



# Discussion 7: Asymptotics

Please fill out the quiz, when you get it.



# Administrivia

- Project 2 Phase 1 due February 26
- Project 2 Phase 2 due March 5th
- Labs this week will be working on project 2
- Don't forget to fill out the partner feedback form!
  - <https://goo.gl/forms/IHu4d2DzldVwtJST2>



# Motivation: Measuring Runtime

- How do we measure the runtime of a program?
- How do we compare the runtime of two programs as the input size grows?



# Measuring Order of Growth

- Determine a cost model (in this case number of operations)
- Define an algebraic expression  $f(n)$  that expresses this cost for some input  $n$ .
- Drop multiplicative constants and lower order terms
- Any exponential dominates any polynomial
- Any polynomial dominates any logarithm



## $\Theta$ (Big Theta)

- Let  $f(n)$  and  $g(n)$  be positive real numbers on inputs of size  $n$
- $f \in \Theta(g)$  if there is a constant  $c_1 > 0$  and  $c_2 > 0$  s.t.
  - $c_1 g(n) \leq f(n) \leq c_2 g(n)$  for all  $c_1 \leq c_2$
- Tightly bounded by  $g(n)$  when  $n$  gets significantly large.



## Problem 1a: Ordering Big- Runtimes

- If you're stuck, think about the big-O runtimes as functions and compare the values of the functions as  $n$  becomes really large.



## Problem 1a: Ordering Big- $\Theta$ Runtimes

$\Theta(1) \rightarrow \Theta(\log n) \rightarrow \Theta(n) \rightarrow \Theta(n \log n) \rightarrow \Theta(n^2 \log n) \rightarrow \Theta(n^3) \rightarrow \Theta(2^n) \rightarrow \Theta(n!) \rightarrow \Theta(n^n)$



# Common Asymptotic Sets

- $\Theta(1)$ : constant
- $\Theta(\log n)$ : logarithmic
- $\Theta(\sqrt{n})$ : square root
- $\Theta(n)$ : linear
- $\Theta(n \log n)$ :  $n \log n$
- $\Theta(n^2)$ : quadratic
- $\Theta(n^3)$ : cubic
- $\Theta(2^n)$ : exponential
- $\Theta(n!)$ : factorial





## Problem 2a: Analyzing Runtime

- What are all the possible things that could happen with the code?
- Remember to think generally in terms of large inputs for  $M$  and  $N$ .



## Problem 2a: Analyzing Runtime

```
1  int j = 0;
2  for (int i = N; i > 0; i--) {
3      for (; j <= M; j++) {
4          if (ping(i, j) > 64) {
5              break;
6          }
7      }
8  }
```



## Problem 2a: Analyzing Runtime

- Worst case:  $M + N$
- Best case:  $N$



## Problem 2b: Analyzing Runtime

- Be sure to consider all operations that contribute to the runtime of the program (running mrpoolsort, for loops, array indexing) and that you can drop lower order terms.



## Problem 2b: Analyzing Runtime

```
1  public static boolean mystery(int[] array) {
2      array = mrpoolsort(array);
3      int N = array.length;
4      for (int i = 0; i < N; i += 1) {
5          boolean x = false;
6          for (int j = 0; j < N; j += 1) {
7              if (i != j && array[i] == array[j])
8                  x = true;
9          }
10         if (!x) {
11             return false;
12         }
13     }
14     return true;
15 }
```



## Problem 2b: Analyzing Runtime

- Worst:  $N^2$
- Best:  $N \log N$
- Returns true if every int has duplicate in array and false if there is any unique int in array
- Linear-time algorithm using map



## Problem 2c: Analyzing Runtime

```
1  for (int i = 0; i < N; i += 1) {  
2      for (int j = 1; j <= M; ) {  
3          if (comeOn()) {  
4              j += 1;  
5          } else {  
6              j *= 2;  
7          }  
8      }  
9  }
```



## Problem 2c: Analyzing Runtime

- Worst:  $NM$
- Best:  $N \log M$





## Problem 3: Fast

- What is the current runtime of the algorithm?
- Remember that the array is sorted!



## Problem 3: Fast

```
1  public static boolean findSum(int[] A, int x) {  
2      for (int i = 0; i < A.length; i++){  
3          for (int j = 0; j < A.length; j++) {  
4              if (A[i] + A[j] == x) {  
5                  return true;  
6              }  
7          }  
8      }  
9      return false;  
10 }
```



## Problem 3: Fast

- Naive
  - Worst:  $N^2$
  - Best: 1
- Optimized
  - Worst:  $N$
  - Best: 1

```
1  public static boolean findSumFaster(int[] A, int x){
2      int left = 0;
3      int right = A.length - 1;
4      while (left <= right) {
5          if (A[left] + A[right] == x) {
6              return true;
7          } else if (A[left] + A[right] < x) {
8              left++;
9          } else {
10             right--;
11         }
12     }
13     return false;
14 }
```



## Problem 4: CTCI

```
1  public static int[] union(int[] A, int[] B) {  
2      HashSet<Integer> set = new HashSet<Integer>();  
3      for (int num : A) {  
4          set.add(num);  
5      }  
6      for (int num : B) {  
7          set.add(num);  
8      }  
9      int[] unionArray = new int[set.size()];  
10     int index = 0;  
11     for (int num : set) {  
12         unionArray[index] = num;  
13         index += 1;  
14     }  
15     return unionArray;  
16 }
```



## Problem 4: CTCI

```
1  public static int[] intersection(int[] A, int[] B) {
2      HashSet<Integer> setOfA = new HashSet<Integer>();
3      HashSet<Integer> intersectionSet = new HashSet<Integer>();
4      for (int num : A) {
5          setOfA.add(num);
6      }
7      for (int num : B) {
8          if (setOfA.contains(num)) {
9              intersectionSet.add(num);
10         }
11     }
12     int[] intersectionArray = new int[intersectionSet.size()];
13     int index = 0;
14     for (int num : intersectionSet) {
15         intersectionArray[index] = num;
16         index += 1;
17     }
18     return intersectionArray;
19 }
```