Big O Notation

Mr. Poole

Pigeons vs the Internet

Big O notation

Internet Transfer	**Pigeon Transfer**
1 GB -----> 30 min	1 GB -----> 60 min
2 GB -----> 60 min	2 GB -----> 60 min
3 GB -----> 90 min	3 GB -----> 60 min
1000 GB -----> 500 hours	1000 GB -----> 60 min

Linear < Constant

Big O notation - O(?)

```
main (String argos[]){
    print(a);
    print(b);
    print(c);
}
```

O(?)

```
while(x > 0){
    print(x);
    x++;
}
```

O(?)

Big O notation - O(?)

```
main (String argos[]){
    print(a);
    print(b);
    print(c);
}
```

O(1) - This happens constant times

```
while(n > 0){
    print(n);
    n++;
}
```

O(n) - This happens n times.

Big O - Rule 1: Different Steps get added

```
function test (){
    printArray(array[a]);
    printArray(array[b]);
}
```

Array of size “a” and “b” would be two different run times.

So we add them together to get **O(a+b)**

Big O - Rule 2: Drop Constants

```
function test (){
    for (array [x])
        findMinimum(array)
    for (array [x])
        findMaximum(array)
}
```

O(?)

```
function test (){
    for (array [x])
        findMinimum(array)
        findMaximum(array)
}
```

O(?)

Big O - Rule 2: Drop Constants

```
function test (){
    for (array [n])
        findMinimum(array)
    for (array [n])
        findMaximum(array)
}
```

O(2n)

```
function test (){
    for (array [n])
        findMinimum(array)
        findMaximum(array)
}
```

O(n)

We get to remove the 2 because in the process of running say 200 million times, there isn't much of a time difference between 200 and 400

Big O - Rule 3: Different inputs get different variables

```
function test (){
    for (array [a])
        print(array)
    for (array [b])
        print(array)
}
```

O(?)

Big O - Rule 3: Different inputs get different variables

```
function test (){
    for (array [a])
        print(array)
    for (array [b])
        print(array)
}
```

O(a+b)

Since a and b are different variables, one may be vastly larger than the next. This means we must represent them differently.

Big O - Rule 4: Drop non-dominant terms

```
function test (){
    for (array [n])
        print(array)
    for (array [n])
        for (array [n])
            print(array)
}
```

O(?)

O(?)

O(?)

Big O - Rule 4: Drop non-dominant terms

```
function test (){
    for (array [n])
        print(array)
    for (array [n])
        for (array [n])
            print(array)
}
```

O(n)

$O(n^2)$

Overall - $O(n + n^2)$

But overall we can write the equation

$O(n^2) <= O(n^2 + n) <= O(n^2 + n^2)$

$O(n^2 + n^2)$ also equals $O(\underline{2}n^2)$ but from rule 2 we drop constants

So overall we drop n since it isn't the dominant term. This equals $O(n^2)$

Sorting - Bubble Sort

Sorting - Bubble Sort

Example:

First Pass:

(**5 1** 4 2 8) -> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 **5 4** 2 8) -> (1 **4 5** 2 8), Swap since 5 > 4

(1 4 **5 2** 8) -> (1 4 **2 5** 8), Swap since 5 > 2

(1 4 2 **5 8**) -> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) -> (**1 4** 2 5 8)

(1 **4 2** 5 8) -> (1 **2 4** 5 8), Swap since 4 > 2

(1 2 **4 5** 8) -> (1 2 **4 5** 8)

(1 2 4 **5 8**) -> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) -> (**1 2** 4 5 8)

(1 **2 4** 5 8) -> (1 **2 4** 5 8)

(1 2 **4 5** 8) -> (1 2 **4 5** 8)

(1 2 4 **5 8**) -> (1 2 4 **5 8**)

Sorting - Bubble Sort

Runtime = O(?)

```
// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)

    // Last i elements are already in place
    for (j = 0; j < n-i-1; j++)
        if (arr[j] > arr[j+1])
            swap(&arr[j], &arr[j+1]);
}
```

Sorting - Bubble Sort

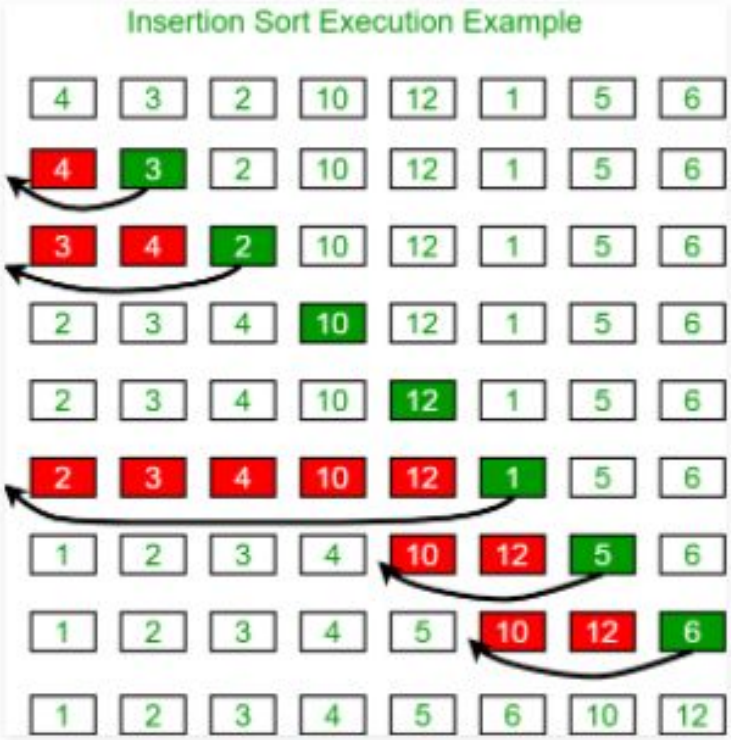
```
// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)

    // Last i elements are already in place
    for (j = 0; j < n-i-1; j++)
        if (arr[j] > arr[j+1])
            swap(&arr[j], &arr[j+1]);
}
```

Runtime = $O(n^2)$

Because of 2 nested loops

Sorting - Insertion Sort

Sorting - Insertion Sort



Sorting - Insertion Sort

```
/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Runtime = O(?)

Sorting - Insertion Sort

```
/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Runtime = $O(n^2)$

Because of 2 nested loops

Sorting - Selection Sort

Sorting - Selection Sort

```
arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

```
void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}

/* Function to print an array */
```

Sorting - Selection Sort

```
void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}
```

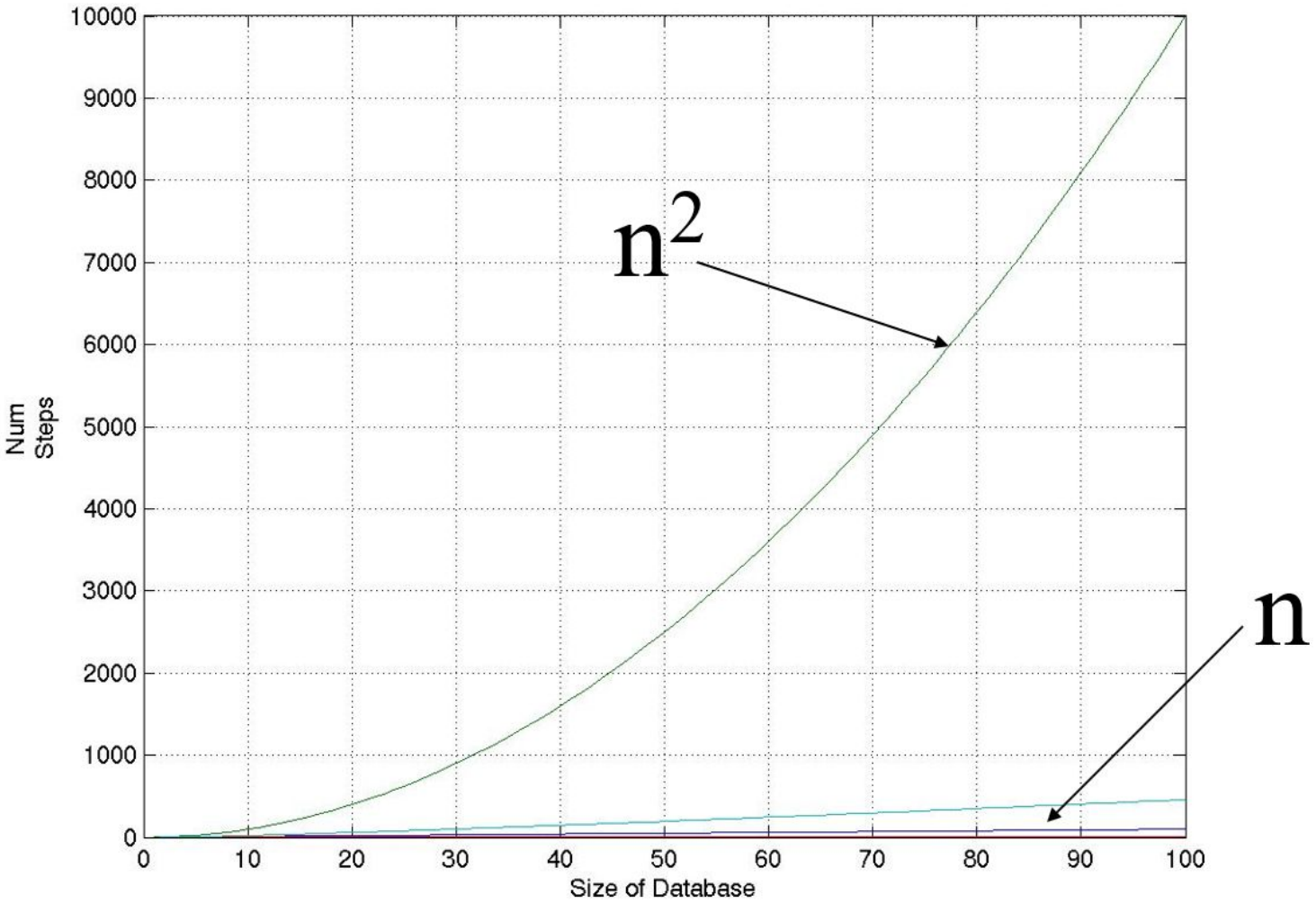
Runtime = $O(n^2)$

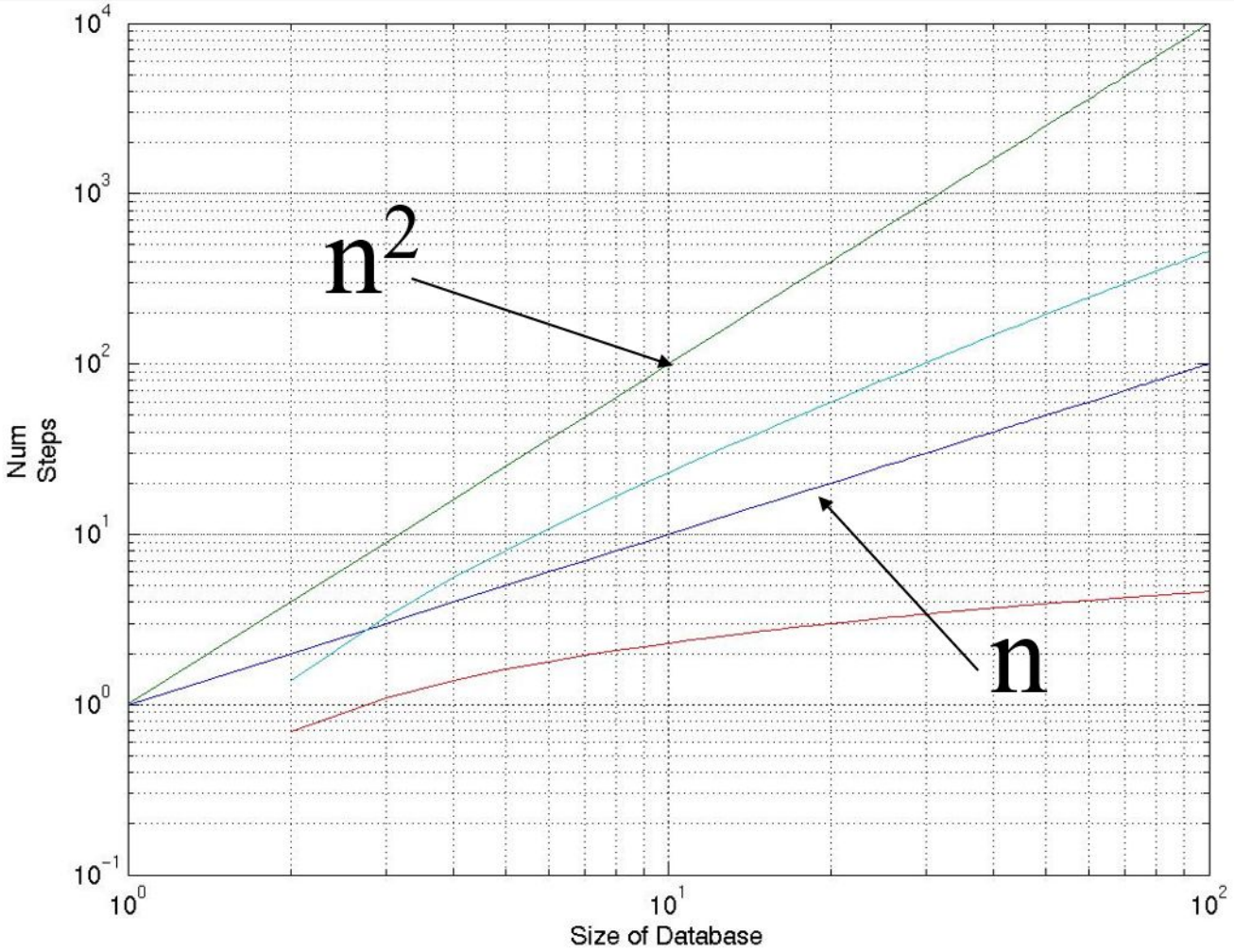
Because of 2 nested loops

What’s the worst runtime?
Bubble, Insertion, Selection?

Big O of Sorting Algorithms

Algorithm	Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	O(1)
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)


10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
Num Steps
0
10
20
30
40
50
60
70
80
90
100
Size of Database
n2
n


10^4
10^3
10^2
10^1
10^0
10^{-1}
Num Steps
n^2
n
10^0
10^1
10^2
Size of Database

Sorting Video