Chapter 10-4 – Algorithm Efficiency

What is a Good Solution?

- A program incurs a real and tangible cost.
  - Computing time
  - Memory required
  - Difficulties encountered by users
  - Consequences of incorrect actions by program
- A solution is good if …
  - The total cost incurs …
  - Over all phases of its life … is minimal
- Important elements of the solution
  - Good structure
  - Good documentation
  - Efficiency
- Be concerned with efficiency when
  - Developing underlying algorithm
  - Choice of objects and design of interaction between those objects

Measuring Efficiency of Algorithms

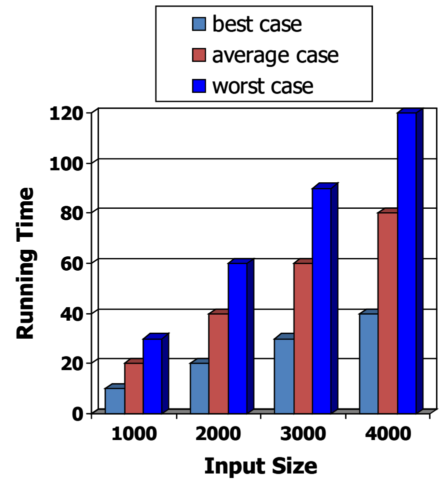
- Important because
  - Choice of algorithm has significant impact
- Examples
  - Responsive word processors
  - Grocery checkout systems
  - Automatic teller machines
  - Video machines
  - Life support systems
- Analysis of algorithms
  - The area of computer science that provides tools for contrasting efficiency of different algorithms
  - Comparison of algorithms should focus on significant differences in efficiency
  - We consider comparisons of *algorithms*, not programs
- Difficulties with comparing programs (instead of algorithms)
  - How are the algorithms coded?
  - What computer will be used
  - What data should the program use
- Algorithm analysis should be independent of
  - Specific implementations, computers, and data

The Execution Time of Algorithms

- An algorithm's execution time is related to the number of operations it requires.
- Usually expressed in terms of the number, n of items the algorithm must process.
- Counting an algorithm's operations is a way to access its efficiency.

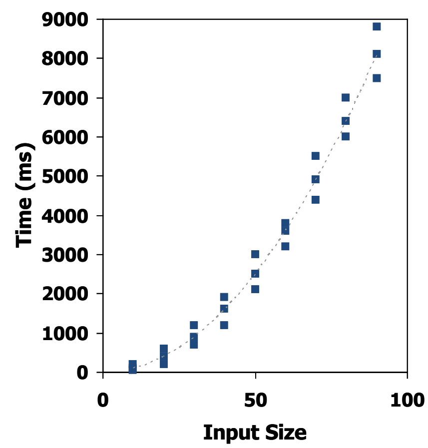
Running Time

- 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 such as games, finance and robotics



Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a function, like the built-in clock() function, 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

- 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

Pseudocode

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- 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 $\leftarrow A[0]$
 for $i \leftarrow 1$ **to** $n - 1$ **do**
  if $A[i] >$ *currentMax* **then**
   currentMax $\leftarrow A[i]$
 return *currentMax*

Pseudocode Details

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

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

- Method/Function call

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

  return *expression*
- Expressions

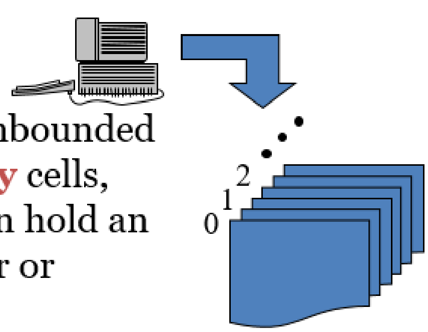
  $\leftarrow$ Assignment (like = in C++)

  = Equality testing (like == in C++)

  n^2 Superscripts and other mathematical formatting allowed

The Random Access Machine (RAM) Model

- A **CPU**
- An potentially unbounded bank of **memory** cells, each of which can hold an arbitrary number or character



- Memory cells are numbered and accessing any cell in memory takes unit time.

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model

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

Counting Primitive Operations

- 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 $arrayMax(A, n)$	# operations
$currentMax \leftarrow A[0]$	1
for $i \leftarrow 1$ **to** $n - 1$ **do**	$n - 1$
if $A[i] > currentMax$ **then**	$(n - 1)$
$currentMax \leftarrow A[i]$	$(n - 1)$
return $currentMax$	1
Total	$3n - 2$

Estimating Running Time

- Algorithm ***arrayMax*** executes $3n - 2$ 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

$$a\,(3n - 2) \le T(n) \le b\,(3n - 2)$$

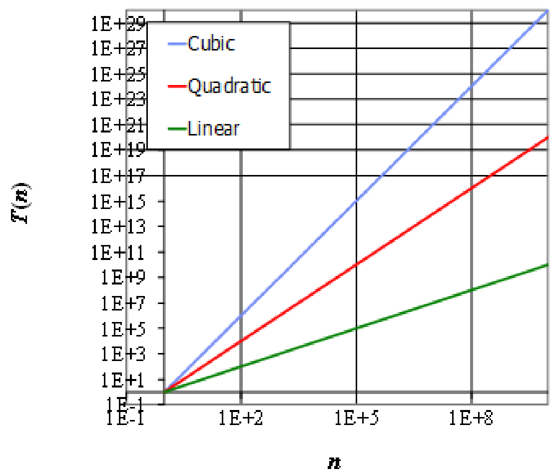
- Hence, the running time $T(n)$ is bounded by two linear functions

Growth Rate of Running Time

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

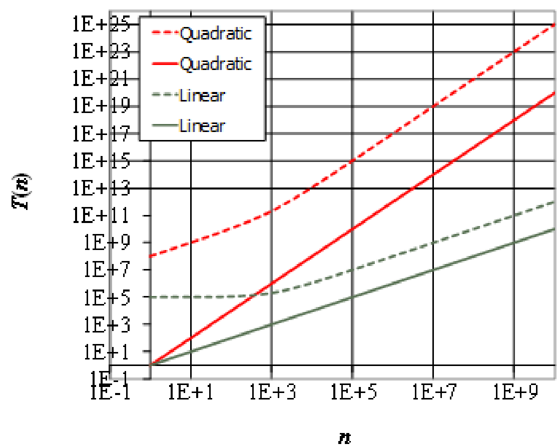
Growth Rates

- Growth rates of functions:
  - Linear $\approx n$
  - Quadratic $\approx n^2$
  - Cubic $\approx n^3$

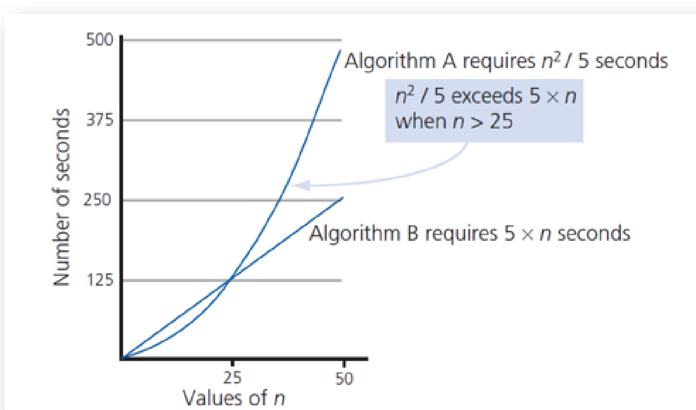


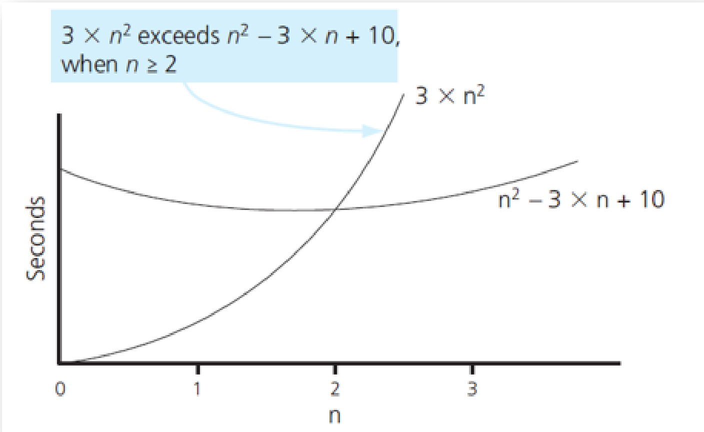
Constant Factors

- 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 (- - -)



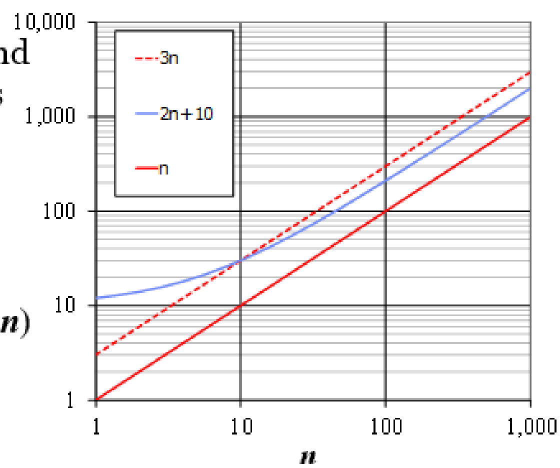
- FIGURE 10-1 Time requirements as a function of problem size n





- FIGURE 10-2 The graphs of $3 \times n^2$ and $n^2 - 3 \times n + 10$

Big-Oh Notation

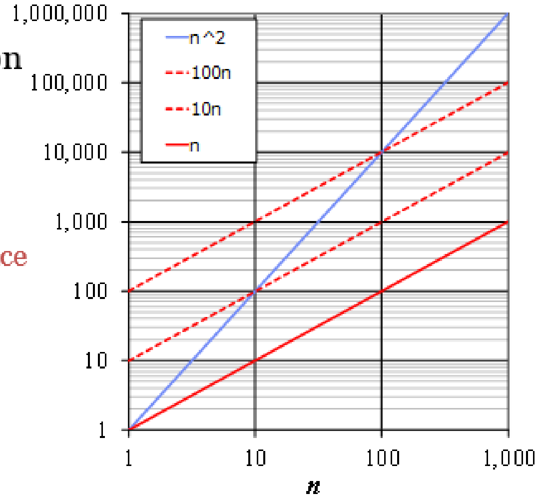


- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

  $f(n) \leq cg(n)$ for $n \geq n_0$
- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$

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



More Big-Oh Examples

- 7n-2

  7n-2 is O(n)

  need $c > 0$ and $n_0 \ge 1$ such that $7n-2 \le c \bullet n$ for $n \ge n_0$

  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 $n_0 \ge 1$ such that $3n^3 + 20n^2 + 5 \le c \bullet n^3$ for $n \ge n_0$

  this is true for $c = 4$ and $n_0 = 25$

- 3 log n + log log n

  3 log n + log log n is O(log n)

  need $c > 0$ and $n_0 \ge 1$ such that $3 \log n + \log \log n \le c \bullet \log n$ for $n \ge n_0$

  this is true for $c = 4$ 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 $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Big-Oh Rules

- If $f(n)$ is 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 $O(3n)$"

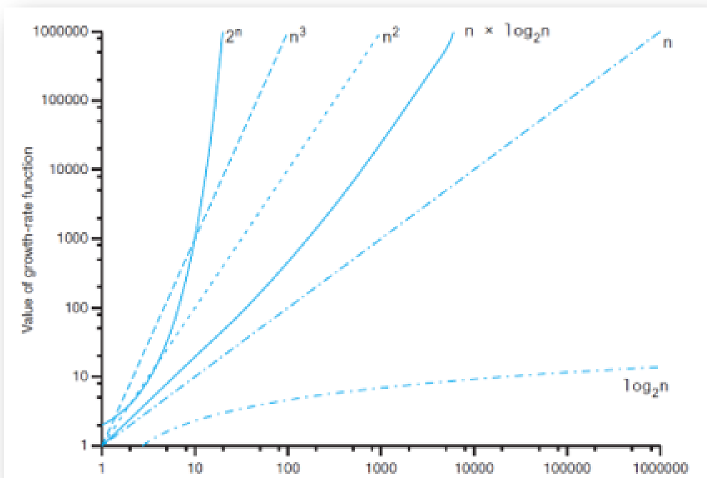
Order-of-Magnitude Analysis and Big O Notation

- Order of growth of some common functions
  - $O(1) < O(\log_2 n) < O(n) < O(n * \log_2 n) < O(n^2) < O(n^3) < O(2^n)$
- Properties of growth-rate functions
  - $O(n^3 + 3n)$ is $O(n^3)$: ignore low-order terms
  - $O(5 f(n)) = O(f(n))$: ignore multiplicative constant in the high-order term
  - $O(f(n)) + O(g(n)) = O(f(n) + g(n))$

Analysis and Big O Notation

Function	n: 10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n \times \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

- FIGURE 10-4 A comparison of growth-rate functions



Order-of-Magnitude Analysis and Big O Notation

- Worst-case analysis
  - A determination of the maximum amount of time that an algorithm requires to solve problems of size n
- Average-case analysis
  - A determination of the average amount of time that an algorithm requires to solve problems of size n
- Best-case analysis
  - A determination of the minimum amount of time that an algorithm requires to solve problems of size n

Analysis and Big O Notation

- Worst-case analysis
  - Worst case analysis usually considered
  - Easier to calculate, thus more common
- Average-case analysis
  - More difficult to perform
  - Must determine relative probabilities of encountering problems of given size

Keeping Your Perspective

- If problem size is always small
  - Possible to ignore algorithm's efficiency
- Weigh trade-offs between
  - Algorithm's time and memory requirements
- Compare algorithms for style and efficiency

Asymptotic Algorithm Analysis

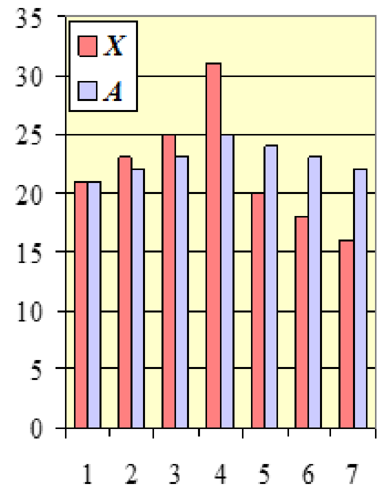
- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We determine that algorithm ***arrayMax*** executes at most $7n - 1$ 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

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:

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

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



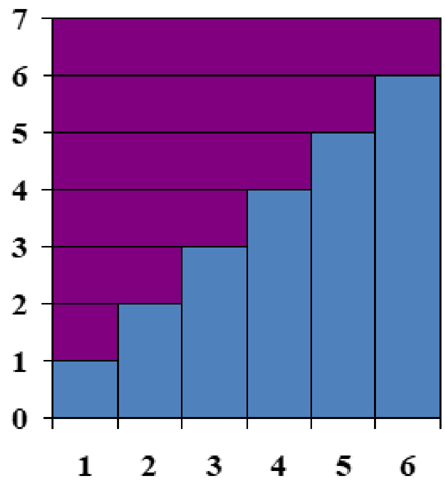
Prefix Averages (Quadratic)

- 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 ← 0                                          n
        for j ← 0 to i do                     1+ 2 + 3 …+ (n - 1)
            s ← s + X[j]                      1 + 2 + …+ (n − 1)
        A[i] ← s / (i + 1)                             n
    return A                                           1
```

Arithmetic Progression

- The running time of ***prefixAverages1*** is $O(1 + 2 + \ldots + n - 1)$
- The sum of the first n integers is $(n-1)\, n / 2$
  - There is a simple visual proof of this fact
- Thus, algorithm ***prefixAverages1*** runs in $O(n^2)$ time



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

Relatives of Big-Oh

- **big-Omega**
  - f(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$
- **big-Theta**
  - f(n) is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)$ for $n \geq n_0$
- **little-oh**
  - f(n) is $o(g(n))$ if, for any constant $c > 0$, there is an integer constant $n_0 \geq 0$ such that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$
- **little-omega**
  - f(n) is $\omega(g(n))$ if, for any constant $c > 0$, there is an integer constant $n_0 \geq 0$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

Intuition for Asymptotic Notation

Big-Oh
- f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)

big-Omega
- f(n) is $\Omega(g(n))$ if f(n) is asymptotically **greater than or equal** to g(n)

big-Theta
- f(n) is $\Theta(g(n))$ if f(n) is asymptotically **equal** to g(n)

little-oh
- f(n) is o(g(n)) if f(n) is asymptotically **strictly less** than g(n)

little-omega
- f(n) is $\omega(g(n))$ if is asymptotically **strictly greater** than g(n)

Efficiency of Searching Algorithms

- Sequential search
  - Worst case: $O(n)$
  - Average case: $O(n)$
  - Best case: $O(1)$
- Binary search
  - $O(\log_2 n)$ in worst case
  - At same time, maintaining array in sorted order requires overhead cost … can be substantial