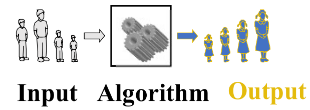
Analysis of Algorithms

Introduction



An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.

Time and space

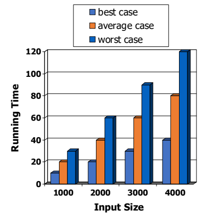
- To analyze an algorithm means:
  - Developing a formula for predicting *how fast* an algorithm is, based on the size of the input (time complexity), and/or
  - Developing a formula for predicting *how much memory* an algorithm requires, based on the size of the input (space complexity)
- Usually time is our biggest concern

What does "size of the input" mean?

- If we are searching an array, the "size" of the input could be the size of the array
- If we are merging two arrays, the "size" could be the sum of the two array sizes
- We choose the "size" to be the parameter that most influences the actual time/space required

Running Time of Algorithm

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications



Average vs worst cases

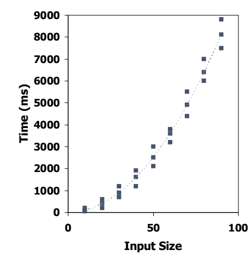
- Usually we would like to find the *average* time to perform an algorithm
- However,
  - Sometimes the “average” isn’t well defined
    - Example: Sorting an “average” array
      - Time typically depends on how out of order the array is
      - How out of order is the “average” unsorted array?
  - Sometimes finding the average is too difficult
- Often we have to be satisfied with finding the worst (longest) time required
  - Sometimes this is even what we want (say, for time-critical operations)
- The *best* (fastest) case is seldom of interest

Analyzing Time Efficiency of an Algorithm

- Two ways:
  - Experimental study
  - Theoretical analysis

Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use Python methods like datetime.now() and time() to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

- 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
- Must run on many data sets to see effects of scaling

Theoretical Analysis Time Efficiency

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

Theoretical Analysis of Time Efficiency

- Time *efficiency* is analyzed by determining the number of repetitions of the *primitive operations* as a function of *input size*
- *Primitive/basic operation*: the operation that contributes most towards the running time of the algorithm

Algorithms - Example

- Step-by-step procedure for solving a problem
- Less detailed than a program
- Hides program design issues

Example: find max element of an array

```
Algorithm arrayMax(A, n)
  Input array A of n integers
  Output maximum element of A

  currentMax ← A[0]
  for i ← 1 to n − 1 do
    if A[i] > currentMax then
      currentMax ← A[i]
  return currentMax
```

Algorithms - Components

- Flow Control
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
- Method declaration

  Algorithm *method* (*arg* [, *arg*…])
  Input …
  Output …
- Method call

  var.method (*arg* [, *arg*…])
- Return value

  return *expression*
- Expressions

  ← Assignment (like = in Java)

  = Equality testing (like == in Java)

  n^2 Superscripts and other mathematical formatting allowed

Primitive/Basic Operations

- Basic computations performed by an algorithm
- Identifiable in algorithm
- Largely independent from the programming language
- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning a value from a method

Counting Primitive Operations

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

Algorithm *arrayMax*(A, n)	# operations
currentMax ← $A[0]$	2
for i ← 1 **to** $n - 1$ **do**	$2(n-1)$
if $A[i]$ > *currentMax* **then**	$2(n-1)$
currentMax ← $A[i]$	$2(n-1)$
{ increment counter i }	$2(n-1)$
return *currentMax*	1
Total	$8n - 1$

Counting Operations

- Consider counting steps in FindMax
- Precise information may not be needed i.e precise details less relevant than order growth
- More interested in growth rates with respect to n ((i.e Big O))

Example: Sequential search

```
ALGORITHM SequentialSearch(A[0..n − 1], K)
    //Searches for a given value in a given array by sequential search
    //Input: An array A[0..n − 1] and a search key K
    //Output: The index of the first element of A that matches K
    //        or −1 if there are no matching elements
    i ← 0
    while i < n and A[i] ≠ K do
        i ← i + 1
    if i < n return i
    else return −1
```

- Worst case: Element being searched is the last one
- Best case: Element being searched is first one
- Average case: Not clear.

17

Critical Factor for Analysis: Growth Rate

- Most important: *Order of growth* as $n \to \infty$
  - What is the growth rate of time as input size increases?
  - How does time increase as input size increases?
- We are interested in asymptotic order of growth

18

Asymptotic Order of Growth

- Critical factor for problem size *n*:
  - Is NOT the exact number of basic ops executed for given *n*
  - It is how number of basic ops GROWS as *n* increases
- Constant factors and constants do not change growth RATE
- Rate most relevant for large input sizes, so ignore small sizes
- Example: 5*n*^2 and 100*n*^2 +1000 are both *n*^2
- Call this: Asymptotic Order of Growth -> how number of basic ops GROWS as *n* increases

19

Growth Rate of Running Time

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

- Focus: asymptotic order of growth:
  - Main concern: which function describes behavior.
  - Less concerned with constants

20

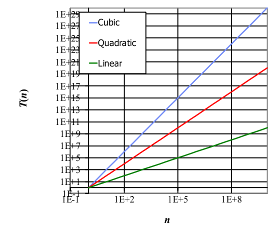
Growth Rate of Running Time

- The linear growth rate $(8n - 2)$ of the running time $T(n)$ is an intrinsic property of algorithm ***arrayMax***
- Changing the hardware/software environment:
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$

21

Seven Important Functions

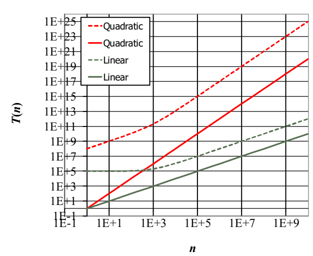
- Seven functions that often appear in algorithm analysis:
  - Constant ≈ *1*
  - Logarithmic ≈ log n
  - Linear ≈ n
  - N-Log-N ≈ n log n
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n
- These are the basic asymptotic efficiency classes



22

Constant Factors

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2n + 10^5$ is a linear function
  - $10^5n^2 + 10^8n$ is a quadratic function



23

Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

$f(n) \leq cg(n)$ for $n \geq n_0$

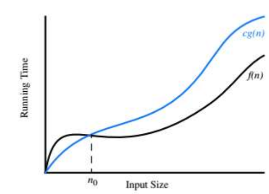
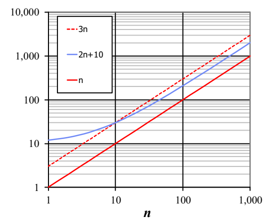


Figure 4.5: Illustrating the "big-Oh" notation. The function $f(n)$ is $O(g(n))$, since $f(n) \leq c \cdot g(n)$ when $n \geq n_0$.

24

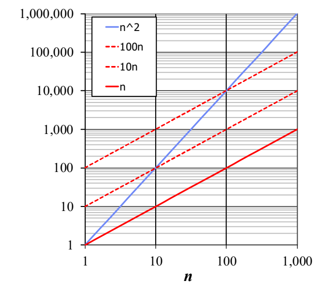
Big-Oh Notation

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \le cn$
  - *What is c and n_0?*
  - Pick $c = 3$ and $n_0 = 10$
- $f(n) \le cg(n)$ for $n \ge n_0$



Big-Oh Example

- Example: the function n^2 is not $O(n)$
  - $n^2 \le cn$
  - $n \le c$
  - The above inequality cannot be satisfied since c must be a constant



Big-Oh Example

$$f(n) = 2n^2 + 3n + 1 = O(n^2)$$

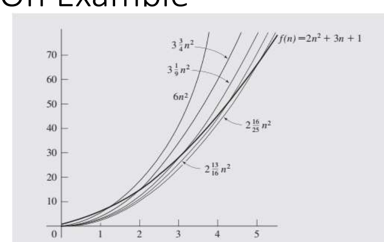
Different values of c and N for function $f(n) = 2n^2 + 3n + 1 = O(n^2)$ calculated according to the definition of big-O.

c	≥ 6	$\ge 3\frac{3}{4}$	$\ge 3\frac{1}{9}$	$\ge 2\frac{13}{16}$	$\ge 2\frac{16}{25}$	...	$\rightarrow$	2
N	1	2	3	4	5	...	$\rightarrow$	∞

We obtain these values by solving the inequality:

$$2n^2 + 3n + 1 \le cn^2$$

Big-Oh Example



n_o = N = 2 and c $\ge \frac{3}{4}$

c	≥ 6	$\ge 3\frac{3}{4}$	$\ge 3\frac{1}{9}$	$\ge 2\frac{13}{16}$	$\ge 2\frac{16}{25}$	...	$\rightarrow$	2
N	1	2	3	4	5	...	$\rightarrow$	∞

$$f(n) = 2n^2 + 3n + 1 = O(n^2)$$

Big-Oh Rules

- If $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

More Big-Oh Examples

- 7n-2
  7n-2 is O(n)
  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \cdot n$ for $n \geq n_0$
  this is true for $c = 7$ and $n_0 = 1$
- $3n^3 + 20n^2 + 5$
  $3n^3 + 20n^2 + 5$ is $O(n^3)$
  need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$
  this is true for $c = 4$ and $n_0 = 21$
- 3 log n + 5
  3 log n + 5 is O(log n)
  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \cdot \log n$ for $n \geq n_0$
  this is true for $c = 8$ and $n_0 = 2$

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$

Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis:
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We determine that algorithm ***arrayMax*** executes at most $8n - 2$ primitive operations
  - We say that algorithm ***arrayMax*** "runs in $O(n)$ time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Counting Primitive Operations

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

Algorithm *arrayMax*(A, n)	# operations
currentMax ← $A[0]$	2
for i ← 1 **to** $n - 1$ **do**	$2(n - 1)$
if $A[i] >$ *currentMax* **then**	$2(n - 1)$
currentMax ← $A[i]$	$2(n - 1)$
{ increment counter i }	$2(n - 1)$
return *currentMax*	1
Total	$8n - 1$

Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages
- The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:

$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$

- Computing the array A of prefix averages of another array X has applications to financial analysis

Prefix Averages (Quadratic)

- The following algorithm computes prefix averages in quadratic time by applying the definition

Algorithm *prefixAverages1*(X, n)	
Input array X of n integers	
Output array A of prefix averages of X	#operations
A ← new array of n integers	n
for i ← 0 **to** $n - 1$ **do**	n
s ← $X[0]$	n
for j ← 1 **to** i **do**	$1 + 2 + \ldots + (n - 1)$
s ← $s + X[j]$	$1 + 2 + \ldots + (n - 1)$
$A[i]$ ← $s / (i + 1)$	n
return A	1

Example array: 12 10 16 20 30

Prefix Averages (Quadratic)

- Thus, algorithm ***prefixAverages1*** runs in $O(n^2)$ time
- Why? - > Inner loop operations - n(n-1)

Prefix Averages (Linear)

- The following algorithm computes prefix averages in linear time by keeping a running sum

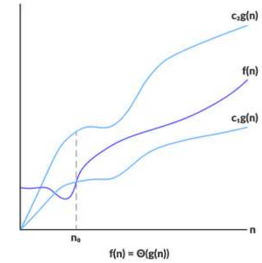
Algorithm *prefixAverages2*(*X, n*)	
Input array *X* of *n* integers	
Output array *A* of prefix averages of *X*	#operations
$A \leftarrow$ new array of n integers	n
$s \leftarrow 0$	1
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
$s \leftarrow s + X[i]$	n
$A[i] \leftarrow s / (i + 1)$	n
return A	1

- Algorithm ***prefixAverages2*** runs in $O(n)$ time
- O(n)

37

Relatives of Big-Oh

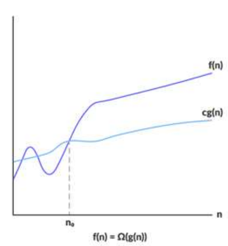
- **1) Big-Theta**
  - f(n) is Θ(g(n)) if there are constants c′ > 0 and c″ > 0 and an integer constant $n_0 \geq 1$ such that c′•g(n) ≤ f(n) ≤ c″•g(n) for n ≥ n_0



38

Relatives of Big-Oh

- **2)Big-Omega**
  - f(n) is Ω(g(n)) if there is a constant c > 0 and an integer constant $n_0 \geq 1$ such that f(n) ≥ cg(n) for n ≥ n_0



39