SLMP and Binary Search and Algorithm Analysis 1

Tanvir Ahmed, PhD

Department of Computer Science,

University of Central Florida

Sorted List Matching Problem (SLMP)

- Given two sorted lists of distinct numbers, output the numbers common to both lists.
- How would you attack the problem?
  - For each number on list 1, do the following:
    - a) Search for the current number in list 2.
    - b) If the number is found, output it.
- If a list is unsorted, steps a and b might take n steps (n = number of elements in list 2)
-

SLMP

- If you don't use the information that the list is sorted, we can do a brute force solution:

```
void printMatches(int list1[], int list2[], int len1, int len2)
{
        int i,j;
        for (i=0; I < len1; i++)
        {
                for (j=0; j<len2; j++)
                {
                        if (list1[i] == list2[j])
                        {
                                printf("%d", list1[i]);
                                break;
                        }
                }
        }
}
```

- How many steps it might take in total?
- n^2

SLMP

- But we know both lists are already sorted.
- Thus we can use binary search in step a.
- Assuming both list are of same sizes, the binary search takes about log n steps. (We will see binary search in next slide)
- We have to repeat n times. So, total around *n log n.* Much better than n^2.
- ***But how binary search work?***
  - ***We will see next***
- *What is the opposite of binary search?*
  - *Sequential search*

Linear/sequential search

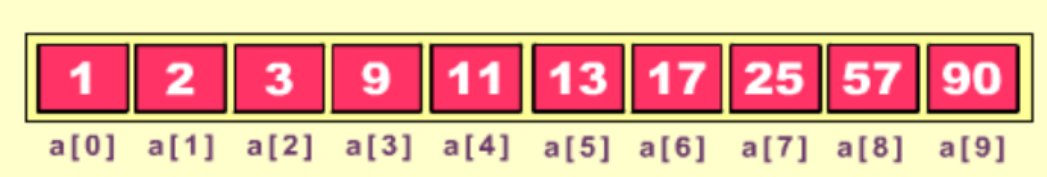
- Can you write a function that takes an array, its length, and a number and return whether the item exist in the array or not?

```
int search(int array[], int len, int value)
{
        int i;
        for (i=0; i<len; i++)
        {
                if (array[i] == value)
                        return 1;
        }
        return 0;
}
```

- Clearly, if your array is unsorted, the above solution is optimal as you don't know what part of the array has the item unless you look one after another.
- Can you do better if we have a sorted array?

Binary Search

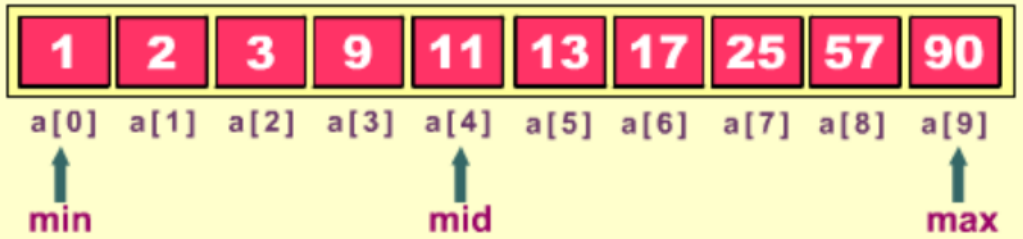
- If you know that the array is sorted, we can guess better what part of the array the data should be located
- For example see the following array:



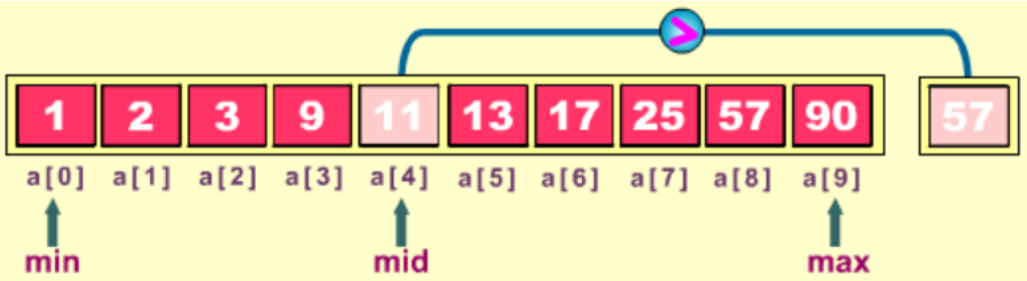
- If you want to search for 2, we can directly start our search in the lower half of the array
- That lower half of the array also can be treated as another array and we can even look lower half of that new array
- and so on….
- We divide our search space like this until we find the item or we are sure that our item does not exist
- So, what is the mid point of the above array?
  - (9+0)/2 = 9/2 = 4
- So, we need two numbers, the ***low index and high index*** and calculate:
  - mid index = (low index + high index)/2
- What would be your mid point if your low=4 and high = 9?
  - (4+9)/2 = 6

Binary Search Simulation

Suppose the data that is to be searched is 57

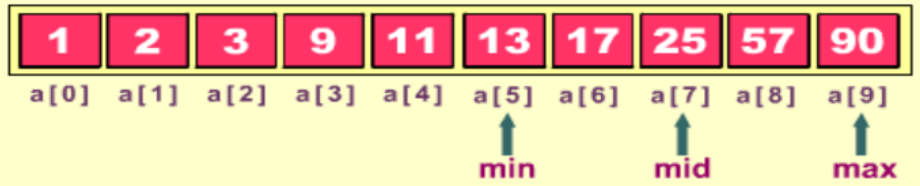


mid = (min + max) / 2
mid = (0 + 9) / 2
mid = 4



a[mid] is compared with 57

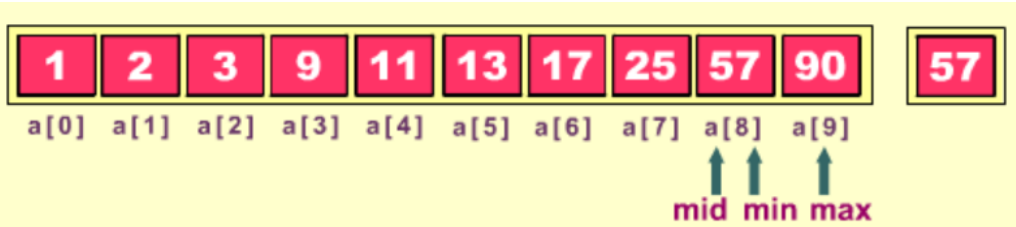
As 11 is less than 57, lower bound min is pointed to a[5]



mid = (min + max) / 2
mid = (5 + 9) / 2
mid = 7

a[mid] is compared with 57

As 25 is less than 57, lower bound min is pointed to a[8]



mid = (min + max) / 2
mid = (8 + 9) / 2
mid = 8

a[mid] is compared with 57

Search is successful as 57 is found

How about if the item search is 26?

Binary search. Search for item = 20 in the bellow array:

0	1	2	3	4	5	6
-15	18	20	25	30	35	112

- Lets work with the array index:

  l = 0 and h = 6. So, mid = (l+h)/2 = 3

- What is in Array[mid]?
  - It is 25.
  - It is > 20.
  - So, our item is actually before it.
  - So, our new h = mid-1 = 2.
  - New mid = (0+2)/2 = 1
  - Now, Array[mid] = 18 < 20
  - So, our item is actually after mid.
  - So, l = mid + 1 = 2
  - Array[mid] = 20 == item. So, return 1 (success)
- Let's try searching for 32
- How would you know that the item is not available?
  - If low>high

Binary search

- So, during the iteration, we update our low or high based on the condition
- Because we want to check the data in that part of the array and we want new mid

```
int binarySearch(int list[], int item, int len)
{
    int l = 0, h = len - 1;
    int mid;
    while (l <= h)
    {
        mid = (l + h) / 2;
        // Check if item is present at mid
        if (list[mid] == item)
            return mid;
         // If item greater, ignore left half
        if (list[mid] < item)
            l = mid + 1;

        // If item is smaller, ignore right half
        else
            h = mid - 1;
    }
    // if we reach here, then element was
    // not present
    return -1;
}
```

```
#include <Stdio.h>

int binarySearch(int list[], int item, int len)
{
    int l = 0, h = len - 1;
    int mid;
    while (l <= h)
    {
        mid = (l + h) / 2;
        // Check if item is present at mid
        if (list[mid] == item)
            return mid;
         // If item greater, ignore left half
        if (list[mid] < item)
            l = mid + 1;

        // If item is smaller, ignore right half
        else
            h = mid - 1;
    }
    // if we reach here, then element was
    // not present
    return -1;
}


int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int searchitem = 10;

    int result = binarySearch(arr, searchitem, 4);
    (result == -1) ? printf("Element is not present"
                            " in array")
                   : printf("Element is present at "
                            "index %d",
                            result);
    return 0;
}
```

Binary search recursive code

```
int binSearch(int *values, int low, int high, int searchval)
{
        int mid;
        if (low <= high)
        {
                mid = (low+high)/2;
                if (searchval < values[mid])
                        return binSearch(values, low, mid-1, searchval);
                else if (searchval > values[mid])
                        return binSearch(values, mid+1, high, searchval);
                elsereturn 1;
        }
        return 0;
}
```

Going back to SLMP

Enhance your code to find common items in two arrays (n * log n)
So, just using binary search in our last SLMP code can result in n * log n as binary search works for log n and we want to use binary search n times.

#Try at home:
Consider both of the arrays are sorted. Can you even reduce more iterations?
1) Start two “markers”, one for each list, at the beginning of both lists.
2) Repeat the following steps until one marker has reached the end of its list.
    a) Compare the two items that the markers are pointing at.
    b) If they are equal, output the number and advance BOTH markers one spot.
    If they are NOT equal, simply advance the marker pointing to the number that comes earlier one spot.

This will improve the run time and will result in 2n steps.
More hints in page 3 and 4 of this link:
http://www.cs.ucf.edu/~dmarino/ucf/transparency/cop3502/lec/IntroAlgorithms-02.pdf

Algorithm Analysis 1

Order Notation and Estimating Complexity

- We have seen and will see various algorithms
- Last example we got idea about SLMP, Linear search, and binary search.
- How to judge the efficiency or speed of the algorithm?
- We will use order notation to approximate:
  - How much time they take
  - How much memory (space) they use.
- It is nearly impossible to find exact timing.
- Also machine dependent
- Our judge should be independent of their implementation
- So, approximation is important
  - We call it Big-O approximation

Order Notation and Estimating Complexity

- We assume each statement and each comparison takes some constant amount of time
- Algorithms have some type of input:
  - For example, a sorting algorithm might have an array of size *n*.
  - Performance in most cases depends on *n*.
- We want to know how performance of an algorithm responds to changes in problem size.

Big-O Notation

- Goal: Provide a qualitative insight of number of Operations of a problem size of n elements.
- Describe the number of operations in a mathematical expression in terms of n.
- This expression is known as Big-O
- It is a way to measuring the order of magnitude of a mathematical expression.
- O(n) means “Order of n”

Big-O Notation

- Consider the expression:
  - $f(n) = 4n^2 + 3n + 10$
  - How fast is this "growing"?
    - There are three terms:
      - the $4n^2$, the 3n, and the 10
    - As n gets bigger, which term makes it get larger fastest?
      - Let's look at some values of n and see what happens?

n	$4n^2$	3n	10
1	4	3	10
10	400	30	10
100	40,000	300	10
1000	4,000,000	3,000	10
10,000	400,000,000	30,000	10
100,000	40,000,000,000	300,000	10
1,000,000	4,000,000,000,000	3,000,000	10

Big-O notation

- Who has the most influence?
- To find the order:
  - Eliminate any term whose contribution to the total cases to be less significant as n becomes large.
  - Eliminate constant factors.
- For: $4n^2+3n + 10$,
- We will:
  - Ignore 3n + 10 because that accounts for a small number of steps as n gets large
  - Eliminate the constant factor of 4 in front of the n^2 term.

- In doing so, we will conclude that the algorithm takes $O(n^2)$ steps.
- We can say, $O(4n^2+3n + 10) = O(n^2)$
- We can say that f(n) takes $O(n^2)$ steps to execute.

Big-O notation

- Some basic examples:
  - What is the Big-O of the following functions:
    - $f(n) = 4n^2 + 3n + 10$
      - Answer: $O(n^2)$
    - $f(n) = 76{,}756{,}234n^2 + 427{,}913n + 7$
      - Answer: $O(n^2)$
    - $f(n) = 74n^8 - 62n^5 - 71562n^3 + 3n^2 - 5$
      - Answer: $O(n^8)$
    - $f(n) = 42n^4*(12n^6 - 73n^2 + 11)$
      - Answer: $O(n^{10})$
    - $f(n) = 75n*\log n - 415$
      - Answer: $O(n*\log n)$

Big-O notation

- Consider the expression: $f(n) = 4n^2 + 3n + 10$
  - How fast is this "growing"?
    - We can say, $O(4n^2 + 3n + 10) = O(n^2)$
    - Till now, we have one function:
      - $f(n) = 4n^2 + 3n + 10$
    - Let us make a second function, **g(n)**
      - It's just a letter right? We could have called it r(n) or x(n)
        - Don't get scared about this
    - Now, let g(n) equal n^2
      - **$g(n) = n^2$**
    - So now we have two functions: **f(n)** and **g(n)**
      - We said (above) that $O(4n^2 + 3n + 10) = O(n^2)$
    - Similarly, we can say that the **order of f(n) is O[g(n)].**

Big-O Notation

- Definition:
  - ***f(n) is O[g(n)] if there exists positive integers c and N, such that f(n) <= c*g(n) for all n>=N.***
    - Think about the two functions we just had:
      - **$f(n) = 4n^2 + 3n + 10$**, and **$g(n) = n^2$**
      - We agreed that $O(4n^2 + 3n + 10) = O(n^2)$
      - Which means we agreed that the order of **f(n) is O(g(n)**
    - That's all this definition says!!!
    - f(n) is big-O of g(n), if there is a c,
      - (c is a constant)
    - such that f(n) is not larger than c*g(n) for sufficiently large values of n (greater than N)

Big-O

- Definition:
  - ***f(n) is O[g(n)] if there exists positive integers c and N, such that f(n) <= c*g(n) for all n>=N.***
    - Think about the two functions we just had:
      - **f(n) = $4n^2 + 3n + 10$**, and **g(n) = n^2**
    - f is big-O of g, if there is a c such that f is not larger than c*g for sufficiently large values of n (greater than N)
      - So given the two functions above, **does there exist** some **constant**, **c**, that would make the following statement true?
      - f(n) <= c*g(n)
      - **$4n^2 + 3n + 10 <= c*n^2$**
      - If there does exist this c, **then f(n) is O(g(n))**
    - Let's go see if we can come up with the constant, c

Big-O

- Definition:
  - ***f(n) is O[g(n)] if there exists positive integers c and N, such that f(n) <= c*g(n) for all n>=N.***
    - PROBLEM: Given our two functions,
      - **f(n) = $4n^2 + 3n + 10$**, and **g(n) = n^2**
    - **Find the c** such that **$4n^2 + 3n + 10 <= c*n^2$**
    - Clearly, c cannot be 4 or less
      - Cause even if it was 4, we would have:
        - **$4n^2 + 3n + 10 <= 4n^2$**
        - This is NEVER true for any positive value of n!
      - So **c must be greater than 4**
    - Let us try with c being equal to 5
      - **$4n^2 + 3n + 10 <= 5n^2$**

- Definition:
  - ***f(n) is O[g(n)] <u>if there exists</u> positive integers c and N, such that <u>f(n) <= c*g(n)</u> for all n>=N.***
    - PROBLEM: Given our two functions,
      - **$f(n) = 4n^2 + 3n + 10$**, and **$g(n) = n^2$**
    - **<u>Find the c</u>** such that **$4n^2 + 3n + 10 <= c*n^2$**
      - **$4n^2 + 3n + 10 <= 5n^2$**
      - For what values of n, if ANY at all, is this true?

n	$4n^2 + 3n + 10$	$5n^2$
1	4(1) + 3(1) + 10 = 17	5(1) = 5

- Definition:
  - ***f(n) is O[g(n)] <u>if there exists</u> positive integers c and N, such that <u>f(n) <= c*g(n)</u> for all n>=N.***
    - PROBLEM: Given our two functions,
      - **f(n) = $4n^2 + 3n + 10$**, and **g(n) = n^2**
    - **<u>Find the c</u>** such that **$4n^2 + 3n + 10 <= c*n^2$**
      - **$4n^2 + 3n + 10 <= 5n^2$**
      - For what values of n, if ANY at all, is this true?

n	$4n^2 + 3n + 10$	$5n^2$
1	4(1) + 3(1) + 10 = **17**	5(1) = **5**
2	4(4) + 3(2) + 10 = **32**	5(4) = **20**
3	4(9) + 3(3) + 10 = **55**	5(9) = **45**
4	4(16) + 3(4) + 10 = **86**	5(16) = **80**

      - For n = 1 through 4, this statement is NOT true

But now let's try larger values of n.

- Definition:
  - ***f(n) is O[g(n)] <u>if there exists</u> positive integers c and N, such that <u>f(n) <= c*g(n)</u> for all n>=N.***
    - PROBLEM: Given our two functions,
      - **f(n) = $4n^2 + 3n + 10$**, and **g(n) = n^2**
    - **<u>Find the c</u>** such that **$4n^2 + 3n + 10 <= c*n^2$**
      - **$4n^2 + 3n + 10 <= 5n^2$**
      - For what values of n, if ANY at all, is this true?

n	$4n^2 + 3n + 10$	$5n^2$
1	4(1) + 3(1) + 10 = **17**	5(1) = **5**
2	4(4) + 3(2) + 10 = **32**	5(4) = **20**
3	4(9) + 3(3) + 10 = **55**	5(9) = **45**
4	4(16) + 3(4) + 10 = **86**	5(16) = **80**
5	4(25) + 3(5) + 10 = **125**	5(25) = **125**

- Definition:
  - ***f(n) is O[g(n)] <u>if there exists</u> positive integers c and N, such that <u>f(n) <= c*g(n)</u> for all n>=N.***
    - PROBLEM: Given our two functions,
      - **$f(n) = 4n^2 + 3n + 10$**, and **$g(n) = n^2$**
    - **<u>Find the c</u>** such that **$4n^2 + 3n + 10 <= c*n^2$**
      - **$4n^2 + 3n + 10 <= 5n^2$**
      - For what values of n, if ANY at all, is this true?

n	$4n^2 + 3n + 10$	$5n^2$
1	4(1) + 3(1) + 10 = **17**	5(1) = **5**
2	4(4) + 3(2) + 10 = **32**	5(4) = **20**
3	4(9) + 3(3) + 10 = **55**	5(9) = **45**
4	4(16) + 3(4) + 10 = **86**	5(16) = **80**
5	4(25) + 3(5) + 10 = **125**	5(25) = **125**
6	4(36) + 3(6) + 10 = **172**	5(36) = **180**

- Definition:
  - ***f(n) is O[g(n)] if there exists positive integers c and N, such that f(n) <= c*g(n) for all n>=N.***
    - PROBLEM: Given our two functions,
      - **f(n) = $4n^2 + 3n + 10$**, and **g(n) = n^2**
    - **Find the c** such that **$4n^2 + 3n + 10 <= c*n^2$**
      - **$4n^2 + 3n + 10 <= 5n^2$**
      - For what values of n, if ANY at all, is this true?
      - So when n = 5, the statement finally becomes **true**
      - And **when n > 5, it remains true**!
    - So our constant, 5, works for all n >= 5.

- Definition:
  - ***f(n) is O[g(n)] <u>if there exists</u> positive integers c and N, such that <u>f(n) <= c*g(n)</u> for all n>=N.***
    - PROBLEM: Given our two functions,
      - **f(n) = $4n^2 + 3n + 10$**, and **g(n) = n^2**
    - **<u>Find the c</u>** such that **$4n^2 + 3n + 10 <= c*n^2$**
    - So our constant, 5, works for all n >= 5.
    - Therefore, **<u>f(n) is O(g(n))</u>** per our definition!
    - Why?
    - Because <u>there exists positive integers, c and N</u>,
      - Just so happens in this case that c = 5 and N = 5
    - **<u>such that f(n) <= c*g(n)</u>**.

Who actually got that ?

- Definition:
  - ***f(n) is O[g(n)] if there exists positive integers c and N, such that f(n) <= c*g(n) for all n>=N.***
    - What can we take from this?

- What we can gather is that:
- **c*g(n)** is an **upper bound** on the value of **f(n)**.
  - It represents the **worst possible scenario** of running time.
- The number of operations is, at worst, proportional to g(n) for all large values of n.

- Summing up the basic properties for determining the order of a function:
  1) If you've got multiple functions added together, the fastest growing one determines the order
  2) Multiplicative constants don't affect the order
  3) If you've got multiple functions multiplied together, the overall order is their individual orders multiplied together

- Some basic examples:
  - What is the Big-O of the following functions:
    - $f(n) = 4n^2 + 3n + 10$
      - Answer: $O(n^2)$
    - $f(n) = 76{,}756{,}234n^2 + 427{,}913n + 7$
      - Answer: $O(n^2)$
    - $f(n) = 74n^8 - 62n^5 - 71562n^3 + 3n^2 - 5$
      - Answer: $O(n^8)$
    - $f(n) = 42n^4*(12n^6 - 73n^2 + 11)$
      - Answer: $O(n^{10})$
    - $f(n) = 75n*\log n - 415$
      - Answer: $O(n*\log n)$

- Quick Example of Analyzing Code:
  - This is just to show you how we use Big-O
    - we'll do more of these (a lot more) next time
  - Use big-O notation to analyze the time complexity of the following fragment of C code:

```
for (k=1; k<=n/2; k++) {
        sum = sum + 5;
}

for (j = 1; j <= n*n; j++) {
        delta = delta + 1;
}
```

- Quick Example of Analyzing Code:
  - So look at what's going on in the code:
    - We care about the total number of REPETITIVE operations.
      - Remember, we said we care about the running time for LARGE values of n
      - So in a **for loop**, with n as part of the comparison value determining when to stop `for (k=1; k<=n/2; k++)`
      - Whatever is INSIDE that loop will be executed a LOT of times
      - So we examine the code within this loop and see how many operations we find
        - When we say operations, we're referring to mathematical operations such as +, -, *, /, etc.

- Quick Example of Analyzing Code:
  - So look at what's going on in the code:
    - The number of operations executed by these loops is the sum of the individual loop operations.
    - We have 2 loops,

```
for (k=1; k<=n/2; k++) {
        sum = sum + 5;
}

for (j = 1; j <= n*n; j++) {
        delta = delta + 1;
}
```

- Quick Example of Analyzing Code:
  - So look at what's going on in the code:
    - The number of operations executed by these loops is the sum of the individual loop operations.
    - We have 2 loops,
      - The first loop runs n/2 times
      - Each iteration of the first loop results in one operation
        - The + operation in: `sum = sum + 5;`
      - So there are n/2 operations in the first loop
      - The second loop runs n^2 times
      - Each iteration of the second loop results in one operation
        - The + operation in: `delta = delta + 1;`
      - So there are n^2 operations in the second loop.

- Quick Example of Analyzing Code:
  - So look at what's going on in the code:
    - The number of operations executed by these loops is the sum of the individual loop operations.
    - The first loop has n/2 operations
    - The second loop has n^2 operations
    - They are NOT nested loops.
      - One loop executes AFTER the other completely finishes
    - So <u>we simply ADD their operations</u>
    - The total number of operations would be $n/2 + n^2$
    - In Big-O terms, we can express the number of operations as $O(n^2)$

- Common orders (listed from slowest to fastest growth)

Function	Name
1	Constant
log n	Logarithmic
n	Linear
n log n	Poly-log
n^2	Quadratic
n^3	Cubic
2^n	Exponential
n!	Factorial

- **O(1)** or “Order One”: **Constant time**
  - does not mean that it takes only one operation
  - does mean that the work doesn’t change as n changes
  - is a notation for “constant work”
  - An example would be finding the smallest element in a sorted array
    - There’s nothing to search for here
    - The smallest element is always at the beginning of a sorted array
    - So this would take O(1) time

- **O(n)** or "Order n": **Linear time**
  - does not mean that it takes n operations
    - maybe it takes 3*n operations, or perhaps 7*n operations
  - does mean that the **work changes in a way that is proportional to n**
  - Example:
    - If the input size doubles, the running time also doubles
  - is a notation for "**work grows at a linear rate**"
  - You usually can't really do a lot better than this for most problems we deal with
    - After all, you need to at least examine all the data right?

- **$O(n^2)$** or "Order n^2 ": **Quadratic time**
  - If input size doubles, running time increases by a factor of 4
- **$O(n^3)$** or "Order n^3 ": **Cubic time**
  - If input size doubles, running time increases by a factor of 8
- **$O(n^k)$**: Other polynomial time
  - Should really try to avoid high order polynomial running times
    - However, it is considered good from a theoretical standpoint

- **O(2^n)** or "Order 2^n ": **Exponential time**
  - more theoretical rather than practical interest because they cannot reasonably run on typical computers for even for moderate values of n.
  - Input sizes bigger than 40 or 50 become unmanageable
    - Even on faster computers
- **O(n!)**: even worse than exponential!
  - Input sizes bigger than 10 will take a long time

- **O(n logn)**:
  - Only slightly worse than O(n) time
    - And O(n logn) will be much less than $O(n^2)$
    - This is the running time for the better sorting algorithms we will go over (later)
- **O(log n)** or "Order log n": **Logarithmic time**
  - If input size doubles, running time increases ONLY by a constant amount
  - **any algorithm that halves the data** remaining to be processed on each iteration of a loop will be an **O(log n) algorithm**.

- Practical Problems that can be solved utilizing order notation:
  - Example:
    - You are told that algorithm A runs in O(n) time
    - You are also told the following:
      - For an input size of 10
      - The algorithm runs in 2 milliseconds
    - As a result, you can expect that for an input size of 500, the algorithm would run in 100 milliseconds!
      - Notice the input size jumped by a multiple of 50
        - From 10 to 500
      - Therefore, given a O(n) algorithm, the running time should also jump by a multiple of 50, **which it does**!

- Practical Problems that can be solved utilizing order notation:
  - General process of solving these problems:
    - We know that Big-O is NOT exact
      - It's an upper bound on the actual running time
    - So when we say that an **algorithm runs in O(f(n)) time**,
    - **Assume** the EXACT **running time is c*f(n)**
      - where c is some constant
    - Using this assumption,
      - we can use the information in the problem to solve for c
      - Then we can use this c to answer the question being asked
    - Examples will clarify…

- Practical Problems that can be solved utilizing order notation:
  - Example 1: Algorithm A runs in **$O(n^2)$** time
    - For an input size of 4, the running time is 10 milliseconds
    - How long will it take to run on an input size of 16?
    - **Let $T(n) = c*n^2$**
      - T(n) refers to the running time (of algorithm A) on input size **n**
      - Now, plug in the given data, and **find the value for c!**
    - $T(4) = c*4^2 = 10$ milliseconds
      - Therefore, **c = 10/16 milliseconds**
    - Now, answer the question by **using c** and **solving T(16)**
    - **$T(16) = c*16^2$** $= (10/16)*16^2 = 160$ milliseconds

- Practical Problems that can be solved utilizing order notation:
  - Example 2: Algorithm A runs in **O($\log_2 n$)** time
    - For an input size of 16, the running time is 28 milliseconds
    - How long will it take to run on an input size of 64?
    - **Let T(n) = c*$\log_2 n$**
      - Now, plug in the given data, and **find the value for c!**
    - T(16) = c*$\log_2 16$ = 28 milliseconds
      - c*4 = 28 milliseconds
      - Therefore, **c = 7 milliseconds**
    - Now, answer the question by **using c** and **solving T(64)**
    - **T(64) = c*$\log_2 64$** = 7*$\log_2 64$ = 7*6 = 42 milliseconds

Acknowledgement and reference

- Many text and slides are taken from Prof. Jonathan's Slides: https://www.cs.ucf.edu/courses/cop3502/spr2012/notes/COP3502_14_AlgorithmAnalysis1.pdf
- And Prof Arup's notes: http://www.cs.ucf.edu/~dmarino/ucf/transparency/cop3502/lec/AlgorithmAnalysis-1.doc