CSE 3010 – Data Structures & Algorithms

Lecture #5

What will be covered today

- Understanding input dataset
- Common growth rates using Big O
- General rules to calculate running time

Understanding input dataset

- Assumption `n` is large
- Sorting a list
  - Input size – `n`, number of items in the list
  - Running time is measured on `n`
- Create `k` groups from a `n` dataset
  - Input size – `n` and `k`
  - If `k` is small, running time may be measured only on `n`
  - If `k` is large, running time may be measured including `n` and `k`

Common growth rates using Big O

`T(n) = O(f(n))`

Asymptotic behavior of a function *`f(n)`* - Growth of *`f(n)`* as *`n`* gets large

`T(n) = O(1)`	Function runs in constant time relative to its input

```
printf("Middle element = "%d", array[n/2]);
```

`T(n) = O(n)`	Function runs in linear time and in direct proportion to the size of the input

```
for (int i = 0; i < size; i++)
        printf("%d\n", array[i]);
```

Common growth rates using Big O

$T(n) = O(n^2)$	Function runs in quadratic time, directly proportion to the square of the size of input

```
for (int i = 0; i < size; i++)
    for (int j = 0; j < size; j++)
        printf("%d, %d\n", array[i], array[j]);
```

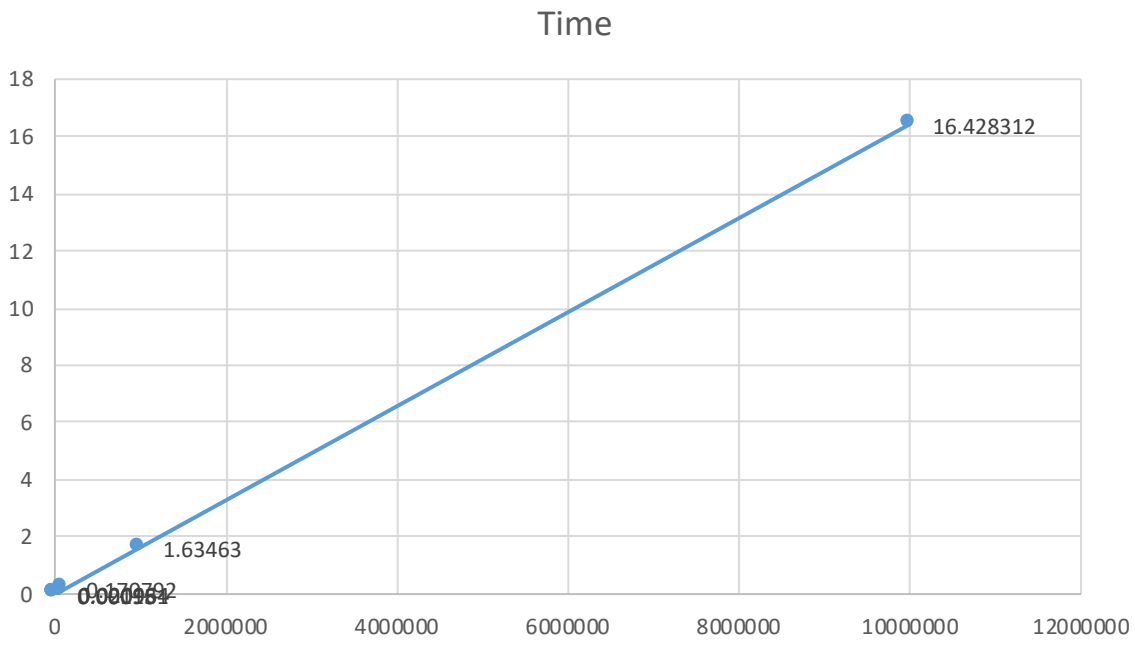
$T(n) = O(2^n)$	Function runs in exponential time, growth doubles with each addition to the input dataset

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

Growth rates – Understanding through examples

`T(n) = O(1)`	Function runs in constant time relative to its input
`T(n) = O(n)`	Function runs in linear time and in direct proportion to the size of the input

Printing elements in a list of n element	
n	**Time**
100	0.00018
1000	0.001951
10000	0.020964
100000	0.170792
1000000	1.63463
10000000	16.428312



Common growth rates using Big O

$T(n) = O(n^2)$

Function runs in quadratic time, directly proportion to the square of the size of input

Sorting a list of n elements using Bubble Sort			
n	**Run 1 - Time**	**Run 2 - Time**	**Run 3 - Time**
100	0.000069	0.000174	0.000087
500	0.000763	0.000892	0.000848
2500	0.018427	0.013226	0.01778
12500	0.364612	0.372255	0.351064
62500	8.726408	8.613476	8.497274

$T(n) = O(2^n)$

Function runs in exponential time, growth doubles with each addition to the input dataset

Recursive function for Fibonacci numbers		
n	**Run 1 - Time**	**Run 2 - Time**
3	0.000039	0.000033
48	38.461742	38.849747

General rules to calculate running time

Rule 1: `for` loops	Running time of a `for` loop is **at most** the running time of the statements inside the loop (including tests) times the number of iterations

```
for (i = 0; i < 10; i++)
    printf("%d\n", i+1);
```

Rule 2: Nested `for` loops	Running time of a statements inside nested loops is the running time of the statements multiplied by the product of the sizes of all the `for` loops

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

General rules to calculate running time

Rule 3: Consecutive statements	Add the running time for each block and take the maximum as the running time

```
for (i = 0; i < N; i++)
    a[i] = 0;

for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        a[i] = a[i] + a[j] + i + j;
```

Rule 4: `if … then … else`	Never more than the running time of the test plus the larger of the running times of *S1* and *S2*

```
if (number mod 2 == 0)
    printf("Number is even!");
else
    printf("Number is odd!");
```