AST20105 Data Structures & Algorithms

CHAPTER 3 –
DESIGN & ANALYSIS OF ALGORITHMS

Instructed by Garret Lai

Helpful sites: http://graphsketch.com/ and https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/asymptotic-notation

Before Start…

- What is Algorithm?
  - Recall, an algorithm is a clearly specified set of simple instructions for solving a problem

Problem example:
Calculate discount-rate
for customers in a retail store

Before Start…

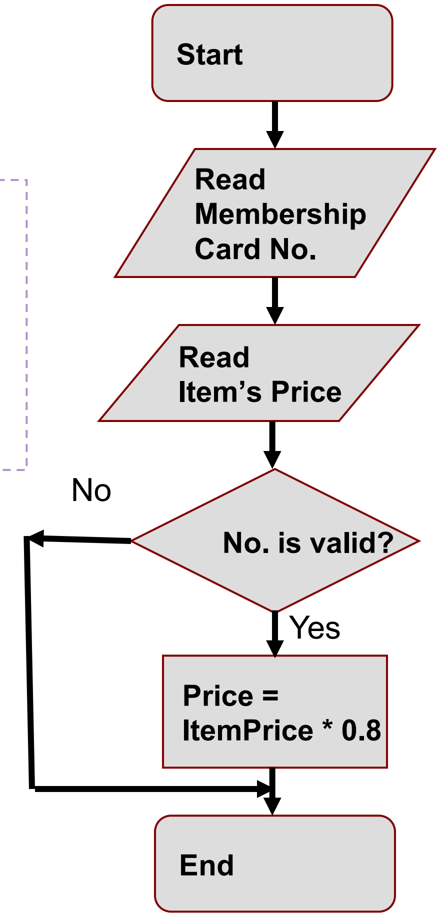
- It can be specified using
  - Flowchart
  - Pseudo-code
  - Computer program

READ Membership Card No.
READ ItemPrice
IF Membership Card No. is valid **THEN**
  price = ItemPrice * 0.8;
ELSE
  price = ItemPrice;
END IF

Pseudo code

```
int main() {
    int cardNo;
    cin >> cardNo;
    int itemPrice;
    cin >> itemiPrice;
    int price;
      if (is_valid(cardNo))
         price = itemPrice * 0.8;
      else
         price = itemPrice;
}
```

Computer program



Flowchart

Common Algorithm Design Paradigms

- Brute force
  - A straightforward approach to solve a problem based on the problem's statement
- Divide and conquer (D&C)
  - Recursively break down a problem into two or more sub-problems of the same type until those problems are simple enough to be solved directly
  - The solutions of the sub-problems are combined to give a solution of the original problem

Common Algorithm Design Paradigms

- Greedy
  - Making the local optimal choice at each stage with the hope of finding global optimum for the problem
  - But the final solution may not be optimal

- Dynamic programming
  - For solving complex problems by breaking down the problem into a number of simpler sub-problems
  - Only applicable to problems exhibiting the properties of overlapping sub-problems which are only slightly small and optimal sub-structure

We will only focus on doing Divide and Conquer in this course!

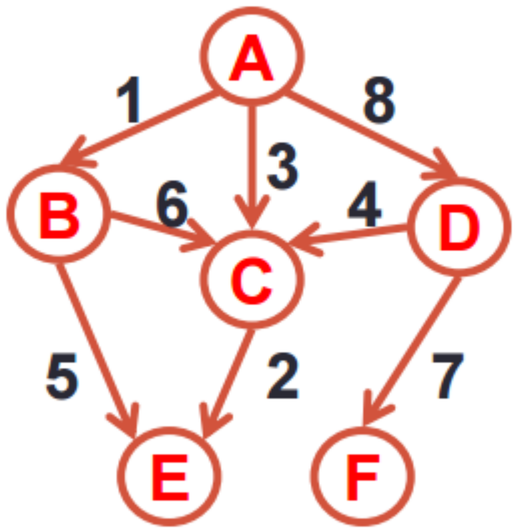
Common Algorithm Design Methodologies

- Apart from design paradigms, design methodologies are also important for coming up good algorithm
  - Top-down approach
    - Progress from the initial problem down to the smallest sub-problems via intermediate sub-problems
    - Example: Divide & Conquer
  - Bottom-up approach
    - Smallest sub-problems are solved first and then the results used to construct solutions to progressively larger sub-problems
    - Example: Dynamic programming
  - Others
    - Hierarchical modularization
    - Stepwise refinement

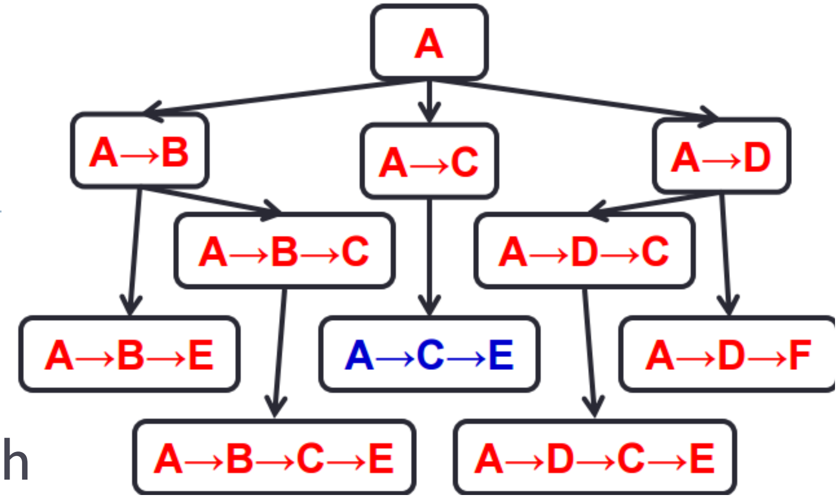
Analysis of Algorithms

- Why algorithm analysis?
  - Writing a working program is not sufficient
  - The program may not be efficient
  - If the program is run on a large set of data, then the running time of a program becomes a problem

- Example:
  - Suppose we wish to find the shortest path to go from point A to E.



Example



- A brute-force approach:
  - Examine all paths in graph
  - Compute the travel time for each
  - Choose the shortest one
  - How many paths are there?
    - Number of path = n! (for every node *i*, you have to compare n – i nodes)
  - How many paths if n = 50?
    - Number of path is around 3.04×10^{64}
- Other better approach exists

Analysis of Algorithms

- The same problem can frequently be solved with algorithms that differ in efficiency.
- The differences between the algorithms may be
  - immaterial for processing a small number of data items,
  - but these differences grow with the amount of data.

Analysis of Algorithms

- To compare the efficiency of algorithms,
  - a measure of the degree of difficulty of an algorithm called computational complexity was developed by
    - Juris Hartmanis and Richard E.Stearns.

Computational Complexity

Computational Complexity

- Computational complexity indicates how much effort is needed to apply an algorithm or how costly it is.
- This cost can be measured in a variety of ways, and the particular context determines its meaning.

Computational Complexity

- In this course, we concern with two efficiency criteria:
  - Time
  - Space

Computational Complexity

- The factor of time is usually more important than that of space, so efficiency considerations usually focus on the amount of time elapsed when processing data.

Computational Complexity

- The most inefficient algorithm run on a Cray computer can execute much faster than the most efficient algorithm run on a PC, so run time is always system-dependent.

- For example, to compare 100 algorithms, all of them would have to be run on the same machine.

Computational Complexity

- Furthermore, the results of run-time tests depend on the language in which a given algorithm is written, even if the tests are performed on the same machine.

- If programs are compiled, they execute much faster than when they are interpreted.

  - A program written in C or Ada may be 20 times faster than the same program encoded in BASIC or LISP.

Computational Complexity

- To evaluate an algorithm's efficiency, real-time units such as microseconds and nanoseconds should not be used.

- Rather, logical units that express a relationship between the size n of a file or an array and the amount of time t required to process the data should be used.

Computational Complexity

- If there is a linear relationship between the size n and time t – that is $t_1 = cn_1$, then an increase of data by a factor of 5 results in the increase of the execution time by the same factor;
  - If $n_2 = 5n_1$, then $t_2 = 5t_1$.
- Similarly, if $t_1 = \log_2 n$, then doubling n increases t by only one unit of time. Therefore,
  - if $t_2 = \log_2(2n)$, then $t2 = t1 + 1$.

Computational Complexity

- A function expressing the relationship between n and t is usually much more complex, and calculating such a function is important only in regard to large bodies of data;

- Any terms that do not substantially change the function's magnitude should be eliminated from the function.

Computational Complexity

- The resulting function gives only an approximate measure of efficiency of the original function.

- However, this approximation is sufficiently close to the original, especially for a function that processes large quantities of data.

Computational Complexity

- This measure of efficiency is called asymptotic complexity and is used when
  - disregarding certain terms of a function to express the efficiency of an algorithm or
  - when calculating a function is difficult or impossible and only approximations can be found.

Computational Complexity

- To illustrate the case, consider the following example:
  - $f(n) = n^2 + 100n + \log_{10}n + 1000$

Computational Complexity

- $f(n) = n^2 + 100n + \log_{10}n + 1000$
  - For small values of n, the last term 1000 is the largest.
  - When n equals 10, the second (100n) and last (1000) terms are on equal footing with the other terms, making a same contribution to the function value.

Computational Complexity

- $f(n) = n^2 + 100n + \log_{10}n + 1000$
  - When n reaches the value of 100, the first and the second terms make the same contribution to the result.
  - But when n becomes larger than 100, the contribution of the second term becomes less significant.

Computational Complexity

- Hence, for large values of n, due to the quadratic growth of the first term (n^2), the value of the function f depends mainly on the value of this first term.

- Other terms can be disregarded for large n.

Machine Independent Algorithm Analysis

- We assume that every basic operation takes constant time

  - Examples:
    - Addition
    - Subtraction
    - Multiplication
    - Comparison
    - Assignment

Machine Independent Algorithm Analysis

- Efficiency of an algorithm is the number of basic operations it performs

  - We do not distinguish the efficiency between basic operations, since we don't care about the exact values, but the asymptotic values (growth rate)

Machine Independent Algorithm Analysis

- If an algorithm needs n basic operations and another needs 2n basic operations, we will consider them to be in the same efficiency category

- However, we distinguish between $\log_2 n$, n, 2^n

Input size n	$\log_2 n$	n	2n	2^n
16	4	16	32	2^{16}
64	6	64	128	2^{64}
128	7	128	256	2^{128}
256	8	256	512	2^{256}
512	9	512	1024	2^{512}
1024	10	1024	2048	2^{1024}

Order of Growth

- The running time of an algorithm can be described as a function of n
- To establish a relative order among functions for large n
- Several asymptotic notations are used:
  - Big-Oh: Class of functions f(n) that grow no faster than g(n)
  - Big-Omega: Class of functions f(n) that grow at least as fast as g(n)
  - Big-Theta: Class of functions f(n) that grow at the same rate as g(n)

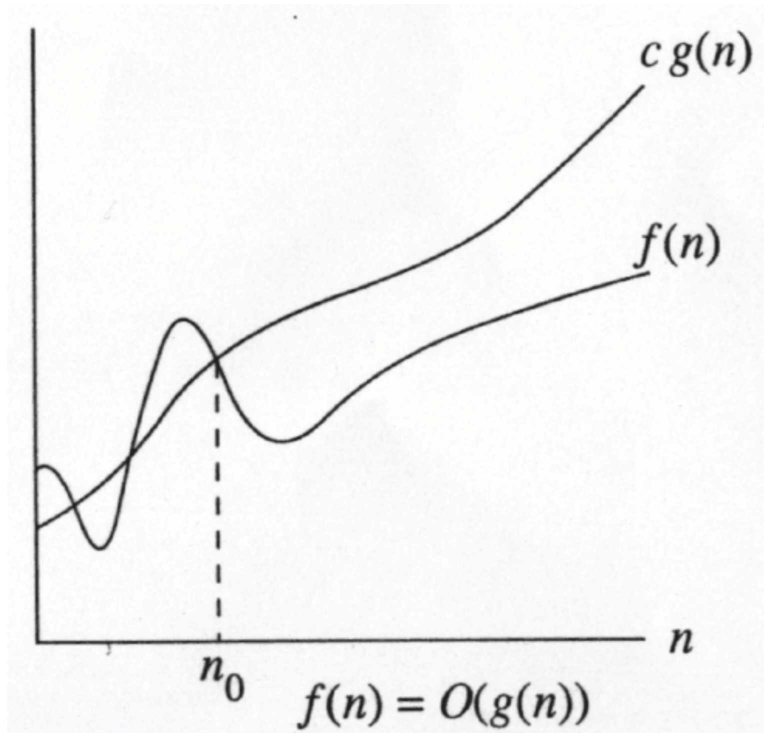
Asymptotic Notation: Big-Oh

- $f(n) = O(g(n))$
  The growth rate of f(n) is less than or equal to the growth rate of g(n)

- There are positive constant c and n_0 such that

  $$f(n) \le cg(n), \text{when } n \ge n_0$$

- g(n) is an upper bound of f(n)



Big-Oh Examples

- Let $f(n) = 2n^2$, then
  - $f(n) = O(n^2)$
    Since $2n^2 <= cn^2$, where $c >= 2$ and $n>=1$

    Best answer, tight!

  - $f(n) = O(n^3)$
    Since $2n^2 <= cn^3$, where $c >= 2$ and $n>=1$
  - $f(n) = O(n^4)$
    Since $2n^2 <= cn^4$, where $c >= 2$ and $n>=1$
  - $f(n) = O(n^5)$
    Since $2n^2 <= cn^5$, where $c>= 2$ and $n>=1$

Rules for Big-Oh

- When considering the growth rate of a function using Big-Oh
  - Ignore the lower order terms
  - Ignore the coefficients of the highest order term
  - Not necessary to specify the base of logarithm
    - Since the base change only change the value of logarithm by a constant factor
    - Recall:

$$\log_a b = \frac{\log_c b}{\log_c a}$$

  - If $f_1(n) = O(g(n))$ and $f_2(n) = O(h(n))$, then
    - $f_1(n) + f_2(n) = \max(O(g(n)), O(h(n)))$
    - $f_1(n) * f_2(n) = O(g(n) * h(n))$

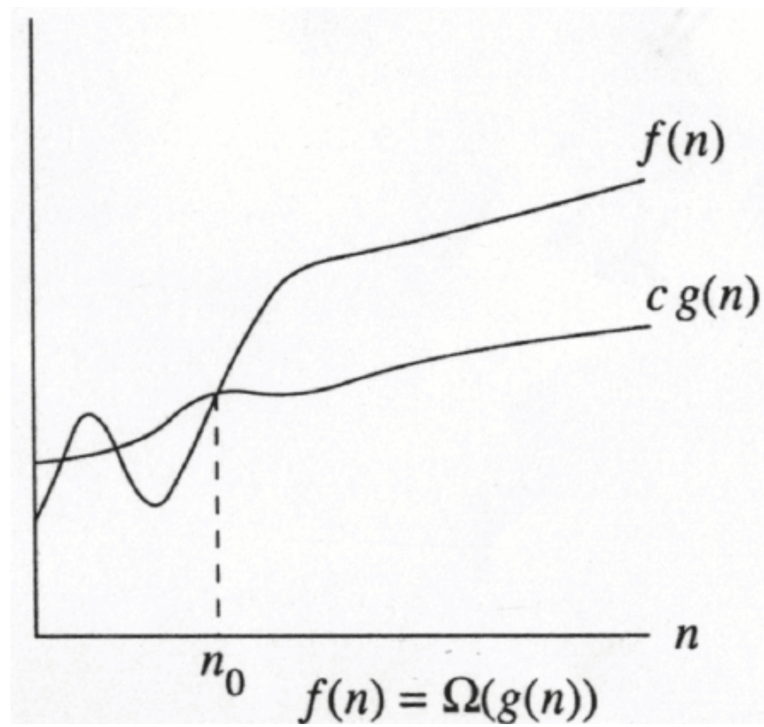
Asymptotic Notation: Big-Omega

- $f(n) = \Omega(g(n))$
  The growth rate of f(n) is greater than or equal to the growth rate of g(n)

- There are positive constant c and n_0 such that

  $$f(n) \geq cg(n), \text{when } n \geq n_0$$

- g(n) is a lower bound of f(n)



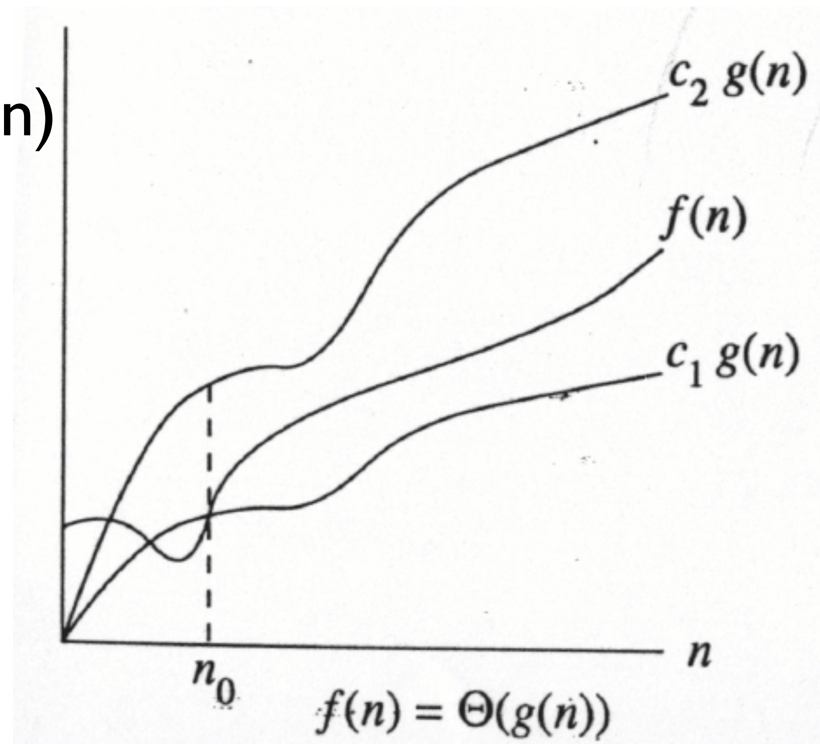
Big-Omega Examples

- Let $f(n) = 2n^2$, then
  - $f(n) = \Omega(n)$
    Since $2n^2 >= cn$, where $c <= 2$ and $n>=1$
  - $f(n) = \Omega(n^2)$
    Since $2n^2 >= cn^2$, where $c <= 2$ and $n>=1$

Best answer, tight!

Asymptotic Notation: Big-Theta

- f(n) = Θ(g(n))
  The growth rate of f(n) is the same as the growth rate of g(n)



- f(n) = Θ(g(n)) if and only if
  - f(n) = O(g(n)) and
  - f(n) = Ω (g(n))

Using L' Hopital's Rule

Only for students with calculus background!

- L' Hopital's rule

  $$\text{If } \lim_{n\to\infty} f(n) = \infty \text{ and } \lim_{n\to\infty} g(n) = \infty$$

  $$\text{then } \lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

- Determine the relative growth rates by using L' Hopital's rule, compute $\lim_{n\to\infty} \frac{f(n)}{g(n)}$

  - If it is 0, f(n) = O(g(n))
  - If it is ∞, f(n) = Ω(g(n))
  - If it is a constant not equal to 0, f(n) = Θ(g(n))

Examples

Only for students with calculus background!

$$1.\ \lim_{n\to\infty}\frac{\sqrt{n}}{n}=\lim_{n\to\infty}\frac{n^{1/2}}{n}=\lim_{n\to\infty}\frac{\frac{1}{2}n^{-1/2}}{1}=\lim_{n\to\infty}\frac{1}{2}\frac{1}{\sqrt{n}}=0$$

$$\boxed{\sqrt{n}=O(n)}$$

$$2.\ \lim_{n\to\infty}\frac{n}{\sqrt{n}}=\lim_{n\to\infty}\frac{n}{n^{1/2}}=\lim_{n\to\infty}\frac{1}{\frac{1}{2}n^{-1/2}}=\lim_{n\to\infty}2\sqrt{n}=\infty$$

$$\boxed{n=\Omega(\sqrt{n})}$$

$$3.\ \lim_{n\to\infty}\frac{n}{2n}=\lim_{n\to\infty}\frac{1}{2}=\frac{1}{2}$$

$$\boxed{n=\Theta(2n)}$$

$$4.\ \lim_{n\to\infty}\frac{2n}{n}=\lim_{n\to\infty}2=2$$

$$\boxed{2n=\Theta(n)}$$

Common Growth Rates

Function	Name
c	Constant
logn	Logarithmic
$\log^2 n$	Log-squared
n	Linear
nlogn	
n^2	Quadratic
n^3	Cubic
2^n	Exponential
n!	Factorial

n	c=10	$\log_2 n$	$\log^2{}_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
10	10	3.3	10.89	10	3.3x10	10^2	10^3	2^{10}
10^2	10	6.6	43.56	10^2	$6.6\text{x}10^2$	10^4	10^6	2^{100}
10^3	10	9.9	98.01	10^3	$9.9\text{x}10^3$	10^6	10^9	2^{1000}
10^4	10	13.2	174.24	10^4	$1.32\text{x}10^5$	10^8	10^{12}	2^{10000}
10^5	10	16.5	272.25	10^5	$1.65\text{x}10^6$	10^{10}	10^{15}	2^{100000}
10^6	10	19.8	392.04	10^6	$1.98\text{x}10^7$	10^{12}	10^{18}	$2^{1000000}$

Analysis Algorithm

- Simple Statement Sequence

  - First note that a sequence of statements which is executed once only is $O(1)$.

  - It doesn't matter how many statements are in the sequence – only that the number of statements (or the time that they take to execute) is constant for all problems.

Analysis Algorithm

- Simple Loops

  - If a problem of size n can be solved with a simple loop:

```
for(i = 0; i < n; i++) {
      s;
}
```

where s is an O(1) sequence of statements, then the time complexity is nO(1) or O(n).

Analysis Algorithm

- Simple Loops

  - If we have two nested loops:

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

then we have n repetitions of an O(n) sequence, giving a complexity of: nO(n) or $O(n^2)$.

Analysis Algorithm

- Simple Loops

  - Where the index 'jumps' by an increasing amount in each iteration, we might have a loop like:

```
h = 1;
while( h <= n ) {
     s;
     h = 2*h;
}
```

in which h takes values 1, 2, 4, … until it exceeds n. This sequence has 1 + [$\log_2$ n] values, so the complexity is O($\log_2$ n).

Analysis Algorithm

- Simple Loops

  - If the inner loop depends on an outer loop index:

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

The inner loop for(i = 0; .. gets executed i times, so the total is:

$$\sum_{1}^{n} i = \frac{n(n+1))}{2}$$

and the complexity is $O(n^2)$.

Analysis Algorithm

- Simple Loops

  - If the inner loop depends on an outer loop index:

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

We see that this is the same as the result for two nested loops above, so the variable number of iterations of the inner loop doesn't affect the
'big picture'.

Analysis Algorithm

- Simple Loops

  - However, if the number of iterations of one of the loops decreases by a constant factor with every iterations:

```
h = n;
while(h > 0)
{
     for(i = 0; i < n; i++)
          s;
     h = h/2;
}
```

Analysis Algorithm

- Simple Loops
  - Then
    - There are $\log_2 n$ iterations of the outer loop and
    - The inner loop is $O(n)$.
  - So the overall complexity is $O(n \log n)$.
  - This is substantially better than the previous case in which the number of iterations of one of the loops decreased by a constant for each iteration!

Analysis Algorithm

- Algorithms with nested loops usually have a larger complexity than algorithms with one loop, but it does not have to grow at all.

- For example, we may request printing sums of numbers in the last five cells of the subarrays starting in position 0.

Analysis Algorithm

```
for (i = 4; i < n; i++) {
  for (j = i - 3, sum = a[i - 4]; j <= i; j++)
       sum += a[j];
  cout << "sum for subarray "<<i-4<<" through "<< i
  <<" is ""<<sum<<endl;
}
```

- The outer loop is executed n - 4 times.
- For each i, the inner loop is executed only four times;
- Therefore, the complexity is (n-4) O(1), i.e. O(n).

Analysis Algorithm

- Analysis of the above examples is relatively uncomplicated because the number of times the loops executed did not depend on the ordering of the arrays.

- Computation of asymptotic complexity is more involved if the number of iterations is not always the same.

Analysis Algorithm

- This point can be illustrated with a loop used to determine the length of the longest subarray with the numbers in increasing order.

- For example, in [1 8 1 2 5 0 11 12], it is three, the length of subarray [1 2 5].

Analysis Algorithm

- The code is

```
for (i = 0, length = 1; i < n - 1; i++) {
  for (i1 = i2 = k = i;
  k < n - 1 && a[k] < a[k+1]; k++, i2++);
  if (length < i2 - i1 + 1)
       length = i2 - i1 + 1;
}
```

Analysis Algorithm

- Notice that if all numbers in the array are in decreasing order, the outer loop is executed $n - 1$ times, but in each iteration, the inner loop executes just one time. Thus, the algorithm is $O(n)$.

- The algorithm is least efficient if the numbers are in increasing order. In this case, the outer for loop is executed $n - 1$ times, and the inner loop is executed $n - 1 - i$ times for each $i \in \{0, \dots, n - 2\}$. Thus, the algorithm is $O(n^2)$.

Analysis Algorithm

- In most cases, the arrangement of data is less orderly, and measuring the efficiency in these cases is of great importance.

- However, it is far from trivial to determine the efficiency in the average cases.

Algorithm Analysis Example

- Let say we would like to compute:
  $1^3 + 2^3 + 3^3 + \ldots + n^3$

```
int sum(int n) {
   int partialSum;
   partialSum = 0;               // Line 1:  1 operation
   for(int i=1; i<=n; i++)       // Line 2:  3n+2 operations
      partialSum += i*i*i;       // Line 3:  4 operations
   return partialSum;            // Line 4:  1 operation
}
```

- Line 1 & 4: 1 unit each (assignment) — total = 2
- Line 3: Executed n times, each time 4 units (Two multiplications, 1 addition and 1 assignment) — total = 4n
- Line 2: 1 for initialization, n+1 for checking, 2n for increments — total = 3n+2
- Total cost: $2 + 4n + 3n + 2 = 7n + 4 = O(n)$

General Rules

- Consecutive statements

```
for(i=0; i<n; i++)
    arr[i] = 0;
for(i=0; i<n; i++)
    for(j=0; j<n; j++)
        arr[i] += arr[j] + i + j;
```

  - We could simply add them together, i.e. $O(n) + O(n^2) = O(n^2)$

- If-else
  - No more than the running of the test plus the larger of the running times of statements in the if block and else block

Analysis of Recursive Algorithms

- Running time of algorithm with recursion could be analyzed using recurrence equation
- Example: Recursive version of factorial

```
int factorial(int n) {
    if(n==0 || n==1)          // base case
        return 1;
    else
        return n * factorial(n-1); // recursive case
}
```

  - The checking using if statement and multiplying by n is O(1), i.e. constant time
  - Time for factorial(n) = Time for factorial(n-1) + O(1)
  - Dropping the Big-Oh for the moment, we have the recurrence equation $T(n) = T(n-1) + 1$

Analysis of Recursive Algorithms

- Repeatedly unrolling the recurrence
  - $T(n) = T(n-1) + 1$
    $= T(n-2) + 1 + 1 = T(n-2) + 2$
    $= T(n-3) + 1 + 2 = T(n-3) + 3$
  - Now, it is pretty easy to observe the pattern
    $T(n) = T(n-k) + k$
  - Let $k = n$
    $T(n) = T(0) + n$
  - $T(0)$ is the time to compute the base case factorial(0), which is just $O(1)$
    $T(n) = 1 + n = O(n)$

CHAPTER 3 END