

CS1102 – Lecture 7

Analysis of Algorithms

Chapter 10: pages 464 – 475

(2nd Edition)



About Me

- Ling Tok Wang
- Office: COM1 03-14
- Email: lingtw@comp.nus.edu.sg
- Consultation hours: 4:30pm-6pm Wed and Fri



Topics covered in the first half of the semester:

- Principles of Programming and Software Engineering
- Java
 - Java Fundamentals, Classes and Interfaces, Generics
- Abstract Data Types (ADT)
- Linked Lists
 - Singly linked lists
 - Tailed linked lists
 - Doubly linked lists
 - Circular linked lists
- Stacks
- Queues
- Recursions

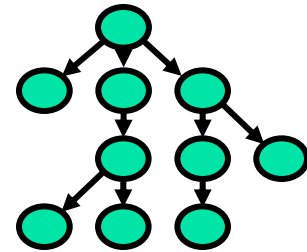
Topics to be covered in the second half of the semester:

- Analysis of algorithms (Complexity)

- Sorting

29	10	14	37	13
----	----	----	----	----

- Trees and Binary Trees
(sorting and searching)

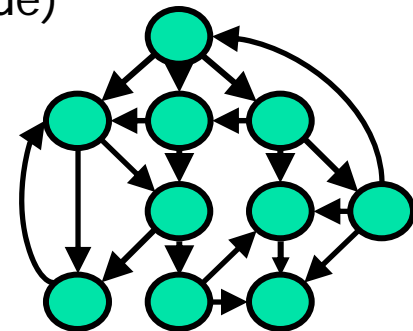


- Heaps and Priority Queues

A Special form of queue from which **items are removed according to their designated priority** and not the order in which they entered.

- Hashing (a very fast searching technique)

- Graphs



Lecture 7 - Analysis of Algorithms:

Outline

- What is an **algorithm**?
- What do we mean by **analysis of algorithms**?
- How to analyze an algorithm?
- **Big O** notation
- Examples



You are expected to know

- Proof by induction
- Proof by contradiction
- Operations on logarithm function
- Arithmetic and geometric progressions
 - Their sums
- Linear, quadratic, cubic, polynomial functions
- ceiling, floor, absolute value



What is an algorithm?

*A step-by-step procedure
for solving a problem.*



Determining the Efficiency of Algorithms

- **Analysis of algorithms**

- Provides tools for contrasting the efficiency of different methods of solution (rather than programs)
- **Complexity** of algorithms

- **A comparison of algorithms**

- Should focus on significant differences in the efficiency of the algorithms
- Should not consider reductions in computing costs due to clever coding tricks. Tricks may reduce the readability of an algorithm.

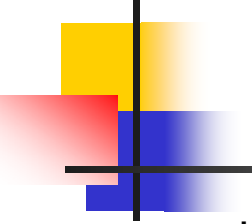


What do we mean by Analysis of Algorithms?

- To evaluate rigorously the *resources* (time and space) needed by an algorithm and represent the result of the analysis with a formula
- Emphasize more on the time requirement
- The time requirement of an algorithm is also called its time complexity

Analysis of Algorithms:

By measuring the Run time?



```
class TimeTest1 {  
    public static void main(String[] args) {  
        long startTime = System.currentTimeMillis();  
        long total = 0;  
        for (int i = 0; i < 10000000; i++) {  
            total += i;  
        }  
        long stopTime = System.currentTimeMillis();  
        long elapsedTime = stopTime - startTime;  
        System.out.println(elapsedTime);  
    }  
}
```

Note: The run time depends on the compiler and computer used, and the current work load of the computer system.



Exact run time is **not** always needed

Using exact run time is not meaningful when we want to **compare** two algorithms

- coded in different languages,
- using different data sets, or
- running on different computers.



Determining the Efficiency of Algorithms

- Difficulties with comparing **programs** instead of **algorithms**
 - How are the algorithms coded?
 - Which compiler is used?
 - What computer should you use?
 - What data should the programs use?
- Algorithm analysis should be **independent of**
 - Specific implementations
 - Compilers and their optimizers
 - Computers
 - Data

The Execution Time of Algorithms



- Instead of working out the exact timing, we count the number of some or all of the **primitive operations** (e.g. +, -, *, /, assignment, ...) needed.
- Counting an algorithm's **operations** is a way to assess its efficiency
 - An algorithm's execution time is related to the number of operations it requires.
 - Examples
 - Traversal of a linked list
 - The Towers of Hanoi
 - Nested Loops



Algorithm Growth Rates

- An algorithm's time requirements can be measured as a function of the **problem size**
- An algorithm's **growth rate**
 - Enables the comparison of one algorithm with another
 - Examples
 - Algorithm A requires time proportional to n^2
 - Algorithm B requires time proportional to n
- Algorithm efficiency is typically a concern for **large problems** only. **Why?**

Algorithm Growth Rates

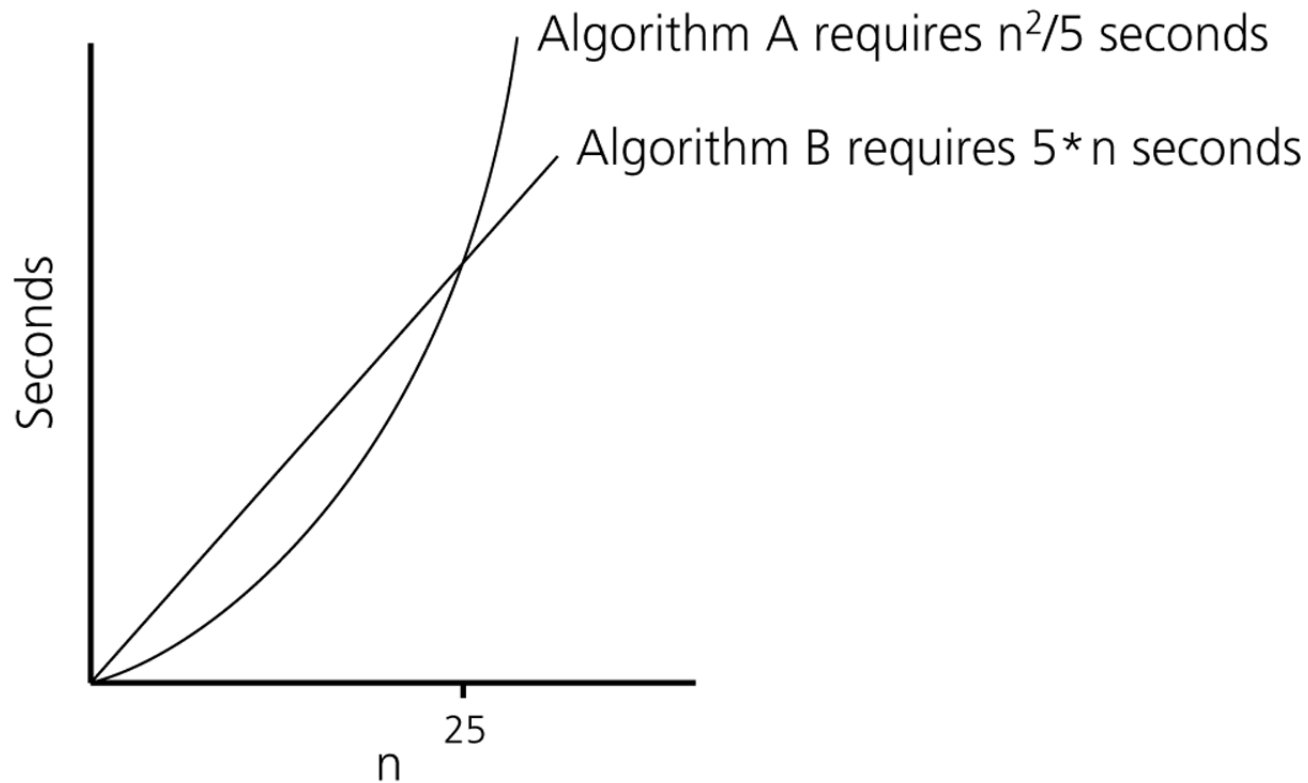


Figure - Time requirements as a function of the problem size n

Computation cost of an algorithm

How many operations are required?

```
for (int i=1; i<=n; i++) {  
    perform 100 operations;           // A  
    for (int j=1; j<=n; j++) {  
        perform 2 operations;        // B  
    }  
}
```

$$\begin{aligned}\text{Total Ops} &= A + B \\ &= \sum_{i=1, n} 100 + \sum_{i=1, n} (\sum_{j=1, n} 2) \\ &= 100n + \sum_{i=1, n} (2n) \\ &= 100n + 2n^2 \\ &= 2n^2 + 100n\end{aligned}$$



Counting the number of statements

To simplify the counting further, we can ignore

- the different types of operations and
- different number of operations in a statement,

and simply **count the number of statements executed**.

So total number of statements executed in the previous example

$$= 2n^2 + 100n$$



Approximation of analysis results

Very often, we are interested only in using a simple term to **indicate how efficient an algorithm is**. The exact formula of an algorithm's performance is not really needed.

Example:

Given the formula: $3n^2 + 2n + \log n + 1/(4n)$,

the **dominating term** $3n^2$ can tell us approximately how an algorithm performs.

What kind of approximation of the analysis of algorithms do we need?



Asymptotic analysis

Asymptotic analysis is an analysis of algorithms that focuses on

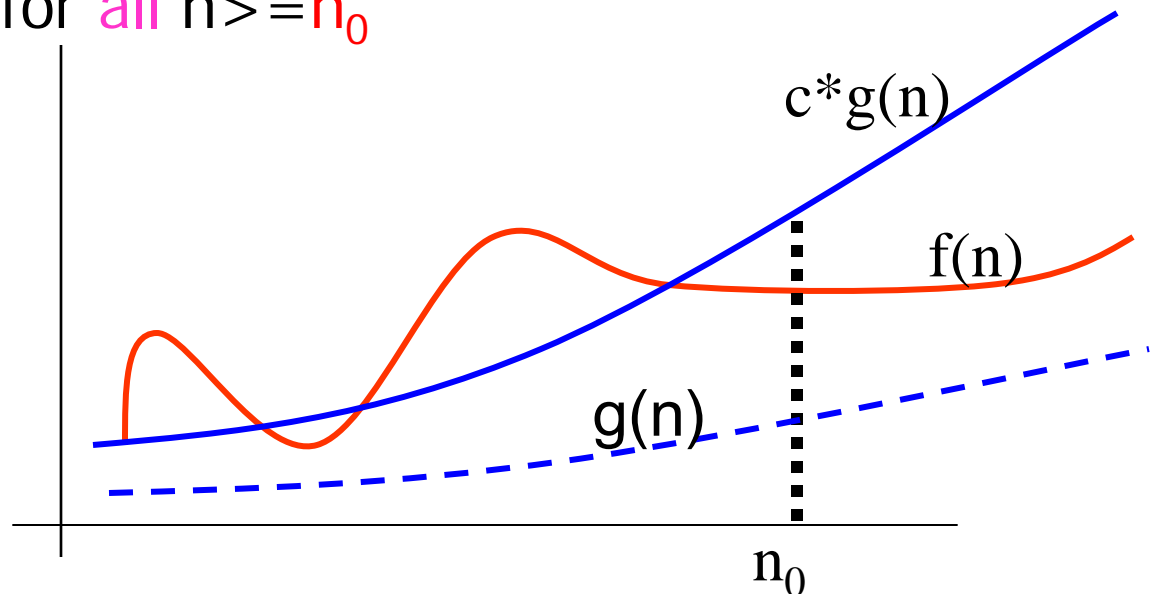
- analyzing problems of **large input size**
- consider only the **leading term** of the formula
- **ignore** the **coefficient** of the leading term

Some notations are needed in asymptotic analysis

Upper Bound

Definition: Given a function $f(n)$, $g(n)$ is an (asymptotic) **upper bound** of $f(n)$, denoted as $f(n) = O(g(n))$, if there exist a constant $c > 0$, and a positive integer n_0 such that $f(n) \leq c * g(n)$ for all $n \geq n_0$.

- $f(n)$ is said to be **bounded from above** by $g(n)$.
- $O()$ is called the **big O notation** (or called "**big Oh**" notation).





Ignore the coefficients of all terms

Based on the definition, $2n^2$ and $30n^2$ have the same upper bound n^2 , i.e.,

- $2n^2 = O(n^2)$.

- $30n^2 = O(n^2)$.

They differ only in the choice of c .

Therefore, in big O notation, we can throw away the coefficients of all terms in a formula.

Example: $f(n) = 2n^2 + 100n$
 $= O(n^2) + O(n)$



Finding the constants c and n_0

Given $f(n) = 2n^2 + 100n$

Prove that $f(n) = O(n^2)$.

[Solution]

$2n^2 + 100n < 2n^2 + n^2 = 3n^2$ whenever $n > 100$.

Set the constants to be $c=3$ and $n_0=100$.

By definition, we have $f(n) = O(n^2)$.

Note:

1. $n^2 < 2n^2 + 100n$ for all n , i.e., $g(n) < f(n)$, and yet $g(n)$ is an asymptotic upper bound of $f(n)$
2. c and n_0 are not unique.

For example, we can choose $c=2+100=102$ and $n_0=1$

Q: Can we write $f(n) = O(n^3)$?



Is the bound tight?

The complexity of an algorithm can be bounded by many functions.

Example

$2n^2 + 100n$ is bounded by n^2 , n^3 , n^4 and many others according to the definition of big O notation.

We are more interested in the **tightest bound** which is n^2 for this case.



Growth Terms (Order-of-Magnitude)

In asymptotic analysis, a formula can be simplified to a single term with coefficient 1

Such a term is called a **growth term** (rate of growth, order of growth, order-of-magnitude)

The most common **growth terms** can be ordered as follows:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

Note

- “log” = log base 2, or \log_2 . “ \log_{10} ” = log base 10, “ln” = log base e. In big O, all these log functions are the same.

Why?



More examples on Big O notation

- $f_1(n) = \frac{1}{2}n + 4$
 $= O(n)$

- $f_2(n) = 240n + 0.001n^2$
 $= O(n^2)$

- $f_3(n) = n \log n + \log n + n \log (\log n)$
 $= O(n \log n)$

Why?



Exponential Time Algorithms

Suppose we have a problem that, for an input consisting of n items, can be solved by going through 2^n cases (we say the complexity is exponential time)

Q: What sort of problems?

We use a supercomputer, that analyses 200 million cases per second

- Input with 15 items, 163 microseconds
- Input with 30 items, 5.36 seconds
- Input with 50 items, more than two months
- Input with 80 items, 191 million years !



Quadratic Time Algorithms

Suppose solving the same problem with another algorithm will use $300n^2$ clock cycles (the complexity is quadratic) on a 80386, running at 33 MHz (very slow)

- Input with 15 items, 2 milliseconds
- Input with 30 items, 8 milliseconds
- Input with 50 items, 22 milliseconds
- Input with 80 items, 58 milliseconds

It is very important to use an efficient algorithm to solve a problem.

Q: What observations do you have from the results of these two pages?
What if the supercomputer speed is increased by 1000 times?

Order-of-Magnitude Analysis and Big O Notation

(a)

Function	n					
	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Figure - A comparison of growth-rate functions in tabular form

Order-of-Magnitude Analysis and Big O Notation

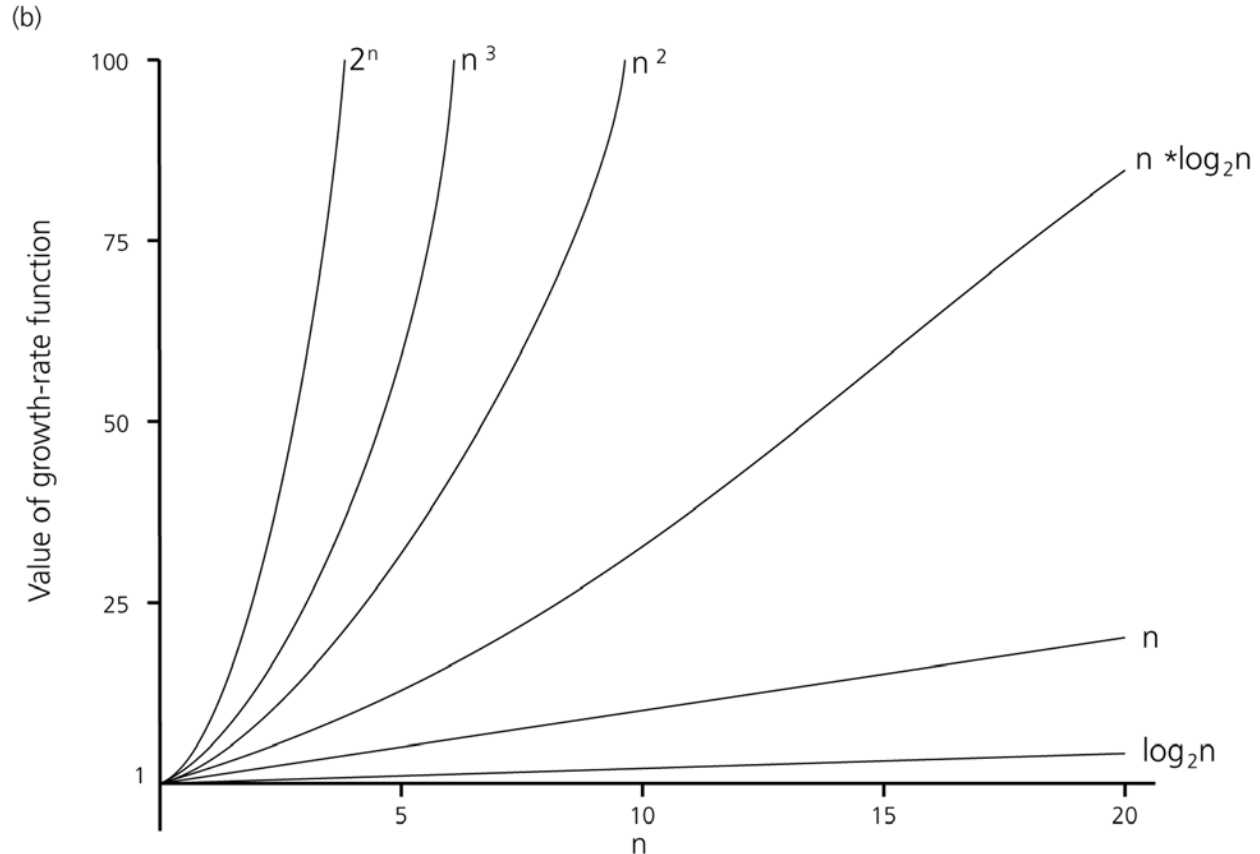


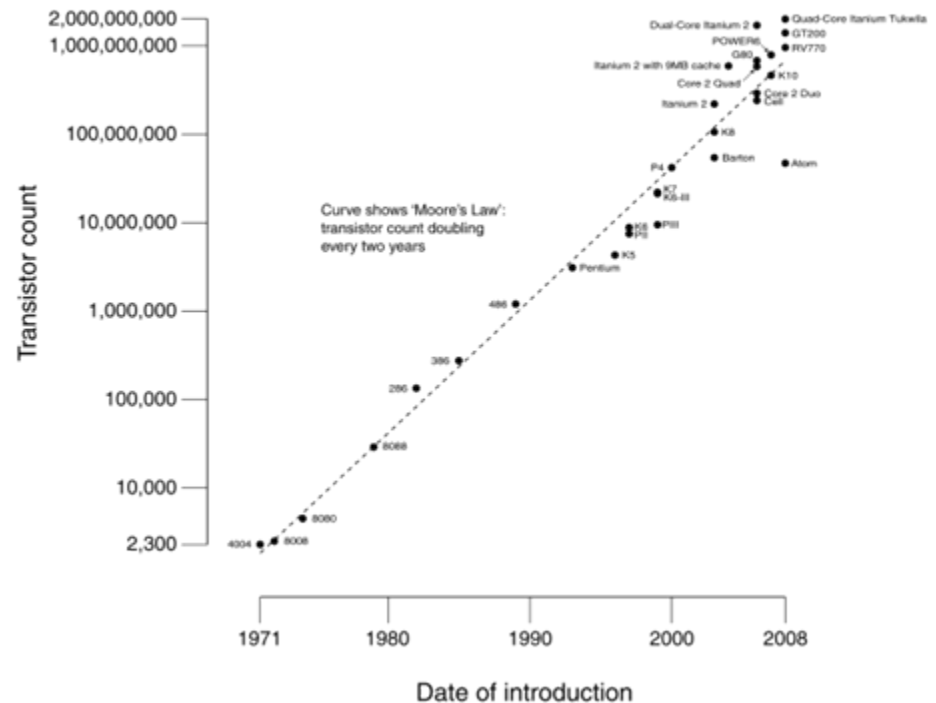
Figure - A comparison of growth-rate functions in graphical form

Example:

Moore's law

Intel co-founder **Gordon Moore** is a visionary. In 1965, his prediction, popularly known as Moore's Law, states that the number of transistors per square inch on an integrated circuit chip will be **increased exponentially, double about every two years**. Intel has kept that pace for nearly 40 years.

CPU Transistor Counts 1971-2008 & Moore's Law





Summary: Order-of-Magnitude Analysis and Big O Notation

- Order of growth of some common functions
 - $O(1) < O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < O(2^n)$
- Properties of growth-rate functions
 - You can ignore low-order terms
 - You can ignore a multiplicative constant in the high-order term
 - $O(f(n)) + O(g(n)) = O(f(n) + g(n))$



How to find the complexity of a program? Some rules of thumb:

- Basically just count number of statements executed.
- If there are only a small number of simple statements in a program
 - $O(1)$
- If there is a for loop dictated by a loop index that goes up to n
 - $O(n)$
- If there is a nested for loop with outer one controlled by n and the inner one controlled by m
 - $O(m * n)$
- For a loop with a range of value n , and each iteration reduces the range by a fixed fraction (usually it is 0.5, i.e., half)
 - $O(\log n)$
- For a recursive method, each call is usually $O(1)$. So
 - if n calls are made – $O(n)$
 - if $n \log n$ calls are made – $O(n \log n)$



Examples on finding complexity

What is the complexity of each of the following code fragment?

```
sum = 0;  
For (i=1; i<n; i=i*2)  
    sum++;
```

It is clear that sum is incremented only when
 $i = 1, 2, 4, 8, \dots, 2^k = n$ where $k = \log n$.
So, the complexity is therefore $O(\log_2 n)$

Note: When 2 is replaced by 10 in the for loop, the complexity is $O(\log_{10} n)$ which is the same as $O(\log_2 n)$.
Why?



Examples on finding complexity

```
sum = 0;
For (i=1; i<n; i=i*3)
    {for (j=1; j<i; j++)
        sum++;
    }
```

Q: What is the complexity of this code fragment?

$$\begin{aligned} O(n) &= 1 + 3 + 9 + 27 + \dots + 3^{(\log_3 n)} \\ &= n + n/3 + n/9 + \dots + 1 \\ &= n(1 + 1/3 + 1/9 + \dots) \\ &\leq 3/2 n \\ &= O(n) \end{aligned}$$

Example:

Tower of Hanoi analysis

- Number of moves made by the algorithm is $2^n - 1$. Prove it!
 - **Hints:** $f(1)=1$, $f(n)=f(n-1)+1+f(n-1)$, and proof by induction
- Assume each move takes t time, then:
$$f(n) = t * (2^n - 1) = O(2^n).$$
- The towers of Hanoi algorithm is an **exponential time algorithm**.

Example:

Sequential Search

```
int seqSearch (int[] a, int len, int x)
    throws ItemNotFound {
    for (int i = 0; i < len; i++) {
        if (a[i] == x) { return i; }
    }
    throw ItemNotFound ("Not found");
}
```



Analysis of Sequential Searching

- Time spent in each iteration through the loop is at most some constant t_1
- Time spent outside the loop is at most some constant t_2
- **Maximum number of iterations** is n , the length of the array
- Hence, the asymptotic upper bound is:

$$t_1 n + t_2 = O(n)$$

- **Rule of Thumb:**

In general, a loop of n iterations will lead to $O(n)$ growth rate (complexity is linear).

```
int seqSearch (int[] a, int len, int x)
    throws ItemNotFound {
    for (int i = 0; i < len; i++) {
        if (a[i] == x) { return i; }
    }
    throw ItemNotFound ("Not found");
}
```

Example:

Binary Search Algorithm

Requires array to be *sorted*.

Maintain subarray where x might be located

Repeatedly compare x with m , the middle of current subarray

- If $x = m$, found it!
- If $x > m$, continue search in subarray after m
- If $x < m$, continue search in subarray before m

Example:

Non-recursive Binary Search

```
static int binSearch (int[] a, int len, int x)
                        throws ItemNotFound {
    int mid, low = 0;
    int high = len - 1;
    while (low <= high) {
        mid = (low + high) / 2;
        if (x == a[mid]) { return mid; }
        else if (x > a[mid]) low = mid + 1;
        else high = mid - 1;
    }
    throw ItemNotFound ("Not found");
}
```



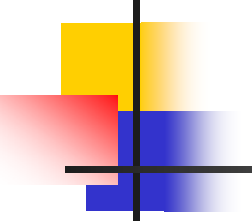
Analysis of Binary Search

- Time spent outside the loop is at most t_1
- Time spent in each iteration of the loop is at most t_2
- For inputs of size n , if we go through at most $f(n)$ iterations, then the complexity is

$$t_1 + t_2 f(n)$$

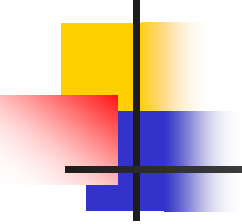
$$\text{or } O(f(n))$$

```
static int binSearch (int[] a, int len, int x)
    throws ItemNotFound {
    int mid, low = 0;
    int high = len - 1;
    while (low <= high) {
        mid = (low + high) / 2;
        if (x == a[mid]) { return mid; }
        else if (x > a[mid]) low = mid + 1;
        else high = mid - 1;
    }
    throw ItemNotFound ("Not found");
}
```

Bounding $f(n)$, the Number of Iterations

- At any point during binary search, part of array is "*alive*" (might contain the point x)
- Each iteration of loop eliminates at least half of previously "*alive*" elements
- At the beginning, all n elements are "*alive*", and after
 - One iteration, at most $n/2$ are left, or alive
 - Two iterations, at most $(n/2)/2 = n/4 = n/2^2$ are left
 - Three iterations, at most $(n/4)/2 = n/8 = n/2^3$ are left
 - :
 - k iterations, at most $n/2^k$ are left
 - At the final iteration, at most 1 element is left



Bounding $f(n)$, the Number of Iterations (cont.)

In the **worst case**, we have to search all the way up to the last iteration **k** with only one element left.

We have:

$$n/2^k = 1$$

$$2^k = n$$

$$k = \log n$$

Hence, the binary search algorithm takes $O(f(n))$, or $O(\log n)$ time

Rule of Thumb:

In general, when the domain of interest is **reduced by a fraction** for each iteration of a loop, then it will lead to $O(\log n)$ growth rate. The complexity is $\log_2 n$.



Analysis of Different Cases

Worst-Case Analysis

- interested in the worst-case behaviour.
- A determination of the maximum amount of time that an algorithm requires to solve problems of size n

Best-Case Analysis

- interested in the best-case behaviour not useful

Average-Case Analysis

- A determination of the average amount of time that an algorithm requires to solve problems of size n
- have to know the probability distribution the hardest



The Efficiency of Searching Algorithms

Example: Efficiency of Sequential search

- Worst case: $O(n)$

Which case?

- Average case: $O(n)$
- Best case: $O(1)$

Why? Which case?

Q: What is the best case complexity of binary search?
Best case complexity is not interesting.

Why?



Keeping Your Perspective

- If the problem size is always **small**, you can probably ignore an algorithm's efficiency
- Weigh the **trade-offs** between an algorithm's **time** requirements and its **memory** requirements
- Compare algorithms for both style and efficiency
- Order-of-magnitude analysis focuses on **large** problems