CSCI 132:
Basic Data Structures and Algorithms

More Big-O

Reese Pearsall
Spring 2025

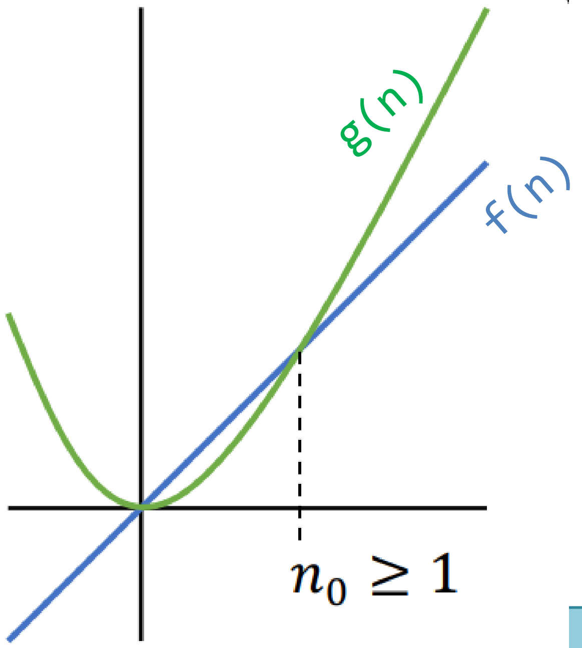
Big O Formal Definition

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers

$f(n)$ is $\mathbf{O}(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n), \text{ for all } n \geq n_0$$

Past a certain spot, g(n) dominates f(n) within a multiplicative constant



$$\forall n \geq 1, n^2 \geq n$$
$$\Rightarrow n \in O(n^2)$$

O -notation provides an upper bound on some function $f(n)$

Big O

Goal: describe the number of operations an algorithm executes in regard to some input n when the input n grows under the worst-case scenario

How many iterations are done?

Are they sequential or nested operations?

Growth Rates

1

n

n^2

x^n

We want an

upper bound

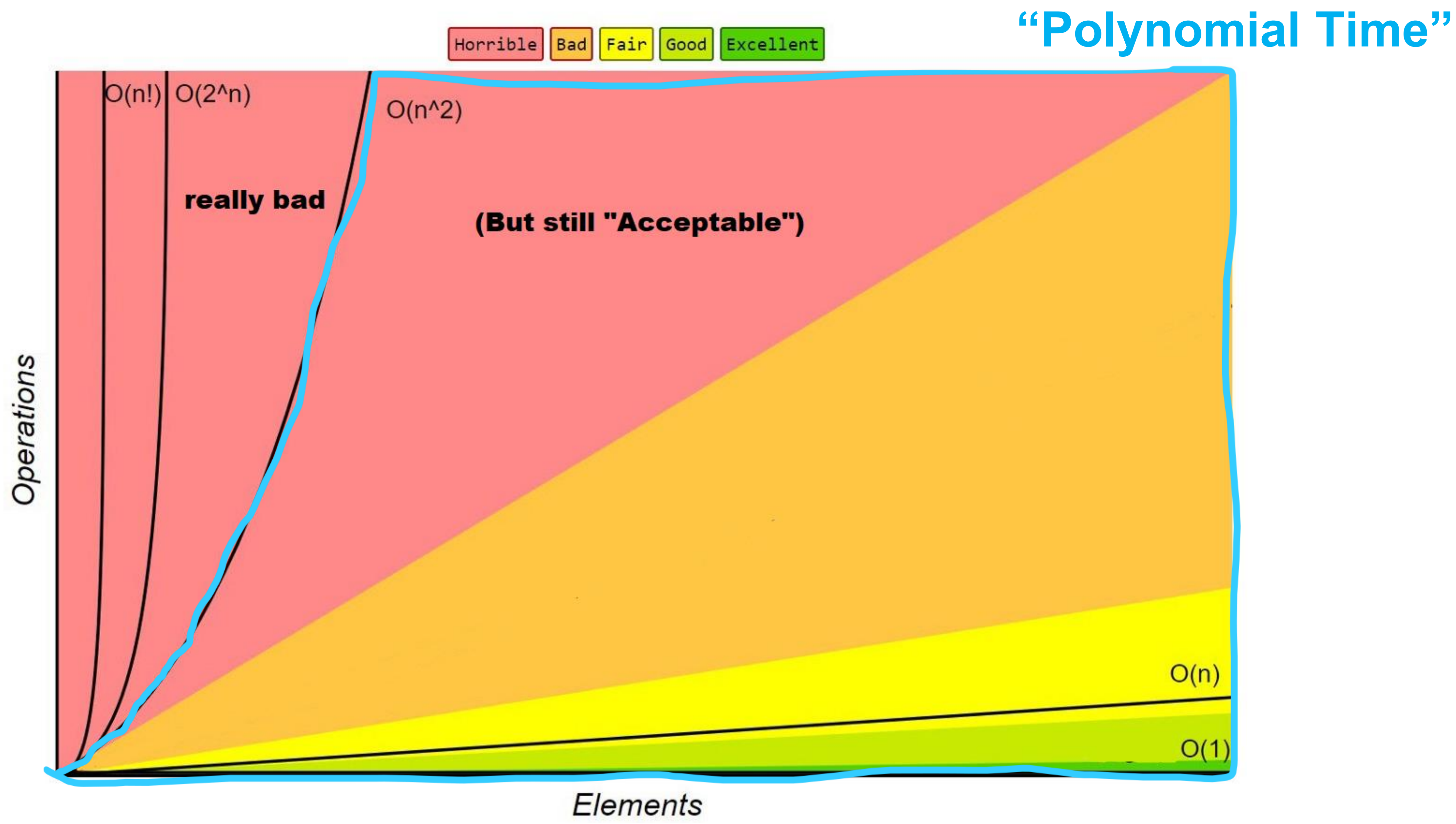
"The algorithm cant do worse than ___ amount of operations"

We can drop multiplicative constants.

We can drop non-dominant factors

- Not 100% precise
- Real-world limitations

Big-O Complexity Chart



Algorithm Analysis: Matrix Multiplication

$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}}$$

Example:

$$\begin{bmatrix} -2 & 1 \\ 0 & 4 \end{bmatrix} \times \begin{bmatrix} 6 & 5 \\ -7 & 1 \end{bmatrix} = \begin{bmatrix} -2\times6+1\times-7 & -2\times5+1\times1 \\ 0\times6+4\times-7 & 0\times5+4\times1 \end{bmatrix}$$

$$= \begin{bmatrix} -19 & -9 \\ -28 & 4 \end{bmatrix}$$

Algorithm Analysis: n*n Matrix Multiplication

```
void matrixMultiply(int[][] A, int[][] B, int[][] C, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = 0;
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Algorithm Analysis: n*n Matrix Multiplication

```
void matrixMultiply(int[][] A, int[][] B, int[][] C, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i][j] = 0;
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Running time?

Algorithm Analysis: n*n Matrix Multiplication

```
void matrixMultiply(int[][] A, int[][] B, int[][] C, int n) {
    for (int i = 0; i < n; i++) {   O(n)
        for (int j = 0; j < n; j++) {
            C[i][j] = 0;
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Running time?

Algorithm Analysis: n*n Matrix Multiplication

```
void matrixMultiply(int[][] A, int[][] B, int[][] C, int n) {
    for (int i = 0; i < n; i++) {         O(n)
        for (int j = 0; j < n; j++) {     O(n)
            C[i][j] = 0;
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Running time?

Algorithm Analysis: n*n Matrix Multiplication

```
void matrixMultiply(int[][] A, int[][] B, int[][] C, int n) {
    for (int i = 0; i < n; i++) {        O(n)
        for (int j = 0; j < n; j++) {    O(n)
            C[i][j] = 0;                 O(1)
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Running time?

Algorithm Analysis: n*n Matrix Multiplication

```
void matrixMultiply(int[][] A, int[][] B, int[][] C, int n) {
    for (int i = 0; i < n; i++) {        O(n)
        for (int j = 0; j < n; j++) {        O(n)
            C[i][j] = 0;        O(1)
            for (int k = 0; k < n; k++) {        O(n)
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Running time?

Algorithm Analysis: n*n Matrix Multiplication

```
void matrixMultiply(int[][] A, int[][] B, int[][] C, int n) {
    for (int i = 0; i < n; i++) {        O(n)
        for (int j = 0; j < n; j++) {        O(n)
            C[i][j] = 0;        O(1)
            for (int k = 0; k < n; k++) {        O(n)
                C[i][j] += A[i][k] * B[k][j];        O(1)
            }
        }
    }
}
```

Running time?

Algorithm Analysis: n*n Matrix Multiplication

```
void matrixMultiply(int[][] A, int[][] B, int[][] C, int n) {
    for (int i = 0; i < n; i++) {          O(n)
        for (int j = 0; j < n; j++) {      O(n)
            C[i][j] = 0;                   O(1)
            for (int k = 0; k < n; k++) {  O(n)
                C[i][j] += A[i][k] * B[k][j];   O(1)
            }
        }
    }
}
```

These O(1)'s won't impact the running time. We need to focus on **loops**

Running time?

Algorithm Analysis: n*n Matrix Multiplication

```
void matrixMultiply(int[][] A, int[][] B, int[][] C, int n) {
    for (int i = 0; i < n; i++) {          O(n)
        for (int j = 0; j < n; j++) {      O(n)
            C[i][j] = 0;                   O(1)
            for (int k = 0; k < n; k++) {  O(n)
                C[i][j] += A[i][k] * B[k][j];   O(1)
            }
        }
    }
}
```

These O(1)'s won't impact the running time. We need to focus on **loops**

Nested for loops = multiply

Running time? $O(n) * O(n) * O(n) \rightarrow \mathbf{O(n^3)}$ where n is size of matrices

Cubic running time!

Algorithm Analysis: n*n Matrix Multiplication

```
void matrixMultiply(int[][] A, int[][] B, int[][] C, int n) {
    for (int i = 0; i < n; i++) {          O(n)
        for (int j = 0; j < n; j++) {      O(n)
            C[i][j] = 0;                   O(1)
            for (int k = 0; k < n; k++) {  O(n)
                C[i][j] += A[i][k] * B[k][j];   O(1)
            }
        }
    }
}
```

These O(1)'s won't impact the running time. We need to focus on **loops**

Nested for loops = multiply

Running time? $O(n) * O(n) * O(n) \rightarrow \mathbf{O(n^3)}$ where n is size of matrices

Cubic running time!

Most efficient Matrix Multiplication (currently known): $O(n^{2.37})$

https://en.wikipedia.org/wiki/Computational_complexity_of_matrix_multiplication#:~:text=The%20optimal%20number%20of%20field,is%20O(n2.371339).

Algorithm Analysis: Intersection size between two sets

```
public static int calculate_matches(int[] array1, int[] array2) {
    int matches= 0;
    for(int i = 0; i < array1.length; i++) {
        for(int j = 0; j < array2.length; j++) {
            if(array1[i] == array2[j]) {
                System.out.println("Match Found:" + array1[i] + " " + array2[j]);
                matches++;
            }
    }
    return matches;
}
```

```
array 1: [1, 7, 5, 3, 2]
array 2: [0, 1, 6, 7, 2]
```

$| A \cap B |$

Value returned: 3

Running time?

Algorithm Analysis: Intersection size between two sets

```
public static int calculate_matches(int[] array1, int[] array2) {
    int matches= 0;                                   O(1)
    for(int i = 0; i < array1.length; i++) {          O(n)
        for(int j = 0; j < array2.length; j++) {      O(n)
            if(array1[i] == array2[j]) {
                System.out.println("Match Found:" + array1[i] + " " + array2[j]);
                matches++;
            }
        }
    }
    return matches;
}
```

```
array 1: [1, 7, 5, 3, 2]
array 2: [0, 1, 6, 7, 2]
```

$| A \cap B |$

Value returned: 3

Running time?

Algorithm Analysis: Intersection size between two sets

```
public static int calculate_matches(int[] array1, int[] array2) {
    int matches= 0;  O(1)
    for(int i = 0; i < array1.length; i++) {  O(n)
        for(int j = 0; j < array2.length; j++) {  O(n)
            if(array1[i] == array2[j]) {  O(1)
                System.out.println("Match Found:" + array1[i] + " " + array2[j]);  O(1)
                matches++;  O(1)
            }
        }
    }
    return matches;  O(1)
}
```

```
array 1: [1, 7, 5, 3, 2]
array 2: [0, 1, 6, 7, 2]
```

$| A \cap B |$

Value returned: 3

Running time?

Algorithm Analysis: Intersection size between two sets

```
public static int calculate_matches(int[] array1, int[] array2) {
    int matches= 0;                                  O(1)
    for(int i = 0; i < array1.length; i++) {         O(n)
        for(int j = 0; j < array2.length; j++) {     O(n)
            if(array1[i] == array2[j]) {             O(1)
                System.out.println("Match Found:" + array1[i] + " " + array2[j]);   O(1)
                matches++;                           O(1)
            }
    }
    return matches;                                  O(1)
}
```

```
array 1: [1, 7, 5, 3, 2]
array 2: [0, 1, 6, 7, 2]
```

$$| A \cap B |$$

Value returned: 3

Running time? **$O(n^2)$ where n = # of elements in the array**

Algorithm Analysis: Intersection size between two sets

```
public static int calculate_matches(int[] array1, int[] array2) {
    int matches= 0;                                   O(1)
    for(int i = 0; i < array1.length; i++) {          O(n)
        for(int j = 0; j < array2.length; j++) {      O(n)
            if(array1[i] == array2[j]) {              O(1)
                System.out.println("Match Found:" + array1[i] + " " + array2[j]);   O(1)
                matches++;                            O(1)
            }
        }
    }
    return matches;                                   O(1)
}
```

```
array 1: [1, 7, 5, 3, 2]
array 2: [0, 1, 6, 7, 2]
```

$| A \cap B |$

Value returned: 3

Running time?

$O(n^2)$ where n = # of elements in the array

Are array1 and array2 always the same size?

Algorithm Analysis: Intersection size between two sets

```
public static int calculate_matches(int[] array1, int[] array2) {
    int matches= 0;                                          O(1)
    for(int i = 0; i < array1.length; i++) {                 O(n)
        for(int j = 0; j < array2.length; j++) {             O(n)
            if(array1[i] == array2[j]) {                     O(1)
                System.out.println("Match Found:" + array1[i] + " " + array2[j]);   O(1)
                matches++;                                   O(1)
            }
        }
    }
    return matches;                                          O(1)
}
```

```
array 1: [1, 7, 5, 3, 2]
array 2: [0, 1, 6]
```

$| A \cap B |$

Value returned: 1

Running time?

$O(n^2)$ where n = # of elements in the array

Are array1 and array2 always the same size?

Algorithm Analysis: Intersection size between two sets

```java
public static int calculate_matches(int[] array1, int[] array2) {
    int matches= 0;                                   O(1)
    for(int i = 0; i < array1.length; i++) {          O(n)
        for(int j = 0; j < array2.length; j++) {      O(m)
            if(array1[i] == array2[j]) {              O(1)
                System.out.println("Match Found:" + array1[i] + " " + array2[j]);   O(1)
                matches++;                            O(1)
            }
    }
    return matches;                                   O(1)
}
```

```
array 1: [1, 7, 5, 3, 2]
array 2: [0, 1, 6]
```

$| A \cap B |$

Value returned: 1

Running time?

O(n * m) where n = # of elements in array1 and m = # of elements in array2

Algorithm Analysis: Escaping Jail

```
public static void escape_jail() {







}
```

Algorithm Analysis: Escaping Jail in Monopoly

```
public static void escape_jail() {
        Random rand = new Random();
        int roll1, roll2;
        do {
                roll1 = rand.nextInt(6) + 1;
                roll2 = rand.nextInt(6) + 1;
        } while (roll1 != roll2); // Keep rolling until doubles

        System.out.println("doubles rolled! you have escaped jail");
}
```

This program loops until doubles are rolled

Algorithm Analysis: Escaping Jail in Monopoly

```
public static void escape_jail() {
        Random rand = new Random();
        int roll1, roll2;
        do {
                roll1 = rand.nextInt(6) + 1;
                roll2 = rand.nextInt(6) + 1;
        } while (roll1 != roll2); // Keep rolling until doubles

        System.out.println("doubles rolled! you have escaped jail");
}
```

This program loops until doubles are rolled

Algorithm Analysis: Escaping Jail in Monopoly

```
public static void escape_jail() {
        Random rand = new Random();     O(1)*
        int roll1, roll2;
        do {
                roll1 = rand.nextInt(6) + 1;
                roll2 = rand.nextInt(6) + 1;
        } while (roll1 != roll2); // Keep rolling until doubles

        System.out.println("doubles rolled! you have escaped jail");
}
```

This program loops until doubles are rolled

Algorithm Analysis: Escaping Jail in Monopoly

```
public static void escape_jail() {
        Random rand = new Random();   O(1)*
        int roll1, roll2;   O(1)
        do {
                roll1 = rand.nextInt(6) + 1;   O(1)
                roll2 = rand.nextInt(6) + 1;   O(1)
        } while (roll1 != roll2); // Keep rolling until doubles

        System.out.println("doubles rolled! you have escaped jail");
}
```

This program loops until doubles are rolled

Algorithm Analysis: Escaping Jail in Monopoly

```
public static void escape_jail() {
       Random rand = new Random();  O(1)*
       int roll1, roll2;  O(1)
       do {
              roll1 = rand.nextInt(6) + 1;  O(1)
              roll2 = rand.nextInt(6) + 1;  O(1)
       } while (roll1 != roll2); // Keep rolling until doubles  O(???)

       System.out.println("doubles rolled! you have escaped jail");
}
```

How many times does the while loop repeat?

This program loops until doubles are rolled

Algorithm Analysis: Escaping Jail in Monopoly

```
public static void escape_jail() {
        Random rand = new Random();     ⟸ O(1)*
        int roll1, roll2;     ⟸ O(1)
        do {
                roll1 = rand.nextInt(6) + 1;     ⟸ O(1)
                roll2 = rand.nextInt(6) + 1;     ⟸ O(1)
        } while (roll1 != roll2); // Keep rolling until doubles     ⟸ O(???)

        System.out.println("doubles rolled! you have escaped jail");
}
```

Worst case scenario: they get <u>really</u> unlucky and never roll doubles → Infinite loop!

This program loops until doubles are rolled

Algorithm Analysis: Escaping Jail in Monopoly

```
public static void escape_jail() {
        Random rand = new Random();          O(1)*
        int roll1, roll2;                    O(1)
        do {
                roll1 = rand.nextInt(6) + 1;     O(1)
                roll2 = rand.nextInt(6) + 1;     O(1)
        } while (roll1 != roll2); // Keep rolling until doubles     O(???)

        System.out.println("doubles rolled! you have escaped jail");
}
```

Worst case scenario: they get really unlucky and never roll doubles → Infinite loop!

This program loops until doubles are rolled

When analyzing an algorithm with Big O notation, there is an assumption that the program will eventually terminate for any input

The Big O for worst case is **unbounded** (infinite rolls is possible, but extremely extremely unlikely

Algorithm Analysis: Escaping Jail in Monopoly

```
public static void escape_jail() {
        Random rand = new Random();          O(1)*
        int roll1, roll2;                    O(1)
        do {
                roll1 = rand.nextInt(6) + 1;    O(1)
                roll2 = rand.nextInt(6) + 1;    O(1)
        } while (roll1 != roll2); // Keep rolling until doubles    O(???)

        System.out.println("doubles rolled! you have escaped jail");
}
```

Worst case scenario: they get really unlucky and never roll doubles → Infinite loop!

This program loops until doubles are rolled

When analyzing an algorithm with Big O notation, there is an assumption that the program will eventually terminate for any input

The Big O for worst case is **unbounded** (infinite rolls is possible, but extremely extremely unlikely

For a program that uses randomness, it is more helpful to look at **expected** number of iterations or average case analysis *(there is about 1/6 chance to roll doubles → 6 iterations)*

Algorithm Analysis: Escaping Jail in Monopoly

```
public static void escape_jail() {
        Random rand = new Random();
        int roll1, roll2;
        do {
                roll1 = rand.nextInt(6) + 1;
                roll2 = rand.nextInt(6) + 1;
        } while (roll1 != roll2); // Keep rolling until doubles

        System.out.println("doubles rolled! you have escaped jail");
}
```

This program loops until doubles are rolled

Running time for worst case: **unbound**

Average case: O(6) $\in O(1)$

There are many algorithms that use randomness, but are guaranteed to have a fixed amount of iterations

Algorithm Analysis: Generating Anagrams

```
String message = “abcde”
```

Algorithm Analysis: Generating Anagrams

```
String message = “abcde”
```

Anagrams of `message`:

```
abcde, abced, abdce, abdec, abecd, abedc,
bacde, baced, badce, badec, baecd, baedc,
cabde, cabed, cadbe, cadeb, caebd, caedb,
dabce, dabec, dacbe, daceb, daebc, daecb,
eabcd, eabdc, eacbd, eacdb, eadbc, eadcb,
bacde, baced, badce, badec, baecd, baedc,
bcade, bcaed, bcdae, bcdea, bcead, bceda,
bdace, bdaec, bdcae, bdcea, bdeac, bdeca,
beacd, beadc, becad, becda, bedac, bedca,
cabde, cabed, cadbe, cadeb, caebd, caedb,
cbade, cbaed, cbdae, cbdea, cbead, cbeda,
cdabe, cdaeb, cdbae, cdbea, cdeab, cdeba,
ceabd, ceadb, cebad, cebda, cedab, cedba,
dabce, dabec, dacbe, daceb, daebc, daecb,
dbace, dbaec, dbcae, dbcea, dbeac, dbeca,
dcabe, dcaeb, dcbae, dcbea, dceab, dceba,
deabc, deacb, debac, debca, decab, decba,
eabcd, eabdc, eacbd, eacdb, eadbc, eadcb,
ebacd, ebadc, ebcad, ebcda, ebdac, ebdca,
ecabd, ecadb, ecbad, ecbda, ecdab, ecdba,
edabc, edacb, edbac, edbca, edcab, edcba
```

`String message = “abcde”`

Anagrams of `message`:

```
abcde, abced, abdce, abdec, abecd, abedc,
bacde, baced, badce, badec, baecd, baedc,
cabde, cabed, cadbe, cadeb, caebd, caedb,
dabce, dabec, dacbe, daceb, daebc, daecb,
eabcd, eabdc, eacbd, eacdb, eadbc, eadcb,
bacde, baced, badce, badec, baecd, baedc,
bcade, bcaed, bcdae, bcdea, bcead, bceda,
bdace, bdaec, bdcae, bdcea, bdeac, bdeca,
beacd, beadc, becad, becda, bedac, bedca,
cabde, cabed, cadbe, cadeb, caebd, caedb,
cbade, cbaed, cbdae, cbdea, cbead, cbeda,
cdabe, cdaeb, cdbae, cdbea, cdeab, cdeba,
ceabd, ceadb, cebad, cebda, cedab, cedba,
dabce, dabec, dacbe, daceb, daebc, daecb,
dbace, dbaec, dbcae, dbcea, dbeac, dbeca,
dcabe, dcaeb, dcbae, dcbea, dceab, dceba,
deabc, deacb, debac, debca, decab, decba,
eabcd, eabdc, eacbd, eacdb, eadbc, eadcb,
ebacd, ebadc, ebcad, ebcda, ebdac, ebdca,
ecabd, ecadb, ecbad, ecbda, ecdab, ecdba,
edabc, edacb, edbac, edbca, edcab, edcba
```

An algorithm will generate all **permutations** of a string

Permutations: order matters
Combination: order does not matter

Combination:

A pizza with **[Pepperoni, Sausage, Cheese]** is the same pizza with **[Sausage, Cheese, Pepperoni]**

Permutation:

The lock code `8124` is a totally different code than `1824`

Given a set of elements, there will always be more permutations that combinations

Algorithm Analysis: Generating Anagrams

```
String message = “abcde”
```

Anagrams of `message`:

```
abcde, abced, abdce, abdec, abecd, abedc,
bacde, baced, badce, badec, baecd, baedc,
cabde, cabed, cadbe, cadeb, caebd, caedb,
dabce, dabec, dacbe, daceb, daebc, daecb,
eabcd, eabdc, eacbd, eacdb, eadbc, eadcb,
bacde, baced, badce, badec, baecd, baedc,
bcade, bcaed, bcdae, bcdea, bcead, bceda,
bdace, bdaec, bdcae, bdcea, bdeac, bdeca,
beacd, beadc, becad, becda, bedac, bedca,
cabde, cabed, cadbe, cadeb, caebd, caedb,
cbade, cbaed, cbdae, cbdea, cbead, cbeda,
cdabe, cdaeb, cdbae, cdbea, cdeab, cdeba,
ceabd, ceadb, cebad, cebda, cedab, cedba,
dabce, dabec, dacbe, daceb, daebc, daecb,
dbace, dbaec, dbcae, dbcea, dbeac, dbeca,
dcabe, dcaeb, dcbae, dcbea, dceab, dceba,
deabc, deacb, debac, debca, decab, decba,
eabcd, eabdc, eacbd, eacdb, eadbc, eadcb,
ebacd, ebadc, ebcad, ebcda, ebdac, ebdca,
ecabd, ecadb, ecbad, ecbda, ecdab, ecdba,
edabc, edacb, edbac, edbca, edcab, edcba
```

How many possible anagrams (permutations) are there?

`String message = “abcde”`

Anagrams of `message`:

```
abcde, abced, abdce, abdec, abecd, abedc,
bacde, baced, badce, badec, baecd, baedc,
cabde, cabed, cadbe, cadeb, caebd, caedb,
dabce, dabec, dacbe, daceb, daebc, daecb,
eabcd, eabdc, eacbd, eacdb, eadbc, eadcb,
bacde, baced, badce, badec, baecd, baedc,
bcade, bcaed, bcdae, bcdea, bcead, bceda,
bdace, bdaec, bdcae, bdcea, bdeac, bdeca,
beacd, beadc, becad, becda, bedac, bedca,
cabde, cabed, cadbe, cadeb, caebd, caedb,
cbade, cbaed, cbdae, cbdea, cbead, cbeda,
cdabe, cdaeb, cdbae, cdbea, cdeab, cdeba,
ceabd, ceadb, cebad, cebda, cedab, cedba,
dabce, dabec, dacbe, daceb, daebc, daecb,
dbace, dbaec, dbcae, dbcea, dbeac, dbeca,
dcabe, dcaeb, dcbae, dcbea, dceab, dceba,
deabc, deacb, debac, debca, decab, decba,
eabcd, eabdc, eacbd, eacdb, eadbc, eadcb,
ebacd, ebadc, ebcad, ebcda, ebdac, ebdca,
ecabd, ecadb, ecbad, ecbda, ecdab, ecdba,
edabc, edacb, edbac, edbca, edcab, edcba
```

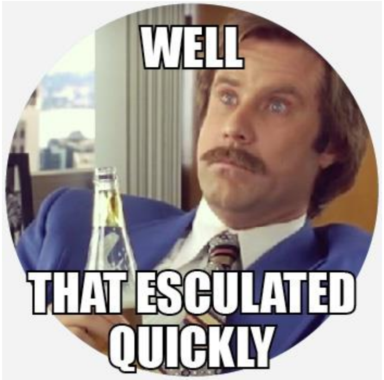
How many possible anagrams (permutations) are there?

$$\underline{5} \times \underline{4} \times \underline{3} \times \underline{2} \times \underline{1}$$

= 120 permutations

$5!$ “five factorial”

Algorithm Analysis: Generating Anagrams




```
String message = “abcdef”
```

Anagrams of `message`:

abcdef, abcedf, abcfde, abcfed, abdcfe, abdecf, abdfce, abdfec, abecdf, abecfd, abedcf, abedfc, abefcd, abefdc, acbdef, acbdfe, acbedf, acbefd, acdbef, acdbfe, acdebf, acdefb, acebdf, acebfd, acedbf, acedfb, acefbd, acefdb, adbcef, adbcfe, adbecf, adbefc, adcbef, adcbfe, adcdeb, adcdef, adcebf, adcefb, adcfbe, adcfeb, adebcf, adebfc, adecbf, adecfb, adefbc, adefcb, aebcdf, aebcfd, aebdcf, aebdfc, aebfcd, aebfdc, aecbdf, aecbfd, aecdbf, aecdfb, aecfbd, aecfdb, aedbcf, aedbfc, aedcbf, aedcfb, aedfbc, aedfcb, aefbcd, aefbdc, aefcbd, aefcdb, aefdbc, aefdcb, bacdef, bacdfe, bacedf, bacefd, bacfde, bacfed, badcef, badcfe, badecf, badefc, badfce, badfec, baecdf, baecfd, baedcf, baedfc, baefcd, baefdc, bcadef, bcadfe, bcaedf, bcaefd, bcafde, bcafed, bcdaef, bcdafe, bcdeaf, bcdefa, bcdfea, bcdfae, bceadf, bceafd, bcedaf, bcedfa, bcefad, bcefda, bdacef, bdacfe, bdaecf, bdaefc, bdafce, bdafec, bdcaef, bdcafe, bdceaf, bdcefa, bdcfea, bdcfae, bdeacf, bdeafc, bdecaf, bdecfa, bdefac, bdefca, bdface, bdfaec, bdfcae, bdfcea, bdfeac, bdfeca, beacdf, beacfd, beadcf, beadfc, beafcd, beafdc, becadf, becafd, becdaf, becdfa, becfad, becfda, bedacf, bedafc, bedcaf, bedcfa, bedfac, bedfca, befacd, befadc, befcad, befcda, befdac, befdca, cabdef, cabdfe, cabedf, cabefd, cabfde, cabfed, cadbef, cadbfe, cadebf, cadefb, cadfbe, cadfeb, caebdf, caebfd, caedbf, caedfb, caefbd, caefdb, cbadef, cbadfe, cbaedf, cbaefd, cbafde, cbafed, cbdaef, cbdafe, cbdeaf, cbdefa, cdbfea, cdbfae, cbeadf, cbeafd, cbedaf, cbedfa, cbefad, cbefda, cdabef, cdabfe, cdaebf, cdaefb, cdafbe, cdafeb, cdbaef, cdbafe, cdbeaf, cddefa, cdfeab, cdfeba, cdebaf, cdeafb, cdefba, cefbad, cdefba, cfabde, cfabed, cfadbe, cfadeb, cfaebd, cfaedb, cfbade, cfbaed, cfbdae, cfbdea, cfbead, cfbeda, cfdabe, cfdaeb, cfdbae, cfdbea, cfdeab, cfdeba, dabcde, dabced, dabdce, dabdec, dabecd, dabedc, dacbde, dacbed, dacdbe, dacdeb, dacebd, dcaedb, dbaecd, dbaedc, dbaecr, dbcade, dbcaed, dbcade, dbcdea, dceadb, dcebad, dceabd, dceadb, dcebad, dceabf, deabcd, deabdc, deacbd, deacdb, deadbc, deadcb, debacd, debadc, debcad, debcda, debdac, debdca, decabd, decadb, decbad, decbda, decdab, decdba, defabc, defacb, defbac, defbca, defcab, defcba, eabcdf, eabcfb, eabdcf, eabdfc, eabfcd, eabfdc, eacbdf, eacbfd, eacdbf, eacdfb, eacfbd, eacfdb, eadbcf, eadbfc, eadcbf, eadcfb, eadfbc, eadfcb, eafbcd, eafbdc, eafcbd, eafcdb, eafdbc, eafdcb, ebacdf, ebacfd, ebadcf, ebadfc, ebafcd, ebafdc, ebcadf, ebcafd, ebcdaf, ebcdfa, ebcfad, ebcfda, ebdacf, ebadfc, ebdcaf, ebdcfa, ebdfac, ebdfca, efbcad, efbacd, efcdbd, efcdba, efcdab, efcbda, efacdb, efadbc,

Anagrams of length 6?

6! = 720 💀

Algorithm Analysis: Generating Anagrams

```
public static List<String> generateAnagrams(String str) {
		List<String> anagrams = new ArrayList<>();
		anagrams.add(String.valueOf(str.charAt(0)));
		for (int i = 1; i < str.length(); i++) {
			char currentChar = str.charAt(i);
			List<String> newAnagrams = new ArrayList<>();
			for (String anagram : anagrams) {
				for (int j = 0; j <= anagram.length(); j++) {
					String newAnagram = anagram.substring(0, j) + currentChar + anagram.substring(j);
					newAnagrams.add(newAnagram);
				}
			}
			anagrams = newAnagrams;
		}
		return anagrams;
}
```

Running time?

Algorithm Analysis: Generating Anagrams

```
public static List<String> generateAnagrams(String str) {
        List<String> anagrams = new ArrayList<>();
        anagrams.add(String.valueOf(str.charAt(0)));
        for (int i = 1; i < str.length(); i++) {
                char currentChar = str.charAt(i);
                List<String> newAnagrams = new ArrayList<>();
                for (String anagram : anagrams) {
                        for (int j = 0; j <= anagram.length(); j++) {
                                String newAnagram = anagram.substring(0, j) + currentChar + anagram.substring(j);
                                newAnagrams.add(newAnagram);
                        }
                }
                anagrams = newAnagrams;
        }
        return anagrams;
}
```

Running time?

Theres a lot of loops and weird stuff happening… let's focus on how much output is created!

Algorithm Analysis: Generating Anagrams

```
public static List<String> generateAnagrams(String str) {
		List<String> anagrams = new ArrayList<>();
		anagrams.add(String.valueOf(str.charAt(0)));
		for (int i = 1; i < str.length(); i++) {
				char currentChar = str.charAt(i);
				List<String> newAnagrams = new ArrayList<>();
				for (String anagram : anagrams) {
						for (int j = 0; j <= anagram.length(); j++) {
								String newAnagram = anagram.substring(0, j) + currentChar + anagram.substring(j);
								newAnagrams.add(newAnagram);
						}
				}
				anagrams = newAnagrams;
		}
		return anagrams;
}
```

Running time?

Theres a lot of loops and weird stuff happening… let's focus on how much output is created!

str length	# of string made
1	1
2	2
3	6
5	24
6	720
7	5040

This is not exponential growth… this is far worse

Algorithm Analysis: Generating Anagrams

```
public static List<String> generateAnagrams(String str) {
		List<String> anagrams = new ArrayList<>();
		anagrams.add(String.valueOf(str.charAt(0)));
		for (int i = 1; i < str.length(); i++) {
			char currentChar = str.charAt(i);
			List<String> newAnagrams = new ArrayList<>();
			for (String anagram : anagrams) {
				for (int j = 0; j <= anagram.length(); j++) {
					String newAnagram = anagram.substring(0, j) + currentChar + anagram.substring(j);
					newAnagrams.add(newAnagram);
				}
			}
			anagrams = newAnagrams;
		}
		return anagrams;
}
```

Running time?

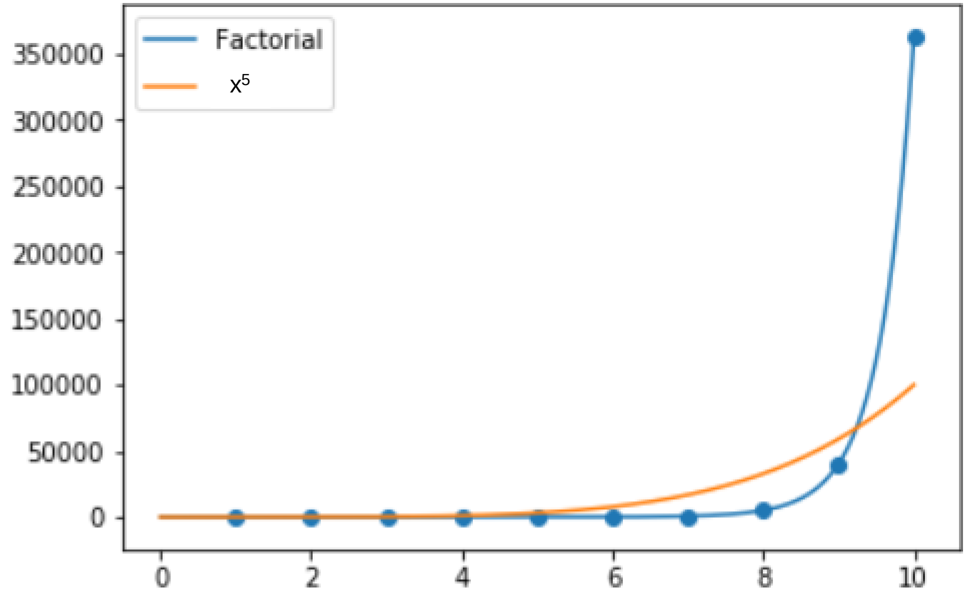
Theres a lot of loops and weird stuff happening… let’s focus on how much output is created!

str length	# of string made
1	1
2	2
3	6
5	24
6	720
7	5040

This is not exponential growth… this is far worse

For a string length n, this algorithm does **O(n!)** amount of work

For a string length n, this algorithm does **O(n!)** amount of work



Algorithms with `O(n!)` running time are **terrible** and **infeasible**

Sometimes they might be the only option available…

Given input n

(There are a few other running times we will see later this semester)

"Best"

$O(1)$ — No loops
Constant amount of work

$O(n)$ — Single loop or sequential for loops

$O(n^2)$ — Nested for loops
Quadratic growth

$O(n^3)$ — Triple nested for loops
Cubic growth

$O(x^n)$ — Brute Forcing combinations
Exponential growth

$O(n!)$ — Brute Forcing permutations
Factorial growth

"Worst"

Given input n

(There are a few other running times we will see later this semester)

"Best"

O(1) — No loops
Constant amount of work

O(n) — Single loop or sequential for loops

O(n^2) — Nested for loops
Quadratic growth

O(n^3) — Triple nested for loops
Cubic growth

Polynomial time
"Acceptable"

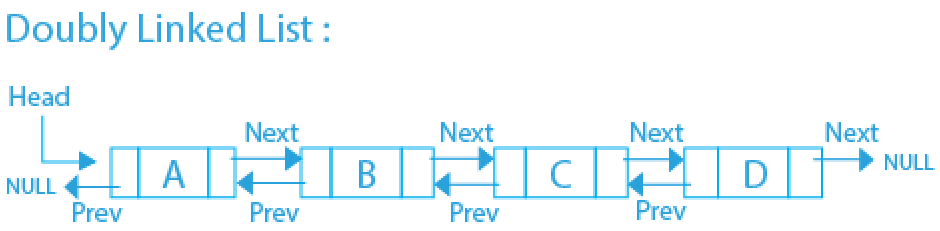
O(x^n) — Brute Forcing combinations
Exponential growth

"Unacceptable"
You are bad at coding (maybe)

O(n!) — Brute Forcing permutations
Factorial growth

"Worst"

List Running Times



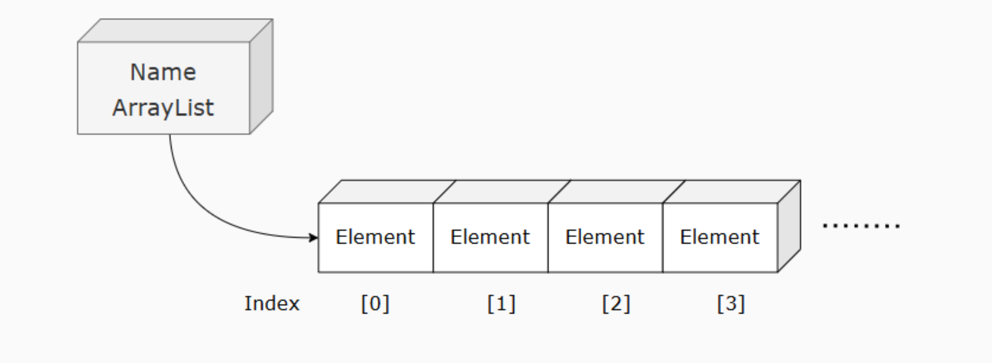


Figure 1: ArrayList in Java

Operation	Running Time
Creation	O(1)
Adding an element	O(1)
Removing head or tail	O(1)
Searching / Contains	O(n) must check every node

Operation	Running Time
Creation of empty list	O(1)
Adding an element	O(n) must grow array and copy
Removing first or last	O(n) must shrink array and copy
Searching / Contains	O(n) must check every node

Where n = # of nodes in Linked List

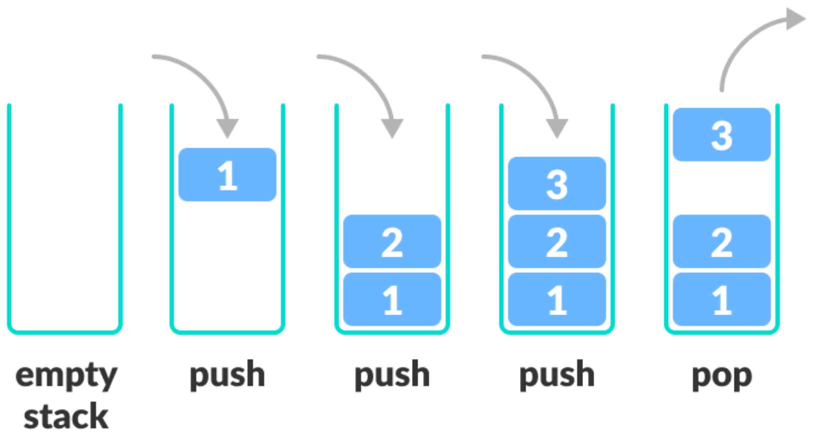
Where n = # of nodes in arraylist

Adding to a linked list is more efficient than adding to a filled array

Stack running time analysis

(Array Implementation)

```
public StackArray() {
    data = new Element[8];
    top_of_stack = -1;
    size = 0;
}
```



(Linked List Implementation)

```
public StackLinkedList() {
   data = new LinkedList<Element>();
   this.size = 0;
}
```

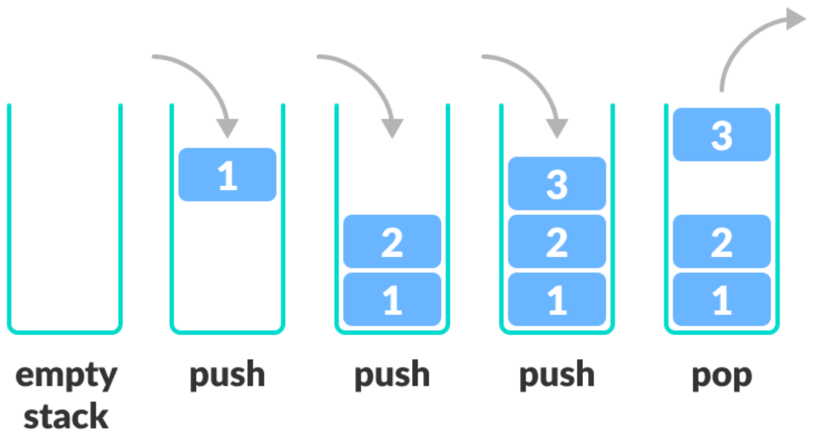
Algorithm	w/ Array	w/ Linked List
Creation		
Push()		
Pop()		
peek()		
Print()		

Stack running time analysis

(Array Implementation)

```
public StackArray() {
    data = new Element[8];  O(n)
    top_of_stack = -1;  O(1)
    size = 0;  O(1)
}
```

Total Running time: O(n) n = | array |



(Linked List Implementation)

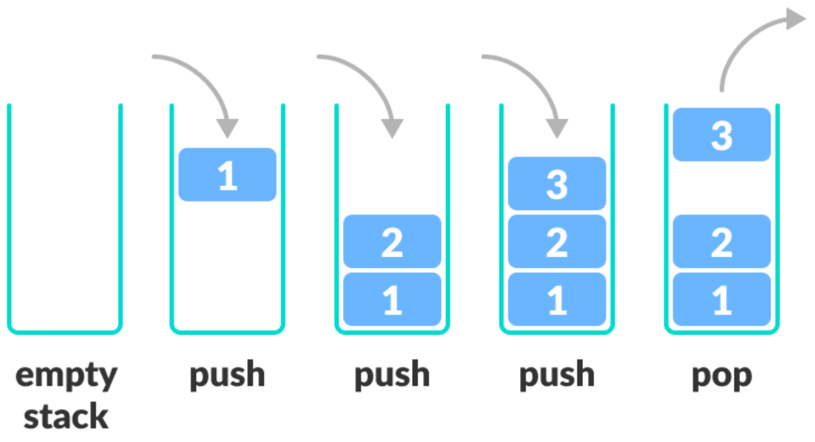
```
public StackLinkedList() {
    data = new LinkedList<Element>(); O(1)
    this.size = 0; O(1)
}
```

Total Running time: O(1)

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()		
Pop()		
peek()		
Print()		

Stack running time analysis

(Array Implementation)

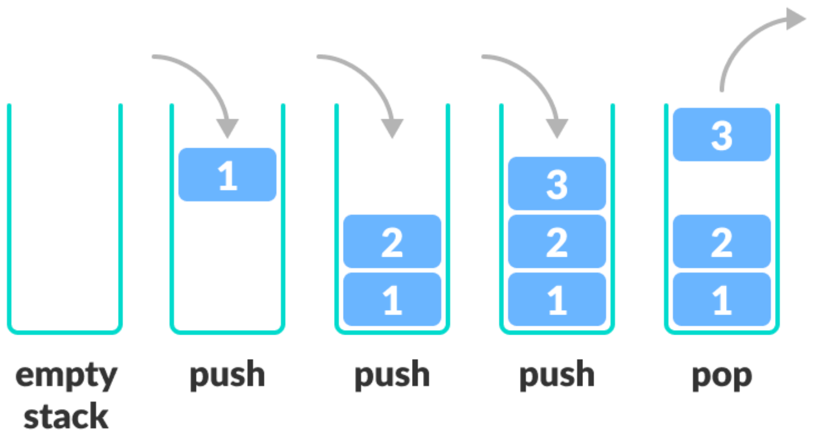



```
public void push(Element newElement) {

   if(this.size == this.data.length) {
     return;
   }
   else {
     this.top_of_stack++;
     data[this.top_of_stack] = newElement;
     this.size++;
   }
}
```

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()		
Pop()		
peek()		
Print()		

Stack running time analysis

(Array Implementation)



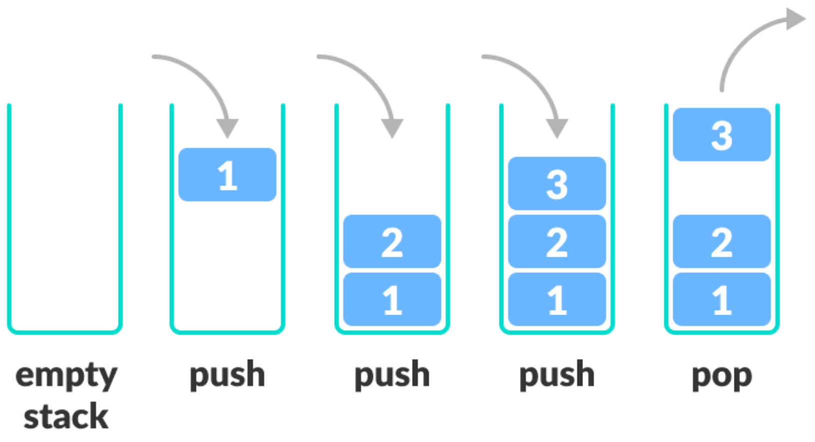

```
public void push(Element newElement) {

   if(this.size == this.data.length) { O(1)
     return;  O(1)
   }
   else {
     this.top_of_stack++;   O(1)
     data[this.top_of_stack] = newElement; O(1)
     this.size++; O(1)
   }
}
```

Total Running Time: O(1)

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	
Pop()		
peek()		
Print()		

Stack running time analysis

(Linked List Implementation)

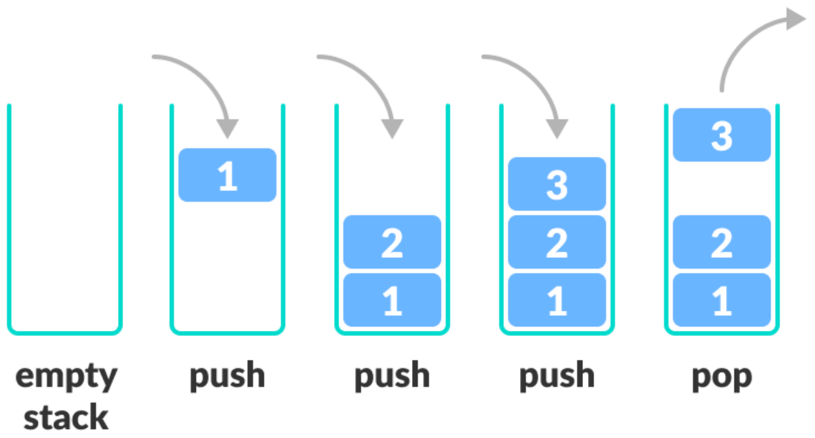



```
public void push(Element newElement) {
    data.addFirst(newElement);
    this.size++;
}
```

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	
Pop()		
peek()		
Print()		

Stack running time analysis

(Linked List Implementation)




```
public void push(Element newElement) {
    data.addFirst(newElement); O(1)
    this.size++; O(1)
}
```

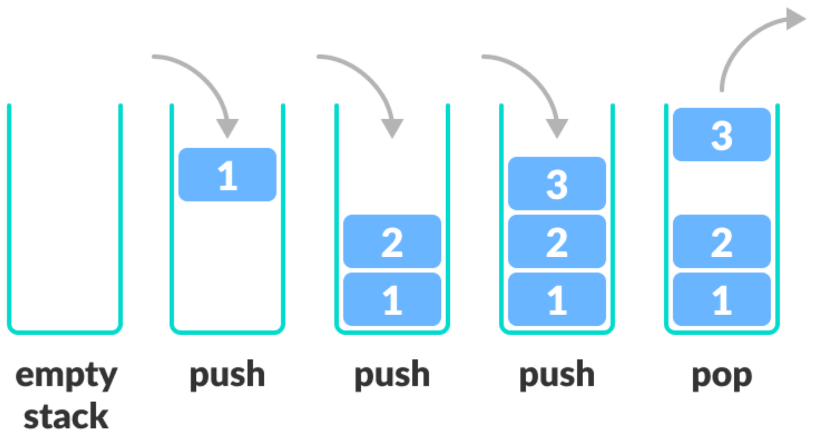
Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()		
peek()		
Print()		

Total Running Time: O(1)

Stack running time analysis

(Array)

```
public void pop() {
    if(this.size == 0) {
        return;
    }
    else {
        this.data[this.top_of_stack] = null;
        this.top_of_stack--;
        this.size--;
    }
}
```



Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()		
peek()		
Print()		

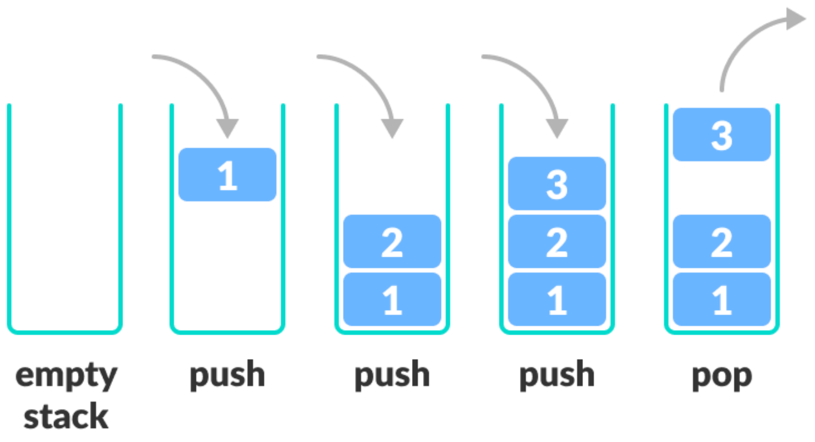
(Linked List)

```
public void pop() {
    if(this.size == 0) {
        return;
    }
    else {
        this.data.removeFirst();
        this.size--;
    }
}
```

Stack running time analysis

(Array)

```
public void pop() {
    if(this.size == 0) {
        return;            O(1)
    }
    else {
        this.data[this.top_of_stack] = null; O(1)
        this.top_of_stack--; O(1)
        this.size--; O(1)
    }
}
```



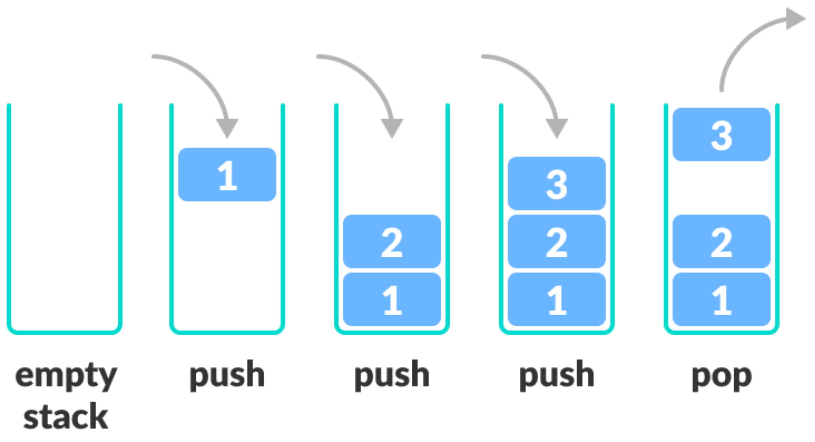
Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()		
Print()		

```
public void pop() {  (Linked List)
    if(this.size == 0) {
        return;            O(1)
    }
    else {
        this.data.removeFirst();  O(1)
        this.size--;  O(1)
    }
}
```

Stack running time analysis

(Array)

```
public Element peek() {
   if(this.size != 0) {
      return this.data[this.top_of_stack];
   }
   else {
     return null;
   }
}
```



(Linked List)

```
public Element peek() {
   if(this.size != 0) {
      return this.top_of_stack;
   }
   else {
      return null;
   }
}
```

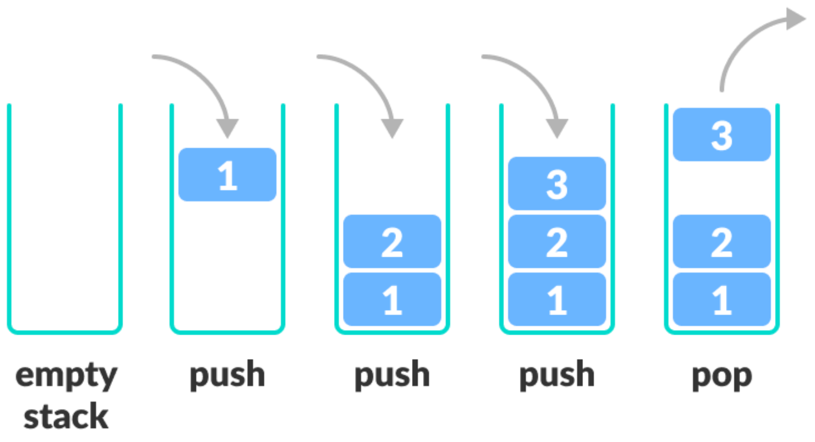
Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()		
Print()		

Stack running time analysis

(Array)

```
public Element peek() {
   if(this.size != 0) {
      return this.data[this.top_of_stack];
   }
   else {
     return null;
   }
}
```

O(1)



(Linked List)

```
public Element peek() {
   if(this.size != 0) {
      return this.top_of_stack;
   }
   else {
      return null;
   }
}
```

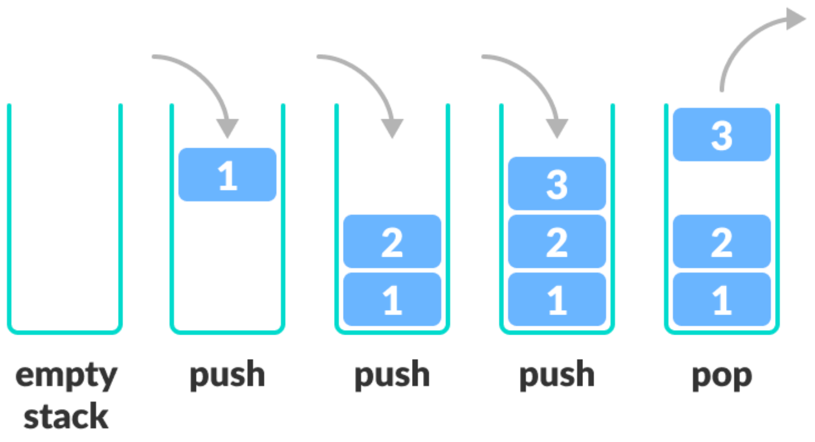
O(1)

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()		

Stack running time analysis

(Array)

```
public void printStack() {
   for(int i = this.size-1; i >= 0; i--) {
      this.data[i].printElement();
   }
}
```



(Linked List)

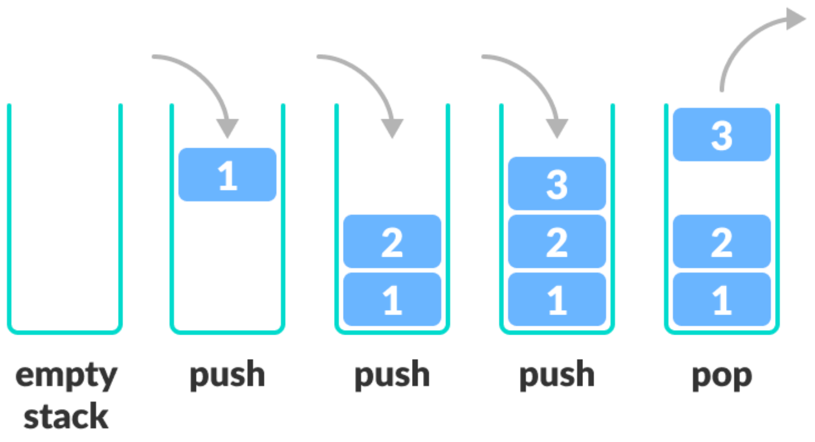
```
public void printStack() {
    for(Element each :  this.data) {
       each.printElement();
    }
}
```

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()		

Stack running time analysis

(Array)

```
public void printStack() {
    for(int i = this.size-1; i >= 0; i--) { O(n)
        this.data[i].printElement();
    }
}
```



(Linked List)

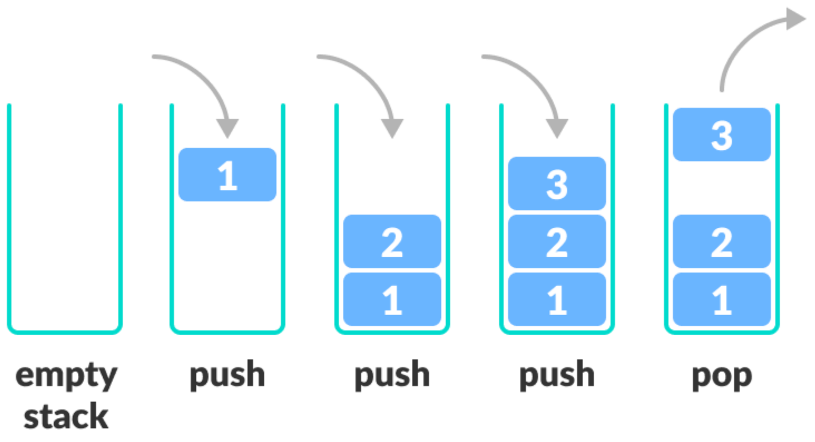
```
public void printStack() {
    for(Element each :  this.data) { O(n)
        each.printElement();
    }
}
```

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()		

Stack running time analysis

(Array)

```
public void printStack() {
    for(int i = this.size-1; i >= 0; i--) { O(n)
        this.data[i].printElement();
    }
}
```



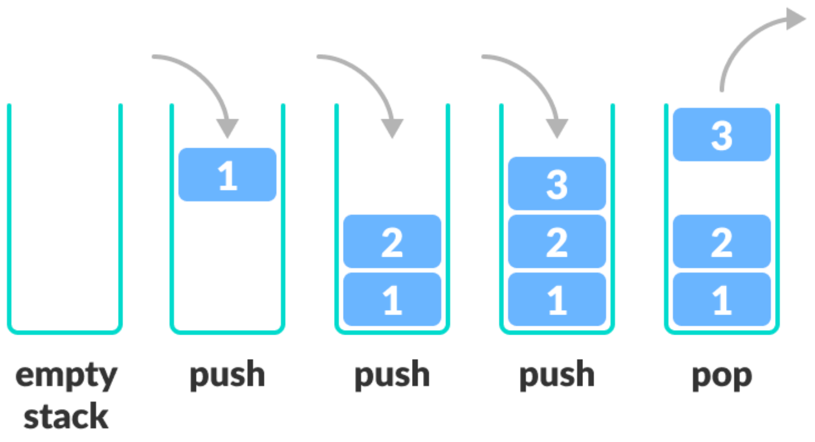
(Linked List)

```
public void printStack() {
    for(Element each :  this.data) { O(n)
        each.printElement();
    }
}
```

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()	O(n)	O(n)

Where n = # of elements in the stack

Stack running time analysis



Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()	O(n)	O(n)

Where n = # of elements in the stack

Adding and removing elements from a stack runs in **`O(1)`** no matter if array or linked list is used

An array will be fixed-sized, linked list will be dynamic

Linked List

Operation	Running Time
Creation	O(1)
Adding an element	O(1)
Removing head or tail	O(1)
Searching / Contains	O(n) must check every node

Where n = # of nodes in Linked List

Dynamic Array / ArrayList

Operation	Running Time
Creation of empty list	O(1)
Adding an element	O(n) must grow array and copy
Removing first or last	O(n) must shrink array and copy
Searching / Contains	O(n) must check every node

Where n = # of nodes in arraylist

Stack

Algorithm	w/ Array	w/ Linked List
Creation	O(n)	O(1)
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
peek()	O(1)	O(1)
Print()	O(n)	O(n)

Where n = # of elements in the stack

I don’t force your to memorize things, but you should memorize running times for basic operations of fundamental data structures