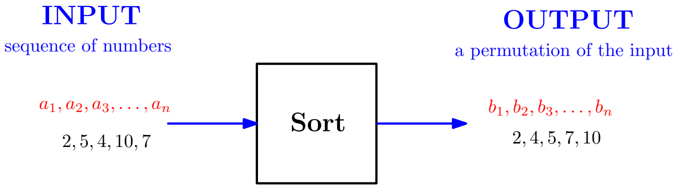
- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Algorithm arrayMax(A, n)	
Input: An array A storing n integers. Output: The maximum element in A.	# operations
currentMax $\leftarrow$ A[0]	2
for $i \leftarrow 1$ to $n-1$ do	$2n$
if currentMax $<$ A[i] then	$2(n-1)$
currentMax $\leftarrow$ A[i]	$2(n-1)$
return currentMax	1
	Total $6(n-1)$

Estimating Running Time

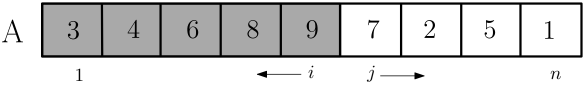
- Algorithm arrayMax executes $6(n-1)$ primitive operations in the worst case. Define:
  $a =$ Time taken by the fastest primitive operation $b =$ Time taken by the slowest primitive operation
- Let $T(n)$ be worst-case time of arrayMax. Then
  $a(6n-1) \leq T(n) \leq b(6n-1)$
- Hence, the running time $T(n)$ is bounded by two linear functions

Example 2: Sorting



- Correctness: For any given input the algorithm halts with the output
  - $b_1 < b_2 < b_3 < \ldots b_n$
  - $b_1 < b_2 < b_3 < \ldots b_n$ should be a permutation of $a_1 < a_2 < a_3 < \ldots a_n$
- Running Time: depends on the number of elements (n)
- how (partially) sorted they are
- also depend upon what particular algorithm is used.

Insertion Sort



Strategy

- Start empty handed
- Insert a card in the right position of the already sorted cards
- Continue until all cards are inserted/sorted

```
INPUT: A[1 : n] -an array of integers
OUTPUT: a permutation of A such that
A[1]≤ A[2] ≤ ... ≤ A[n]
for j ← 2 to n do
      key ← A[j]
      insert A[j] into the
      sorted sequence A[1, j − 1]
      i ← j − 1
      while i > 0 and A[i] > key
           do A[i + 1]← A[i]
              i − −

     A[i + 1]← key
```

Analysis of Insertion Sort

	cost	times
for $j \leftarrow 2$ to n do	C_1	n
key $\leftarrow$ A[j]	C_2	$n-1$
insert A[j] into the sorted sequence A[$1, j-1$]	0	$n-1$
$i \leftarrow j-1$	C_3	$n-1$
while $i > 0$ and A[i] $>$ key	C_4	$\sum_{j=2}^{n} t_j$
do A[$i+1$] $\leftarrow$ A[i]	C_5	$\sum_{j=2}^{n}(t_j - 1)$
$i--$	C_6	$\sum_{j=2}^{n}(t_j - 1)$
A[$i+1$] $\leftarrow$ key	C_7	$n-1$

Total time $= n(C_1 + C_2 + C_3 + C_7) + \sum_{j=2}^{n} t_j(C_4 + C_5 + C_6)$
$-(C_2 + C_3 + C_5 + C_6 + C_7)$

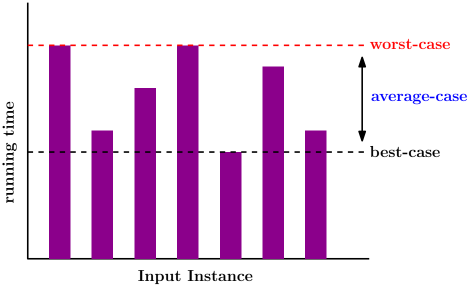
Best/Worst/Average Case

Total time$= n(C_1 + C_2 + C_3 + C_7) + \sum_{j=2}^{n} t_j(C_4 + C_5 + C_6)$
$-(C_2 + C_3 + C_5 + C_6 + C_7)$

- Best case: Elements are already sorted; $t_j = 1$, running time $= f(n)$ i.e. linear time.
- Worst case: Elements are sorted in reverse order; $t_j = j$, running time $= f(n^2)$ i.e. quadratic time.
- Average case: $t_j = j/2$, running time $= f(n^2)$ i.e. quadratic time.

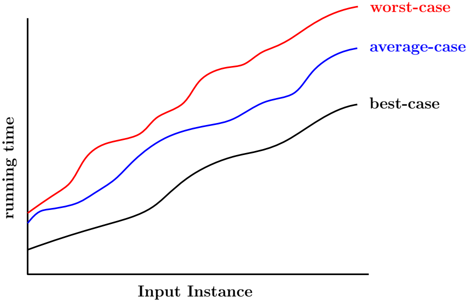
Best/Worst/Average Case

- For a specific size of input n, investigate running times for different input instances:



Best/Worst/Average Case

- For inputs of all size



Best/Worst/Average Case

By running time $T(n)$ of an algorithm we mean $T(n)$ is the upper bound/worst case

- The algorithm may very well take less time on some inputs of size n, but it doesn't matter.
- If an algorithm takes $T(n) = c * n^2 + k$ steps on only a single input of size n and only n steps on the rest, we still say that it is a quadratic algorithm.

Why Worst Case?

- It gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer.
- We need not make some educated guess about the running time and hope that it never gets much worse.
- For some algorithms, the worst case occurs fairly often. For example, in searching a database.
- The “average case” is often roughly as bad as the worst case.
- Finding average case can be very difficult.

Average Case

- An alternative to worst-case analysis is average-case analysis.
- Here we do not bound the worst case running time, but try to calculate the expected time spent on a randomly chosen input.
- Analysis is harder, since it involves probabilistic arguments and often requires assumptions about the distribution of inputs that may be difficult to justify.

Why Average Case?

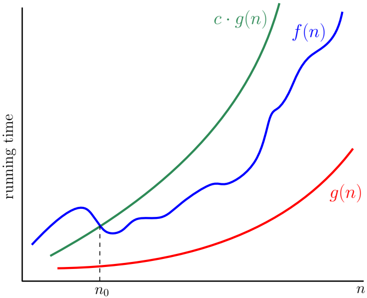
- Average-case analysis is useful because sometimes the worst-case behaviour of an algorithm is misleadingly bad.
- A good example of this is the popular quicksort algorithm, whose worst-case running time on an input sequence of length n is proportional to n^2 but whose expected running time is proportional to $n \lg n$.

Asymptotic Analysis

- Goal: to simplify analysis of running time by getting rid of "details", which may be affected by specific implementation and hardware
- Capturing the essence: how the running time of an algorithm increases with the size of the input in the limit.
  - Asymptotically more efficient algorithms are best for all but small inputs

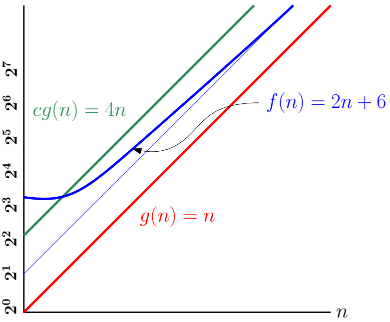
Asymptotic Notation

- The "big-Oh" O-Notation
  - asymptotic upper bound
  - $f(n)$ is $O(g(n))$, if there exist constants c and n_0, s.t. $f(n) \leq c \cdot g(n)$ for $n \geq n_0$
  - $f(n)$ and $g(n)$ are functions over non- negative integers
- Used for worst-case analysis



Example

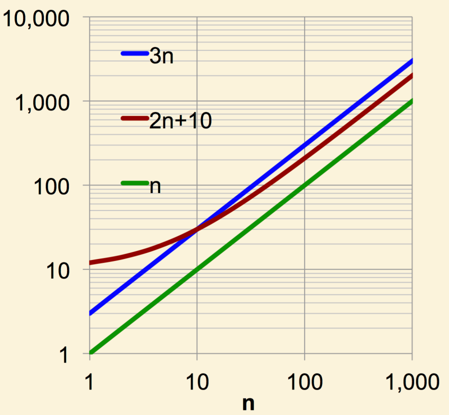
For functions $f(n)$ and $g(n)$ there are positive constants c and n_0 such that: $f(n) \leq cg(n)$ for $n \geq n_0$



conclusion:
$2n + 6$ is $O(n)$.

Another Example

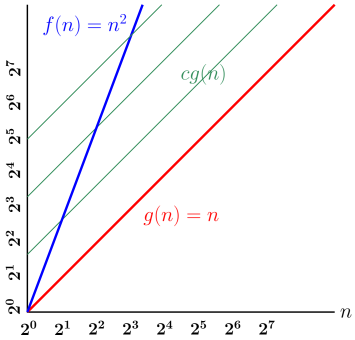
$2n + 10$ is $O(n)$
$2n + 10 \leq cn$
$(c - 2)n \geq 10$
$n \geq 10/(c - 2)$
Pick $c = 3$ and $n_0 = 10$



yet another Example

n^2 is not $O(n)$ because there is no c and n_0 such that: $n^2 \leq c \cdot n$ for $n \geq n_0$

The graph bellow illustrates that no matter how large a c is chosen there is an n big enough such that $n^2 > c \cdot n$



Asymptotic Notation

- Simple Rule: Drop lower order terms and constant factors.
  - $50n \lg n$ is $O(n \lg n)$
  - $7n - 3$ is $O(n)$
  - $8n^2 \lg n + 5n^2 + n$ is $O(n^2 \lg n)$
- Note: We generally specify the tightest bound possible
  - Say $2n$ is $O(n)$ instead of $2n$ is $O(n^2)$
  - Similarly, even though $(50n \lg n)$ is $O(n^5)$, it is expected that such an approximation be of as small an order as possible.
- Use the simplest expression of the class
  - Say $3n + 5$ is $O(n)$ instead of $3n + 5$ is $O(3n)$

Asymptotic Analysis of Running Time

- Use O-notation to express number of primitive operations executed as function of input size.
- Comparing asymptotic running times
  - an algorithm that runs in $O(n)$ time is better than one that runs in $O(n^2)$ time
  - similarly, $O(\lg n)$ is better than $O(n)$ hierarchy of functions: $\lg n < n < n^2 < n^3 < 2^n$
- Caution! Beware of very large constant factors. An algorithm running in time $1,000,000n$ is still $O(n)$ but might be less efficient than one running in time $2n^2$, which is $O(n^2)$

Example of Asymptotic Analysis

Algorithm prefixAverages1(X):

INPUT: X[$1:n$] -An n-element array X of numbers.

OUTPUT: An n-element array A of numbers such that A[i] is the average of elements X[0], ... , X[i].

```
1.    for i ← 0 to n − 1 do
2.        a ← 0
3.        for j ← 0 to n − 1 do
4.            a ← a + X[j]
5.        A[i] ← a/(i + 1)
6.    return array A
```

Analysis:

Steps $2-4$ executes n times.

Step 4 executes i times for each i where $i = 0, 1, \ldots, n-1$

running time of prefixAverages1(X): $O(n^2)$

Example of Asymptotic Analysis

Algorithm prefixAverages2(X):

INPUT: X[$1:n$] -An n-element array X of numbers.
OUTPUT: An n-element array A of numbers such that A[i] is the average of elements X[0], ... , X[i].

1. $s \leftarrow 0$
2. for $i \leftarrow 0$ to n do
3. $\quad s \leftarrow s + X[i]$
4. $\quad A[i] \leftarrow s/(i+1)$
5. return array A

Analysis:

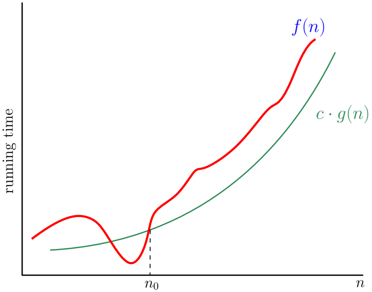
Steps $3-4$ executes $n+1$ times.
running time of prefixAverages2(X): $O(n)$

Common Terminology

- Few special classes of algorithms
  - Logarithmic: $O(\lg n)$
  - Linear: $O(n)$
  - Quadratic: $O(n^2)$
  - Polynomial: $O(n^k), k \geq 1$
  - Exponential: $O(a^n), a > 1$
- Other classes (relatives) of the Big-Oh
  - $\Omega(f(n))$: Big Omega -asymptotic lower bound
  - $\Theta(f(n))$: Big Theta -asymptotic tight bound

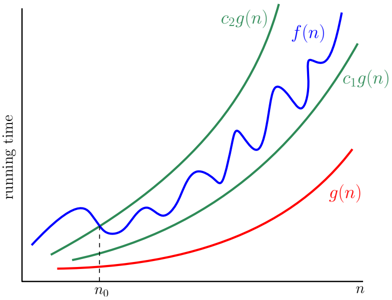
Asymptotic Notation

- The Big-Omega Ω-Notation
  - asymptotic lower bound
  - $f(n)$ is $\Omega(g(n))$, if there exist constants c and n_0, s.t. $c \cdot g(n) \leq f(n)$ for $n \geq n_0$
-
- Used to describe best-case running times or lower bounds for algorithmic problems
  - E.g., lower-bound for searching in an unsorted array is $\Omega(n)$.



Asymptotic Notation

- The big-Theta Θ-Notation
  - asymptotic tight bound
  - $f(n)$ is $\Theta(g(n))$, if there exist constants $c_1, c_2,$ and n_0, s.t. $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for $n \geq n_0$
- $f(n)$ is $\Theta(g(n))$ if and only if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$
- $f(n)$ is sandwiched between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$



Example

$3n^2 + 7n + 8 = \Theta(n^2)$
There should exist $c_, c_2, n_0 > 0$ s.t.
$c_1 n^2 \leq 3n^2 + 7n + 8 \leq c_2 n^2 \ \forall n \geq n_0$
let us pick $c_1 = 3$ and $c_2 = 4$
$3n^2 \leq 3n^2 + 7n + 8 \leq 4n^2$
let $n = 1$
$3 \cdot 1^2 \leq 3 \cdot 1^2 + 7 \cdot 1 + 8 \leq 4 \cdot 1^2$ which is false
let $n = 7$
$3 \cdot 7^2 \leq 3 \cdot 7^2 + 7 \cdot 7 + 8 \leq 4 \cdot 7^2$ which is false again
let $n = 8$
$3 \cdot 8^2 \leq 3 \cdot 8^2 + 7 \cdot 8 + 8 \leq 4 \cdot 8^2$ which is True
This is also True for $n = 9, 10, \ldots$
this is satisfied for $c_1 = 3, c_2 = 4,$ and $n_0 = 8$
Final Conclusion
$3n^2 + 7n + 8 = \Theta(n^2)$

Example

$3n^2 + 7n + 8 = O(n^2)$
$3n^2 + 7n + 8 \leq 3n^2 + 7n^2 + 8n^2$
This is true because if we compare term by term of both sides
we get $3n^2 \leq 3n^2;\;\; 7n \leq 7n^2;\; and\; 8 \leq 8n^2$
so we can write
$3n^2 + 7n + 8 \leq 18n^2$
so if we chose $c = 18$ and $n \geq n_0 = 1$ the inequality
$3n^2 + 7n + 8 \leq cn^2$ is satisfied, so we can conclude that
$3n^2 + 7n + 8 = O(n^2)$