Section 3

CSCI E-22

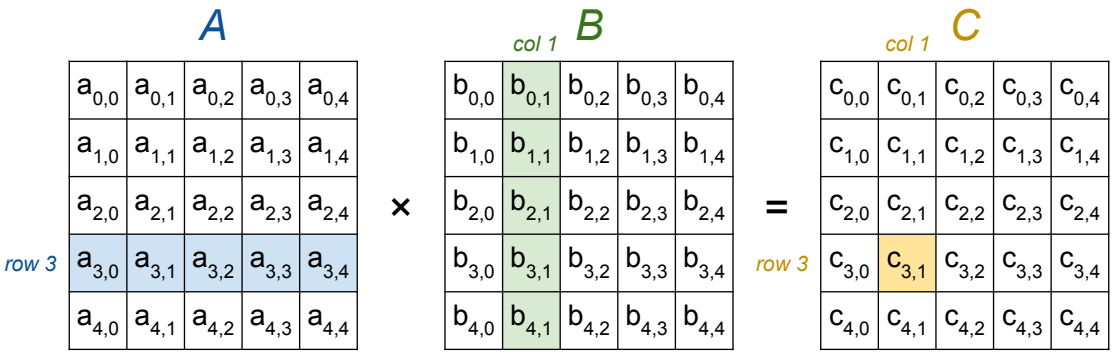
Will Begin Shortly

Big O notation

- Constant multipliers do not “affect” big O
  - $600n^2$ and $0.00005n^2$ are both $O(n^2)$
- Lower order terms do not “affect” big O
  - $2^n + n^{100}$ is still $O(2^n)$
- General ordering of functions, from slowest-growing to fastest:
  - $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^k) < O(c^n) < O(n!)$
    $c > 1$, $k > 2$ constant

Matrix multiplication

- For two $n \times n$ matrices A and B, the product C is another $n \times n$ matrix where each element is obtained by multiplying elements of A row-wise with B column-wise, adding up the individual products:



$$c_{3,1} = a_{3,0} \times b_{0,1} + a_{3,1} \times b_{1,1} + a_{3,2} \times b_{2,1} + a_{3,3} \times b_{3,1} + a_{3,4} \times b_{4,1}$$

Matrix multiplication

- For two $n \times n$ matrices A and B, the product C is another $n \times n$ matrix where each element is obtained by multiplying elements of A row-wise with B column-wise, adding up the individual products:

$c_{0,0}$	$c_{0,1}$	$c_{0,2}$	...	$c_{0,n-1}$
$c_{1,0}$	$c_{1,1}$	$c_{1,2}$	...	$c_{1,n-1}$
$c_{2,0}$	$c_{2,1}$	$c_{2,2}$	...	$c_{2,n-1}$
⋮	⋮	⋮	$c_{i,j}$	⋮
$c_{n-1,0}$	$c_{n-1,1}$	$c_{n-1,2}$	...	$c_{n-1,n-1}$

$$c_{i,j} = a_{i,0} \times b_{0,j} + a_{i,1} \times b_{1,j} + a_{i,2} \times b_{2,j} + \ldots + a_{i,n-1} \times b_{n-1,j}$$

$$c_{i,j} = \sum_{k=0}^{n-1} a_{i,k} \times b_{k,j}$$

row col

Matrix multiplication

- In Java, if we represent a matrix as a two-dimensional array, we can determine the entries of the matrix *AB* using nested for loops in the following way:

```
int[][] matrixA = new int[n][n];     // A is an n-by-n matrix
int[][] matrixB = new int[n][n];     // B is another n-by-n matrix
int[][] matrixC = new int[n][n];     // C will hold the matrix product A * B

for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        int total = 0;

        for (int k = 0; k < n; k++) {
            total += matrixA[i][k] * matrixB[k][j];
        }

        matrixC[i][j] = total;
    }
}
```

General case of nested loops

- Nested loops, both dependent on **`n`**: statements inside the inner loop will run exactly **`n`** × **`n`** times:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        // statements here are executed n^2 times
    }
}
```

General case of nested loops

- Nested loops, both dependent on **n**: statements inside the inner loop will run exactly **n** × **n** times:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        // statements here are executed n^2 times
    }
}
```

- Three nested loops, all dependent on **n**: statements inside the innermost loop will run exactly **n** × **n** × **n** times:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            // statements here are executed n^3 times
        }
    }
}
```

General case of nested loops

- What if the outer loop is dependent on **n** and the inner loop runs a constant number of times (here, only 3 times)?

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < 3; j++) {
        // statements here are executed ??? times
    }
}
```

General case of nested loops

- What if the outer loop is dependent on `n` and the inner loop is dependent on the current value of `i`?

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        // statements here are executed ??? times
    }
}
```

Sorting algorithms

Selection sort
Insertion sort
Bubble sort
Shell sort

```
private static int smallest(
  int[] arr, int lower, int upper
) {
    int indexMin = lower;
    for (int i = lower + 1; i <= upper; i++) {
        if (arr[i] < arr[indexMin]) {
            indexMin = i;
        }
    }

    return indexMin;
}

public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int j = smallest(arr, i, arr.length - 1);
        swap(arr, i, j);
    }
}
```

0	1	2	3	4	5
7	39	20	11	16	5

```
private static int smallest(
  int[] arr, int lower, int upper
) {
    int indexMin = lower;
    for (int i = lower + 1; i <= upper; i++) {
        if (arr[i] < arr[indexMin]) {
            indexMin = i;
        }
    }

    return indexMin;
}

public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int j = smallest(arr, i, arr.length - 1);
        swap(arr, i, j);
    }
}
```

	comparisons	moves (swap is 3 moves)
best case		
worst case		
average case		

```
public static void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] < arr[i - 1]) {
            int toInsert = arr[i];
            int j = i;

            do {
                arr[j] = arr[j - 1];
                j = j - 1;
            } while (
                j > 0 &&
                toInsert < arr[j - 1]
            );

            arr[j] = toInsert;
        }
    }
}
```

i toInsert j

0	1	2	3	4	5
7	39	20	11	16	5


```
public static void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] < arr[i - 1]) {
            int toInsert = arr[i];
            int j = i;

            do {
                arr[j] = arr[j - 1];
                j = j - 1;
            } while (
                j > 0 &&
                toInsert < arr[j - 1]
            );

            arr[j] = toInsert;
        }
    }
}
```


comparisons moves (swap is 3 moves)

best case

worst case

average case

```
public static void bubbleSort(int[] arr) {
    for (int i = arr.length - 1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (arr[j] > arr[j + 1])
                swap(arr, j, j + 1);
        }
    }
}
```

0	1	2	3	4	5
7	39	20	11	16	5

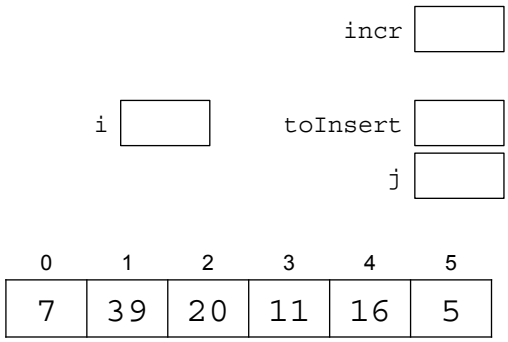
```
public static void bubbleSort(int[] arr) {
    for (int i = arr.length - 1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (arr[j] > arr[j + 1])
                swap(arr, j, j + 1);
        }
    }
}
```

	comparisons	moves (swap is 3 moves)
best case		
worst case		
average case		

```
public static void shellSort(int[] arr) {
    int incr = 1;
    while (2 * incr <= arr.length)
        incr = 2 * incr;
    incr--;

    while (incr >= 1) {
        for (int i = incr; i < arr.length; i++) {
            if (arr[i] < arr[i - incr]) {
                int toInsert = arr[i];
                int j = i;

                while (
                    j > incr - 1 &&
                    toInsert < arr[j - incr]
                ) {
                    arr[j] = arr[j - incr];
                    j = j - incr;
                }
                arr[j] = toInsert;
            }
        }
        incr = incr / 2;
    }
}
```




```
public static void shellSort(int[] arr) {
    int incr = 1;
    while (2 * incr <= arr.length)
        incr = 2 * incr;
    incr--;

    while (incr >= 1) {
        for (int i = incr; i < arr.length; i++) {
            if (arr[i] < arr[i - incr]) {
                int toInsert = arr[i];
                int j = i;

                while (
                    j > incr - 1 &&
                    toInsert < arr[j - incr]
                ) {
                    arr[j] = arr[j - incr];
                    j = j - incr;
                }
                arr[j] = toInsert;
            }
        }
        incr = incr / 2;
    }
}
```

comparisons and moves

best case

worst case

average case