Analysis of Algorithms

(Chapter 2)

What did we learn last lesson?

1. Efficiency of an algorithm depends on **input size**
2. Efficiency of an algorithm also depends on **basic operation**
3. Efficiency can be expressed by **counting** the basic operation

Example1

- Problem: find the max element in a list
- Input size measure:
  - *Number of list's items, i.e. n*
- Basic operation:
  - *Comparison*

```
ALGORITHM MaxElement(A[0..n − 1])
    maxval ← A[0]
    for i ← 1 to n − 1 do
        if A[i] > maxval
            maxval ← A[i]
    return maxval
```

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1$$

Example2

- Problem: *Multiplication of two matrices*
- Input size measure:
  - *Matrix dimensions or total number of elements*
- Basic operation:
  - *Multiplication of two numbers*

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$
    for $i \leftarrow 0$ **to** $n-1$ **do**
        for $j \leftarrow 0$ **to** $n-1$ **do**
            $C[i, j] \leftarrow 0.0$
            for $k \leftarrow 0$ **to** $n-1$ **do**
                $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
    return C

$$C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = n^3$$

Example3

- Problem: calculating sum
- Input size measure:
  - *Number n*
- Basic operation:
  - *Addition*

```
Example3(n)
 0 ← sum
 i ← n
 while i ≥ 1
    sum ← sum +1
    i ← i/2
 return sum
```

$$C(n) = \log n$$

Example4

- Problem: Searching for key in a list of n items
- Input size measure:
  - *Number of list's items, i.e. n*
- Basic operation:
  - *Key comparison*

ALGORITHM $SequentialSearch(A[0..n-1], K)$

$i \leftarrow 0$

while $i < n$ **and** $A[i] \neq K$ **do**

$\quad i \leftarrow i + 1$

if $i < n$ **return** i

else return -1

Depends on form of input

$C_{worst}(n) = n$

$C_{best}(n) = 1$

Notation

For some algorithms efficiency depends on form of input:

- Worst case: $C_{worst}(n)$ – maximum over inputs of size n
- Best case: $C_{best}(n)$ – minimum over inputs of size n

Which to use: best, worst

- ***We will focus on*** ***worst-case*** ***analysis in this course***
  - unless otherwise specified, you should always analyze the worst case

- there are many situations where:

  best case = worst case

- consider the algorithm to find the largest element in an unordered array

Try it!

For each of the following algorithms determine:

a) its basic operation
b) basic operation count
c) if basic op count depends on input form

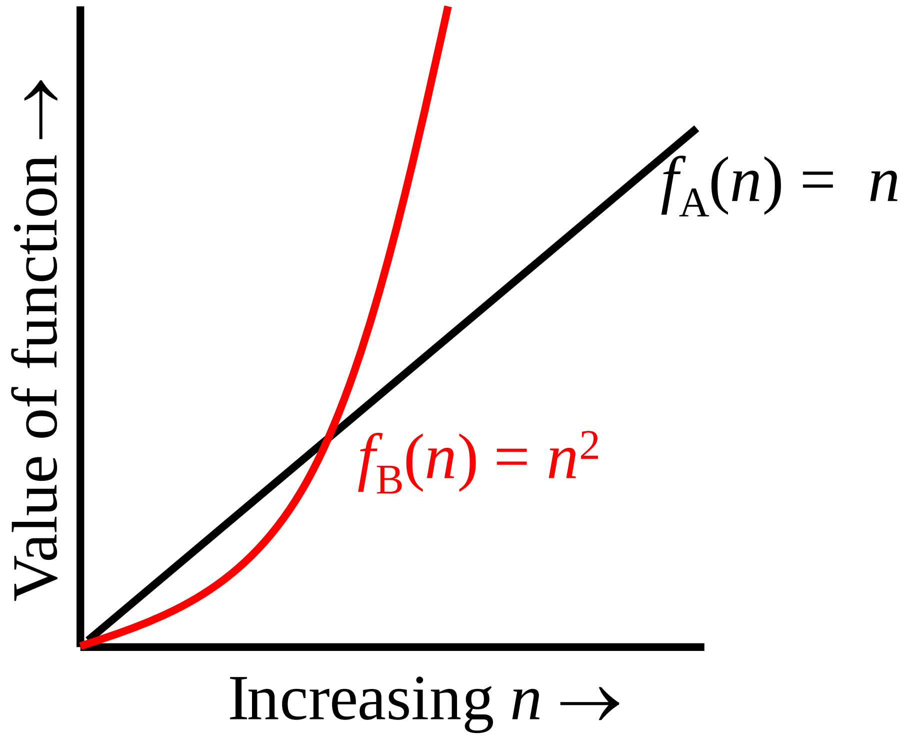
1. computing the sum of a set of numbers
2. computing n! (n factorial)
3. check weather all the elements in a given array are distinct

Counts you might see

- $C(n) = n(n-1)/2$
- $C(n) \approx 0.5n^2$
- $C(n) = log\ n + 5$
- $C(n) =$ n!

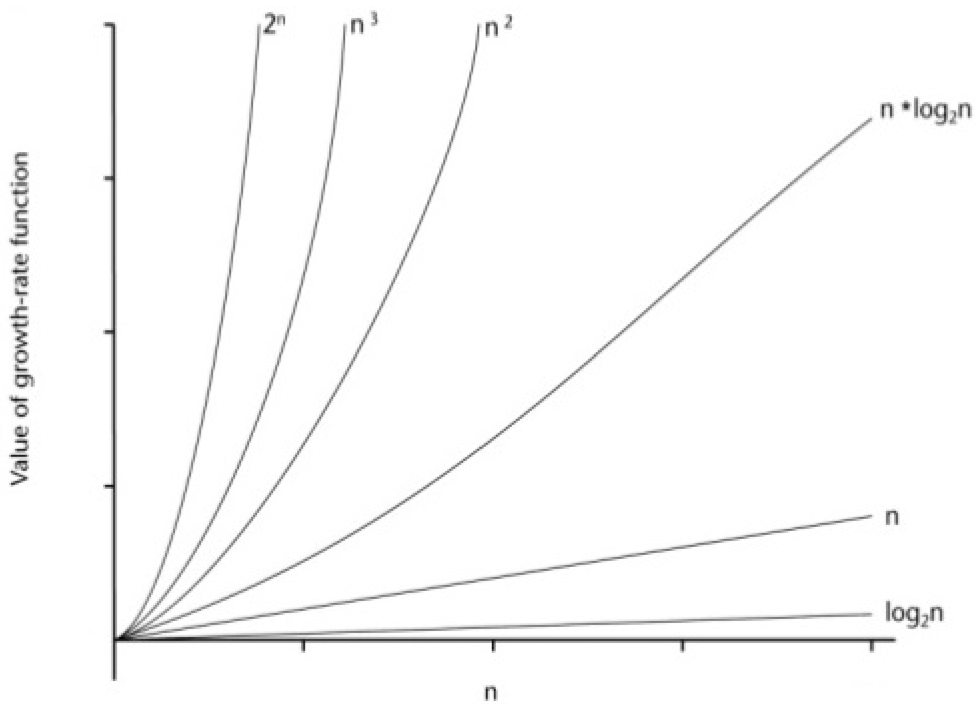
- Which one is better algorithm?

Order of growth



Order of growth

- What we really care about:
  - Order of growth as $n \to \infty$



Orders of Growth

- consider table 2.1 from your textbook

these represent possible functions that classify basic ops counts

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

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}		

it would take more than 5 billion years to make 100! calculations*

* – on at 1000 MHz CPU

Orders of Growth

$$\log_2 n < n < n \log_2 n < n^2 < n^3 < 2^n < n!$$

Base Efficiency Classes (part 1)

Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 5.5). Note that a logarithmic algorithm cannot take into account all its input (or even a fixed fraction of it): any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	"*n-log-n*"	Many divide-and-conquer algorithms (see Chapter 4), including mergesort and quicksort in the average case, fall into this category.

Base Efficiency Classes (part 2)

n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on n-by-n matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

General Strategy for Analysis of Non-recursive Algorithms

This strategy is taken from page 62 of your textbook:

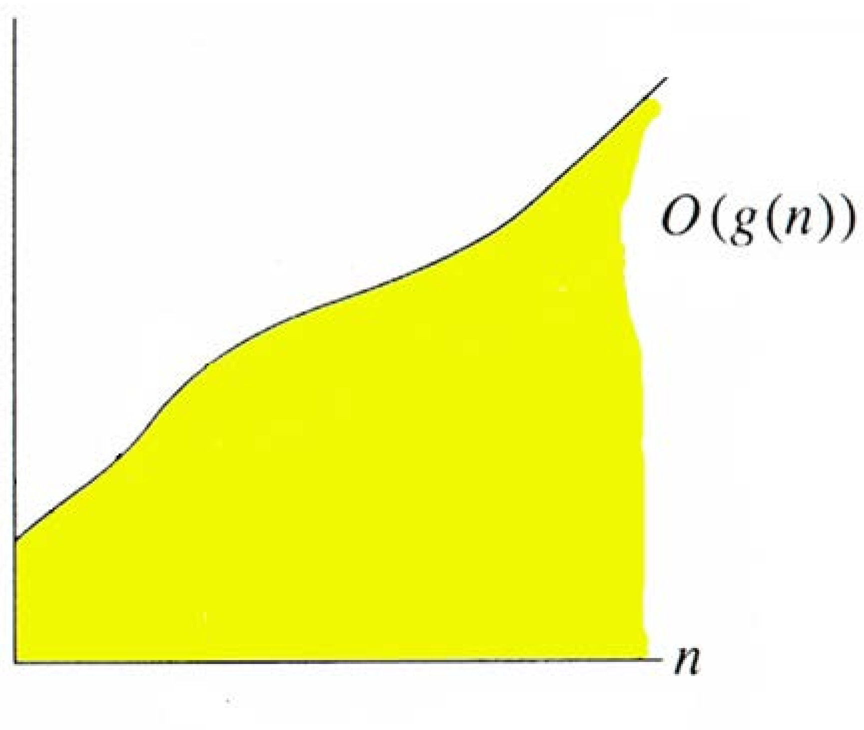
1. Decide on a parameter indicating the inputs size.
2. Identify the algorithms basic operation.
3. Check whether the number of times the basic operation is executed depends only on the size of the input.
   - if it depends on some other property, the best/worst/average case efficiencies must be investigated separately
4. Set up a sum expressing the number of times the basic operation is executed.
5. Use standard formulas and rules of sum manipulation to find a closed form formula c(n) for the sum from step 4 above.
6. Determine the efficiency class of the algorithm using asymptotic notations

Asymptotic order of growth

A way of comparing functions

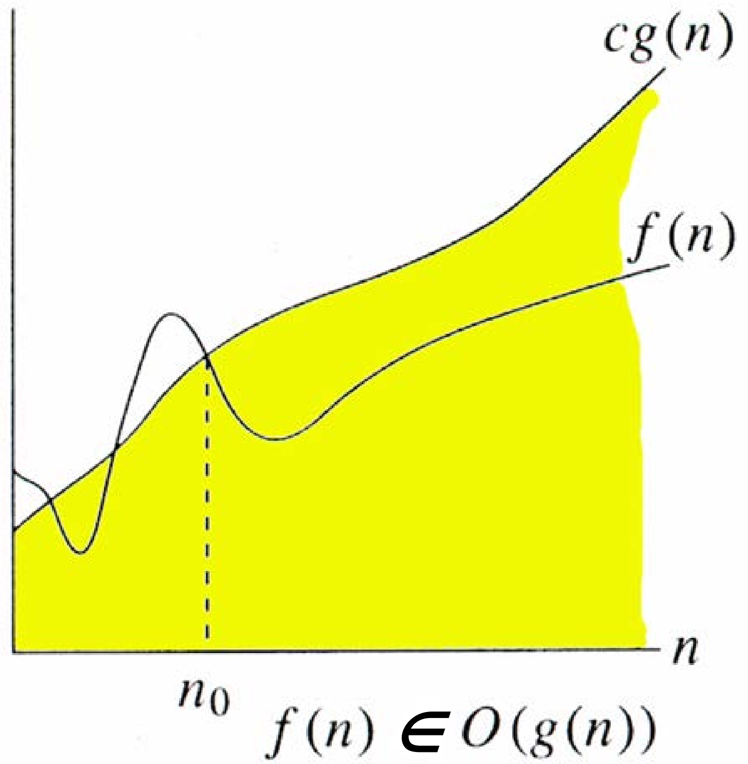
- Big O
- Big Θ
- Big Ω

Big-Oh in Pictures



Set of all functions whose *rate of growth* is the same as or lower than that of $g(n)$.

Big-Oh in Pictures



$f(n) \leq c * g(n)$, for all $n \geq n_0$

Big-Oh

- Simple Rule: Drop lower order terms and constant factors

1. $50n^3 + 20n + 4 \in O(n^3)$
2. $4n^2 + 10 \in O(n^2)$
3. $n(2n + 1) \in O(n^2)$
4. $3\log n + 1 \in O(\log n)$
5. $3\log n + n \in O(n)$
6. $1 + \log 6 \in O(1)$
7. $5! + 3^2 \in O(1)$

Examples

What is the order of the following:

- $10n$ O(n)
- $5n^2+20$ O(n^2)
- $10000n + 2^n$ O(2^n)
- log(n) * (1+ n) O(nlog(n))

General Plan for Analysis

- Decide on parameter *n* indicating *input size*
- Identify algorithm's *basic operation*
- Determine *worst* and/or *best* cases for input of size *n*
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules
- Determine the efficiency class of the algorithm using asymptotic notations

Example1

- Problem: find the max element in a list
- Input size measure:
  - *Number of list's items, i.e. n*
- Basic operation:
  - *Comparison*

ALGORITHM *MaxElement*$(A[0..n-1])$
    $maxval \leftarrow A[0]$
    for $i \leftarrow 1$ **to** $n-1$ **do**
        if $A[i] > maxval$
            $maxval \leftarrow A[i]$
    return $maxval$

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in O(n)$$

Example2

- Problem: *Multiplication of two matrices*
- Input size measure:
  - *Matrix dimensions or total number of elements*
- Basic operation:
  - *Multiplication of two numbers*

```
ALGORITHM  MatrixMultiplication(A[0..n − 1, 0..n − 1], B[0..n − 1, 0..n − 1])
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            C[i, j] ← 0.0
            for k ← 0 to n − 1 do
                C[i, j] ← C[i, j] + A[i, k] * B[k, j]
    return C
```

$$C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = n^3 \in O(n^3)$$

Example 5: Element uniqueness problem

ALGORITHM $UniqueElements(A[0..n-1])$

//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
//   and "false" otherwise

for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] = A[j]$ **return false**
return true

Example 5

ALGORITHM *UniqueElements*$(A[0..n-1])$

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//        and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

- Parameter for input size:

  n, the size of the array

- Basic operation:

  Comparison in the innermost loop

- Worst case efficiency count… nested loop:

$$\sum_{i=0}^{n-2}\sum_{J=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i-1+1) = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i = n(n-1) - (n-1) - (n-2)(n-1)/2$$

$$= n^2 - n - n + 1 - n^2/2 + 3n/2 - 1$$

$$= n^2/2 - n/2 \in O(n^2)$$

Try it/ Homework

1. Chapter 2.1, page 50, question 2
2. Chapter 2.2, page 60, question 5
3. Chapter 2.3, page 68, question 5,6

QUIZ Announcement

- There will be a quiz in the lab next week.

- It will be 5 questions, on D2L
  - It will take 10-20 minutes
  - Followed by a lab activity