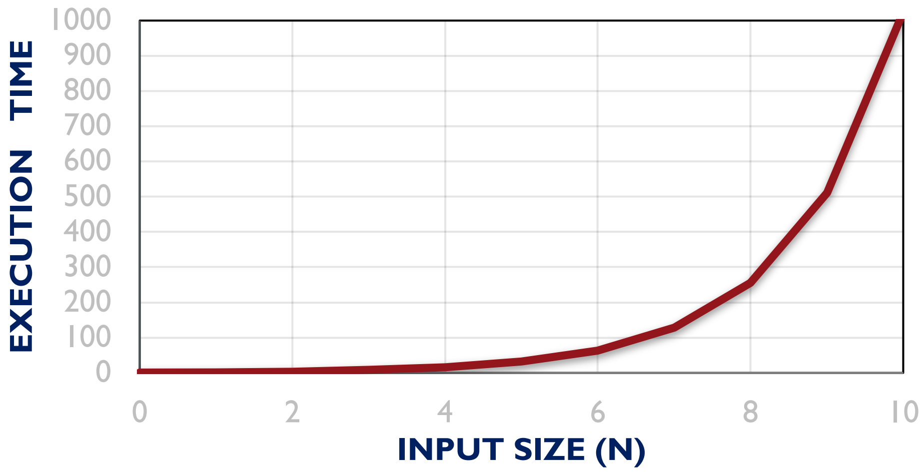
ASYMPTOTIC ANALYSIS

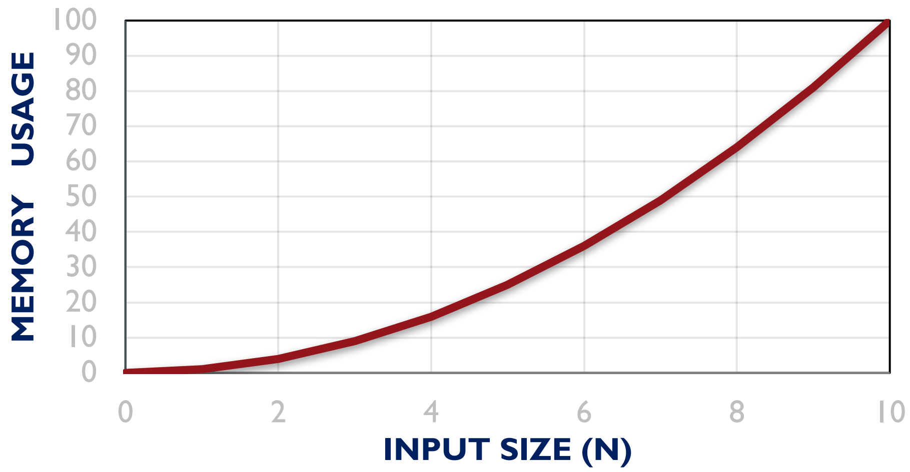
INTRODUCTION TO ASYMPTOTIC ANALYSIS

Goals

To gain a sense of how algorithms are evaluated which will be further developed in a future Algorithms course.

To gain a sense of time complexity and space complexity.





TIME COMPLEXITY

Concept quantify the execution time of an algorithm by evaluating the order of growth as the input size is modified

rate of change in input size vs. rate of change in execution time

Examples if the size of the input set N is doubled and the execution time remains the same, the order of growth is constant

if the size of the input set N is doubled and the execution time is also doubled, the order of growth is linear

TIME COMPLEXITY

Algorithms are evaluated for worst-case, average-case or best-case performance

Example linear search of an unsorted array of size n

```
for(int i=0; i<size; ++i) {
    if(a[i] == value) cout << i;      // element found
}
cout << "Not found";                  // element not found
```

Worst case the value is last or not in the array at all (n iterations)

Average case the value is around the middle of the array (n+1/2 iterations)

Best case the value is the first element in the array (1 iterations)

TIME COMPLEXITY

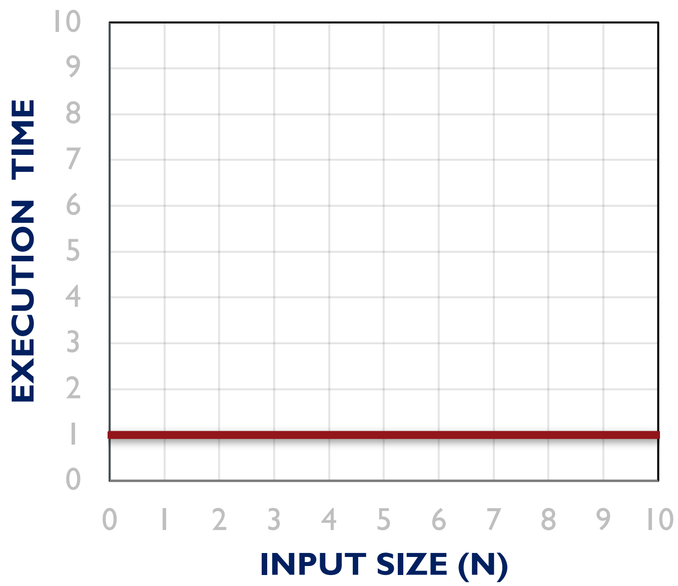
Best case	known as Big-Θ (big-theta) analysis
Average case	known as Big-Ω (big-omega) analysis
Worst case	known as Big-O (big-Oh) analysis

this introduction will focus on Big-O analysis
which approximates worst-case time complexity

CONSTANT TIME O(1)

Print an element in a one-dimensional array:

```
void printValue(int a[], int n) {
      cout << a[n] << “\n”;
}
```



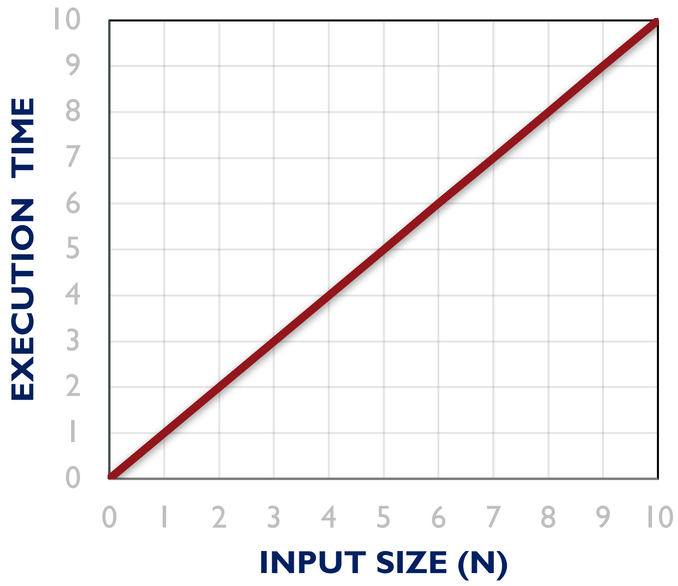
as the index of the string increases the execution time of the function does not change

the order of growth in Big-O notation is O(1) or constant time

LINEAR TIME O(n)

A function to print a one-dimensional array:

```
void print (int a[], int n) {
    for(int i=0; i<n; ++i) {
        cout << a[i] << “\n”;
    }
}
```



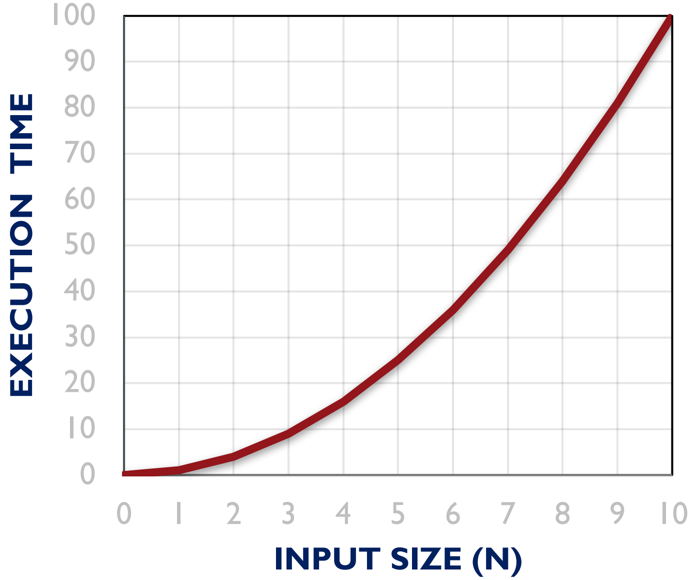
as the size of the array increases,
the execution time increases at the same rate

the order of growth in Big-O notation is O(n) or linear time

QUADRATIC TIME O(n^2)

A function to print a two-dimensional array of equal proportions:

```
void print (int a[][], int n) {
     for(int i=0; i<n; ++i) {
          for(int j=0; j<n; ++j) {
               cout << a[i][j]<< " \n";
          }
     }
}
```



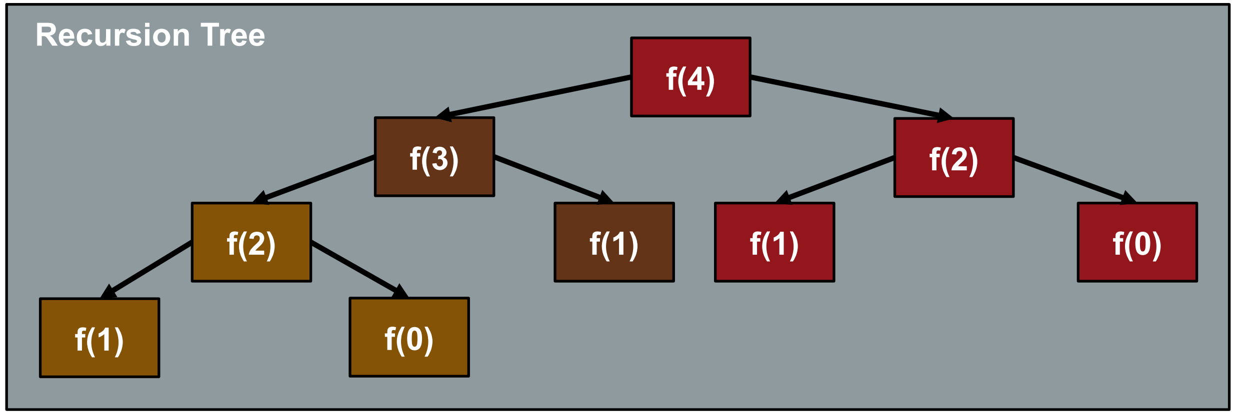
as the size of a two-dimensional array of equal proportion increases, the execution time increases by n^2

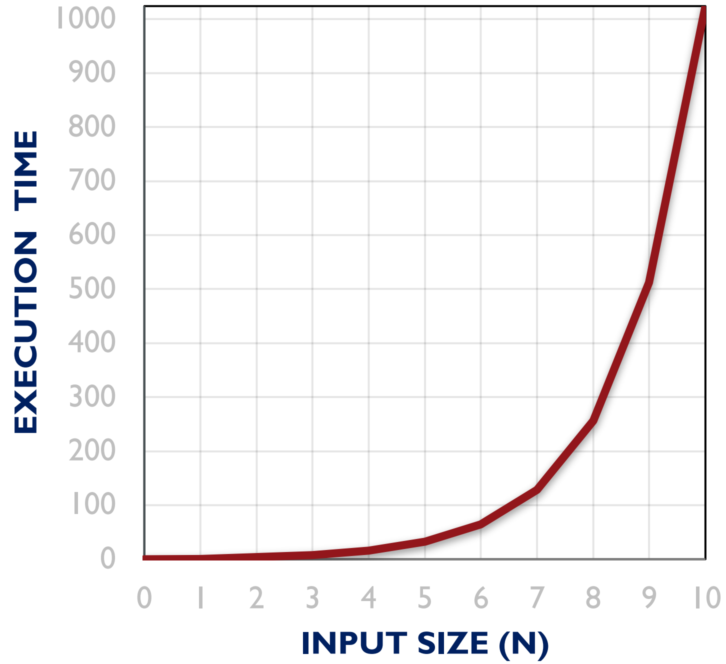
the order of growth in Big-O notation is O(n^2) or quadratic time

EXPONENTIAL TIME $O(2^n)$

A recursive function to calculate the nth value of the fibonacci sequence:
0 1 1 2 3 5 8 13 21 34 55

```
long fibonacci(int n) {
    if(n<2) { return n; }
    return fibonacci(n-1) + fibonacci(n-2);
}
```



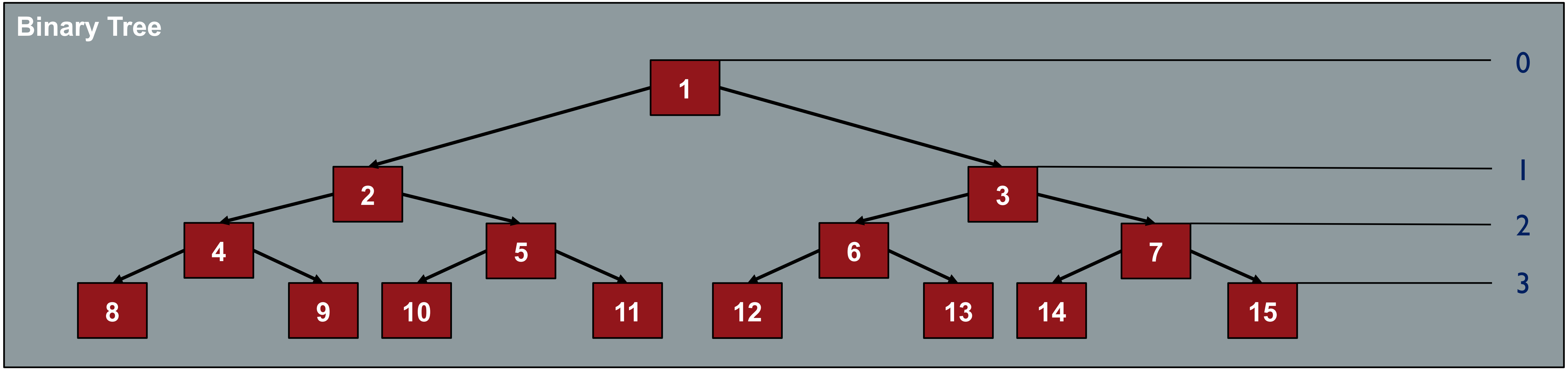


as the value of n increases, execution time increases by 2^n
the order of growth in Big-O notation is $O(2^n)$ or exponential time

EXPONENTIAL TIME $O(2^n)$

Binary Tree a structure in which each node has at most two children

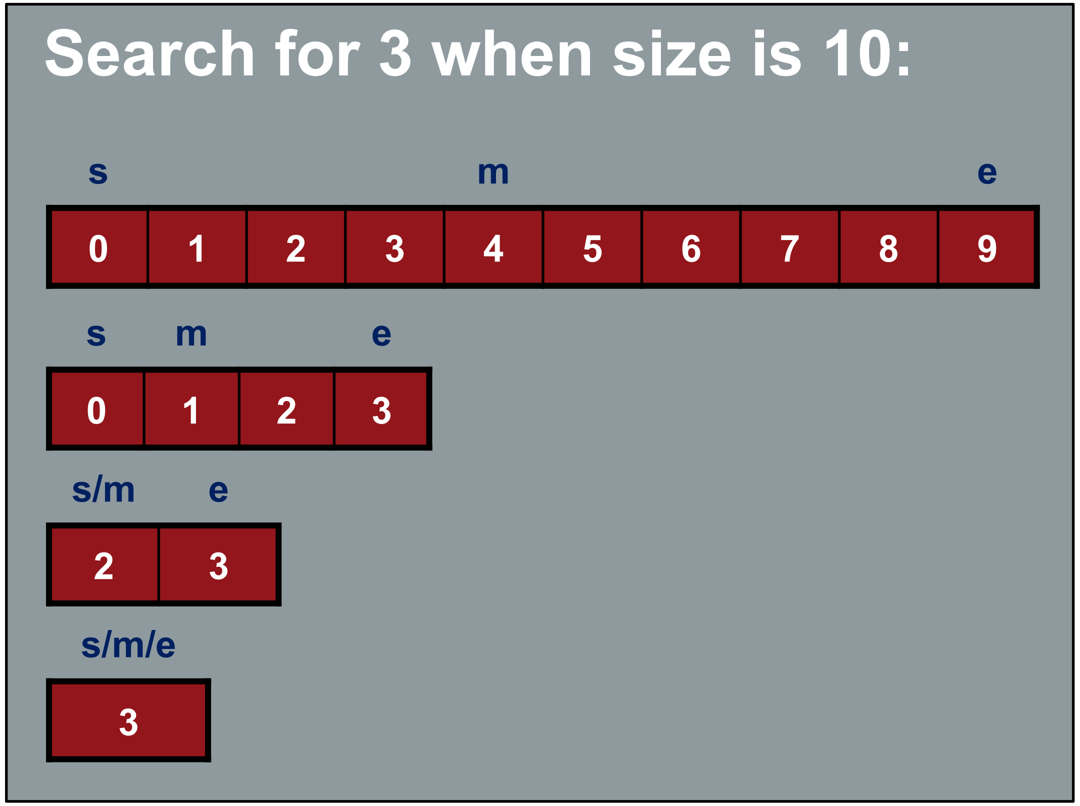
a binary tree has at most $2^{(h+1)}-1$ nodes in a tree of h levels



LOGARITHMIC TIME O(log n)

A function to search for a value in a sorted array of size n:

```
int binarySearch(int a[], int size, int value) {
    int start = 0, end = size - 1;
    while(start <= end) {
        int middle = (start + end) / 2;
        if(a[middle] == value) {
            return middle;
        } else if(a[middle] < value) {
            start = middle + 1;
        } else { end = middle - 1; }
    }
    return -1;
}
```



LOGARITHMIC TIME O(log n)

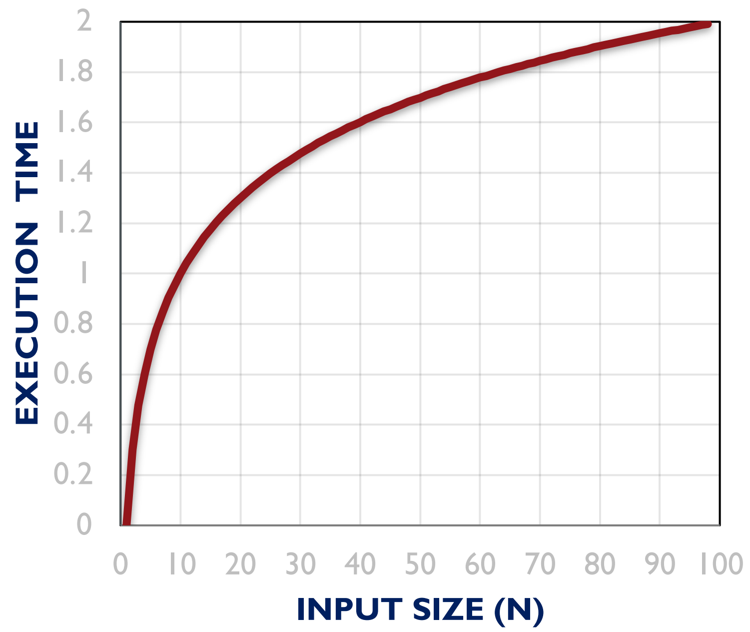
Worst-case search where the value is not in an array of size n:

n=10 (4 iterations)

0 4 9
5 7 9
8 8 9
9 9 9

n=1000 (10 iterations)

0 499 999
500 749 999
750 874 999
875 937 999
938 968 999
969 984 999
985 992 999
993 996 999
997 998 999
999 999 999



as the value of n increases the execution time decreases by log n
the order of growth in Big-O notation is O(log n) or logarithmic time