

1 Asymptotic Notation

1.1 Order the following big- O runtimes from smallest to largest.

$O(\log n), O(1), O(n^n), O(n^3), O(n \log n), O(n), O(n!), O(2^n), O(n^2 \log n)$

$O(1), O(\log n), O(n), O(n \log n), O(n^2 \log n), O(n^3), O(2^n), O(n!), O(n^n)$

1.2 Are the statements in the right column true or false? If false, correct the asymptotic notation ($\Omega(\cdot), \Theta(\cdot), O(\cdot)$). Be sure to give the tightest bound. $\Omega(\cdot)$ is the opposite of $O(\cdot)$, i.e. $f(n) \in \Omega(g(n)) \iff g(n) \in O(f(n))$.

$f(n) = 20501$	$g(n) = 1$	$f(n) \in \Theta(g(n))$ Big Theta; but true
$f(n) = n^2 + n$	$g(n) = 0.000001n^3$	$f(n) \in \Omega(g(n))$ Big O
$f(n) = 2^{2n} + 1000$	$g(n) = 4^n + n^{100}$	$f(n) \in O(g(n))$ Big Theta; but true
$f(n) = \log(n^{100})$	$g(n) = n \log n$	$f(n) \in \Theta(g(n))$ Big O
$f(n) = n \log n + 3^n + n$	$g(n) = n^2 + n + \log n$	$f(n) \in \Omega(g(n))$
$f(n) = n \log n + n^2$	$g(n) = \log n + n^2$	$f(n) \in \Theta(g(n))$
$f(n) = n \log n$	$g(n) = (\log n)^2$	$f(n) \in \Omega(g(n))$ Big Omega

2 Analyzing Runtime

2.1 Give the worst case and best case runtime in terms of M and N . Assume `ping` is in $\Theta(1)$ and returns an `int`.

```

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  }
```

Best: $O(1)$
Worst: $O(N \cdot M)$

notice that `j` is initialized outside of the loop so it will not naturally go back to 0!

Best: $O(N)$
Worst: $O(N + M)$

I can understand the worst case, but I cannot understand the best case

- 2.2 Give the worst case and best case runtime where $N = \text{array.length}$. Assume $\text{mrpoolsort}(\text{array})$ is in $\Theta(N \log N)$.

```

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 }

```

Best: $N \log N$
Worst: N^2

Achilles Added Additional Amazing Asymptotic And Algorithmic Analysis Achievements

- (a) What is `mystery()` doing?
`mystery()` returns false if any element in the input array is unique; otherwise returns true.
- (b) Using an ADT, describe how to implement `mystery()` with a better runtime.
 Then, if we make the assumption an **int** can appear in the **array** at most twice, develop a solution using only constant memory.
1. we can use a set and its `.contains()` method, which is $O(n)$
 2. sorting with constant space algorithm and check neighbourhood elements

- 2.3 Give the worst case and best case running time in $\Theta(\cdot)$ notation in terms of M and N . Assume that `comeOn()` is in $\Theta(1)$ and returns a boolean.

```

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  }

```

Best: $O(N * \log M)$
Worst: $O(N * M)$

3 Have You Ever Went Fast?

- 3.1 Given an **int** x and a *sorted* array A of N distinct integers, design an algorithm to find if there exists indices i and j such that $A[i] + A[j] == x$.

Let's start with the naive solution.

```

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 }
```

- (a) How can we improve this solution? *Hint*: Does order matter here?
 since it's a sorted array, instead of traversal, I think we can do binary search

this is still not efficient enough. we can traverse all the elements once, by incrementing the first element or decrementing the last element

- (b) What is the runtime of both the original and improved algorithm?

original: $O(N^2)$

improved: $O(N \log N)$

$O(N)$

```

public class FindSum {
    1 usage
    public static boolean findSum(int[] A, int x) {
        int first = 0;
        int last = A.length - 1;
        while (first < last) {
            if (A[first] + A[last] == x) {
                return true;
            } else if (A[first] + A[last] < x) {
                first++;
            } else {
                last--;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        int[] A = {1, 2, 3, 4, 5, 6};
        System.out.println(findSum(A, x: 1));
    }
}
```

4 CTCI *Extra*

- 4.1 **Union** Write the code that returns an array that is the union between two given arrays. The union of two arrays is a list that includes everything that is in both arrays, with no duplicates. Assume the given arrays do not contain duplicates. For example, the union of {1, 2, 3, 4} and {3, 4, 5, 6} is {1, 2, 3, 4, 5, 6}.

Hint: The method should run in $O(M + N)$ time where M and N is the size of each array.

- 4.2 **Intersect** Now do the same as above, but find the intersection between both arrays. The intersection of two arrays is the list of all elements that are in both arrays. Again assume that neither array has duplicates. For example, the intersection of {1, 2, 3, 4} and {3, 4, 5, 6} is {3, 4}.

Hint: Think about using ADTs other than arrays to make the code more efficient.

```
import java.util.HashSet;

public class Union {
    1 usage
    public static int[] union(int[] arr1, int[] arr2) {
        // The method should run in O(N + M) time
        // Hint: use ADT
        HashSet<Integer> set = new HashSet<>();
        for (int element : arr1) {
            set.add(element);
        }
        for (int element : arr2) {
            set.add(element);
        }
        int[] ans = new int[set.size()];
        int index = 0;
        for (int element : set) {
            ans[index] = element;
            index++;
        }
        return ans;
    }

    public static void main(String[] args) {
        int[] A = {1, 2, 3, 4};
        int[] B = {3, 4, 5, 6};
        int[] result = union(A, B);
        for (int element : result) {
            System.out.println(element);
        }
    }
}
```

```
import java.util.HashSet;

public class Intersect {
    1 usage
    public static int[] intersect(int[] A, int[] B) {
        HashSet<Integer> set1 = new HashSet<>();
        HashSet<Integer> set2 = new HashSet<>();
        for (int element : A) {
            set1.add(element);
        }
        for (int element : B) {
            if (set1.contains(element)) {
                set2.add(element);
            }
        }
        int[] ans = new int[set2.size()];
        int index = 0;
        for (int element : set2) {
            ans[index] = element;
            index++;
        }
        return ans;
    }

    public static void main(String[] args) {
        int[] A = {1, 2, 3, 4};
        int[] B = {3, 4, 5, 6};
        int[] result = intersect(A, B);
        for (int element : result) {
            System.out.println(element);
        }
    }
}
```