Analysis of Algorithms

Review: Analysis of Algorithms

- Issues:
  - Correctness
  - Time efficiency
  - Space efficiency
  - Optimality
- Approaches:
  - Theoretical analysis
  - Empirical analysis

Review: Basic Approaches to Algorithm Analysis

- Empirical analysis
  - Select a specific (typical) sample of inputs
  - Either
    - Use physical unit of time (e.g., milliseconds)
    - Or count actual number of times “basic” operation is executed
  - Analyze the empirical data
- Theoretical analysis
  - E.g., binary search halves search space every iteration.
    - Hence, for input size N, take ~$\log_2 N$ comparisons.

How to Measure Algorithm Efficiency?

- Space utilization: the amount of *memory* required to store the data
- Time efficiency: the amount of *time* required to accomplish the task
- The tradeoff problem
- Today, space is not as big of a problem as it once was
  - Time efficiency is more emphasized
  - But not always!
    - In embedded computers or sensor nodes, space efficiency is still important

Factors Impacting Time Efficiency

- Algorithm's execution time efficiency depends on :
  - size of input
  - speed of machine
  - quality of source code
  - quality of compiler

These vary from one platform to another

So, we cannot express time efficiency meaningfully in real time units such as seconds!

Theoretical Analysis of Algorithm Efficiency

- But, we can count the number of repetitions of the ***basic operation*** as a function of ***input size***
  - This gives us an indication of how long an algorithm may take
    - a measurement of efficiency of an algorithm

- We can compute *T(n):* running time of the algorithm as a function of n, where *n* is the input size
  - E.g., sorting a list of phone numbers
  - E.g., searching a list of numbers using binary search or linear search
  - T(n) is relevant to all computers a program may execute on

Review: Example: Calculating the Mean

This algorithm finds the mean of x[0], … x[n-1]

Statement	# of times executed
1. Initialize *sum=0.*	
2. Initialize index variable *i=0*	
3. While *i* < *n* do the following	
4. a. Add *x[i]* to *sum*	
5. b. Increment *i* by *1*	
End while.	
6. Calculate and return *mean=sum/n.*	
Total	?

Order of Magnitude

- Let's consider what happens as *n* grows when

$$T(n) = 3n+4$$

- So we say that T(n) has "order of magnitude n"
- This is usually written using the "***big-O notation***" as:
- T(n) is O(n) or $T(n) \in O(n)$

More Generally…

- The computing time **T(n)** of an algorithm is said to have **order of magnitude f(n)** is denoted by

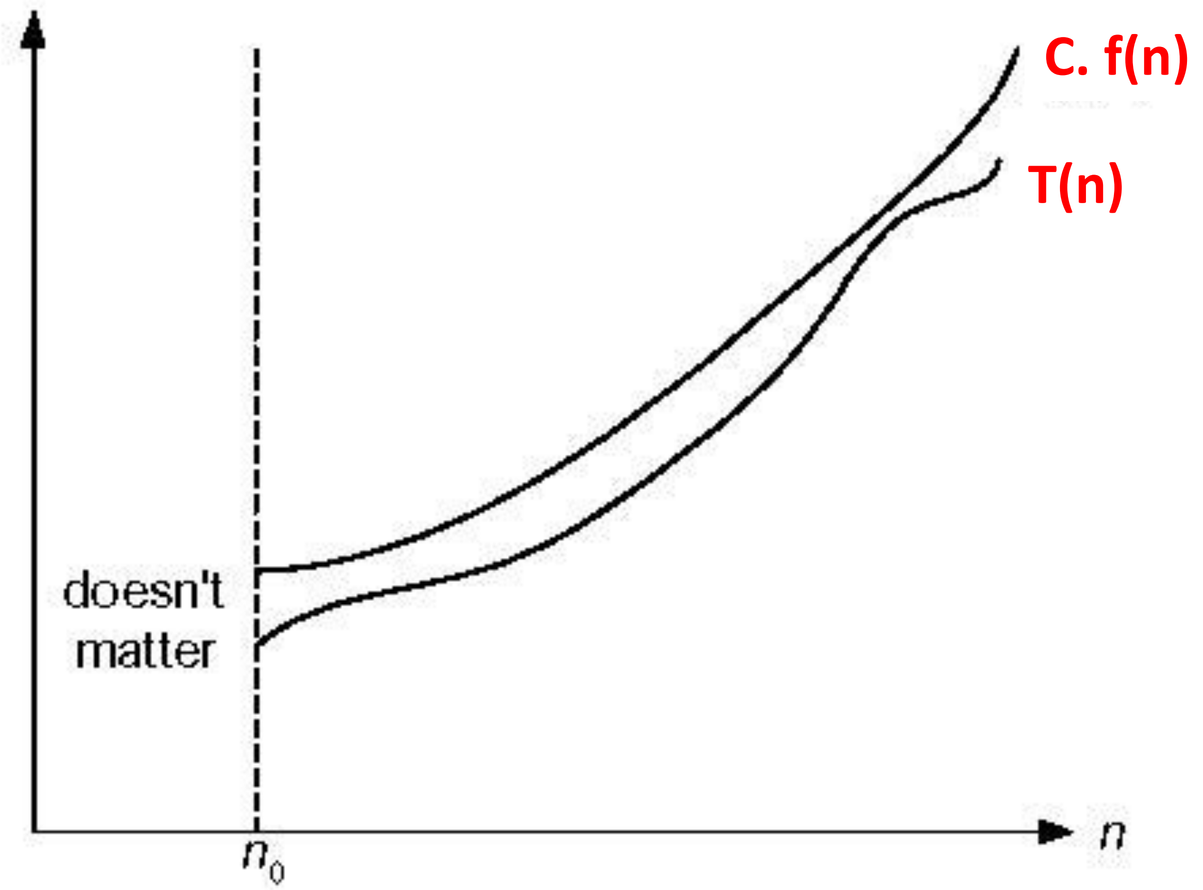
  T(n) is O(f(n))

  if there is some constant **c** such that:

  T(n) ≤ c.f(n)

  *for all sufficiently large values of **n***

Big-O



O(f(n))

Computational Complexity

- **T(n)** is bounded above by some constant **C** times **f(n)** for all values of **n** <u>from some point on.</u>

  → The computational complexity of the algorithm is said to be O(f(n))

Rate of Growth

- Given two functions, there are some points where one function is smaller than the other function.
  - 1000 N vs N^2
    - When N is small? when N is large?

- When T(N) = O(f(N)), then we are guaranteed that the function T(N) grows at a rate no faster than f(N)
  - f(N) is an upper bound on T(N)

Example

- The computing time of the *mean* algorithm was found to be $T(n) = 3n + 4$. What is its computational complexity using the Big-O notation?

Note

- It would also be correct to say that T(n) is O(52761n) or T(n) is O(4n+200)
- … but we prefer to stick with *simple functions* like n, n^2, or $log_2 n$

- **Constants and multiplicative factors are ignored.**
  - **Simple function f(n)**

Also…

- If $T(n)$ is $O(n)$ then:
  - It certainly is *$O(n^2)$*
  - *… and is $O(n^{5/2})$*
  - *… and is $O(2^n)$*

- Why?
  - Because all these other functions actually grow ***faster*** than $f(n) = n$.

- However, report the tightest upper bound.
  - If $T(n)$ is $O(n)$, $O(n^2)$, $O(2^n)$—tightest upper bound for $T(n)$'s growth rate would be $O(n)$
  - Therefore, correct answer would be $T(n)$ is $O(n)$

Another Point to Consider

- Sometimes, the computing time only depends on the size of the input *(n)*.
- But in many other cases it also depends on the arrangement of the input items
  - Example: sorting

Best-case, Average-case, Worst-case

- When the arrangement of data makes a difference, then we may need to find:
  - Best case: minimum over inputs of size *n*
    - *Not too informative!*
  - Average case: "average" over inputs of size *n*
    - *Difficult to determine [see note below]*
  - Worst case: maximum over inputs of size *n*
    - *This is what is used to measure an algorithm's performance*

Note: Number of times the basic operation will be executed on a typical input or random input. This is NOT the average of worst and best case

Let’s Try It!

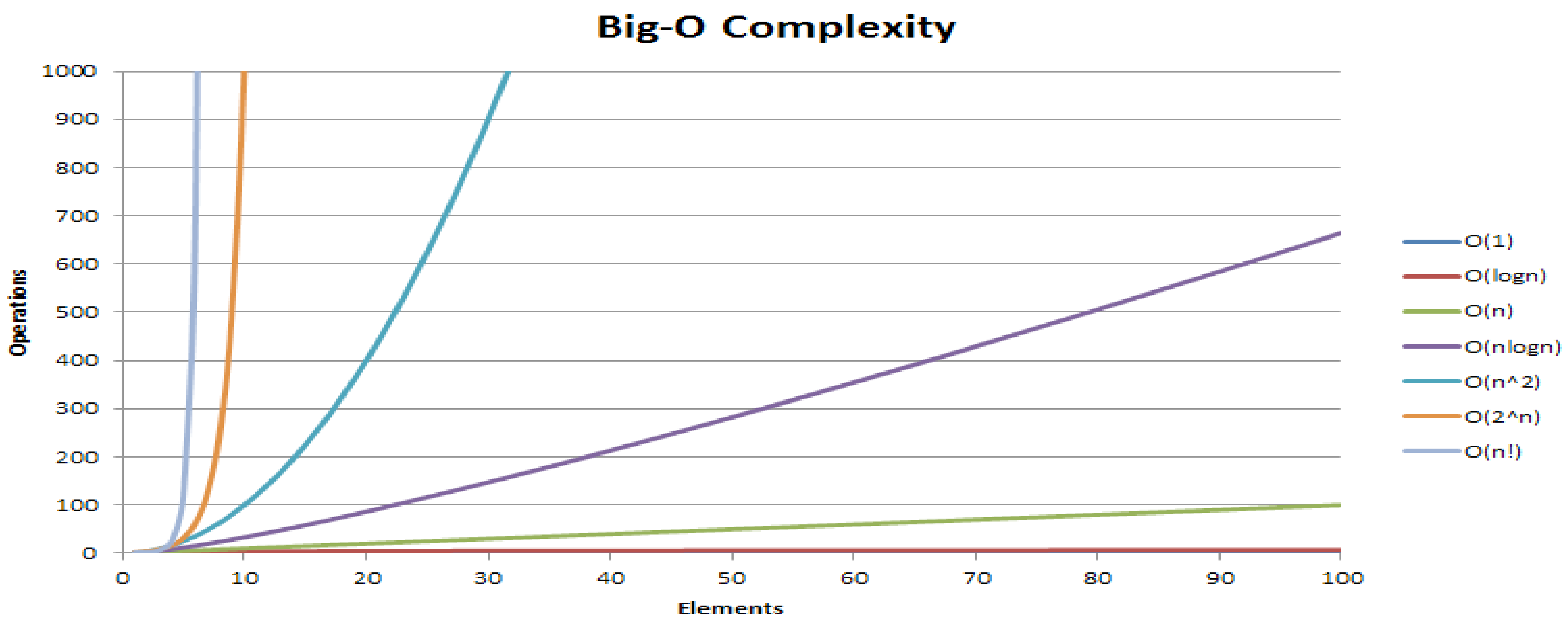
```
""" Search for x in list a of size n
 assuming the elements in list a are unique """

def search (a, x):
   for i in range(n):
      if a[i]==x:
           return i;
   return -1;
```

- What is the Best-case, Worst-case, and Average-case complexity in terms of T(n)

Simple Array Search Algorithm…

- Answers:
  - Best-case: O(1)
  - Average-case: O(n)
  - Worst-case: O(n)
- Do you know how to get these?

Comparison


Big-O Complexity
Operations
1000
900
800
700
600
500
400
300
200
100
0
0
10
20
30
40
50
60
70
80
90
100
Elements
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)

Another Simple Example

```
def sum (int n):
  partialSum =0
  for i in range(1,n+1):
          partialSum += i*i*i
     return partialsum
```

What about this one?

```
def magicSum (int n):
    magic = 0
    for i in range(0,n,2):
        magic+= i*i*i
    return magic
```

Simplifying the Complexity Analysis

- Non-trivial computation in the preceding example
- We can simplifying the Big-O computation considerably
  - Identify the statement executed most often and determine its execution count
    - Ignore items with lower degree
  - Only the highest power of n affects Big-O computation
  - Big-O estimate gives an ***approximate*** measure of the computing time of an algorithm for large inputs
    - Not necessarily for small inputs… but that's ok ☺

General Rule 1 – FOR Loops

- The running time of a **for** loop is at most the running time of the statements inside the for loop (including tests) multiplied by the number of iterations

General Rule 2 – Nested Loops

- Analyze these inside out. Find the running time of the statement inside the innermost loop and multiple that by the product of the sizes of the loops

Example

```
for i in range(n):
    for j in range(n):
        k++;
```

Rule 3 – Consecutive Statements

- These just add.
  - The maximum is the one that counts
- Example:

```
for i in range(n):
     a[i] = 0
for i in range(n):
     for j in range(n):
          a[i] += a[j] + i +j
```

Rule 4 – If/Else

```
if (condition)
      S1
else
      S2
```

- The running time is never more than the running time of the test plus the larger of the running times of S1 and S2

Important Complexity Rules

- If T1(N) = O(f(N)) and T2(N) = O(g(N)), then
  - T1(N) + T2(N) = O(f(N) + g(N))
  - T1(N)*T2(N) = O(f(N)*g(N))
- If T(N) is a polynomial of degree k, then
  - T(N) = O(n^k)
- (logN)^k = O(N), for any constant k

Practice 1

```
for i in range(n):
    for j in range(n):
        m[i][j] = a[i][j] + b[i][j];
```

Practice 1 (Solution)

```
for i in range(n):
    for j in range(n):
        m[i][j] = a[i][j] + b[i][j];
```

- $T(n) = 2n^2$ which is $O(n^2)$

Practice 2

```
for i in range(n):
    for j in range(i,n):
        m[i][j] = a[i][j] + b[i][j];
```

Practice 2 (Solution)

```
for i in range(n):
    for j in range(i,n):
        m[i][j] = a[i][j] + b[i][j];
```

- The statement under the inner for loop has 2 operations (assignment, add)
- For n iterations of the outer loop, the inner for loop will run a total of: $n + (n-1) + (n-2) + \ldots + 1 = n(n+1)/2$
- Therefore, $T(n) = 2n(n+1)/2$; $T(n)$ is $O(n^2)$

Complexity

- What is the complexity O() of the following expressions?

1. $T(n) = n^3 + 1000$
2. $T(n) = 100n^4 + 1000n^3 + 1000000000$
3. $T(n) = n^5(\log_2 n) + 1000n^3 + (\log_2 n)^{1000000}$

Complexity (Keys)

- What is the complexity O() of the following expressions?

- $n^3 + 1000$ is $O(n^3)$
- $100n^4 + 1000n^3 + 1000000000$ is $O(n^4)$
- $n^5(\log_2 n) + 1000n^3 + (\log_2 n)^{1000000}$ is $O(n^5(\log_2 n))$

Asymptotic bounds (I)

- Is $2^{n+1} = O(2^n)$?
  - Yes!

To show that $2^{n+1} = O(2^n)$, we must find constants $c, n_0 > 0$ such that

$0 \leq 2^{n+1} \leq c \cdot 2^n$ for all $n \geq n_0$.

Since $2^{n+1} = 2 \cdot 2^n$ for all n, we can satisfy the definition with $c = 2$ and $n_0 = 1$.