COMP5481
Programming and Problem Solving

1.Algorithm Analysis

Chap 4

Today

- Algorithm Efficiency

  - Beyond Experimental Analysis
- Growth Functions and Big-O Notation
- Asymptotic Analysis
  - Comparing Growth functions

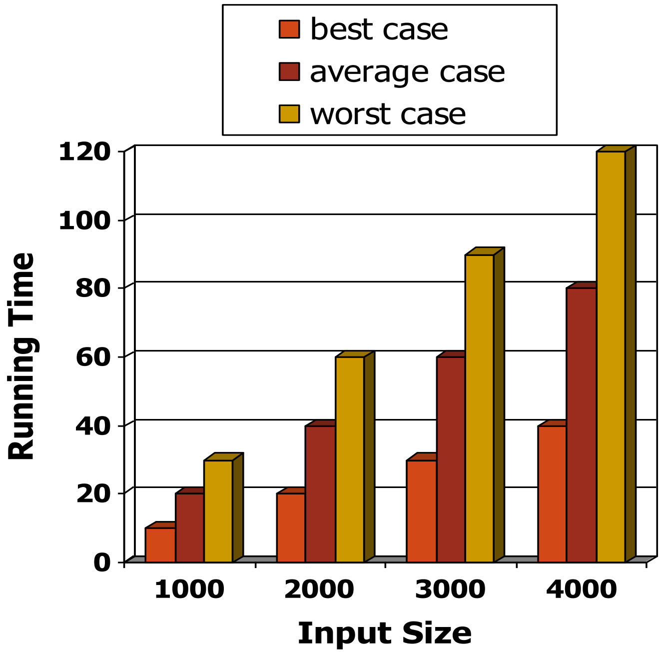
How to estimate Efficiency?

- Correctness of a method depends merely on whether the algorithm performs what it is supposed to do.
- Clearly, efficiency ≠ correctness.
- One algorithm can be said to be more efficiency than another if ….
  - Less memory utilization
  - Faster execution time
  - Quicker release of allocated recourses
  - etc.

- How do we measure efficiency?
  - Measurement should be independent of used software (i.e. compiler, language, etc.) and hardware (CPU speed, memory size, etc.)
  - Particularly, run-time analysis can have serious weaknesses

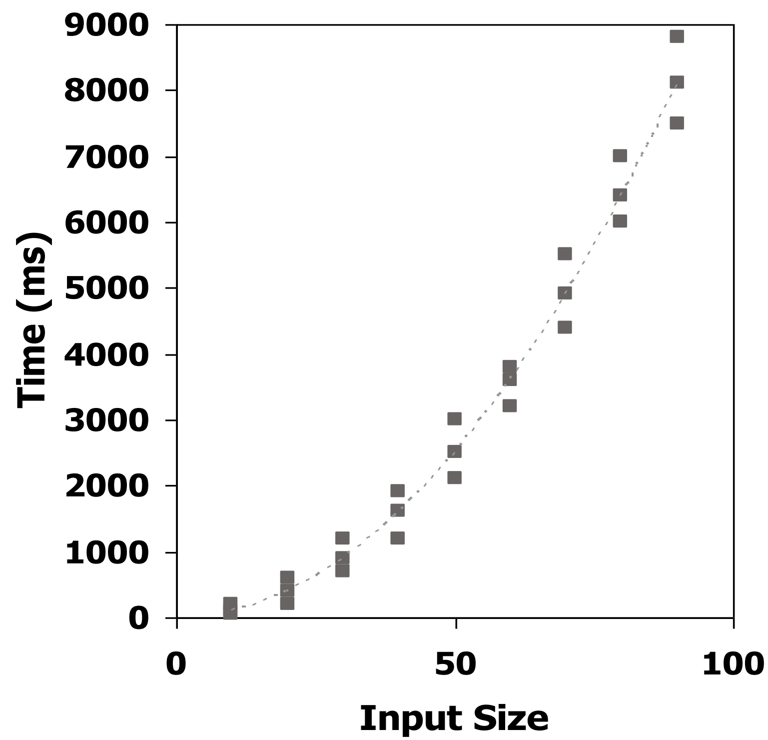
Running Time

- 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, noting the time needed:
- Plot the results




```
long startTime = System.currentTimeMillis();       // record the starting time
/* (run the algorithm) */
long endTime = System.currentTimeMillis();         // record the ending time
long elapsed = endTime − startTime;                // compute the elapsed time
```

Limitations of Experiments

- Need to implement the algorithm, which may be difficult/costly.
- Results may not be indicative of the true running time (can't test all possible types of input)
- In order to compare two algorithms, need to use same hardware and software environments
- In some multiprogramming environments, such as Windows, it is very difficult to determine how long a single task takes(since there is so much happening behind the scene).

How to estimate Efficiency

- Efficiency, to a great extent, depends on how the method is defined.
- An abstract analysis that can be performed by direct investigation of the method definition is hence preferred.
- ✓ Ignore various restrictions; i.e.:
- ✓ CPU speed
- ✓ Memory limits; for instance allow an int variable to take any allowed integer value, and allow arrays to be arbitrarily large
- ✓ etc.
- Since the method is now unrelated to specific computer environment, we refer to it as ***algorithm***.

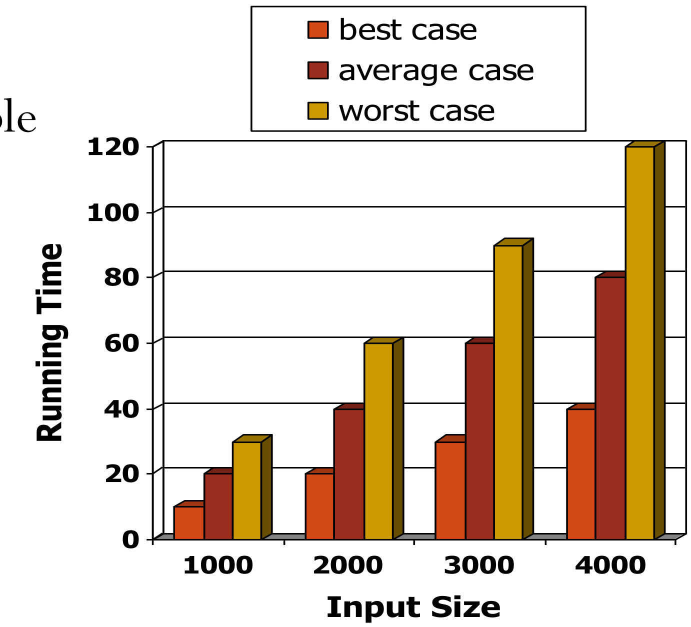
Estimating Running Time

How can we estimate the running/execution-time given the algorithm's definition?

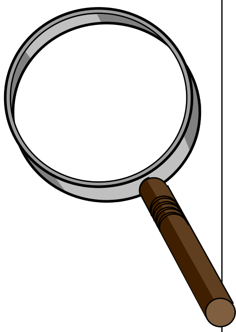
- Consider the number of executed statements, in a trace of the algorithm, as a measurement of running-time requirement.
- This measurement can be represented as function of the " input size" "n" of the problem.
- The running time of an algorithm typically grows with the input size.

Estimating Running Time

- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics, etc.
- Given a method of a problem of size *n*, find ***worstTime(n)*** , which is the maximum number of executed statements in a trace, considering all possible parameters/input values.



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

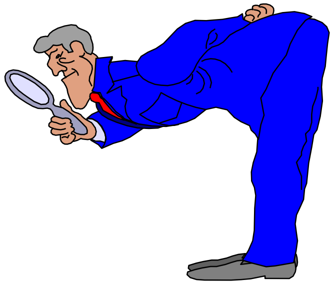
Pseudocode Details

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

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

  Input …

  Output …



- **Method call**

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

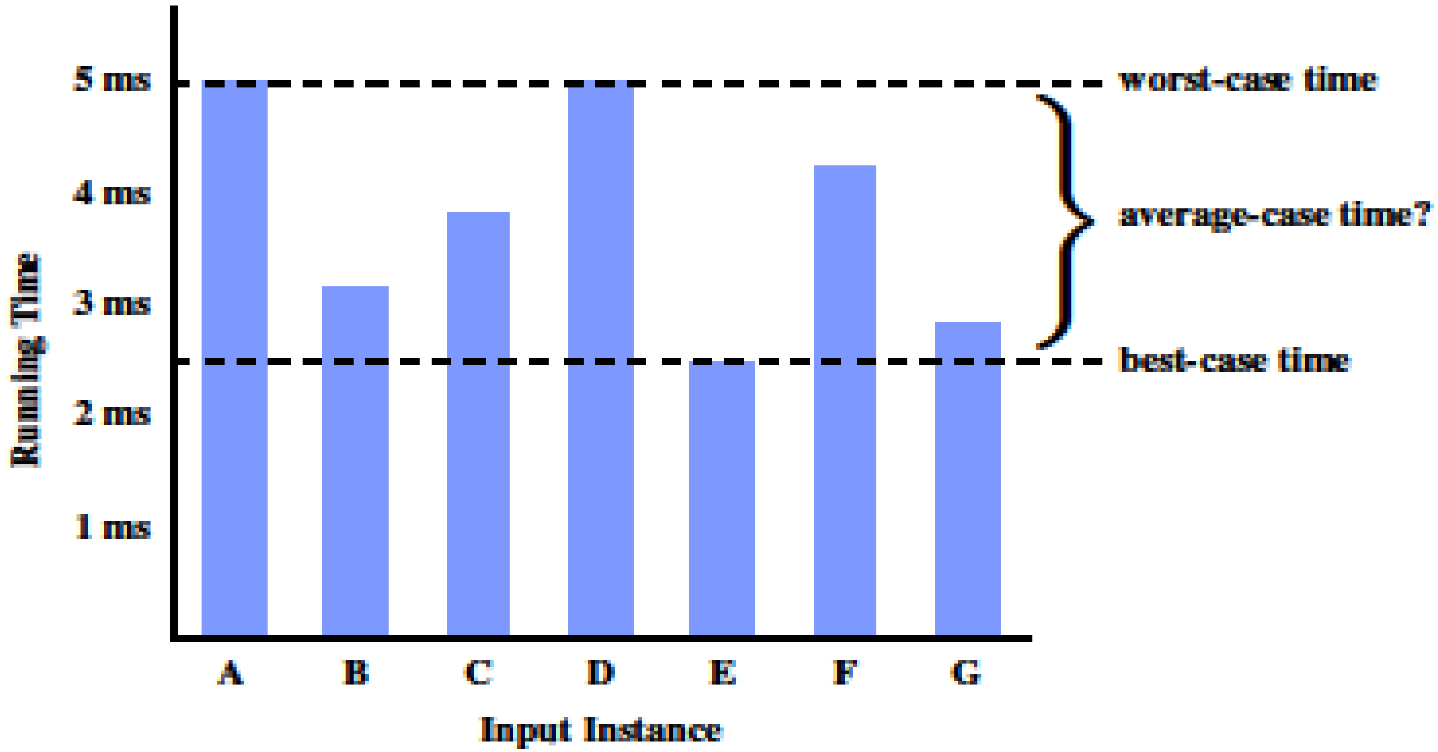
  return *expression*
- **Expressions:**

  ← **Assignment**

  = **Equality testing**

  n^2 **Superscripts and other mathematical formatting allowed**

Best-case vs. Worst-case time



Estimating Running Time

- Example: Assume an array a [0 … n –1] of int, and assume the following code segment:

```
for (int i = 0; i < n - 1; i++)
 if (a [i] > a [i + 1])
   System.out.println (i);
```

- What is worstTime(n)?

Estimating Running Time

```
for (int i = 0; i < n - 1; i++)
  if (a [i] > a [i + 1])
    System.out.println (i);
```

Statement	Worst Case Number of Executions
`i = 0`	1
`i < n - 1`	n
`i++`	n - 1
`a[i] > a[i+1]`	n - 1
`System.out.println()`	n - 1

That is, worstTime(n) is: $4n-2$.

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*

http://www.wikihow.com/Write-Pseudocode

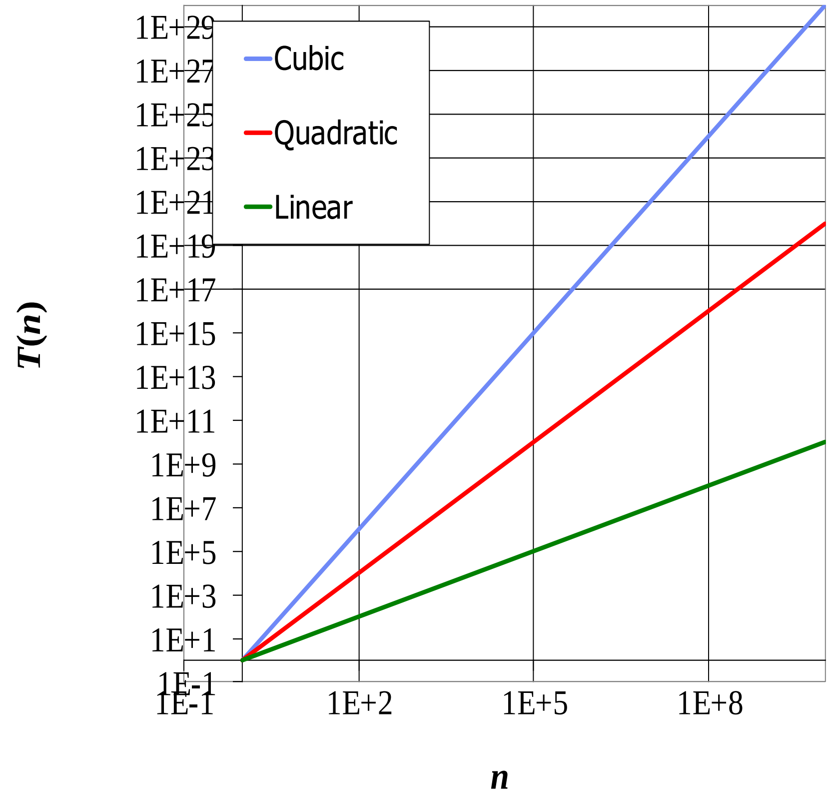
Primitive Operations

- 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

Seven Important Functions

- Seven functions that often appear in algorithm analysis:
  - Constant ≈ ***1***
  - Logarithmic ≈ log ***n***
  - Linear ≈ ***n***
  - N-Log-N ≈ ***n*** log ***n***
  - Quadratic ≈ $\boldsymbol{n}^2$
  - Cubic ≈ $\boldsymbol{n}^3$
  - Exponential ≈ $\mathbf{2}^n$



- In a log-log chart, the slope of the line corresponds to the growth rate

log, base 2

Counting Primitive Operations

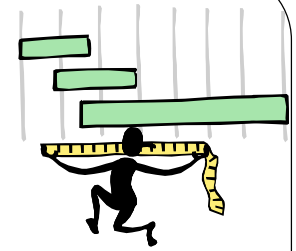
By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size.

Algorithm: ***compareValues*(A, n)**
Input: array A of n integers

Output: display all elements larger than following one

	# of operations
for $i \leftarrow 0$ to $n - 2$ do	$n - 1$
if $A[i] > A[i+1]$ then	$n - 1$
display i	$n - 1$
increment counter i	$n - 1$
	Total : $4n - 4$

Estimating Running Time



Algorithm *compareValues* executes $4n - 4$ 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 the running time of *compareValues*. Then

$$a\,(4n - 4) \le T(n) \le b(4n - 4)$$

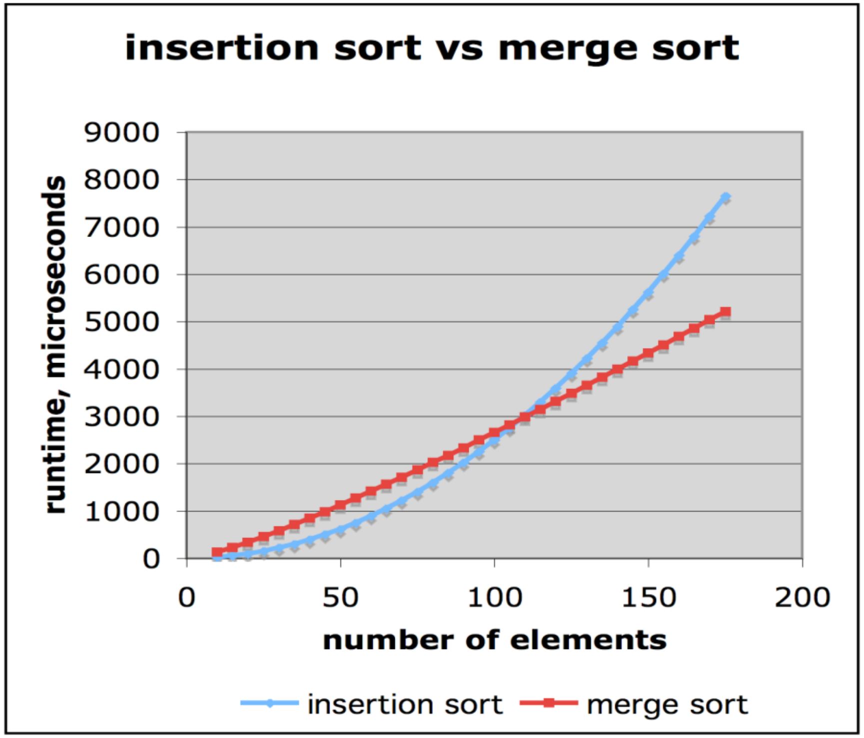
- Hence, the running time $T(n)$ is bounded by two linear functions

Why Growth Rate Matters

if runtime is...	time for n + 1	time for 2 n	time for 4 n
c lg n	c lg (n + 1)	c (lg 2n)	c(lg n + 2)
c n	c (n + 1)	2c n	4c n
c n lg n	~ c n lg n + c n	2c n lg n + 2cn	4c n lg n + 4cn
c n^2	~ c n^2 + 2c n	**4c n^2**	16c n^2
c n^3	~ c n^3 + 3c n^2	8c n^3	64c n^3
c 2^n	c 2^{n+1}	c 2^{2n}	c 2^{4n}

runtime quadruples when problem size doubles

Comparison of Two Algorithms



- insertion sort is $n^2/4$
- merge sort is 2 nlgn

sort a million items?

- insertion sort takes roughly 70 hours

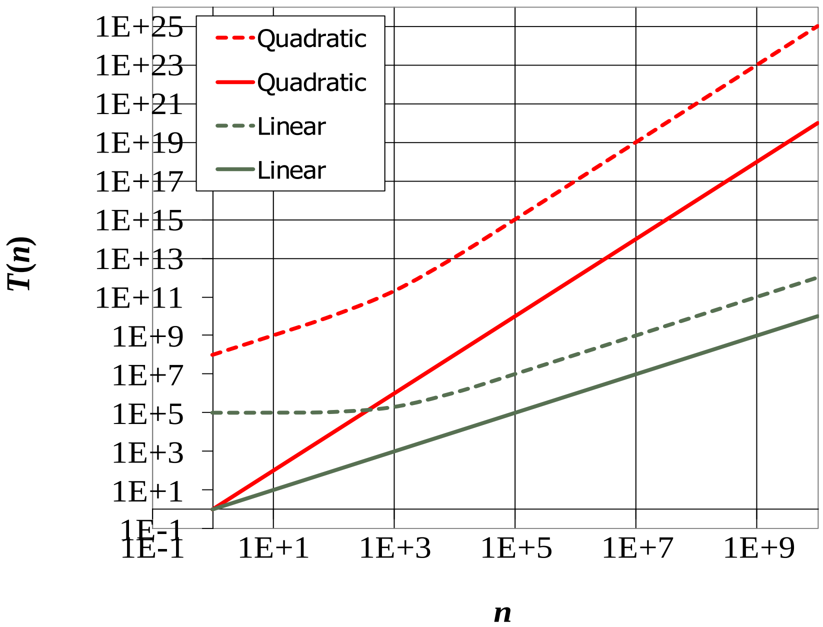
while

- merge sort takes roughly 40 seconds

This is a slow machine, but if 100 x as fast then it's 40 minutes versus less than 0.5 seconds

Constant Factors

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- <u>Examples</u>
  - $10^2 n + 10^5$ is a linear function
  - $10^5 n^2 + 10^8 n$ is a quadratic function



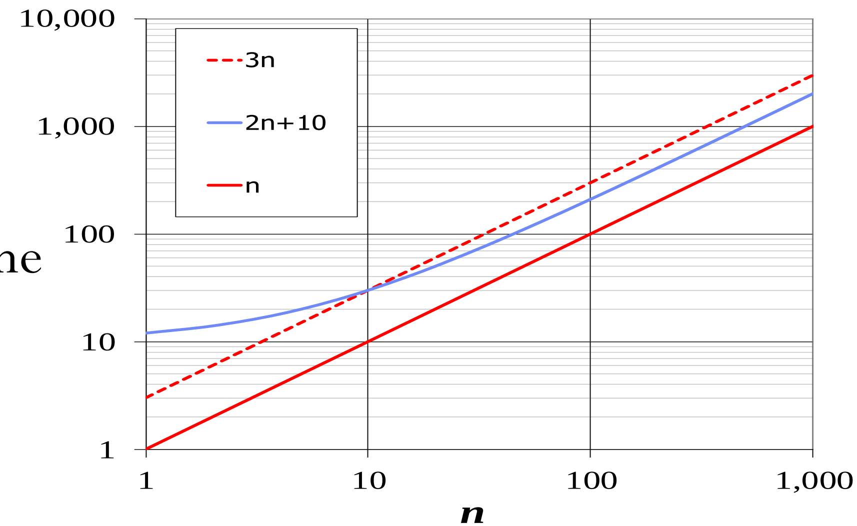
Today

- Algorithm Efficiency
  - Beyond Experimental Analysis
- Growth Functions and Big-O Notation

- Asymptotic Analysis
  - Comparing Growth functions

Big-O Notation

- We do NOT need to calculate the exact worst time since it is only an approximation of time requirements.
- Instead, we can just approximate that time by means of "Big-O" notation.
- That is quite satisfactory since it gives us approximation of an approximation!



Big-O Notation

- Further, by a standard abuse of notation, we often associate a function with the value is calculates.

For instance, if $g(n) = n^3$, for $n = 0, 1, 2, \ldots.,$
then instead of saying that $f(n)$ is $O(g(n))$, we say
$f(n)$ is $O(n^3)$.

Big-O Notation

Example

Nested loops are significant when estimating O().

Example 4:

Consider the following loop segment, what is O()?

```
for (int i = 0; i < n; i++ )
  for (int j = 0; j < n; j++ )
          ...............
```

- The outer loop has $1 + (n + 1) + n$ executions.
- The inner loop has $n(1 + (n + 1) + n)$ executions.
- Total is: $2n^2 + 4n + 2 \rightarrow O(n^2)$.

Hint: As seen in Example 2 for polynomial functions

Big-O Notation

Important Note: Big-O only gives an upper bound of the algorithm.

However, if $f(n)$ is $O(n)$, then it is also $O(n + 10)$, $O(n^3)$, $O(n^2 + 5n + 7)$, $O(n^{10})$, etc.

We, generally, choose the smallest element from this hierarchy of orders.

For example, if $f(n) = n + 5$, then we choose $O(n)$, even though $f(n)$ is actually also $O(n \log n)$, $O(n^4)$, etc.

Similarly, we write $O(n)$ instead of $O(2n + 8)$, $O(n - \log n)$, etc.

Big-O Notation

Elements of the Big-O hierarchy can be as:

$$O(1) \subset O(\log n) \subset O(n^{1/2}) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset \ldots\ldots\ldots \subset O(2^n) \subset \ldots\ldots\ldots$$

Where the symbol "$\subset$", indicates "is contained in".

The following table provides some examples:

Sample Functions	Order of O()
$f(n) = 3000$	$O(1)$
$f(n) = (n * \log_2(n+1) + 2) / (n+1)$	$O(\log n)$
$f(n) = (500 \log_2 n) + n / 100000$	$O(n)$
$f(n) = (n * \log_{10} n) + 4n + 8$	$O(n \log n)$
$f(n) = n * (n + 1) / 2$	$O(n^2)$
$f(n) = 3500\, n^{100} + 2^n$	$O(2^n)$

Finding Big-O Estimates Quickly

<u>Case 1:</u> Number of executions is independent of n → $O(1)$

<u>Example:</u>

```
// Constructor of a Car class
  public Car(int nd, double pr)
  {
      numberOfDoors = nd;
      price = pr;
  }
```

Finding Big-O Estimates Quickly

Case 2: The splitting rule → $O(\log n)$

Example:

```
while(n > 1)
{
     n = n / 2;
     ...;
}
```

Example:

See the binary search method in Recursion6.java & Recursion7.java

Finding Big-O Estimates Quickly

Case 3: Single loop, dependent on n → $O(n)$

Example:

```
for (int j = 0; j < n; j++ )
        System.out.println(j);
```

Note: It does NOT matter how many simple statement (i.e. no inner loops) are executed in the loop. For instance, if the loop has k statements, then there is $k*n$ executions of them, which will still lead to $O(n)$.

Finding Big-O Estimates Quickly

<u>Case 4:</u> Double looping dependent on *n* & splitting

→ $O(n \log n)$

<u>Example:</u>

```
for (int j = 0; j < n; j++ )
{
     m = n;
     while (m > 1)
     {
          m = m / 2;
          …;
          // Does not matter how many statements are here
     }
}
```

Finding Big-O Estimates Quickly

<u>Case 4:</u> Double looping dependent on n
➔ $O(n^2)$

<u>Example:</u>

```
for (int i = 0; i < n; i++ )
    for (int j = 0; j < n; j++ )
    {
            …;
            // Does not matter how many statements are here
    }
```

Finding Big-O Estimates Quickly

Case 4 (Continues): Double looping dependent on n
$\rightarrow O(n^2)$

Example:

```
for (int i = 0; i < n; i++ )
    for (int j = i; j < n; j++ )
    {
            ...;
            // Does not matter how many statements are here
    }
```

The number of executions of the code segment is as follows:

$$n + (n-1) + (n-2) + \ldots + 3 + 2 + 1$$

Which is: $n(n + 1) / 2 = \frac{1}{2} n^2 + \frac{1}{2} n \rightarrow O(n^2)$

Finding Big-O Estimates Quickly

Case 5: Sequence of statements with different $O(\ )$

$O(g_1(n)) + O(g_2(n)) + \ldots = O(g_1(n) + g_2(n) + \ldots)$

Example:

```
for (int i = 0; i < n; i++ )
{             . . .
}
for (int i = 0; i < n; i++ )
    for (int j = i; j < n; j++ )
    {         . . .
    }
```

The first loop is $O(n)$ and the second is $O(n^2)$. The entire segment is hence $O(n) + O(n^2)$, which is equal to $O(n + n^2)$, which is in this case $O(n^2)$.

Big-O 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 $O(3n)$"

Today

- Algorithm Efficiency
  - Beyond Experimental Analysis
- Growth Functions and Big-O Notation
- Asymptotic Analysis 

  - Comparing Growth functions

Asymptotic Algorithm Analysis

- In computer science and applied mathematics, asymptotic analysis is a way of describing limiting behaviors (may approach ever nearer but never crossing!).
- Asymptotic analysis of an algorithm determines the running time in big-O notation.
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We then express this function with big-O notation

Asymptotic Algorithm Analysis

- Example:
- We determine that algorithm ***compareValues*** executes at most $4n-4$ primitive operations
- We say that algorithm ***compareValues*** "runs in $O(n)$ time", or has a "complexity" of $O(n)$

- Note: Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations.

Asymptotic Algorithm Analysis

- If two algorithms *A* & *B* exist for solving the same problem, and, for instance, A is $O(n)$ and B is $O(n^2)$, then we say that *A* is asymptotically better than *B* (although for a small time *B* may have lower running time than *A)*.
- To illustrate the importance of the asymptotic point of view, let us consider three algorithms that perform the same operation, where the running time (in μs) is as follows, where n is the size of the problem:
  - Algorithm1: $400n$
  - Algorithm2: $2n^2$
  - Algorithm3: 2^n

Asymptotic Algorithm Analysis

- Which of the three algorithms is faster?
  - *Notice that Algorithm1 has a very large constant factor compared to the other two algorithms!*

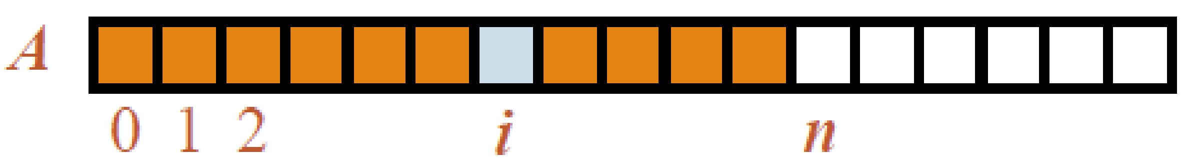
Running Time (μs)	**Maximum Problem Size (n) that can be solved in:**		
	1 Second	1 Minute	1 Hour
Algorithm 1 $400n$	2,500 (400 * 2,500 = 1000,000)	150,000	9 Million
Algorithm 2 $2n^2$	707 ($2 * 707^2 \approx 1000,000$)	5,477	42,426
Algorithm 3 2^n	19 (only 19, since 2^{20} would exceed 1000,000)	25	31

DS Efficiencies Comparison

1. Arrays/ ArrayList
2. BS Trees
3. AVL Trees

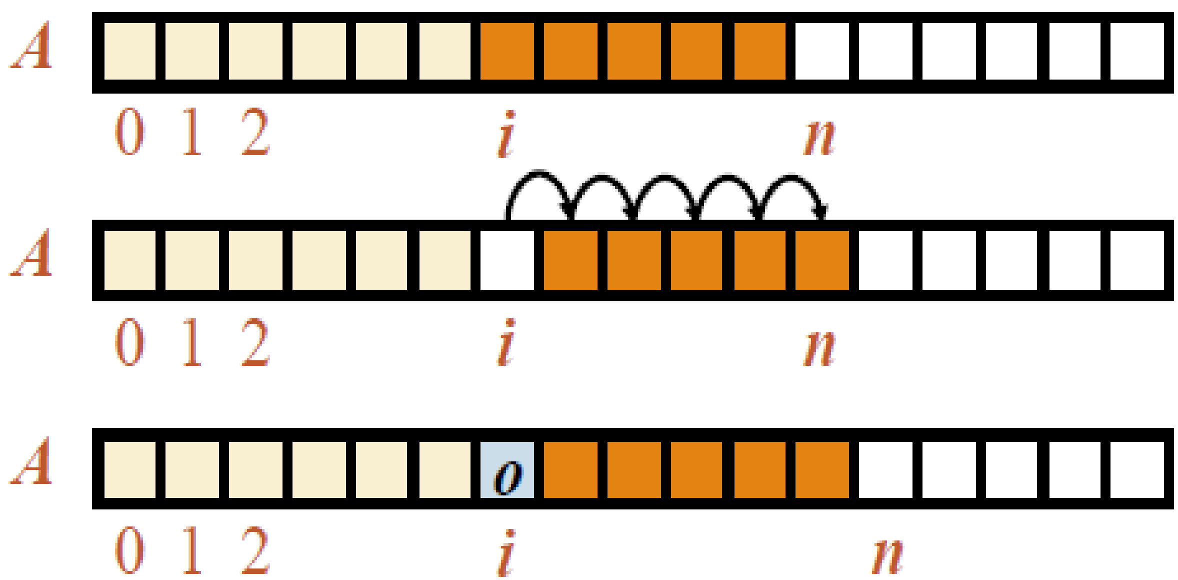
Array-based Implementation

- Use an array A of size N
- A variable n keeps track of the size of the array list (number of elements stored)
- Operation *get*(i) is implemented in $O(\_\_)$ time by returning $A[i]$
- Operation *set*(i,o) is implemented in $O(\_\_)$ time by performing $t = A[i]$, $A[i] = o$, and returning t.



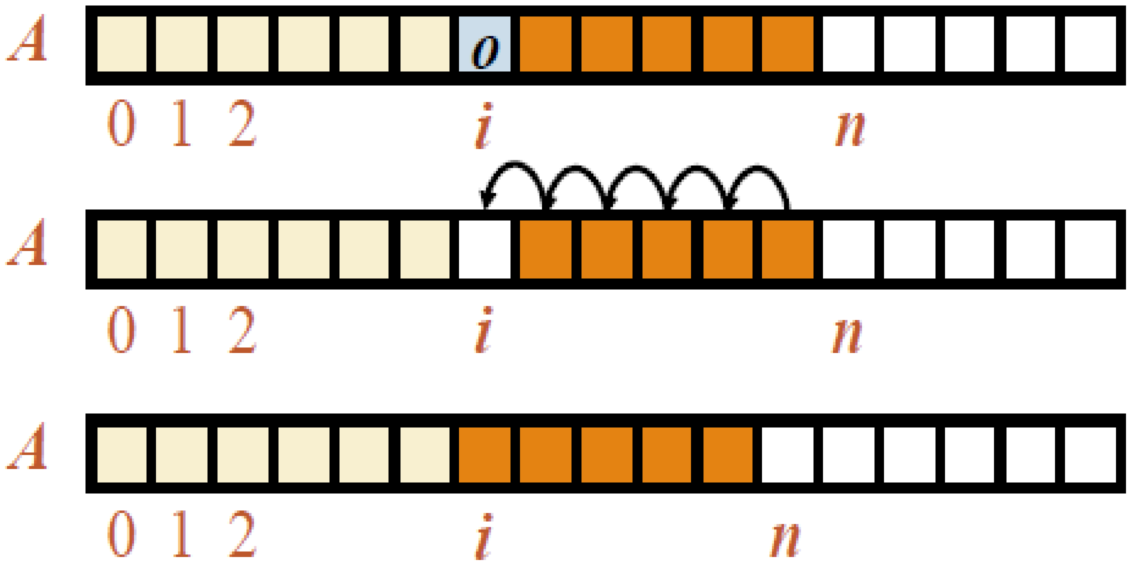
Insertion

- In operation *add*(i, o), we need to make room for the new element by shifting forward the $n - i$ elements $A[i]$, …, $A[n - 1]$
- In the worst case ($i = 0$), this takes $O(\_\_)$ time



Element Removal

- In operation *remove*(i), we need to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i + 1], \ldots, A[n - 1]$
- In the worst case ($i = 0$), this takes $O($___$)$ time

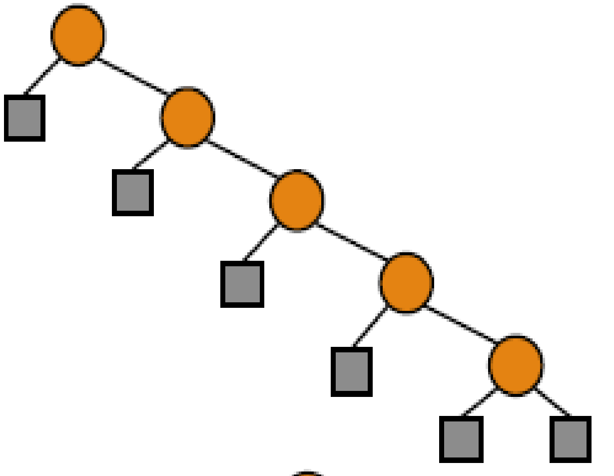


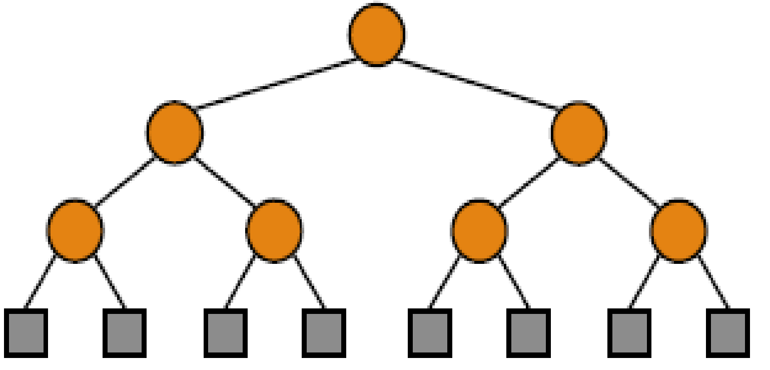
Performance

- In the array based implementation of an array list:
  - The space used by the data structure is $O(n)$
  - *size*, *isEmpty*, *get* and *set* run in $O(1)$ time
  - *add* and *remove* run in $O(n)$ time in worst case
- If we use the array in a circular fashion, operations *add*$(0, x)$ and *remove*$(0, x)$ run in $O(1)$ time
- In an *add* operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one

Performance

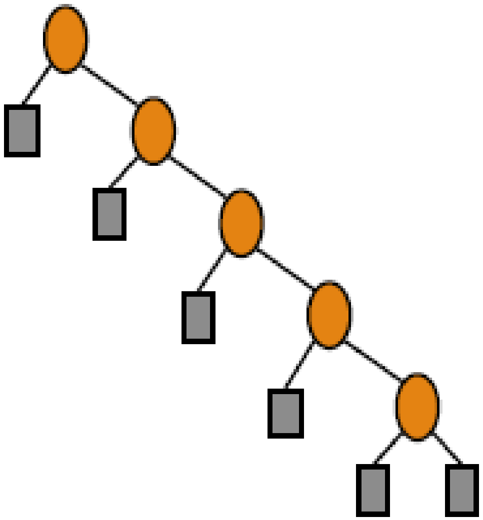
- Consider an ordered map with n items implemented by means of a binary search tree of height h
  - the space used is $O(n)$
  - methods get, floorEntry, ceilingEntry, put and remove take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case





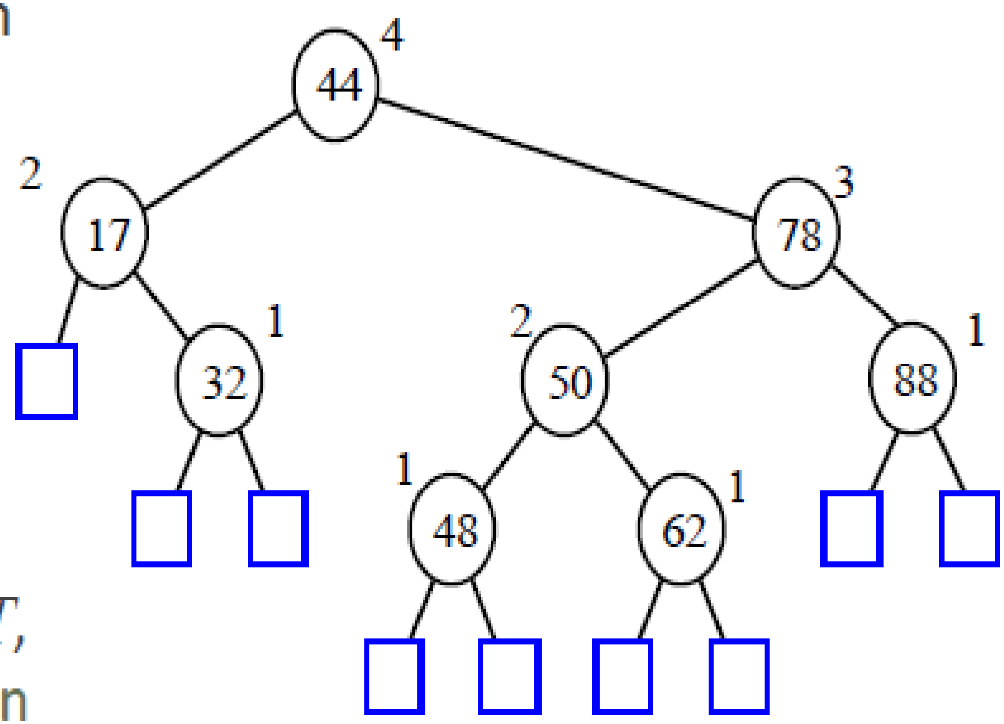
Performance of Binary Search Trees

- A binary search tree should be an efficient data structure for map implementation.
- However, as seen, binary search trees may have $O(n)$ in the worst case, which is not any better than list-based or array-based map implementation.
- That problem is caused by the possibility that the nodes may be arranged such that $n = h + 1$, where h is the height of the tree.



AVL Tree Definition

- The performance problem of binary search trees can be corrected using AVL trees.
- AVL Trees are balanced Trees.
- An AVL Tree is a binary search tree such that for every internal node v of T, the heights of the children of v can differ by at most 1; this is referred to as the ***Height-Balance Property***.



An example of an AVL tree where the heights are shown next to the nodes

AVL Tree Performance

- a single restructure takes O(1) time
  - using a linked-structure binary tree
- get takes O(log n) time
  - height of tree is O(log n), no restructures needed
- put takes O(log n) time
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(1)
  - However, we still need O(log n) to find out if restructuring is needed
- remove takes O(log n) time
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)

Which DS to select for a given problem?

Need to .

- Understand the problem we are solving in terms of :
  1. Data size
  2. Key operations performed on the data
  - Trade off 1 and 2

References

- Textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014