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:

$$
\begin{array}{c} \\ \\ \\ \\ \text{row 3} \\ \\ \end{array}
\overset{A}{\begin{bmatrix}
a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} \\
a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\
a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\
a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\
a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4}
\end{bmatrix}}
\times
\begin{array}{c} \\ \text{col 1} \ \ B \\
\begin{bmatrix}
b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} & b_{0,4} \\
b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\
b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\
b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \\
b_{4,0} & b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4}
\end{bmatrix}\end{array}
=
\begin{array}{c} \\ \\ \\ \\ \text{row 3} \\ \\ \end{array}
\begin{array}{c} \text{col 1} \ \ C \\
\begin{bmatrix}
c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} & c_{0,4} \\
c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} & c_{1,4} \\
c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} & c_{2,4} \\
c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} & c_{3,4} \\
c_{4,0} & c_{4,1} & c_{4,2} & c_{4,3} & c_{4,4}
\end{bmatrix}\end{array}
$$

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

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:

$$
\begin{bmatrix}
c_{0,0} & c_{0,1} & c_{0,2} & \cdots & c_{0,n-1} \\
c_{1,0} & c_{1,1} & c_{1,2} & \cdots & c_{1,n-1} \\
c_{2,0} & c_{2,1} & c_{2,2} & \cdots & c_{2,n-1} \\
\vdots & \vdots & \vdots & c_{i,j} & \vdots \\
c_{n-1,0} & c_{n-1,1} & c_{n-1,2} & \cdots & c_{n-1,n-1}
\end{bmatrix}
$$

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

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

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;
    }
}
```

outer `i`-loop gets values 0, 1, …, n - 1

for each `i`, inner `j`-loop gets values 0, 1, …, n - 1

this code runs for every $c_{i,j}$ matrix element

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;
    }
}
```

how many multiplications are done for each element $c_{i,j}$?

***n** multiplications*

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;
    }
}
```

how many multiplications are done for each element $c_{i,j}$*?*
***n** multiplications*

overall, how many multiplications are required to compute the final result?
n×n elmnts × n multiplications =
n × n × n = $\boldsymbol{n^3}$ ***multiplications***

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 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 3n times
    }
}
```

`n` iterations of outer loop × 3 iterations of inner loop = 3`n` **= still O(*n*)**

General case of nested loops

- What if the outer loop is dependent on `n` and the inner loop is <u>dependent on **the current value of `i`**</u>?

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

when `i` = 0, statements are executed 0 times
when `i` = 1, statements are executed 1 time
when `i` = 2, statements are executed 2 times
when `i` = 3, statements are executed 3 times
when `i` = 4, statements are executed 4 times
...

how many <u>total</u> executions for a given value of n?
1 + 2 + 3 + 4 + ... + n - 1 = ***n(n - 1) / 2***

i	j	exec?	
0	0	✗	
1	0	✓	*1 time*
1	1	✗	+
2	0	✓	
2	1	✓	*2 times*
2	2	✗	+
3	0	✓	
3	1	✓	*3 times*
3	2	✓	
3	3	✗	+
4	0	✓	
4	1	✓	
4	2	✓	*4 times*
4	3	✓	
4	4	✗	
5	0	✓	
...	...	...	

n^2 versus $n(n - 1) / 2$

Both loops dependent on *n*

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        // ...
    }
}
```

when `i` *= 0, statements are executed n times*
when `i` *= 1, statements are executed n times*
when `i` *= 2, statements are executed n times*
when `i` *= 3, statements are executed n times*
...
when `i` *= n - 1, statements are excd n times*

$n \times n = n^2$

$O(n^2)$

Outer loop dependent on *n*, inner loop dependent on current value of `i`

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        // ...
    }
}
```

when `i` *= 0, statements are executed 0 times*
when `i` *= 1, statements are executed 1 times*
when `i` *= 2, statements are executed 2 times*
when `i` *= 3, statements are executed 3 times*
...
when `i` *= n - 1, statements are excd n - 1 times*

$n(n - 1) / 2$

$n^2 / 2 - n / 2$

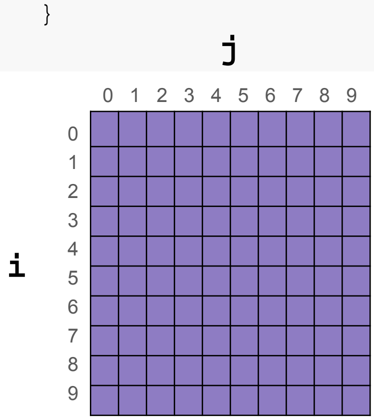
$\frac{1}{2}n^2 - \frac{1}{2}n$

also $O(n^2)$

n^2 versus $n(n - 1) / 2$

Both loops dependent on *n*

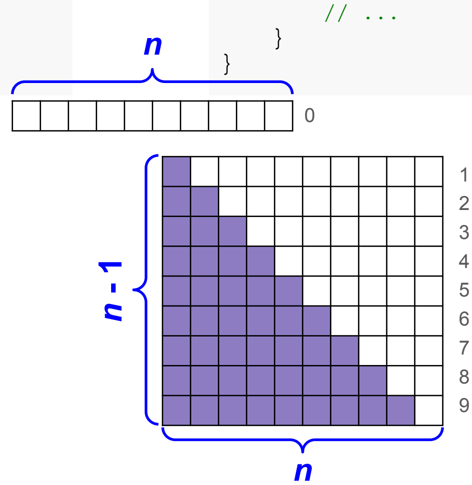
```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        // ...
    }
}
```

$n \times n = n^2$

Outer loop dependent on *n*, inner loop dependent on current value of `i`

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < i; j++) {
        // ...
    }
}
```

width *height*

$$\frac{n \times (n - 1)}{2}$$

Sorting algorithms

Selection sort
Insertion sort
Bubble sort
Shell sort

```java
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

```java
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
5	39	20	11	16	**7**
5	**7**	20	11	16	**39**
5	7	**11**	**20**	16	39
5	7	11	**16**	**20**	39
5	7	11	16	**20**	39

```java
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	$n(n - 1) / 2$ $O(n^2)$	$3(n - 1)$ $O(n)$
worst case	$O(n^2)$	$O(n)$
average case	$O(n^2)$	$O(n)$

```java
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: 1 toInsert: j:

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

```java
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
7	**20**	**39**	11	16	5
7	**11**	**20**	**39**	16	5
7	11	**16**	**20**	**39**	5
5	**7**	**11**	**16**	**20**	**39**

```java
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	$n - 1$ $O(n)$	0
worst case	$O(n^2)$	$O(n^2)$
average case	$O(n^2)$	$O(n^2)$

```java
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

i

j

```java
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);
        }
    }
}
```

i

j

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

```java
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	$n(n - 1) / 2$ $O(n^2)$ *(faster implementations are possible)*	0
worst case	$O(n^2)$	$O(n^2)$
average case	$O(n^2)$	$O(n^2)$

```
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;
    }
}
```

incr

i

toInsert

j

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

```
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;
    }
}
```

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

```java
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	$O(n \log n)$
worst case	$O(n^{1.5})$
average case	$O(n^{1.5})$