CS2040C Data Structures and Algorithms
Lecture 2 - Analysis of Algorithms

Big O!

Lecture Outline

- What is an algorithm?
- What is analysis of algorithms?
- How to analyze an algorithm?
- Big-O notation
- Examples

You are expected to know

- Operations on logarithm function
- Arithmetic and geometric progressions
  - Their sums
- Linear, Quadratic, Cubic, Polynomial
- Ceiling, Floor, Absolute Value

Algorithm and Analysis

- Algorithm:
  - A step-by-step procedure for solving a problem

- Analysis of Algorithm:
  - To evaluate rigorously the *resources* (time and space) needed by an algorithm and represent the result of the evaluation with a formula
  - We focus more on time requirement in our analysis
  - The time requirement of an algorithm is also called the time complexity of the algorithm

Limitation of exact running time

- We can measure the exact running time of a program
  - Use *wall clock time* or code inserted into program
- To compute the exact running time needed by an algorithm, the analysis will *depend* on:
  - Language in which the algorithm is coded
  - Data set: input to the algorithm
  - Computer that the algorithm is executed on
- Such analysis will make it difficult to compare two algorithms
- It is useful to know the behaviour of the algorithm **before** it is coded / executed
  - Poorly designed algorithm may take very long to finish execution

How to Analyse an Algorithm?

- Instead of measuring the exact timing
  - Count the number of operations needed
    - Operations: Arithmetic, Assignment, Comparison, etc.
  - Usually choose the “dominant” operations
- Example:

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

```
Total Ops
= A + B
= Σ{i = 1,n}100 + Σ{i = 1,n} (Σ{j = 1,n} 2)
= 100n + Σ{i = 1,n} (2n)
= 100n + 2n²
= 2n² + 100n
```

Time Requirement

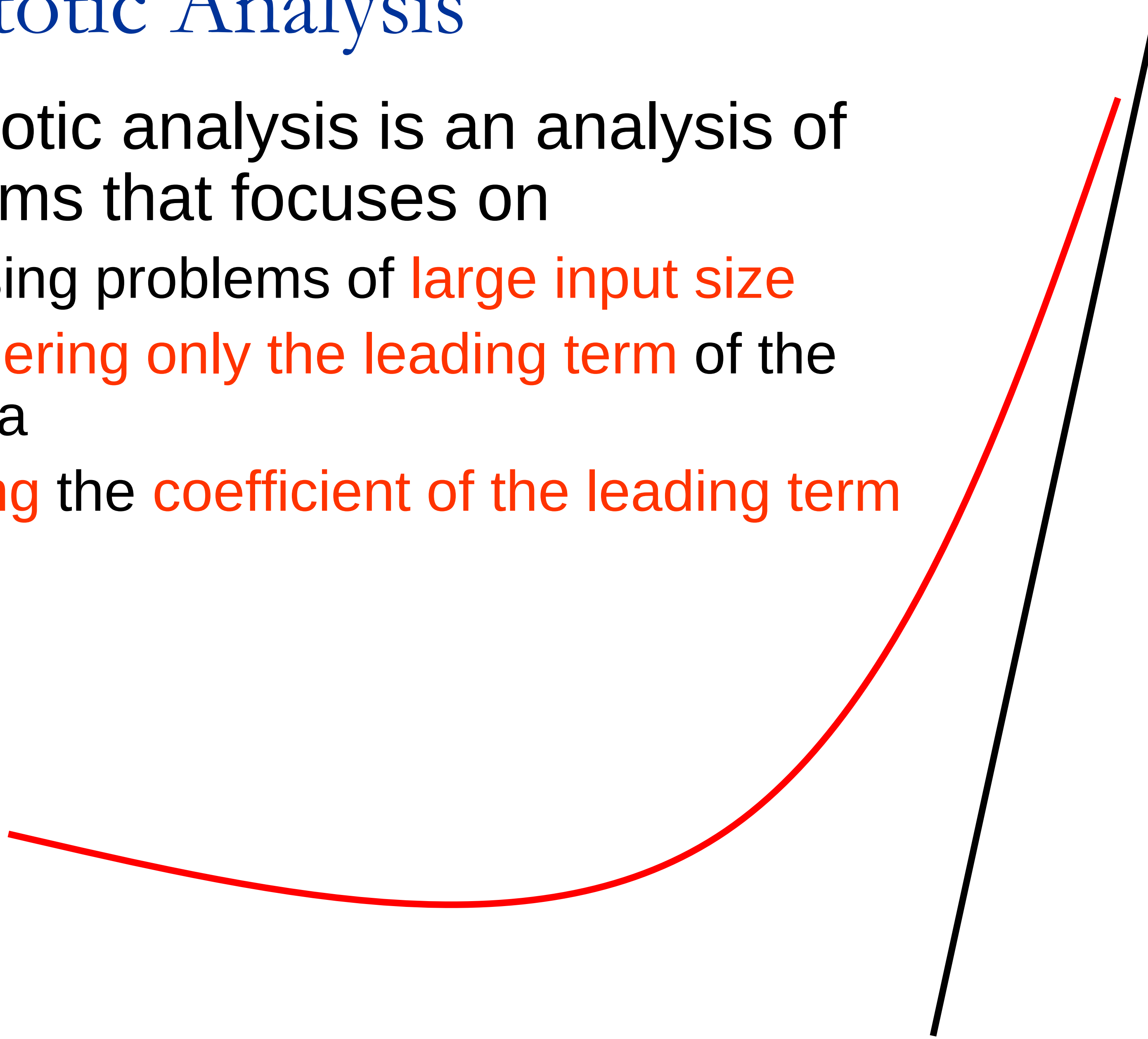
- Knowing the number of operations required for an algorithm A, we can state that:
  - E.g. Algorithm A takes $2n^2 + 100n$ operations to solve a problem of size n
- If the time t needed for one operation is known, then we can state:
  - E.g. Algorithm A takes $(2n^2 + 100n)*t$ time units
- However, time t is directly dependent on the factors mentioned earlier:
  - Computer and Programming Language
- Instead of tying the analysis to actual time t, we can state:
  - Algorithm A takes time that is **proportional** to $2n^2 + 100n$ for solving problem of size n

Approximation of Analysis Results

- Suppose the complexity of
  - Algorithm A is found to be **$3n^2$ + 2n + log n + 1/4n**
  - Algorithm B is found to be **$0.39n^3$ + n**
- Intuitively, we know Algorithm A will outperform B
  - when solving larger problems i.e. larger n
- The **dominating** term $3n^2$ and $0.39n^3$ can tell us approximately how the algorithms perform
  - Algorithm B with dominating term $0.39n^3$ is inferior
- The terms n^2 and n^3 are even simpler and are preferred
  - This term can be obtained through asymptotic analysis

Asymptotic Analysis

- Asymptotic analysis is an analysis of algorithms that focuses on
  - analysing problems of large input size
  - considering only the leading term of the formula
  - ignoring the coefficient of the leading term



Why choose the Leading Term?

- Lower order terms contribute lesser to the overall cost as the input grows larger
- Example:
  - `f(n) = 2n`^{2} `+ 100n`
  - `f(1000)      = 2(1000)`^{2} `+ 100(1000)`
    `= 2000000 + 100000`
  - `f(100000)    = 2(100000)`^{2} `+ 100(100000)`
    `= 20000000000 + 10000000`
- Hence, lower order terms can be **ignored**

Examples: Leading Term

- $a(n) = \frac{1}{2} n + 4$
  - leading term = $\frac{1}{2} n$
- $b(n) = 240n + 0.001n^2$
  - leading term = $0.001n^2$
- $c(n) = n \lg(n) + \lg(n) + n \lg(\lg(n))$
  - leading term = $n \lg(n)$

Coefficient of the Leading Term

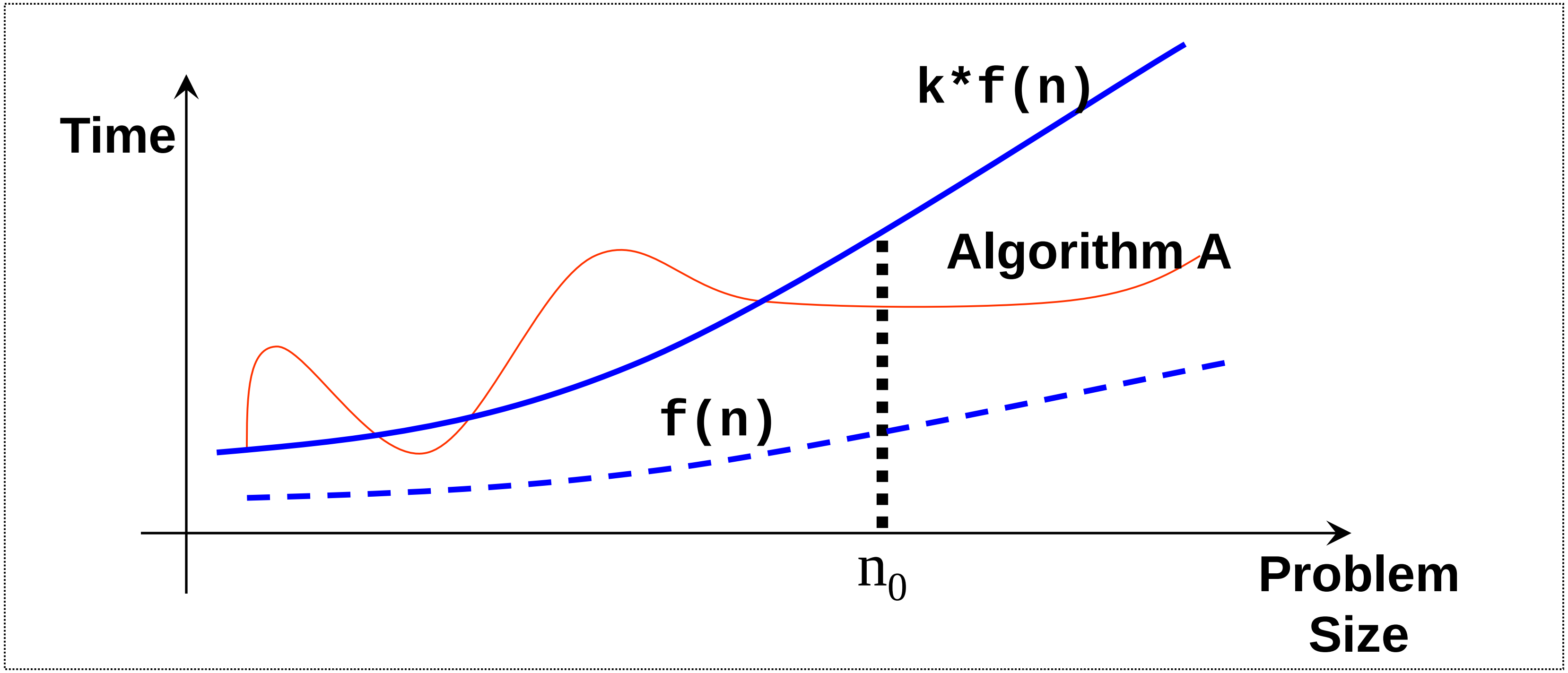
- Suppose two algorithms have $2n^2$ and $30n^2$ as the leading term of their time complexity respectively
  - Although **actual** time will be different due to the constant, the **growth rate** of the algorithms is the same
  - E.g. Compare the algorithms with another algorithm with leading term of $\mathbf{n^3}$, the difference in growth rate is a much more dominating factor
- Hence, we can drop the coefficient of leading term when studying algorithm complexity
  - A more formal reasoning for this decision will be covered later

Upper Bound: the Big-O notation

- If algorithm A requires time proportional to f(n):
  - Algorithm A **is of the order of** `f(n)`
  - Denoted as Algorithm **A is** `O(f(n))`
  - f(n) is the **growth rate function** for Algorithm A
- Formal definition:
  - Algorithm A is of `O(f(n))` if there exist a constant **k**, and a positive integer n_0 such that Algorithm A requires **no more** than `k * f(n)` time units to solve a problem of size $n \geq n_0$

Big-O: Illustration

- When problem size is larger than n_0, Algorithm A is **bounded from above** by `k * f(n)`
- Observations:
  - **n_0** and **k** are **not unique**
  - There are a number of possible `f(n)`



Example: Finding n_0 and constant k

- Given Complexity of algorithm A = $2n^2 + 100n$

Claim: Algorithm A is of $O(n^2)$

[Solution]
$2n^2 + 100n < 2n^2 + n^2 = 3n^2$ whenever $n>100$
Set the constants to be $k = 3$ and $n_0 = 100$
By definition, we say algorithm A is $O(n^2)$

- Question:
  - Can we say A is $O(2n^2)$ or $O(3n^2)$?
  - Can we say A is $O(n^3)$?

Growth Rate Function

- Using the definition of Big-O notation, it is clear that:
  - Coefficient of the f(n) can be absorbed into the constant k
  - E.g. **A** is $O(3n^2)$ with constant k_1

    ➔ **A** is $O(n^2)$ with constant $k = k_1 * 3$
  - So, **f(n)** should be function with coefficient of 1 only
- Such a term is called a **growth term (order of growth, Order-of-Magnitude)**
- The most common **growth terms** can be ordered as follows:

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

Best ⟷ Worst

Note: $\lg(n) = \log_2(n)$

Growth Rate: Terminology

- **O(1): Constant time**
  - independent of n
- **O(n): Linear time**
  - Grows as the same rate of n
  - E.g. double input size == double execution time
- **$O(n^2)$: Quadratic time**
  - Increase rapidly w.r.t. n
  - E.g. double input size == quadruple execution time
- **$O(n^3)$: Cubic time**
  - Increase even more rapidly w.r.t. n
  - E.g. double input size == 8 * execution time
- **$O(2^n)$: Exponential time**
  - Increase very very rapidly w.r.t. n

Example: Exponential Time

- Suppose we have a problem that, for an input consisting of n items, can be solved by going through 2^n cases
- 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

Example: Quadratic Time

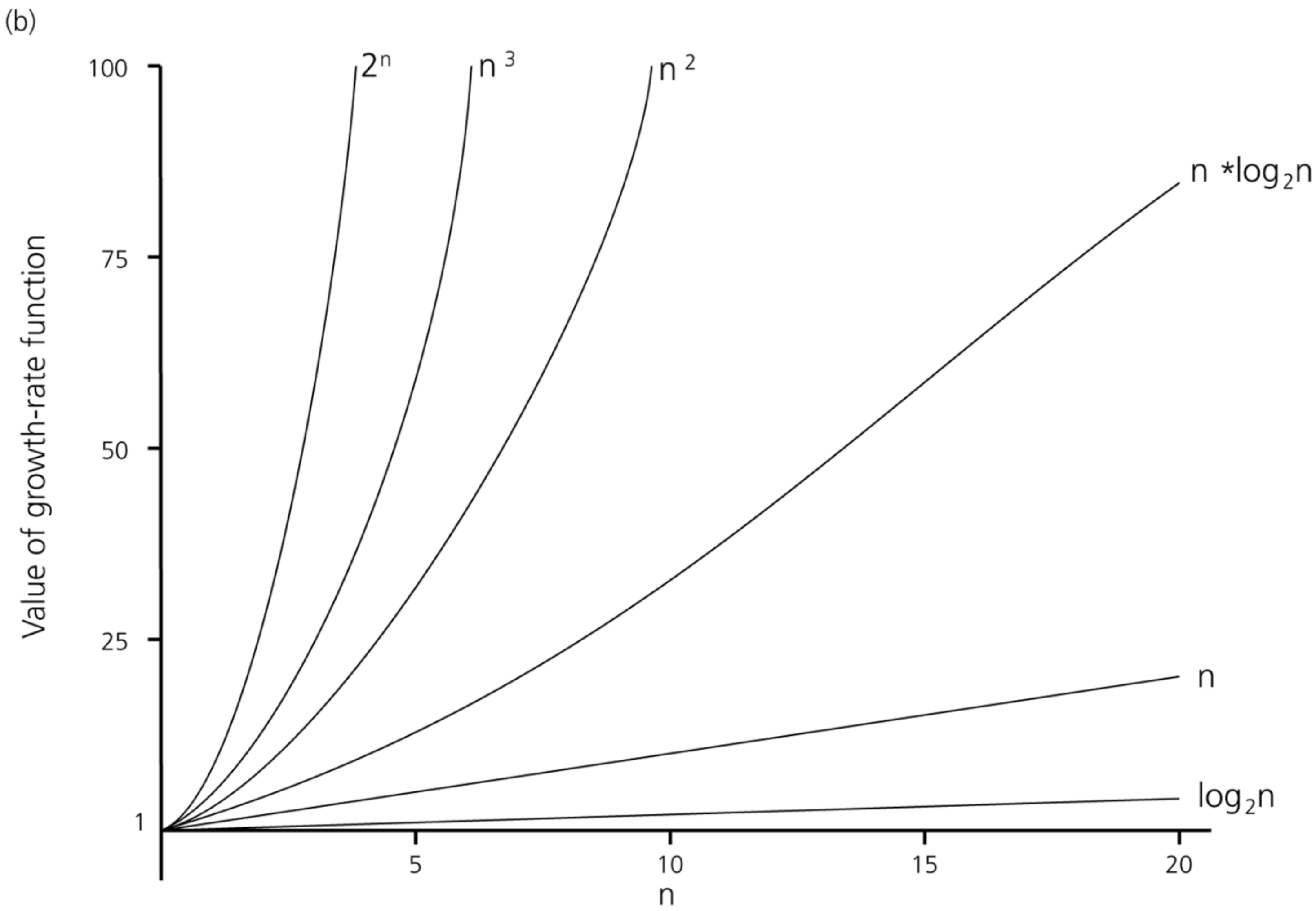
- Suppose solving the same problem with another algorithm will use *$300n^2$* clock cycles on a *Handheld PC*, running at *33* MHz
  - Input with *15* items, *2* milliseconds
  - Input with *30* items, *8* milliseconds
  - Input with *50* items, *22* milliseconds
  - Input with *80* items, *58* milliseconds
- So, to speed up our program, do not simply depend on the raw power of a computer.
- It is very important to use an efficient algorithm to solve a problem

Growth Rate : Illustration

(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}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Growth Rate : Illustration



Algorithm Analysis Examples

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

$i = 1, 2, 4, 8, \ldots, 2^k = n$ where $k = \text{floor}(\log n)$.

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?

$$
\begin{aligned}
f(n) &= 1 + 3 + 9 + 27 + \ldots\ldots + 3\wedge(\log_3 n) \\
&= n + n/3 + n/9 + \ldots + 1 \\
&= n(1 + 1/3 + 1/9 + \ldots) \\
&<= 3/2\, n \\
&= O(n)
\end{aligned}
$$

Analysis 1: Tower of Hanoi

- Number of moves made by the algorithm is $2^n - 1$

  (try to proof it by induction)
- Assume each move takes c_1 time, then Tower of Hanoi takes $c_1(2^n-1) = O(2^n)$

- The Tower of Hanoi algorithm is an **exponential time** algorithm

Analysis 2: Sequential Search

```
int seqSearch (int a[], int len, int x)
{
   for (int i = 0; i < len; i++) {
      if (a[i] == x)
         return i;
   }
   return -1;
}
```

Analysis 2: Sequence Searching

- Time spent in each iteration through the loop is at most some constant c_1
- Time spent outside the loop is at most some constant c_2
- Maximum number of iterations is n
- Hence, the asymptotic upper bound is:

  $c_1 n + c_2 = \mathbf{O(n)}$
- Observations:
  - In general, a loop of n iterations will lead to **O(n)** growth rate.
  - This is an example of **Worst Case Analysis**

Analysis 3: Binary Searching

- Important characteristics:
  - Requires array to be *sorted*
  - Maintains sub-array where x might be located
  - Repeatedly compares x with m, the middle of current sub-array
    - If x == m, found it!
    - If x > m, eliminate m and positions *before* m
    - If x < m, eliminate m and positions *after* m
- Two implementations: iterative and recursive

Binary Search (recursive)

```
int binarySearch (int a[], int x, int low, int high)
{
   if (low > high)   // Base Case 1: item not found
      return -1;

   int mid = (low + high) / 2;

   if (x > a[mid])
      return binarySearch (a, x, mid + 1, high);
   else if (x < a[mid])
      return binarySearch (a, x, low, mid - 1);
   else
      return mid;     // Base Case 2: item found
}
```

Binary Search (iterative)

```
int binSearch (int a[], int len, int x)
{
    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;
    }
    return -1;
}
```

Analysis 3: Iterative Binary Searching

- Time spent outside the loop is at most c_1
- Time spent in each iteration of the loop is at most c_2
- For inputs of size n, if the program goes through at most $f(n)$ iterations, then the complexity is

  $$c_1 + c_2 f(n)$$

  or $\mathbf{O}(f(n))$
- i.e. the complexity is decided by the number of iterations (loops)

Analysis 3: Finding f(n)

- 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

Analysis 3: Finding f(n)

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

$$n/2^k = 1$$

$$2^k = n$$

$$k = \log_2(n) = \lg(n)$$

- Hence, the binary search algorithm takes **O**(f(n)), or **O(lg(n))** time
- Observation:
  - 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.

Analysis of Different Cases

- For any algorithm, there are **three different cases of analysis:**
  - **Worst-Case Analysis:**
    - The worst possible scenario
  - **Best-Case Analysis:**
    - The ideal case
    - Usually not useful
  - **Average-Case Analysis**:
    - probability distribution should be known
    - hardest/impossible to analyse

- E.g. Use linear searching to locate an item
  - **Worst-Case:** target item at the tail of array
  - **Best-Case:** target item at the head of array
  - **Average-Case:** target item can be anywhere

Summary

- Algorithm Definition
- Algorithm Analysis
  - Asymptotic Analysis
  - Big-O notation (Upper-Bound)
- Three cases of analysis
  - Best-case
  - Worst-case
  - Average-case