FPT UNIVERSITY
DREAM OF INNOVATION

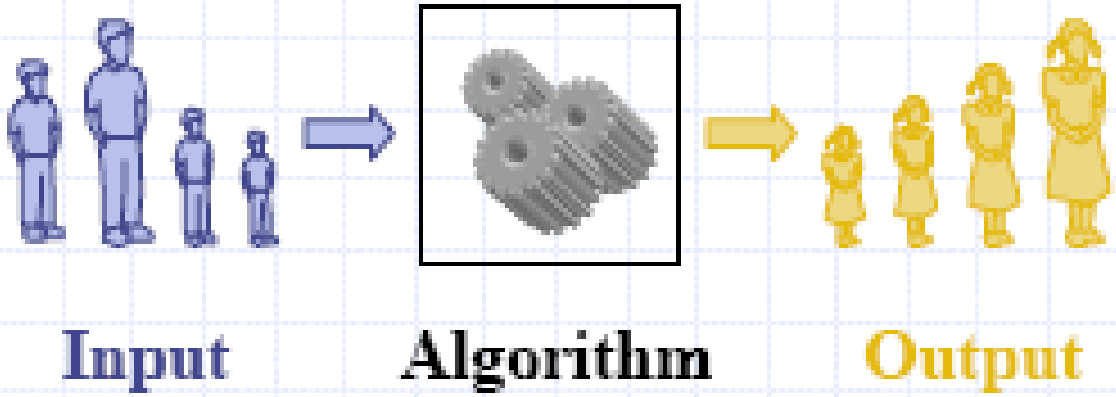
Complexity Analysis

(For reading)

Objectives

- Computational and Asymptotic Complexity
- Big-O, Big-*Ω and Big-Θ* Notations
- The Best, Average, and Worst Cases
- *NP-Completeness*

Simple definition of an algorithm



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

Analysis of Algorithms

What are requirements for a good algorithm?

- Precision:
  - Proved by mathematics
  - Implementation and test
- Simple and public
- Effectiveness:
  - Run time duration (**time complexity**)
  - Memory space (**space complexity**)

FPT UNIVERSITY
DREAM OF INNOVATION

What is a computational complexity?

The same problem can be solved with various algorithms that differ in efficiencies.

The computational complexity (or simply speaking, complexity) of an algorithm is a measure of how "complex" the algorithm is. The complexity answers the question: How difficult is to compute something that we know to be computable?

e.g. What resources (time, space, machines, ...) will it take to get a result?

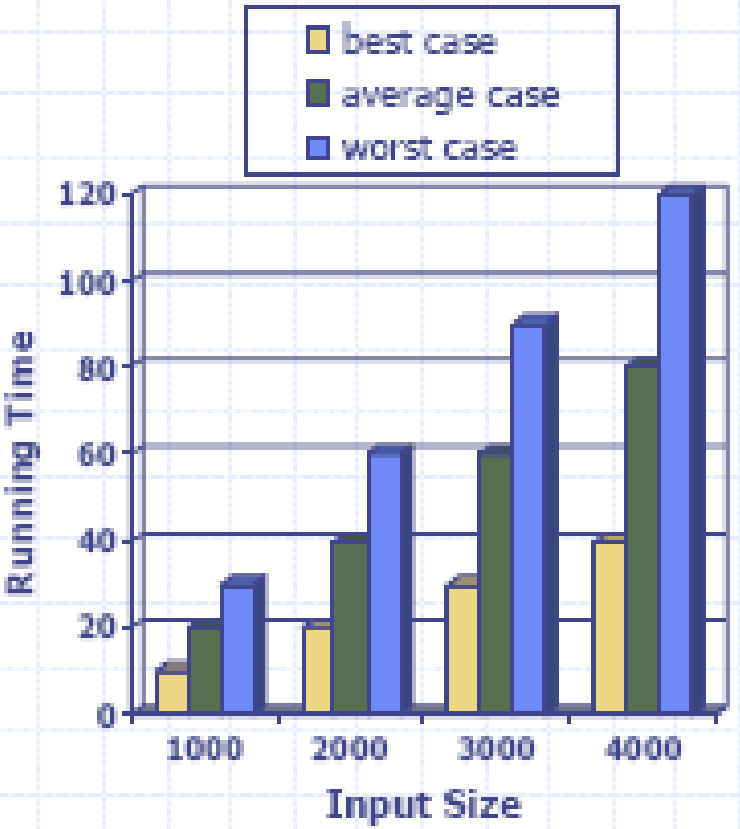
Rather than referring to how "complex" an algorithm is, we often talk instead about how "efficient" the algorithm is. Measuring efficiency (or complexity) allows us to compare one algorithm to another (assuming that both algorithms compute the same result).

There are several measurements to compare algorithms. Here we'll focus on one complexity measure: **the computation time** of an algorithm.

How can we compare program execution on two different machines? For example, if we use wall clock time to compare two different machines, will this tell us which algorithm/program is more efficient?

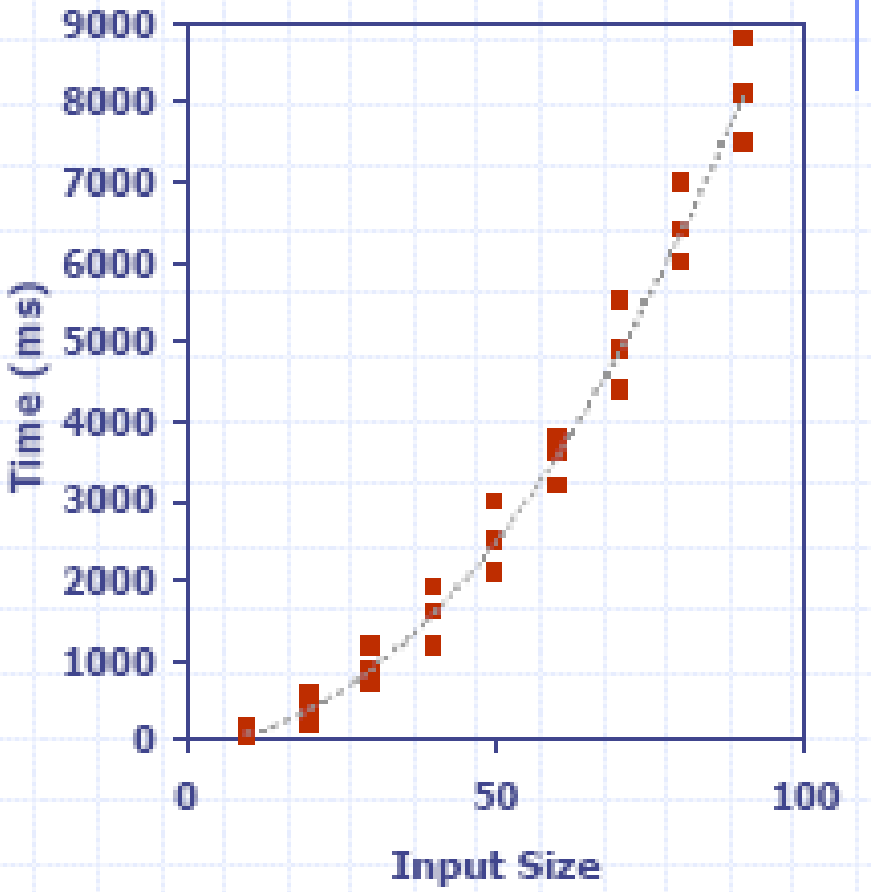
Running time

- 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 method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Plot the results



Experimental Study Example

```
import java.util.Calendar;
public class Main
  { public static void main(String[] args)
      { long beginTimes = Calendar.getInstance().getTimeInMillis();
        long n = 10000;
        for (long i=0; i<n;++i)
            for(long j =0; j<n; ++j);
        long endTimes = Calendar.getInstance().getTimeInMillis();
        System.out.println("The times in ms for run the program are:");
        System.out.println(endTimes - beginTimes);
      }
  }
```

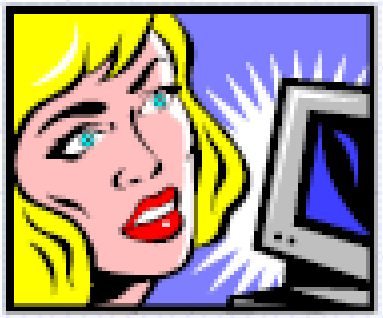
```
C:\Documents and Settings\Hung\Desktop\DSA-PTIT\example
lysis)\01-runningTime>echo off

   The times in ms for run the program are:  172

Press any key to continue . . .
```

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



Time complexity of an algorithm (1)

- Run time duration of a program depend on
  - Size of data input
  - Computing system (platform: operation system, speed of CPU, type of statement…)
  - Programming languages
  - State of data
- => It is necessary to evaluate the run time of a program such that it does not depend on computing system and programming languages.

Time complexity of an algorithm (2)

We'll define a time complexity measure as the number of operations performed by the algorithm on an input of a given size.

What is meant by "number of operations"?

and

What is meant by "size"?

We want to express the number of operations performed as a function of the input size n.

What if there are many different inputs of size n?

Worst case

Best case

Average case

"number of operations" = "running time"?

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

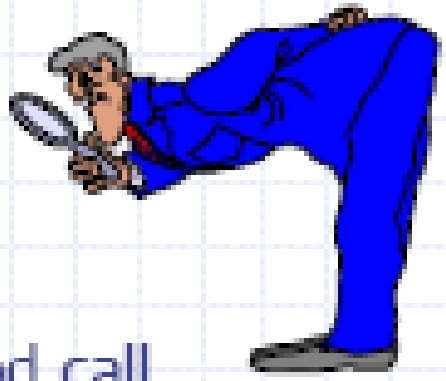
- High level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

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

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

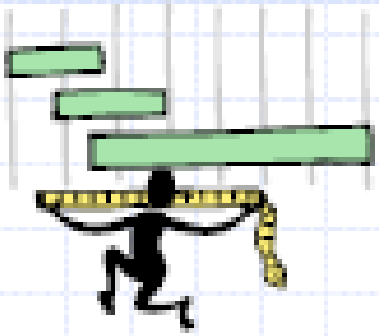
- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Counting Primitive Operation

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

Estimating Running Time



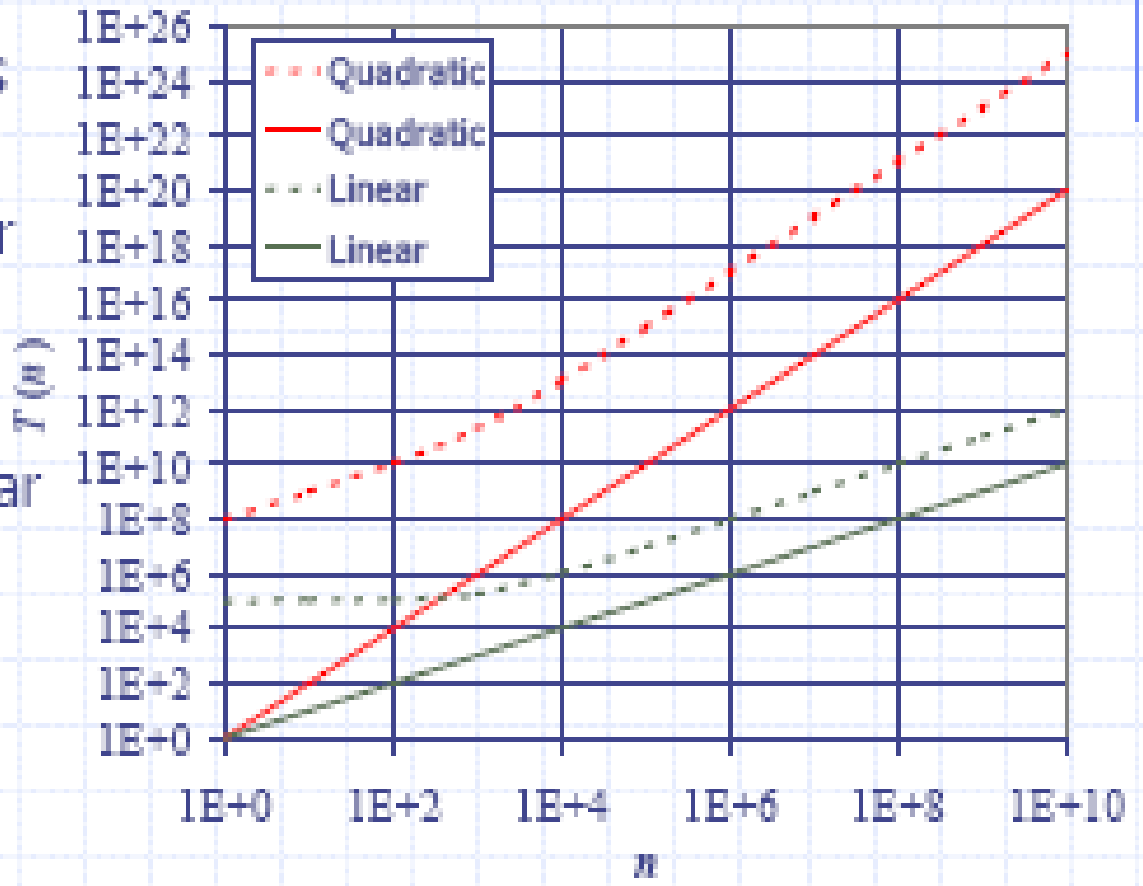
- Algorithm $arrayMax$ executes $8n - 2$ primitive operations in the worst case. Define:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation
- Let $T(n)$ be worst-case time of $arrayMax$. Then
$$a(8n - 2) \leq T(n) \leq b(8n - 2)$$
- Hence, the 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*

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

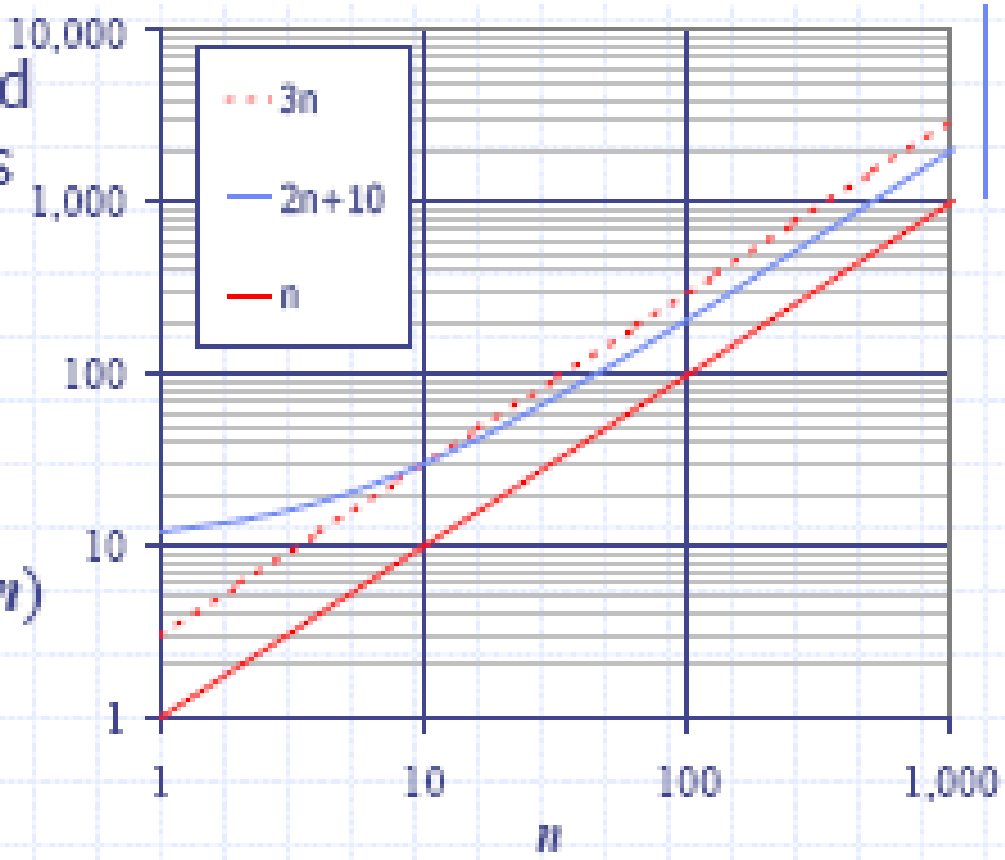


Big-Oh Notation

- 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 $n \geq n_0$

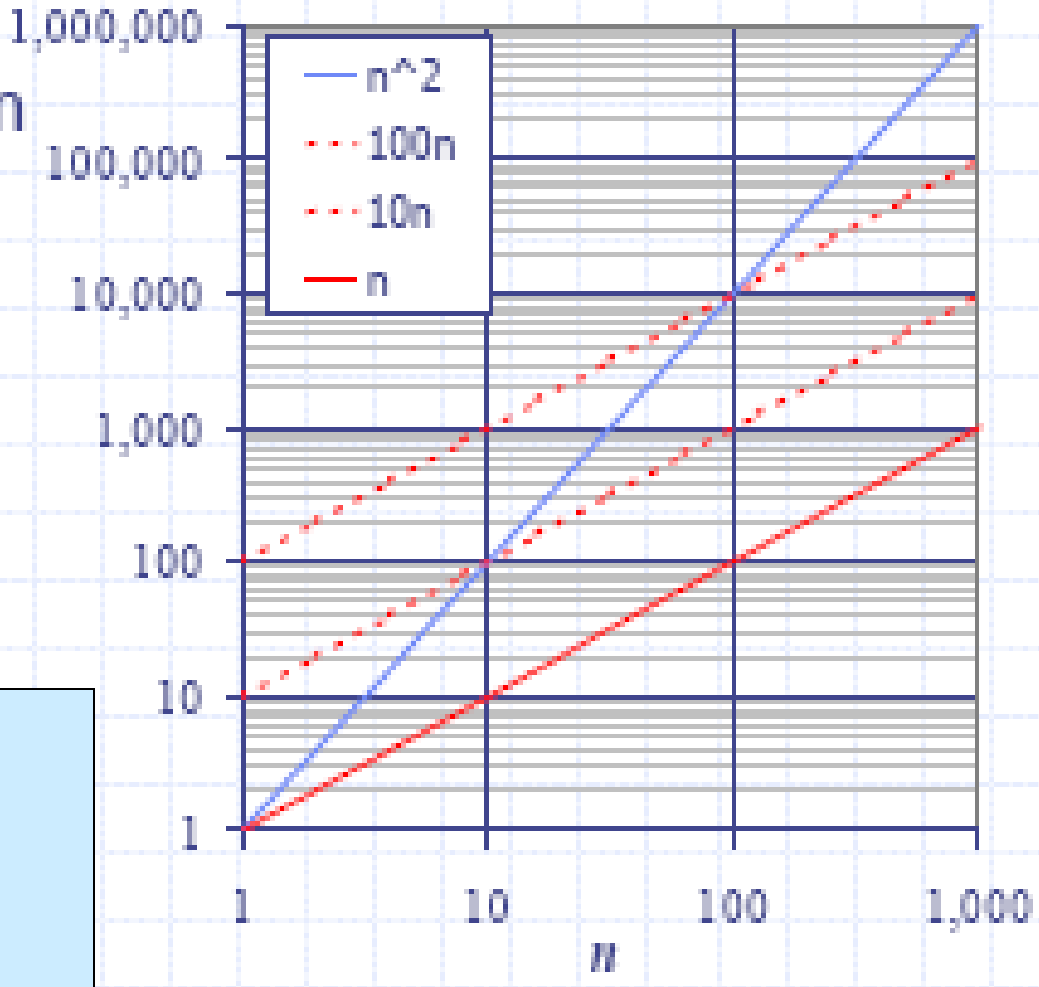
- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$



Big O notation is also called **Big Oh notation**, **Landau notation**, **Bachmann–Landau notation**, and **asymptotic notation**. A description of a function in terms of big O notation usually only provides an upper bound on the growth rate of the function; associated with big O notation are several related notations, using the symbols *o*, Ω, ω, and Θ, to describe other kinds of bounds on asymptotic growth rates.

Big-Oh Example

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



big-oh notation pronouncement :

O(1) : O to the one
O(n) : O to the n, Big-O of n

O($\log_2$ n): O to the $\log_2$ n

More Big-Oh Examples

- $7n-2$

  $7n-2$ is $O(n)$

  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \bullet n$ 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 $3n^3 + 20n^2 + 5 \leq c \bullet n^3$ 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 $3 \log n + 5 \leq c \bullet \log n$ for $n \geq n_0$

  this is true for $c = 8$ and $n_0 = 2$

Big-Oh and Growth Rate

- The big Ohnotation 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 Ohnotation 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

When we say that f(n) is O(g(n)), we use g(n) to provide a limit as to how fast f(n) can grow when n goes to infinity.

Properties of Big-Oh

The first theorem addresses the asymptotic behavior of the sum of two functions whose asymptotic behaviors are known:

Theorem If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then

$$f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n))).$$

The next theorem addresses the asymptotic behavior of the product of two functions whose asymptotic behaviors are known:

Theorem If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then

$$f_1(n) \times f_2(n) = O(g_1(n) \times g_2(n)).$$

The last theorem in this section introduces the *transitive property* of big oh:

Theorem (Transitive Property) If $f(n)=O(g(n))$ and $g(n)=O(h(n))$ then $f(n)=O(h(n))$.

Big-Oh Rules

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

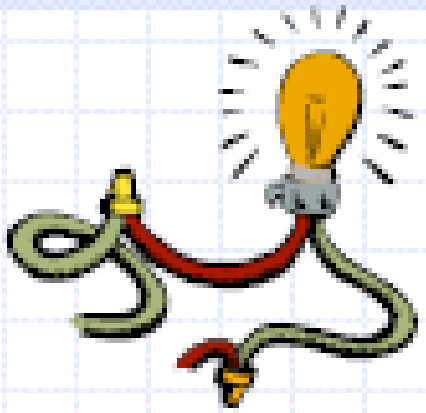
Asymptotic Algorithm Analysis

- 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

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 $f(n) \geq c \bullet g(n)$ 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 $c' \bullet g(n) \leq f(n) \leq c'' \bullet g(n)$ 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 Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

big-Theta

- f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

Example Uses of the Relatives of Big-Oh

- $5n^2$ is $\Omega(n^2)$

  $f(n)$ 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 $n \geq n_0$

  let $c = 5$ and $n_0 = 1$

- $5n^2$ is $\Omega(n)$

  $f(n)$ 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 $n \geq n_0$

  let $c = 1$ and $n_0 = 1$

- $5n^2$ is $\Theta(n^2)$

  $f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$

  Let $c = 5$ and $n_0 = 1$

More notes about Big-Oh notation

Using Big-O notation, we might say that “Algorithm A runs in time Big-O of n log n”, or that “Algorithm B is an order n-squared algorithm”. We mean that the number of operations, as a function of the input size n, is O(n log n) or $O(n^2)$ for these cases.

Notes about style: We write

“$T(n)$ is $O(g(n))$” or “$T(n) \in O(g(n))$”.

Both of these expressions are better than writing

“T(n) = O(g(n))”.

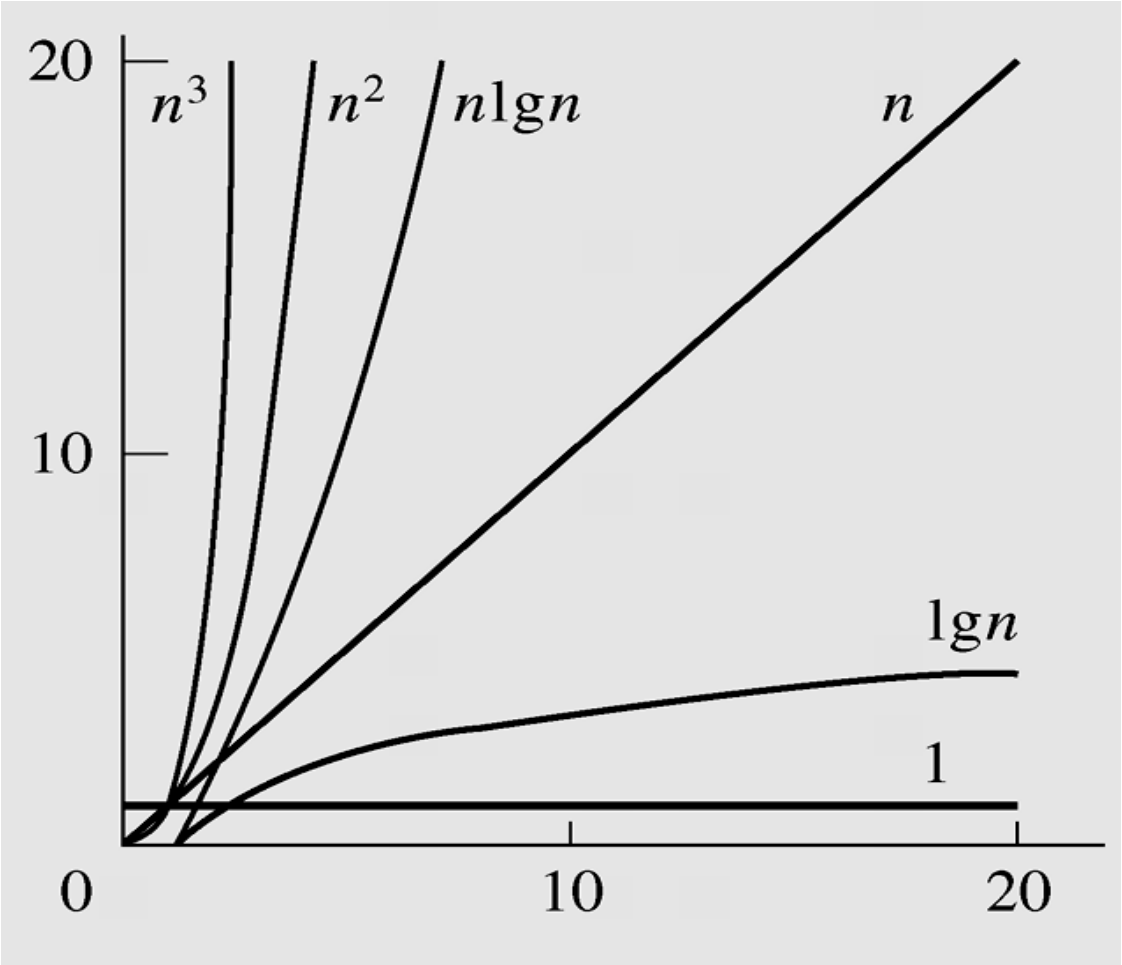
Some authors use the latter, but the first two choices are preferred.

Some common growth orders of functions

To express O(), we often use the following functions:

constant	$O(1)$
logarithmic	$O(\log n)$
linear	$O(n)$
nlogn	$O(n\log n)$
quadratic	$O(n^2)$
polynomial	$O(n^b)$
exponential	$O(b^n)$
factorial	$O(n!)$

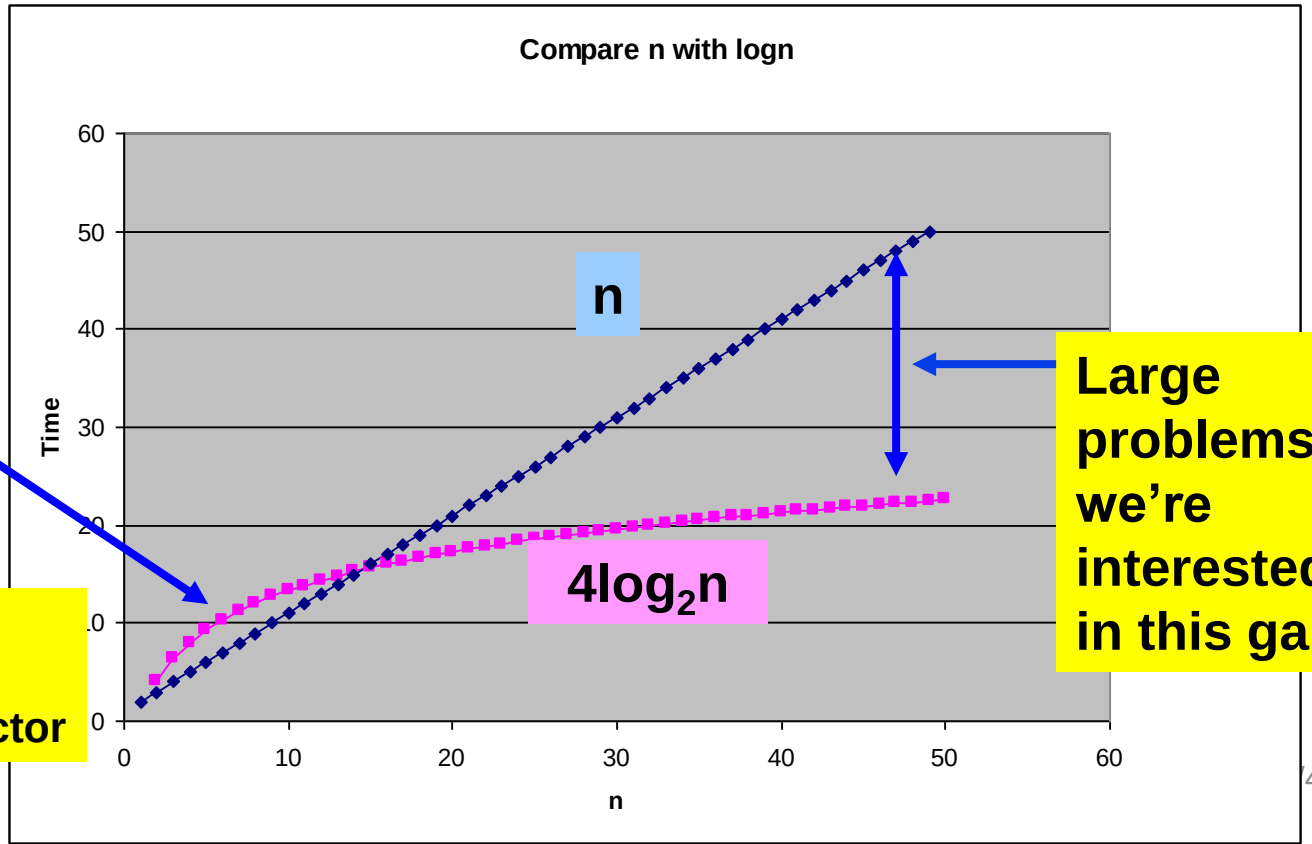
Growth orders of functions in graphics



Typical functions applied in big-O estimates

Binary Search vs Sequential Search

- `Find` method
  - Sequential
    - Worst case time: $c_1 n$
  - Binary search
    - Worst case time: $c_2 \log_2 n$



Small problems - we're not interested!

Binary search
More complex
Higher constant factor

Large problems - we're interested in this gap!

Determining time complexity

To determine time complexity:

- Count operations, and
- Convert to big-Oh notation.

We will use “pseudocode” to describe algorithms.

Examples for determining time complexity - 1

max returns the maximum element in a sequence.

procedure max(a_0 ,a_1, a_2, . . . , a_n: R)

  max := a_0

  for i := 1 to n

    if max < a_i then max := a_i

  end for

  return max

end procedure

The comparison max < a_i is performed n − 1 times.

n − 1 is O(n).

Examples for determining time complexity - 2

Linear search returns the location of key in a sequence, or -1 if key is not in the sequence.

procedure linear search (key, a_0, a_1, . . . , a_n: R)

```
   loc := -1
   i := 0
   while i <= n  and loc = -1
      if key = a_i then loc := i
         i := i + 1
   return loc
```

The comparison key = a_i is performed from 1 to n times.

Both 1 and n are O(n).

Examples for determining time complexity - 3

Binary search returns the location of key in a sorted sequence, or -1 if key is missing.

procedure binary search (key: R; a_0, a_1, . . . , a_n: increasing R)

```
  left := 0; right := n-1
  while left <= right
    begin
       mid := (left + right)/2
       if a_mid = key  return mid
       if key > mid
          then left := mid + 1
             else right := mid -1
     end
   return -1
```

key > amid is performed log2 n times. O(log2 n).

Amortized Complexity – Main idea

Worst case analysis of run time complexity is often too pessimistic. Average case analysis may be difficult because:

(i) it is not clear what is “average data",
(ii) uniformly random data is usually not average,
(iii) probabilistic arguments can be difficult.

Amortized complexity analysis is a different way to estimate run times. The main ideas is to spread out the cost of operations, charging more than necessary when they are cheap – thereby “saving up for later use" when they occasionally become expensive.

Amortized Complexity...

- **Worst case:**

$C(op_1, op_2, op_3, \ldots) = C_{worst}(op_1) + C_{worst}(op_2) + C_{worst}(op_3) + \ldots$

- **Average case:**

$C(op_1, op_2, op_3, \ldots) = C_{avg}(op_1) + C_{avg}(op_2) + C_{avg}(op_3) + \ldots$

- **Amortized**:

$C(op_1, op_2, op_3, \ldots) = C(op_1) + C(op_2) + C(op_3) + \ldots$

Where *C* can be worst, average, or best case complexity

Amortized Complexity...

Consider the operation of adding a new element to a flexible array a. Suppose number of elements in a is count, and the size of a is N. If count<N then the cost of the operation is O(1). If count=N then at first we should create new array with size = N + k and copy all elements on a to the new array and then add new element to new array. Copying all elements from a to new array costs O(count). If k=1 then after the first overflow, each insertion causes overflow and cost O(count). Clearly, this situation should be delayed. One solution is to double the space allocated for the array if overflow occurs. It may be claimed that, in the best case, the cost of inserting m items is O(m), but it is impossible to claim that, in the worst case, it is O(m*count). Therefore, to see better what impact this performance has on the sequence of operations, the amortized analysis should be used.

In amortized analysis, the question is asked: What is the expected efficiency of a sequence of insertions? We know that the best case is O(1) and the worst case is O(count), but also we know that the latter case occurs only occasionally and leads to doubling the size of the array. In this case what is the expected efficiency of one insertion in the series of insertions? We are interested in sequences of insertions to have the worst case senario. The outcome of amortized analysis depends on the assumed amortized cost of one insertion amCost(add(x)). It is clear that we cannot simply take amCost(add(x)=1, because it does not consider the overflow case. In general it is not adequate to choice constant k for amortized cost (i.e. amCost(add(x)=k). Define as *potencial* a function that assigns a number to a particular state of a data structure ds that is a subject of a sequence of operations. The amortized cost is defined as a function:

$$amCost(op_i) = Cost(op_i) + potencial(ds_i) - potencial(ds_{i-1})$$

which is the real cost of executing the operation op_i plus the chage in potencial in the data structure ds as a result of execution of op_i . ($potencial(ds_i)$ = 0 if $count_i = N_i$ and $2count_i - N_i$ otherwise).

In our case, it can be proved that $amCost(add_i() = 3$.

NP - Completeness

What is NP problem?

- The subject of *computational complexity theory* is dedicated to classifying problems by how hard they are. There are many different classifications; some of the most common and useful are the following.
- **P**. Problems that can be solved in polynomial time. ("P" stands for polynomial.) These problems have formed the main material of this course.
- **NP**. This stands for "***nondeterministic polynomial time***" where nondeterministic is just a fancy way of talking about guessing a solution. A problem is in NP if you can quickly (in polynomial time) test whether a solution is correct (without worrying about how hard it might be to find the solution). Problems in NP are still relatively easy: if only we could guess the right solution, we could then quickly test it.

unsolved problem: P=NP?

Reduction (1)

- Formally, NP-completeness is defined in terms of "reduction" which is just a complicated way of saying one problem is easier than another. We say that A is easier than B, and write A < B, if we can write down an algorithm for ***solving A that uses a small number of calls to a subroutine for B*** (with everything outside the subroutine calls being fast, polynomial time). There are several minor variations of this definition depending on the detailed meaning of "small" -- it may be a polynomial number of calls, a fixed constant number, or just one call.
- Then if A < B, and B is in P, so is A: we can write down a polynomial algorithm for A by expanding the subroutine calls to use the fast algorithm for B.
- So "easier" in this context means that if one problem can be solved in polynomial time, so can the other. It is possible for the algorithms for A to be slower than those for B, even though A < B.

Reduction (2)

- As an example, consider the Hamiltonian cycle problem. Does a given graph have a cycle visiting each vertex exactly once? Here's a solution, using longest path as a subroutine:

  for each edge (u,v) of G

  if there is a simple path of length n-1 from u to v

  return yes // path + edge form a cycle

  return no

  This algorithm makes m calls to a longest path subroutine, and does O(m) work outside those subroutine calls, so it shows that Hamiltonian cycle < longest path. (It doesn't show that Hamiltonian cycle is in P, because we don't know how to solve the longest path subproblems quickly.)

What is a NP-complete problem?

We are now ready to formally define NP-completeness.

We say that a problem A in NP is NP-complete when

1. It is NP
2. For every other problem B in NP, B < A.

This seems like a very strong definition. After all, the notion of reduction we've defined above seems to imply that if B < A, then the two problems are very closely related; for instance Hamiltonian cycle and longest path are both about finding very similar structures in graphs. Why should there be a problem that closely related to all the different problems in NP?

Cook's theorem: an NP-complete problem exists.

How to prove NP-completeness in practice

It is easily to prove that if A < B and B < C, then A < C.

As a consequence of this observation,

if A is NP-complete, B is in NP, and A < B, B is NP-complete.

In practice that's how we prove NP-completeness: We start with one specific problem that we prove NP-complete, and we then prove that it's easier than lots of others which must therefore also be NP-complete.

So e.g. since Hamiltonian cycle is known to be NP-complete, and Hamiltonian cycle < longest path, we can deduce that longest path is also NP-complete.