Lecture 9

Analysis of Algorithms

Measuring Algorithm Efficiency

Lecture Outline

- What is an Algorithm?
- What is Analysis of Algorithms?
- How to analyze an algorithm
- Big-O notation
- Example Analyses

You are expected to know…

- Proof by induction
- Operations on logarithm function
- Arithmetic and geometric progressions
  - Their sums
  - See L9 – useful_formulas.pdf for some of these
- Linear, quadratic, cubic, polynomial functions
- 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
  - For **this** module, we focus more on **time** requirement in our analysis
  - The time requirement of an algorithm is also called the **time complexity** of the algorithm

Measure Actual Running Time?

- We can measure the actual running time of a **program**
  - Use **wall clock time** or insert timing code into program
- However, actual running time is not meaningful when **comparing** two **algorithms**
  - Coded in different languages
  - Using different data sets
  - Running on different computers

Counting Operations

- Instead of measuring the actual timing, we count the number of **operations**
  - Operations: arithmetic, assignment, comparison, etc.
- 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

Example: Counting Operations

- 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
  }
}
```

Total Ops = A + B $= \sum_{i=1}^{n} 100 + \sum_{i=1}^{n} (\sum_{j=1}^{n} 2)$

$$= 100n + \sum_{i=1}^{n} 2n \quad = 100n + 2n^2 \quad = 2n^2 + 100n$$

Example: Counting Operations

- Knowing the number of operations required by the algorithm, we can state that
  - Algorithm *X* takes $2n^2 + 100n$ operations to solve problem of size *n*
- If the time *t* needed for one operation is known, then we can state
  - Algorithm *X* takes $(2n^2 + 100n)t$ time units

Example: Counting Operations

- However, time *t* is directly dependent on the factors mentioned earlier
  - e.g. different languages, compilers and computers
- Instead of tying the analysis to actual time *t*, we can state
  - Algorithm *X* takes time that is **proportional to** $2n^2 + 100n$ for solving problem of size *n*

Approximation of Analysis Results

- Suppose the time complexity of
  - Algorithm *A* is $\mathbf{3n^2} + 2n + \log n + 1/(4n)$
  - Algorithm *B* is $\mathbf{0.39n^3} + n$
- Intuitively, we know Algorithm *A* will outperform *B*
  - When solving larger problem, i.e. larger *n*
- The **dominating term** $\mathbf{3n^2}$ and $\mathbf{0.39n^3}$ can tell us approximately how the algorithms perform
- The terms $\mathbf{n^2}$ and $\mathbf{n^3}$ are even simpler and preferred
- These terms can be obtained through **asymptotic analysis**

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

Why Choose Leading Term?

- Lower order terms contribute lesser to the overall cost as the input grows larger
- Example
  - $f(n) = 2n^2 + 100n$
  - $f(1000) = 2(1000)^2 + 100(1000)$
    $= 2{,}000{,}000 + 100{,}000$
  - $f(100000) = 2(100000)^2 + 100(100000)$
    $= 20{,}000{,}000{,}000 + 10{,}000{,}000$
- Hence, lower order terms can be ignored

Examples: Leading Terms

- $a(n) = \frac{1}{2} n + 4$
  - 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: $n \lg(n)$
  - Note that $\lg(n) = \log_2(n)$

Why Ignore Coefficient of Leading Term?

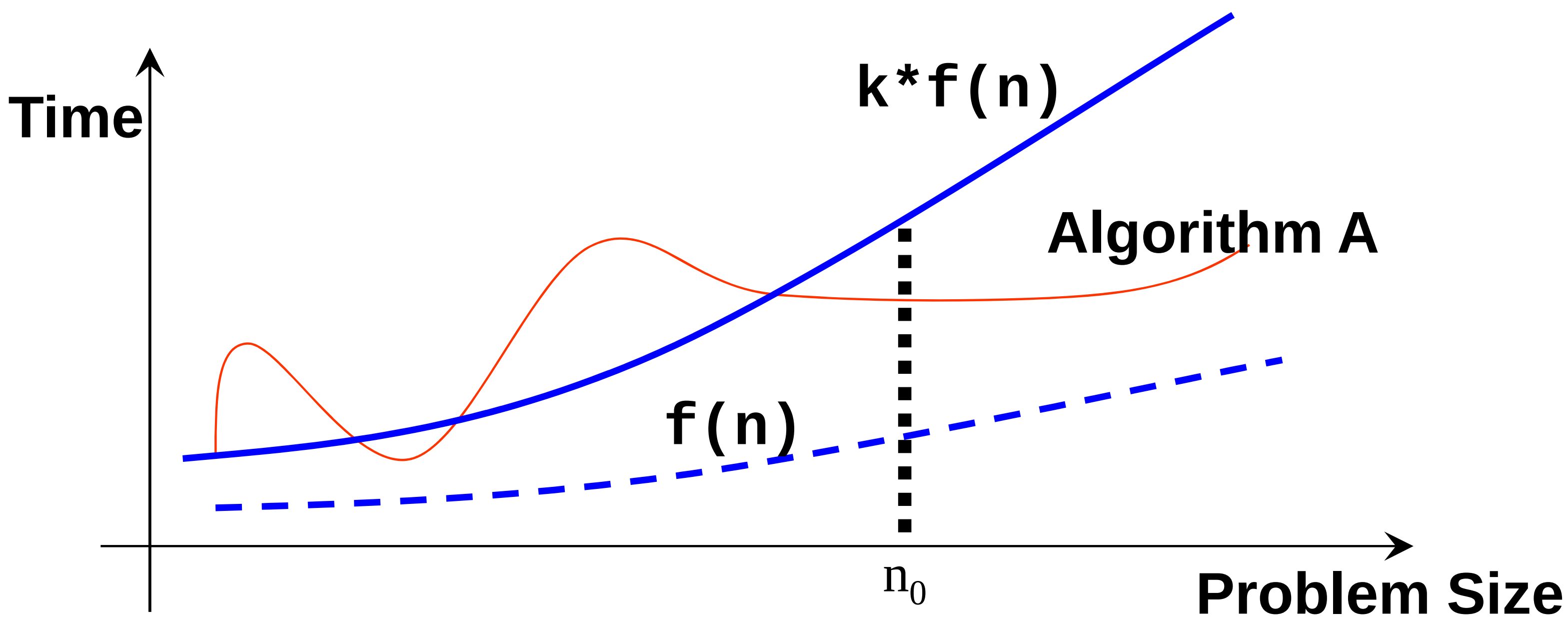
- Suppose two algorithms have $2n^2$ and $30n^2$ as the leading terms, respectively
- Although actual time will be different due to the different constants, the **growth rates** of the running time are the same
- Compare with another algorithm with leading term of n^3, the difference in growth rate is a much more dominating factor
- Hence, we can drop the coefficient of leading term when studying algorithm complexity

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 $\mathbf{O}(f(n))$
  - $f(n)$ is the **growth rate function** for Algorithm *A*

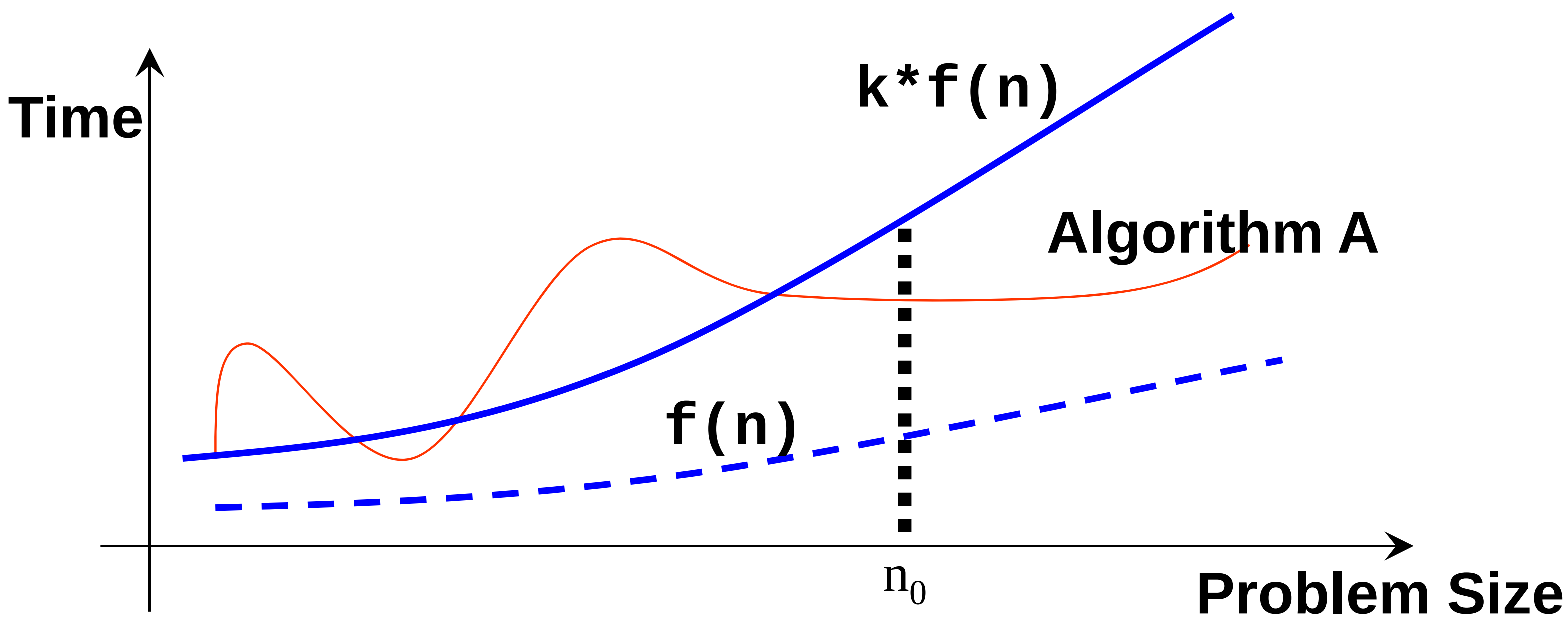
The Big-O Notation

- Formal definition
  - Algorithm *A* is of **O(*f*(*n*))** if there exist a constant ***k***, and a positive integer $\mathbf{n_0}$ such that Algorithm *A* requires no more than $k * f(n)$ time units to solve a problem of size $n >= n_0$



The Big-O Notation

- 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 many possible $f(n)$



Example: Finding n_0 and k

- Given complexity of Algorithm *A* is $2n^2 + 100n$
- **Claim:** Algorithm *A* is of $O(n^2)$
- Solution
  - $2n^2 + 100n < 2n^2 + n^2 = \mathbf{3n^2}$ whenever $\boldsymbol{n > 100}$
  - Set the constants to be $\boldsymbol{k = 3}$ and $\boldsymbol{n_0 = 100}$
  - By definition, we say Algorithm *A* is $\mathbf{O(n^2)}$
- Questions
  - Can we say *A* is $O(2n^2)$ or $O(3n^2)$?
  - Can we say *A* is $O(n^3)$?

Growth Terms

- In asymptotic analysis, a formula can be simplified to a **single term** with **coefficient 1** (how?)
- 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 (note that many others are not shown)

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

*"**fastest**"* *"**slowest**"*

- "log" = $\log_2$
- In big-O, log functions of different bases are all the same (why?)

Common Growth Rates

- **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**
  - Increases rapidly w.r.t. n
  - E.g. double input size → quadruple execution time
- **O(n^3) — cubic time**
  - Increases even more rapidly w.r.t. n
  - E.g. double input size → 8 * execution time
- **O(2^n) — exponential time**
  - Increases very very rapidly w.r.t. n

Example: Exponential-Time Algorithm

- 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 Algorithm

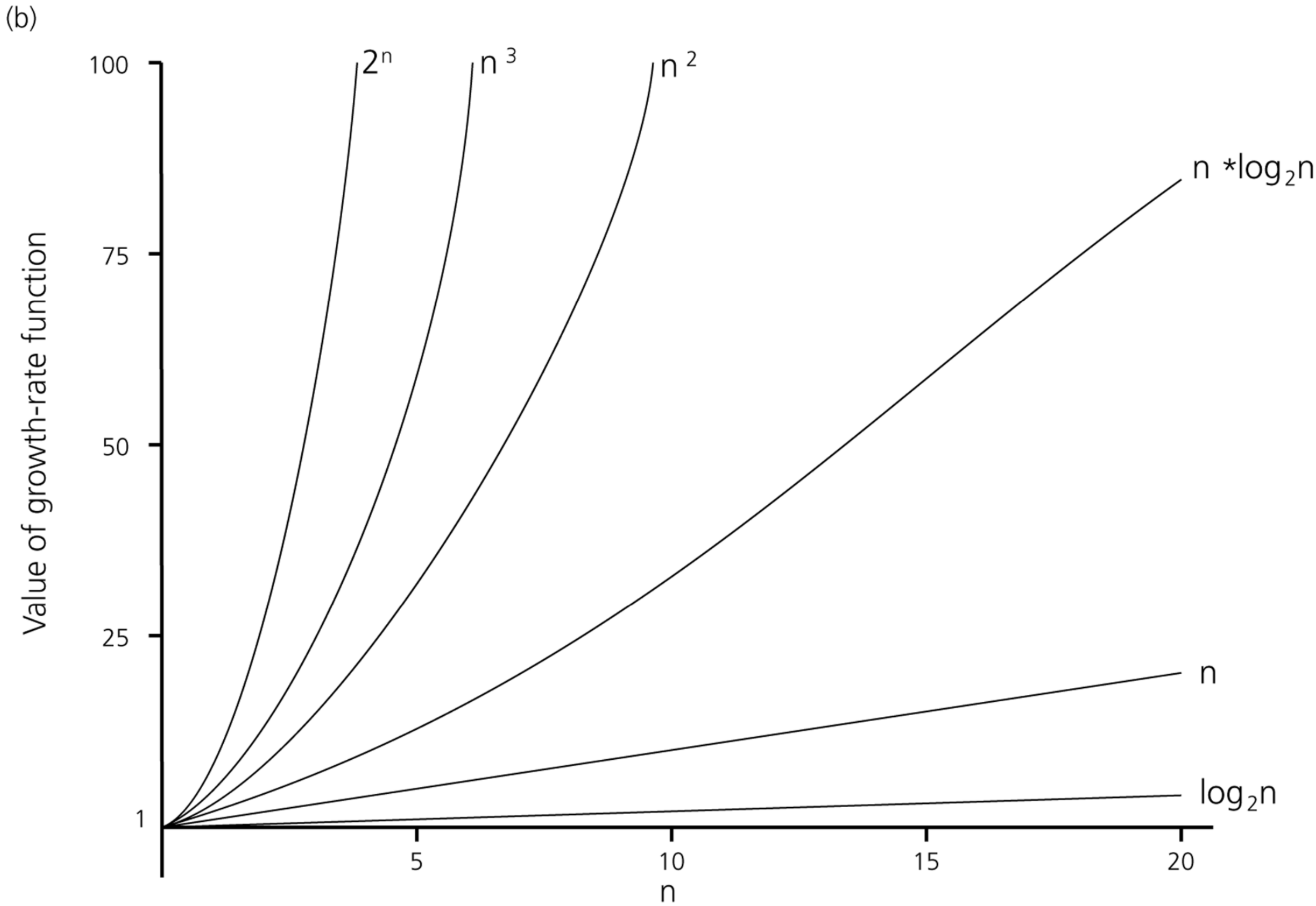
- Suppose solving the same problem with another algorithm will use $300n^2$ clock cycles on a Handheld PC, running at 33 MHz
  - Input with 15 items — 2 milliseconds
  - Input with 30 items — 8 milliseconds
  - Input with 50 items — 22 milliseconds
  - Input with 80 items — 58 milliseconds
- Therefore, to speed up program, don't simply rely on the raw power of a computer
  - Very important to use an efficient algorithm

Comparing Growth Rates

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

Comparing Growth Rates



How to Find Complexity?

- Some rules of thumb
  - Basically just count the 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(*n***m*)**
  - For a loop with a range of values *n*, and each iteration reduces the range by a fixed constant fraction (eg: ½) — **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*)**

Example: Finding Complexity (1/2)

- What is the complexity of the following code fragment?

```
int sum = 0;
for (int i = 1; i < n; i = i*2) {
  sum++;
}
```

- It is clear that **sum** is incremented only when

  i = 1, 2, 4, 8, …, 2^k where $k = \lfloor \log_2 n \rfloor$

  There are k + 1 iterations.
  So the complexity is O(k) or O(log n)

Example: Finding Complexity (2/2)

- What is the complexity of the following code fragment?
  - For simplicity, let's assume that n is some power of 3

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

- $f(n) = 1 + 3 + 9 + 27 + \ldots + 3^{(\log_3 n)}$
  $= 1 + 3 + \ldots + n/9 + n/3 + n$
  $= n + n/3 + n/9 + \ldots + 3 + 1$
  $= n * (1 + 1/3 + 1/9 + \ldots)$
  $\leq n * (3/2)$
  $= 3n/2$
  $= O(n)$

Analysis 1: Tower of Hanoi

- 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 prove by induction

- Assume each move takes c time, then

  $f(n) = c(2^n - 1) = O(2^n)$

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

Analysis 2: Sequential Search

- Check whether an item *x* is in an unsorted array a[ ]
  - If found, it returns position of *x* in array
  - If not found, it returns -1

```
public 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: Sequential Search

- 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
  $c_1 n + c_2 = O(n)$
- Observation
  - 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 Search

- Important characteristics
  - Requires array to be sorted
  - Maintain sub-array where *x* might be located
  - Repeatedly compare *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*
- Iterative and recursive implementations

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;  // item not found
}
```

Analysis 3: Binary Search (Iterative)

- 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

  $c_1 + c_2f(n)$ or $O(f(n))$

- 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 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

$$n/2^k = 1 \quad \Rightarrow \quad 2^k = n \quad \Rightarrow \quad k = \log_2(n) = \lg(n)$$

- 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 an algorithm, three different cases of analysis
  - **Worst-Case Analysis**
    - Look at the worst possible scenario
  - **Best-Case Analysis**
    - Look at the ideal case
    - Usually not useful
  - **Average-Case Analysis**
    - Probability distribution should be known
    - Hardest/impossible to analyze
- Example: Sequential Search
  - **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
  - Counting operations
  - Asymptotic Analysis
  - Big-O notation (Upper-Bound)
- Three cases of analysis
  - Best-case
  - Worst-case
  - Average-case