Data Structures and Algorithms

Alemitu M, and Ashenafi Ch.

Addis Ababa Science & Technology University

2011 E.C., Semester III

Chapter One: Introduction

- Outline:
  - Introduction
  - Abstract Data Type & Data Structure
  - Algorithm
  - Algorithm Analysis

Introduction

A program is written in order to solve a problem.

A solution to a problem actually consists of two things:

- A way to organize the data in memory : Data Structure
- Sequence of steps to solve the problem: Algorithm

▪ Program = Data structure + Algorithm

Why Study Data Structures & Algorithms?

- A primary concern of this course is **efficiency**.

  Efficient data structures
  Efficient Algorithms
  ⇒ Efficient programs

- [You might believe that faster computers make it unnecessary to be concerned with efficiency.]
- More complex applications ⇒ More powerful computers
- **YET more complex applications** demand more calculations, computationally infeasible: taking years, even millions of years.
- So we need special training, efficient program design.

Why…?

- Any organization for a collection of records can be searched, processed in any order, or modified.
- [If you are willing to pay enough in time delay.

  - Ex: Simple unordered array of records.]

- **The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days.**

Efficiency

- A solution is said to be **efficient** if it solves the problem within its **resource constraints**.
- [Alt defn: Better than known alternatives ("relatively" efficient)]
  - Space
  - Time

  [These are typical constraints for programs]
- *[This does not mean always strive for the most efficient program. If the program operates well within resource constraints, there is no benefit to making it faster or smaller.]*
- The **cost** of a solution is the amount of resources that the solution consumes.

Selecting a Data Structure

Select a data structure as follows:

1. Analyze the problem to determine the resource constraints a solution must meet.
2. Determine the basic operations that must be supported. Quantify the resource constraints for each operation.
3. Select the data structure that best meets these requirements.

- [Typically we want the “simplest” data structure that will meet the requirements.]

Some Questions to Ask

- These questions often help to narrow the possibilities
  - Are all data inserted into the data structure at the beginning, or are insertions interspersed with other operations?
  - Can data be deleted? [If so, a more complex representation is typically required]
  - Are all data processed in some well-defined order, or is random access allowed?

Data Structure Philosophy

- Each data structure has costs and benefits.
- Rarely is one data structure better than another in all situations.
- A data structure requires:
  - space for each data item it stores [Data + Overhead],
  - time to perform each basic operation,
  - programming effort. [Some data structures/ algorithms more complicated than others]

Data Structure Philosophy . . .

- Each problem has constraints on available space and time.
- Only after a careful analysis of problem characteristics can we know the best data structure for the task.
- Bank example:
  - Start account: should take only a few minutes
  - Transactions: should take only a few seconds
  - Close/delete account: can take overnight

$\Downarrow$

We should choose a data structure that has little concern for the cost of deletion, but is highly efficient for search and moderately efficient for insertion.

Goals of this Course:

1. Learn the commonly used data structures.
    - These form a programmer's basic data structure ``toolkit.''
2. Reinforce the concept that costs and benefits exist for every data structure.
3. Learn commonly used Algorithms
4. Understand how to measure the cost of a data structure or algorithm.

Definition of Terms

- A **type** is a collection of values. E.g. :
  - Boolean – {true, false}
  - Integer – {-,0,+}
  - Float – {integers, fractions}
- A **data item** is a piece of information or a record whose value drawn from a type.
- A data item is said to be a **member** of a type. [e.g. 2]
- A **simple data item** contains no subparts. [e.g: Integer]
- An **aggregate data item** may contain several pieces of information. e.g: bank account : name, address, account number, balance
- A **data type** is a type and a collection of operations that manipulate the type. [e.g: Addition]

Abstract Data Types

Abstract Data Type (ADT): a definition for a data type solely in terms of a set of values and a set of operations on that data type.

- Realization of a data type as a software component.
- ADT specifies:
  1. What can be stored in the Abstract Data Type – **attributes.**
  2. What **operations** can be done on/by the Abstract Data Type.

E.g: model employees of an organization.

- **Data**: stores employees with relevant attributes.
- **Operations**: supports hiring, firing, retiring, . . .

Encapsulation: Hide implementation details.

Data Structure

Data structure: is the physical implementation of an ADT.

- Each operation associated with the ADT is implemented by one or more subroutines in the implementation – function member or method.
- The variables that define the space required by a data item are referred to as data members (attributes).

- **Data structure** usually refers to an organization for data in main memory.
- **File structure**: an organization for data on peripheral storage, such as a disk drive or tape.

ADT

- **Model** of a data type.
- Collection of data plus operation on that data.
- From users point of view.
- One: Can have different implementations.
- Separates data type definition from its implementation.
- Implementation-independent
- It is like Java Interface.

Data Structure

- Organize the data that the ADT is encapsulating.
  - Concrete implementation
- Data member + functions.
- Programmers point of view.
- Many.

Example: List ADT

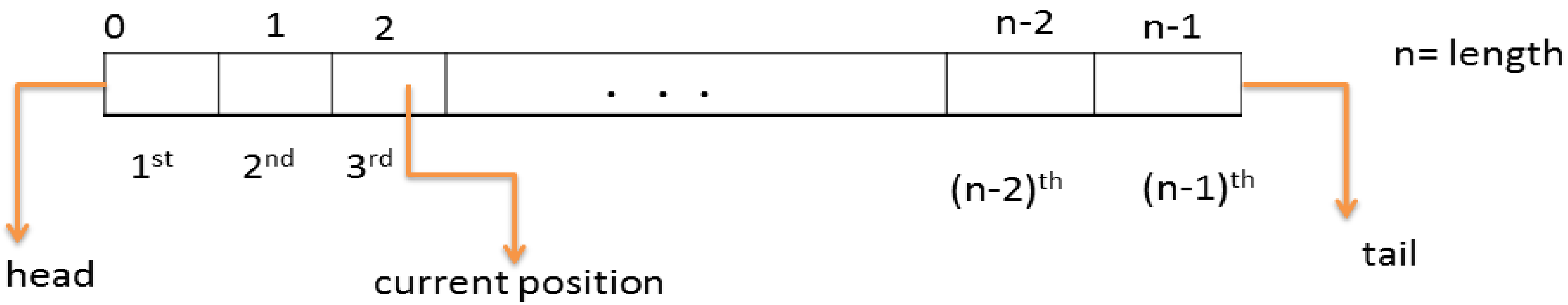
- A list is a finite, ordered sequence of data items called elements, each having a position.
- Let us define an ADT for list of integers:
- Type (values): integer, i.e. list of integers
- Operations:
  - Insert a new integer at a particular position in the list.
  - Return **true** if the list is empty.
  - Reinitialize the list.
  - Return the number of integers currently in the list.
  - Delete the integer at a particular position in the list.

=>No specification of the list's implementation

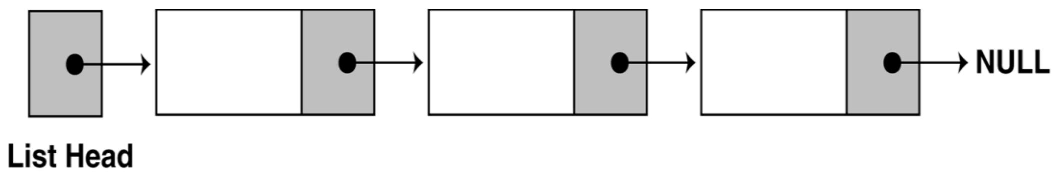
Example: Implementation of the above list ADT

Two ways of doing that:

- **Array-based**: contiguous memory allocation



- **Using linked lists**: dynamic allocation



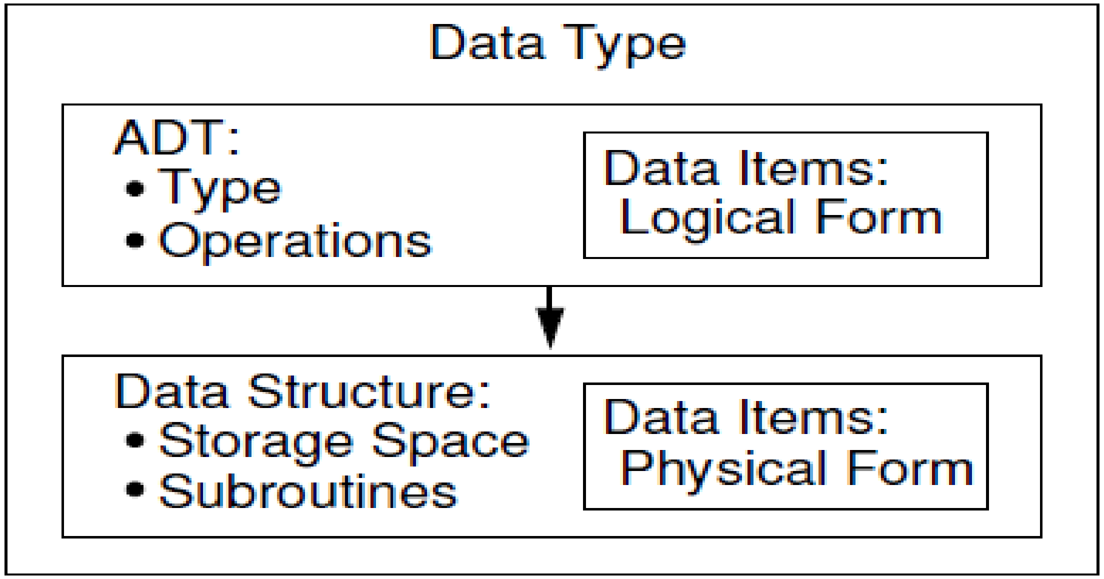
Therefore,

- A List is an ADT, with well defined operations (add, delete, search, etc.) while a linked list is a pointer based data structure that can be used to create a representation of a List.

- **Note:**
  - Different applications have different requirements in using operations (functions) of an ADT (frequency, running time).
  - This is why an ADT might have more than one implementations.

Logical & Physical Forms of a Data Type

- Data types have both a logical and a physical form.
  - The definition of the data type in terms of an ADT is its logical form.
  - The implementation of the data type as a data structure is its physical form.



When you implement an ADT, you are dealing with the physical form of the associated data type.

When you use an ADT elsewhere in your program, you are concerned with the associated data type's logical form.

Abstraction:

- An ADT manages complexity through abstraction, metaphor. [Hierarchies of labels]
- [Ex: transistors → gates → CPU.]
- In a program, implement an ADT, then think only about the ADT, not its implementation].

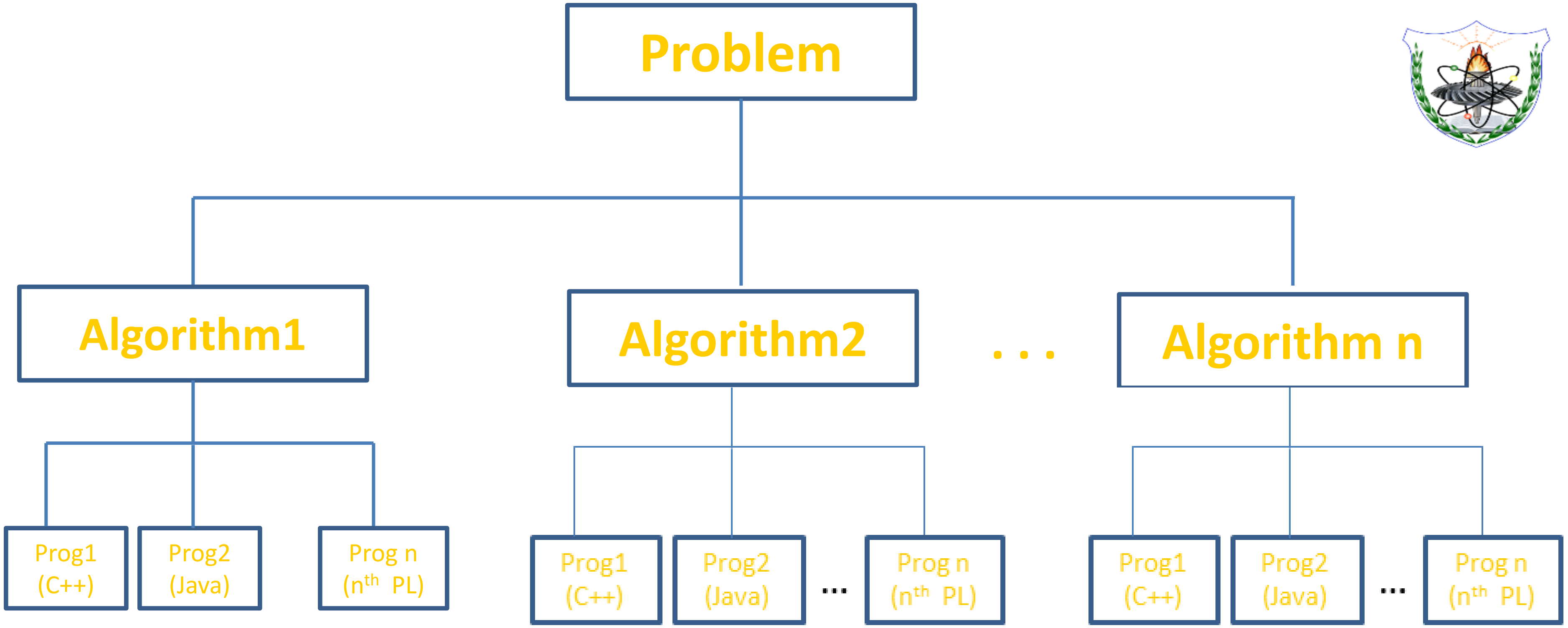
Problems

- Problem: a task to be performed.
  - Best thought of as inputs and matching outputs.
  - Problem definition should include constraints on the resources that may be consumed by any acceptable solution.
  - [But NO constraints on HOW the problem is solved].
- Problems ⇔ mathematical functions
  - A function is a matching between inputs (the **domain**) and outputs (the range).
  - An input to a function may be single number, or a collection of information.
  - The values making up an input are called the **parameters** of the function.
  - A particular input must always result in the same output every time the function is computed.

Algorithms

- What is Algorithm?
  - a clearly specified **set of simple instructions** to be followed to solve a problem.
    - Takes a set of values, as input and
    - produces a value, or set of values, as output
  - May be specified
    - In English
    - As a computer program
    - As a pseudo-code

- A problem can have many algorithms (i.e. solutions).
- **Computer program**: is an instance, or concrete representation, for an algorithm in some programming language.



An Algorithm may be efficient than another for a specific variation of the problem, or for a specific class of inputs.

E.g.: One sorting algorithm might be the best for sorting a *small* collection of integers, another a *large* collection of integers, and a third for sorting a collection of *variable-length strings*.

Algorithm Properties

An algorithm possesses the following properties:

1. It must be **correct**. [Computes proper function].
2. It must be composed of a series of **concrete steps**. [Each step is completely understood & doable by that machine]
3. There can be **no ambiguity** as to which step will be performed next.
4. It must be composed of a **finite** number of steps.
5. It must **terminate**.[Must not go into an *infinite loop*]

Algorithm Analysis

- There are often many approaches (algorithms) to solve a problem. *How do we choose between them?*
- At the heart of computer program design are two (sometimes conflicting) goals:

   1. To design an algorithm that is easy to understand, code and debug.
   2. To design an algorithm that makes efficient use of the computer's resources.

- Goal (1) is the concern of Software Engineering.
- Goal (2) is the concern of data structures and algorithm analysis.

- When goal (2) is important, how do we measure an algorithm's cost?

How to Measure Efficiency?

Two approaches:

1. Empirical comparison (run programs) [Difficult to do "fairly." Time consuming.]
2. Asymptotic Algorithm Analysis

- First method is unsatisfactory:
  - Programming effort.
  - one of the programs might be "better written" than the other.
  - Unfair choice of test cases.

How to Measure Efficiency?

- Factors affecting running time:
  - Machine load
  - OS
  - Compiler
  - Problem size or Specific input values for given problem size.
- Critical resources for a program:
  - Time: required for an algorithm
  - Space (disk, RAM): required for a data structure
  - Programmer’s effort
  - Ease of use (user’s effort)

Complexity Analysis

Two things to consider:

- **Time Complexity**: Determine the approximate number of operations required to solve a problem of size n.
- **Space Complexity**: Determine the approximate memory required to solve a problem of size n.

- For most algorithms, running time depends on “size” of the input.
- Running time is expressed as T(n) for some function T on input size n.

Analysis Rules

[No generally accepted set of rules for algorithm analysis, but an exact count of operations is commonly used.]

1. We assume an arbitrary time unit.
2. Execution of one of the following operations takes time 1:
    - Assignment Operation
    - Single Input/Output Operation
    - Single Boolean Operations
    - Single Arithmetic Operations
    - Function Return
3. Running time of a selection statement (if, switch) is the time for the condition evaluation + the maximum of the running times for the individual clauses in the selection.

Analysis Rules

4. Loops: Running time for a loop is:

(running time for the statements inside the loop * number of iterations)

- The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the loops.
- For nested loops, analyze inside out.
- Always assume that the loop executes the maximum number of iterations possible.

5. Running time of a function call is 1 for setup + the time for any parameter calculations + the time required for the execution of the function body.

Examples

1.

```
int count(){
 int k=0;
 cout<<"Enter an integer";
 cin>>n;
 for (i=0;i<n;i++)
   k=k+1;
  return 0;
}
```

- 1 for the assignment statement: int k=0
- 1 for the output statement.
- 1 for the input statement.
- In the for loop:
  - 1 assignment, *n+1* tests, and *n* increments.
  - *n* loops of 2 units for an assignment, and an addition.
- 1 for the return statement.

T(n) = 1+1+1+(1+(n+1)+n)+2n+1)= 4n+6

Examples…

2.

```
void func(){
 int x=0;
 int i=0;
 int j=1;
 cout<<"Enter an Integer";
 cin>>n;
 while (i<n){
   x++;
   i++;
  }
 while (j<n){
   j++;
  }
}
```

- 1 –each 3 assign stmts
- 1– output stmt
- 1 – input stmt.
- 1st while loop:
  - *n+1* tests
  - *n* loops of 2 units for the two increment (addition) operations
- 2nd while loop:
  - n tests
  - n-1 increments

T (n)= *1+1+1+1+1+n+1+2n+n+n-1 = 5n+5*

Examples...

3.

```
sum = 0;
for (i=1; i<=n; i++)
 for (j=1; j<=n; j++)
     sum++;
```

- 1- assign stmt
- Inner loop:
  - 1 assign, n+1 tests, and n increments.
  - Loop executes n times
- Outer loop:
  - 1 assign, n+1 tests, and n increments.
  - Loop executes n times
- Stmt inside nested loop:
  - 2 units, assign & increment
  - Total: 2 *(n*n) –>size of both loops

$T(n)= 1+2(1+(n+1)+n)+2n^2$

$= 2n^2 +4n+5$

Formal Approach to Analysis

- We can simplify analysis by ignoring:
  - Initializations
  - Loop control condition
  - Loop counter increments, etc.

- Primary consideration when estimating an algorithm's performance is the number of basic operations required by the algorithm to process an input of a certain size.

Formal Approach to Analysis

- Size: the number of inputs processed.
    - E.g. Sorting algorithms, size is the number of records to be sorted.
- A basic operation must have the property that its time to complete does not depend on the particular values of its operands.
    - E.g.
        - » Adding or comparing two integer variables.
        - » Summing the contents of an array containing n integers is not, because the cost depends on the value of n (i.e. the size of the input).

`for` Loops

- A for loop translates to a summation.
- The index and bounds of the summation are the same as the index and bounds of the for loop.

```
for (int i = 1; i<=N; i++){
    sum = sum + i;
}
```

$$\Rightarrow \sum_{i=1}^{N} 1 = N$$

- Suppose we count the number of additions that are done.
- There is 1 addition per iteration of the loop, hence *N* additions in total.

Nested Loops

- Nested for loops translate into multiple summations, one for each for loop.

```
for (int i = 1; i <= N; i++) {
      for (int j = 1; j <= M; j++) {
              sum = sum+i+j;
      }
}
```

$$\Rightarrow \sum_{i=1}^{N}\sum_{j=1}^{M} 2 = \sum_{i=1}^{N} 2M = 2MN$$

- Again, count the number of additions. The outer summation is for the outer for loop.

Consecutive Statements

- Add the running times of the separate blocks

```
for (int i = 1; i <= N; i++) {
     sum = sum+i;
}
for (int i = 1; i <= N; i++) {
     for (int j = 1; j <= N; j++) {
          sum = sum+i+j;
     }
}
```

$$\left[\sum_{i=1}^{N} 1\right] + \left[\sum_{i=1}^{N}\sum_{j=1}^{N} 2\right] = N + 2N^2$$

Conditional Statements

- If (test) s1 else s2: Compute the maximum of the running time for s1 and s2.

```
if (test == 1) {
    for (int i = 1; i <= N; i++) {
         sum = sum+i;
}}
else for (int i = 1; i <= N; i++) {
              for (int j = 1; j <= N; j++) {
                     sum = sum+i+j;
}}
```

$$\max\left(\sum_{i=1}^{N} 1, \sum_{i=1}^{N}\sum_{j=1}^{N} 2\right) =$$

$$\max\left(N, 2N^2\right) = 2N^2$$

Examples

E.g.1: [As n grows, how does T(n) grow?]

```
static int largest(int[] array) { // Find largest val
                                  // all values >=0
  int currLargest = 0;  // Store largest val
  for (int i=0; i<array.length; i++) // For each elem
    if (array[i] > currLargest)      //   if largest
      currLargest = array[i];        //     remember it
  return currLargest;                // Return largest val
}
```

alem

=>the **size** of the problem is:
n= array.length

=>**basic operation**: comparing each value with current largest

[Cost: T teps]

T(n) = cn

Examples

- **E.g.2**: Assignment statement

$$\boxed{T(n) = C}$$

- The size of the input n has no effect on the running time. This is called a **constant running time**.
- **E.g.3**:

```
sum = 0;
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
        sum++;
```

alem

takes longer to run when n is larger

The basic operation in this example is the increment operation for variable sum.

$$\mathbf{T}(n) = c_2 n^2.$$

Asymptotic Analysis

- Estimates the time or space required by a program as function of the input size.

- We analyze:
  - *time* required for an *algorithm.*
  - the *space* required for a *data structure*.

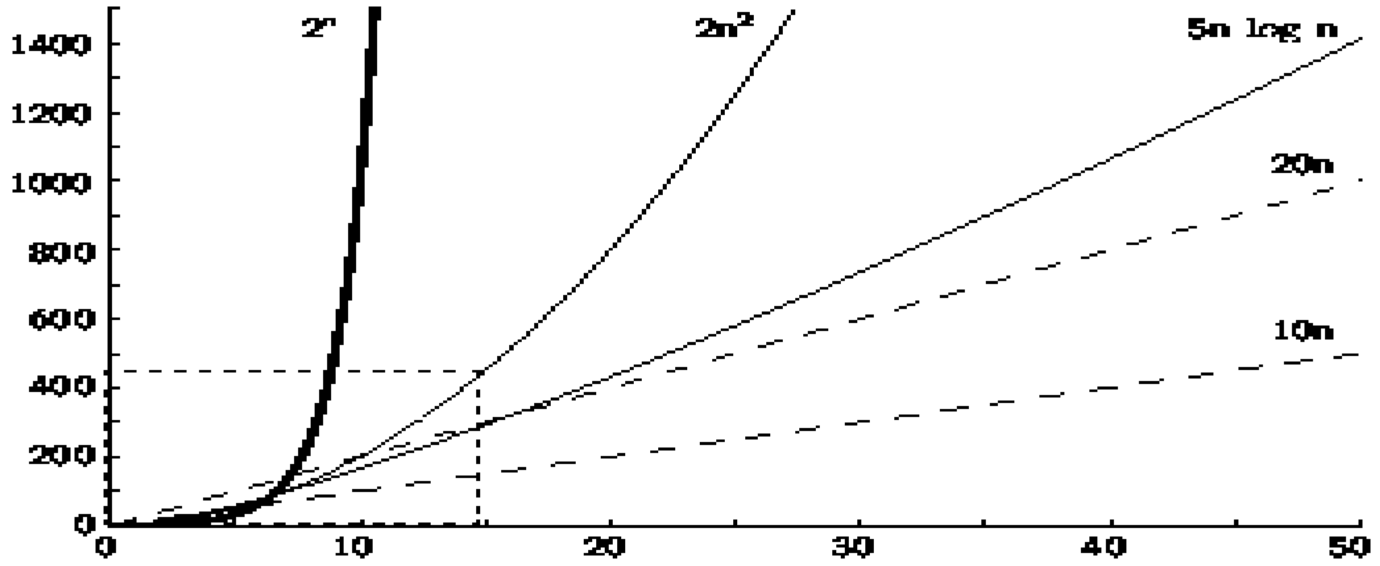
Growth Rate

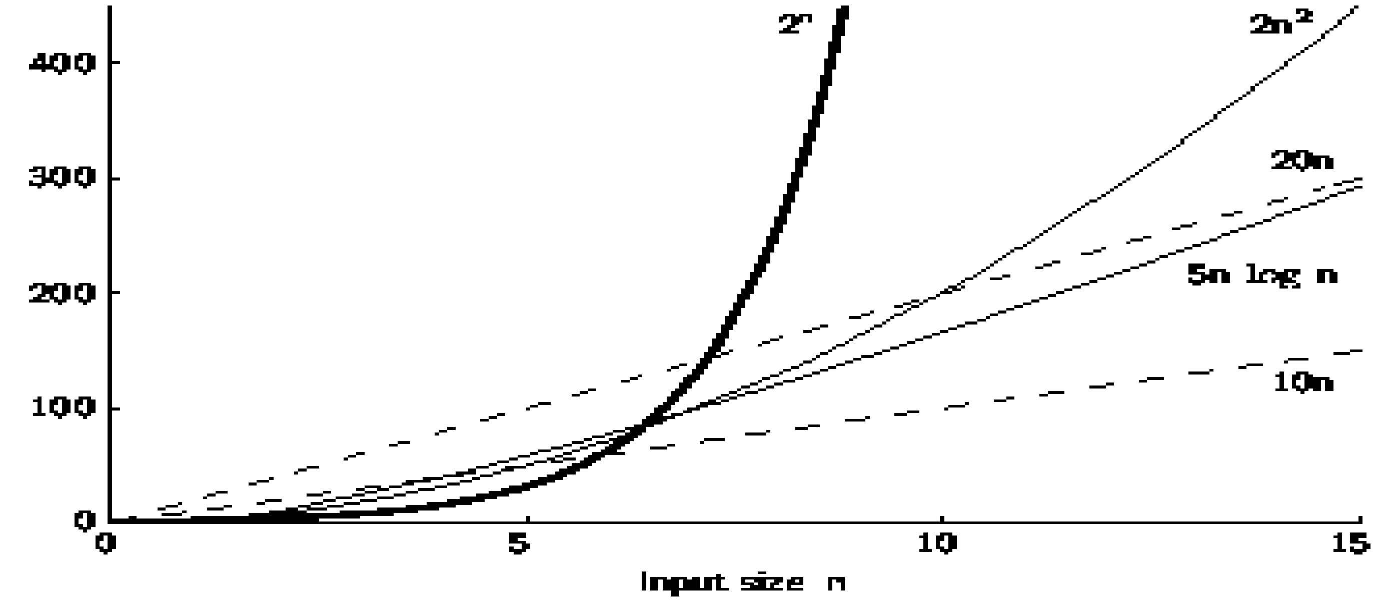
- The growth rate for an algorithm is the rate at which the cost of the algorithm grows as the size of its input grows.

Examples:

T(n)	Type
c	constant
log n	logarithmic
n	linear
n log n	"n log n"
n^2	quadratic
n^3	cubic
2^n	exponential

Growth Rate Graph





Best, Worst and Average Cases

- Largest-value sequential search algorithm - always examines every array value.

▣ However, for some algorithms, different inputs require different amounts of time.

E.g. Sequential search for K in an array of n integers:

- Begin at first element in array and look at each element in turn until K is found.

Best case: Find at first position. Cost is 1 compare.

Worst case: Find at last position. Cost is *n* compares.

Average case:(*n*+1)/2 compares if we assume the element with value *K* is equally likely to be in any position in the array.

Asymptotic Analysis: Upper Bounds

- Several terms are used to describe the running-time equation for an algorithm.

- Upper bound - indicates the upper or highest growth rate that an algorithm can have.
- Measured on the best-case, average-case, or worst-case inputs.
- **Big-Oh notation:**
  - If the upper bound for an algorithm's growth rate (say, the worst case) is f(n), then this algorithm is "in the set O(f(n)) in the worst case"

  (or just "in O(f(n)) in the worst case").

Asymptotic Analysis: Big-oh Notation

Definition: For **T**(n) a non-negatively valued function, **T**(n) is in the set O($f(n)$) if there exist two positive constants c and n_0 such that **T**(n) <= $cf(n)$ for all $n > n_0$.

- n_0 is the smallest value of n for which the claim of an upper bound holds true.

Usage: The algorithm is in O(n^2) in [best, average, worst] case.

[Must pick one of these to complete the statement. Big-oh notation applies to some set of inputs.]

Meaning: For all data sets big enough (i.e., $n>n_0$), the algorithm always executes in less than $c\ f(n)$ steps in [best, average, worst] case.

Big-oh Notation (cont.)

- Big-oh notation indicates an upper bound.

  T(n) grows at a rate no faster than *f* (n)

- …"is in O(f(n))" or " Є O(f(n))", no strict equality actually.

- Example: If **T**(n) = 3n^2 then **T**(n) is in O(n^2).

Wish tightest upper bound:

While **T**(n) = 3n^2 is in O(n^3), we prefer O(n^2).

Big-Oh Examples

- Example 1: Finding value *X* in an array. (average cost, assuming equal probability of appearing in any position).

  T(n) = $c_s n/2$.

  For all values of $n > 1$, $c_s n/2 <= c_s n$.
- Therefore, by the definition, **T**(n) is in O(n) for $n_0 = 1$ and $c = c_s$.

Big-Oh Examples

- Example 2: $\mathbf{T}(n) = c_1n^2 + c_2n$ in average case.

  $c_1n^2 + c_2n <= c_1n^2 + c_2n^2 <= (c_1 + c_2)n^2$ for all $n > 1$.

  $\mathbf{T}(n) <= cn^2$ for $c = c_1 + c_2$ and $n_0 = 1$.

- Therefore, $\mathbf{T}(n)$ is in $O(n^2)$ by the definition.

Example 3: $\mathbf{T}(n) = c$. We say this is in O(1).

A Common Misunderstanding

- “The best case for my algorithm is n=1 because that is the fastest.” WRONG!

- Big-oh refers to a growth rate as n grows to ∞.
    - It states a claim about the greatest amount of some resource (usually time) that is required by an algorithm for some class of inputs of size n (typically the worst such input, the average of all possible inputs, or the best such input).
- Best case is defined as which input of size n is cheapest among all inputs of size n.

Lower Bound

- Describes the least amount of a resource that an algorithm needs for some class of input.
- Like big-Oh notation, this is a measure of the algorithm’s growth rate.
- Measured for some particular class of inputs:

  the worst-, average-, or best-case input of size n.

Big-Omega

Definition: For **T**(n) a non-negatively valued function, **T**(n) is in the set $\Omega(g(n))$ if there exist two positive constants c and n_0 such that **T**(n) >= $cg(n)$ for all $n > n_0$.

Meaning: For all data sets big enough (i.e., $n > n_0$), the algorithm always executes in more than $cg(n)$ steps.

wish to get the "tightest" (for Ω notation, the largest) bound possible.

Big-Omega Example

$\mathbf{T}(n) = c_1n^2 + c_2n$.

$c_1n^2 + c_2n >= c_1n^2$ for all $n > 1$.
$\mathbf{T}(n) >= cn^2$ for $c = c_1$ and $n_0 = 1$.

Therefore, $\mathbf{T}(n)$ is in $\Omega(n^2)$ by the definition.

We want the greatest lower bound.

Theta Notation

- When big-Oh and Ω meet, we indicate this by using Θ (big-Theta) notation.

- Definition: An algorithm is said to be $\Theta(h(n))$ if it is in $O(h(n))$ and it is in $\Omega(h(n))$.

- For polynomial equations on T(n), we always have Θ. There is no uncertainty, since once we have the equation, we have a "complete" analysis.

A Common Misunderstanding

- Confusing worst case with upper bound:
- Upper bound refers to a growth rate.
- Worst case refers to the worst input from among the choices for possible inputs of a given size.

Simplifying Rules

(1) If $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$.

[if some function g(n) is an upper bound for your cost function, then any upper bound for g(n) is also an upper bound for your cost function.]

(2) If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$. [Ignore constants]

(3) If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $(f_1 + f_2)(n)$ is in $O(\max(g_1(n), g_2(n)))$.

[Given two parts of a program run in sequence consider only the more expensive part. Drop lower order terms]

(4) If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$ then $f_1(n)f_2(n)$ is in $O(g_1(n)g_2(n))$.

[If some action is repeated some number of times, and each repetition has the same cost, then the total cost is the cost of the action multiplied by the number of times that the action takes place. Useful for analyzing loops]

Simplifying Rules (cont.)

- Taking the first three rules, ignore all constants and all lower-order terms to determine the asymptotic growth rate for any cost function.
- The higher-order terms soon swamp the lower-order terms in their contribution to the total cost as n becomes larger.
- Example: if $T(n) = 3n^4 + 5n^2$, then $T(n)$ is in $O(n^4)$. n^2 contributes relatively little to the total cost.

Running Time Examples (1)

[Asymptotic analysis is defined for equations. We need to convert programs to equations to analyze them.]

Example 1: `a = b;`

[This assignment takes constant time, so it is $\Theta(1)$.]

Example 2:

```
sum = 0;
for (i=1; i<=n; i++)
   sum += n;
```

[1st line is $\Theta(1)$. The **for** loop is repeated n times. 3rd line takes constant time so, by rule (4), total cost for the two lines is $\Theta(n)$. By rule (3), the cost of the entire code fragment is also $\Theta(n)$.]

Running Time Examples (2)

Example 3:

```
sum = 0;
for (i=1; i<=n; j++)   //first for loop
  for (j=1; j<=i; i++)// is double loop
       sum++;
for (k=0; k<n; k++)//second for loop
       A[k] = k;
```

[First statement is c_1 =>$\Theta(1)$.

Double for loop :

- **sum++** requires constant time,c_2
- inner **for** loop is executed i times, by rule (4), $c_2 i$
- outer **for** loop is executed n times, but each time the cost of the inner loop is different ,$c_2 i$ with i changing each time.

-Thus, the total cost of the loop is c_2 times the sum of the integers 1 through n: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. => $\Theta(n^2)$.

Final for loop is $c_3 n = \Theta(n)$ time $\Theta(n)$. **Result: $\Theta(n^2)$.**]

Running Time Examples (3)

Example 4: Compare.

```
sum1 = 0;
for (i=1; i<=n; i++)
   for (j=1; j<=n; j++)
      sum1++;

sum2 = 0;
for (i=1; i<=n; i++)
   for (j=1; j<=i; j++)
      sum2++;
```

In the first double loop, the inner **for** loop always executes n times

In the second double loop, the inner **for** loop always executes i times

[First loop, sum is n^2. Second loop, sum is $(n+1)(n)/2$. Both are $\Theta(n^2)$.]

Other Control Statements

`while` loop: Analyze like a `for` loop.

`if` statement: Take greater complexity of `then`/`else` clauses.

[The probabilities for the then/else clauses being executed must be independent of *n*.]

`switch` statement: Take complexity of most expensive case.

Function call: add the cost of the subroutine.

Space Bounds

Space bounds can also be analyzed with asymptotic complexity analysis.

Time: Algorithm
Space: Data Structure

[While time requirements are normally measured for an algorithm that manipulates a particular data structure, space requirements are normally determined for the data structure itself.]

- A data structure's primary purpose is to store data in a way that allows efficient access to those data.
- To provide this, it stores additional information where the data are – **overhead.**

Examples

- E.g.1:
- What are the space requirements for an array of n integers?
  - If each integer requires c bytes, then the array requires cn bytes, which is $\Theta(n)$

Examples

- E.g. 2: Imagine that we want to keep track of friendships between n people. We can do this with an array of size nxn.
- Each row of the array represents the friends of an individual, with the columns indicating who has that individual as a friend.
- For example, if person j is a friend of person i, then we place a mark in column j of row i in the array.
- Likewise, we should also place a mark in column i of row j if we assume that friendship works both ways.
- For n people, the total size of the array is $\Theta(n^2)$.

Space/Time Tradeoff Principle

One can often reduce time if one is willing to sacrifice space, or vice versa.

- Encoding or packing information:- reduces storage, but Unpacking" or decoding the requires additional time.

- Lookup Table: for calculating factorials

[For a program that often computes factorials, it is likely to be much more time efficient to simply pre-compute the storable values in a table (up to 12! for 32-bit **`int`**)]

Disk-based Space/Time Tradeoff Principle: The smaller you make the disk storage requirements, the faster your program will run.

- This is because the time to read information from disk is enormous compared to computation time.