INSERTION SORT

Consider an example: arr[]: {12, 11, 13, 5, 6}

1) First Pass: Initially, the first two elements of the array are compared in insertion sort.

12	11	13	5	6

Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.

So, for now 11 is stored in a sorted sub-array.

11	12	13	5	6

2) Second Pass: Now, move to the next two elements and compare them

11	12	13	5	6

Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11

3) Third Pass: Now, two elements are present in the sorted sub-array which are 11 and 12

Moving forward to the next two elements which are 13 and 5

11	12	13	5	6

Both 5 and 13 are not present at their correct place so swap them

11	12	5	13	6

After swapping, elements 12 and 5 are not sorted, thus swap again

11	5	12	13	6

Here, again 11 and 5 are not sorted, hence swap again

5	11	12	13	6

Here, 5 is at its correct position

4) Fourth Pass: Now, the elements which are present in the sorted sub-array are 5, 11 and 12

Moving to the next two elements 13 and 6

5	11	12	13	6

Clearly, they are not sorted, thus perform swap between both

5	11	12	6	13

Now, 6 is smaller than 12, hence, swap again

5	11	6	12	13

Here, also swapping makes 11 and 6 unsorted hence, swap again

5	6	11	12	13

Finally, the array is completely sorted.

Best Case

- The "best case" is that the input array is already sorted
- Then we won't need to execute the inner while loop to insert the chosen element
- Let $t_j = 1$

Worst Case

- In the worst case, the input array is reverse sorted order
- We will need to compare each chosen element to all of the currently-sorted elements
- Let $t_j = j$

Analysis of Efficiency (Best Case)

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} 1$$

$$+c_6 \sum_{j=2}^{n} 0 + c_7 \sum_{j=2}^{n} 0 + c_8(n-1)$$

Summation Identities

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

Analysis of Efficiency (Best Case)

$$T(n) = an - b$$

where

$$a = c_1 + c_2 + c_4 + c_5 + c_8$$

$$b = c_2 + c_4 + c_5 + c_8$$

In the best case, the run time grows *linearly* with input size ($O(n)$)

Analysis of Efficiency (Worst Case)

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right)$$

$$+c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

Analysis of Efficiency (Worst Case)

$$T(n) = an^2 + bn + c$$

where

$$a = \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}$$

$$b = c_1 + c_2 + c_4 + + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8$$

$$c = -\left(c_2 + c_4 + c_5 + c_8\right)$$

n the worst case, the run time grows *quadratically* with input size ($O(n^2)$)

- **When to Use**: Insertion Sort is efficient for small data sets or nearly sorted data.
- **Why**: It has a simple implementation and works well when the input data is mostly sorted because it makes minimal comparisons and swaps.

Insertion Sort

```
INSERTION-SORT(A)

for j = 2 to A.length
  key = A[j]

  // shift elements in sorted sequence A[1..j - 1]
  i = j - 1
  while i > 0 and A[i] > key
    A[i+1] = A[i]
    i = i - 1

  // insert A[j] into the sorted sequence
  A[i+1] = key
```

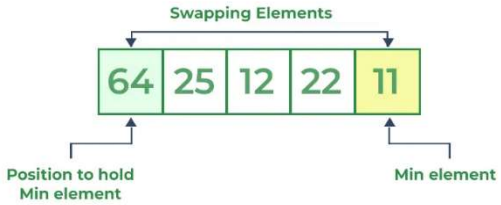
```python
def insertion_sort(lst):
    for i in range(1, len(lst)):
        key = lst[i]
        j = i - 1
        while j >= 0 and key < lst[j]:
            lst[j + 1] = lst[j]
            j -= 1
        lst[j + 1] = key
```

SELECTION SORT

Lets consider the following array as an example: arr[] = {64, 25, 12, 22, 11}

1) For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where 64 is stored presently, after traversing whole array it is clear that 11 is the lowest value.

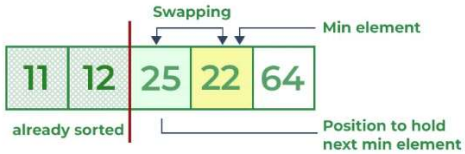
Thus, replace 64 with 11. After one iteration 11, which happens to be the least value in the array, tends to appear in the first position of the sorted list.



2) Second Pass: For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.

After traversing, we found that 12 is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.

3) Third Pass: Now, for third place, where 25 is present again traverse the rest of the array and find the third least value present in the array. While traversing, 22 came out to be the third least value and it should appear at the third place in the array, thus swap 22 with element present at third position.



4) Fourth pass: Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array. As 25 is the 4th lowest value hence, it will place at the fourth position.

5) Fifth Pass: At last the largest value present in the array automatically get placed at the last position in the array. The resulted array is the sorted array.

Analysis of Efficiency

$$T(n) = c_1 + \sum_{j=1}^{n-1} (c_2 + c_3 + c_4 + \sum_{i=j+1}^{n} (c_5 + c_6 + b_1 c_7 + c_8) + c_5 + c_6 + c_7 + c_8 + c_9) + c_2$$

Assume b_1 always equals 1 (list is reversed sorted)

Analysis of Efficiency

$$T(n) = \frac{3}{2}an^2 + (n-1)(b-a) - \frac{3}{2}an + c_1 + c_2$$

$$a = c_5 + c_6 + c_7 + c_8$$

$$b = c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8 + c_9$$

Analysis of Efficiency

```
SELECTION-SORT(A)
for j = 1 to A.length - 1
  smallest = j

  // find smallest element in A[j..n]
  for i = j + 1 to A.length
    if A[i] < A[smallest]
      smallest = i

  // swap A[j] and A[smallest]
  temp = A[j]
  A[j] = A[smallest]
  A[smallest] = temp
```

c	t
c_1	$(n-1)+1 = n$
c_2	$n-1$
c_3	$\sum_{j=1}^{n-1} n-j+1$
c_4	$\sum_{j=1}^{n-1} (n-j+1)-1$
c_5	x
c_6	$n-1$

$j+1$ to n → $n-(j+1)+1+1$ (+1 last cond.)

65 / 85

$$T(n) = c_1 \cdot n + c_3(n-1) + c_3\left(\sum_{j=1}^{n-1} n-j+1\right) + c_4\left(\sum_{j=1}^{n-1} (n-j+1)-1\right) + x c_5 + (n-1)c_6$$

Best Case: $x=0$ (Already sorted)

Worst Case: $x = \sum_{j=1}^{n-1} (n-j+1)+1$

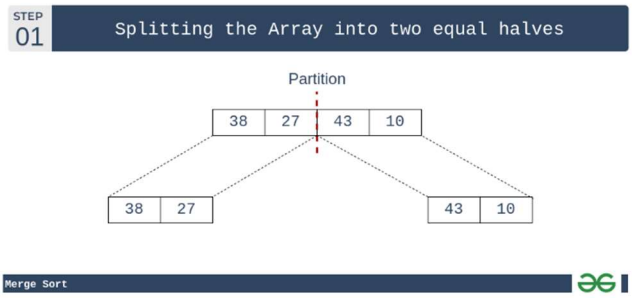
- **When to Use**: Selection Sort is suitable for small data sets and is generally more efficient than Bubble Sort.
- **Why**: It consistently performs the same number of swaps regardless of the initial order of the elements.

```
def selection_sort(lst):
    for i in range(len(lst)):
        min_index = i
        for j in range(i + 1, len(lst)):
            if lst[j] < lst[min_index]:
                min_index = j
        lst[i], lst[min_index] = lst[min_index], lst[i]
```

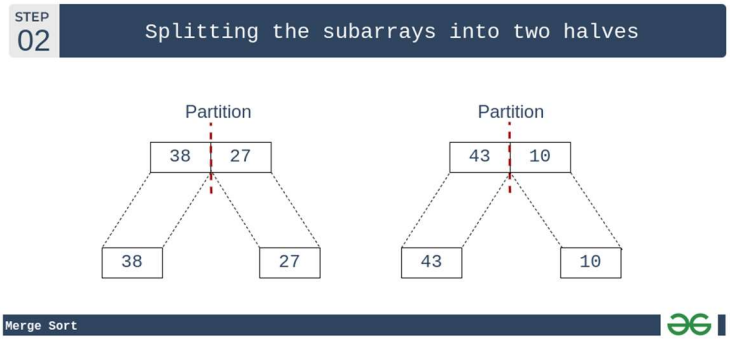
MERGE SORT

Lets consider an array arr[] = {38, 27, 43, 10}

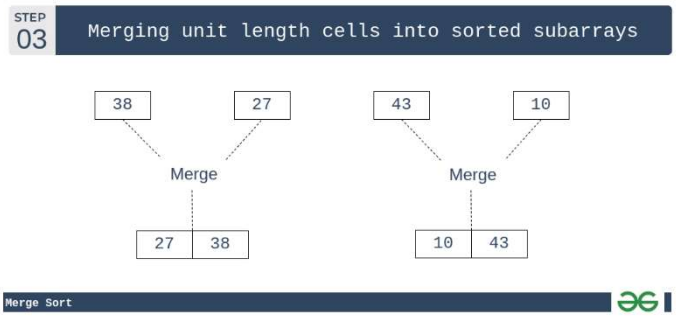
1) Initially divide the array into two equal halves:



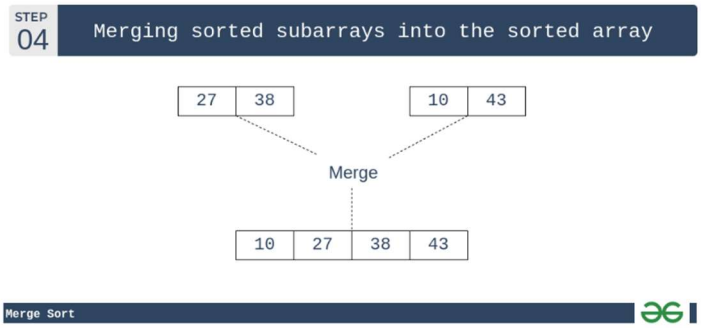
2) These subarrays are further divided into two halves. Now they become array of unit length that can no longer be divided and array of unit length are always sorted.



3) These sorted subarrays are merged together, and we get bigger sorted subarrays.

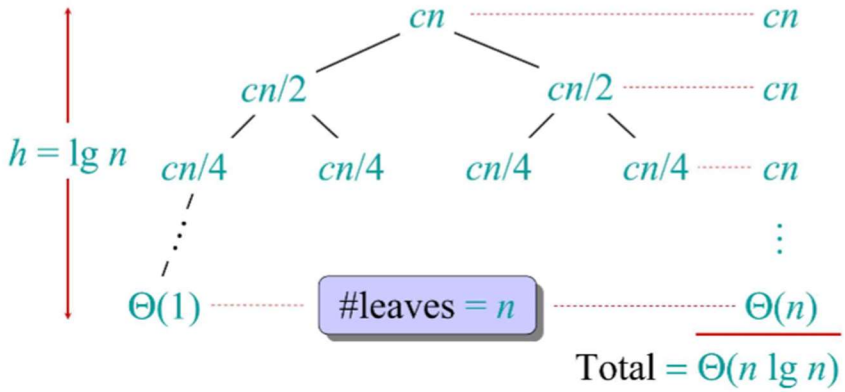


4) This merging process is continued until the sorted array is built from the smaller subarrays.



Solve the Recurrence

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$



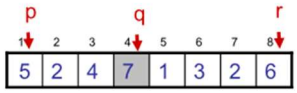

```python
def merge_sort(arr):
    if len(arr) > 1:
        # Divide the array into two halves
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        # Recursively sort each half
        merge_sort(left_half)
        merge_sort(right_half)

        # Merge the sorted halves
        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
```



Alg.: MERGE-SORT(A, p, r)

```
if p < r                          ▷ Check for base case
then q ← ⌊(p + r)/2⌋              ▷ Divide
     MERGE-SORT(A, p, q)          ▷ Conquer
     MERGE-SORT(A, q + 1, r)      ▷ Conquer
     MERGE(A, p, q, r)            ▷ Combine
```

- Initial call: MERGE-SORT(A, 1, n)

- **When to Use**: Merge Sort is efficient for large data sets and is a good choice for sorting linked lists.
- **Why**: It's a divide-and-conquer algorithm that guarantees a stable sort and has a consistent O(n log n) time complexity.

- Selection sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Running time: $\Theta(n^2)$

- Merge Sort
  - Design approach: divide and conquer
  - Sorts in place: No
  - Running time: $\Theta(n \log n)$

- Insertion sort
  - Design approach: incremental
  - Sorts in place: Yes
  - Best case: $\Theta(n)$
  - Worst case: $\Theta(n^2)$

BIG O Notation

Asymptotic Notation

- O notation: asymptotic “less than”:
  - f(n)=O(g(n)) implies: f(n) “≤” g(n)
- Ω notation: asymptotic “greater than”:
  - f(n)= Ω (g(n)) implies: f(n) “≥” g(n)
- Θ notation: asymptotic “equality”:
  - f(n)= Θ (g(n)) implies: f(n) “=” g(n)

Prove $\Theta(g(n)) \subset O(g(n))$

- Let x be any element in $\Theta(g(n))$
- Since $x \in \Theta(g(n))$, let's apply the definition of Θ
- There exist c_1, c_2, and n_0 such that

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0$$

- Now, let's use the constants for the definition of Θ to show that the definition for O is satisfied
- Let $c = c_2$. Then

$$0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0$$

- Since the inequality is satisfied, then $x \in O(g(n))$
- Therefore, by the definition of a subset, $\Theta(g(n)) \subset O(g(n))$

Problem Set 1 **CS 3851** **Name** Hudson Amey

Submission instructions: Your written answers should be submitted to Canvas as a PDF.

Assumption array is not empty

1. Write the equation T(n) modelling the run time in terms of the size n of the input array.

```
 public static int findMinimum(int[] input) {
1    int minimum = Integer.MAX_VALUE;
2    for(int i = 0; i < input.length; i++) {
3        if(input[i] < minimum) {
4            minimum = input[i];
5        }
6    }
7    return minimum;
 }
```

c	#t
c_1	1
c_2	1
c_3	$n+1$
c_4	n
c_5	n
c_6	x
c_7	1

$1+1+n(1+1+1+1)+1+1$ where $x=n$ worst

$= 4n+4$

$= 4(n+1)$

$T(n) = c_1 + c_2 + (n+1)c_3 + nc_4 + nc_5 + xc_6 + c_7$

$T(n) = O(n)$

Best $= (c_3+c_4+c_5)n + c_1+c_2+c_3+c_7$

$3n+4$

$x=1$

[max, ..., min]

2. Write the equation T(n) modelling the run time in terms of n.

```
 public static long fibonacci(int n) {
1     int a = 0;
2     int b = 1;
3     for(int i = 0; i < n; i++) {
4         int c = a + b;
5         a = b;
6         b = c;
7     }
8     return a;
 }
```

$1+1+1+n(1+1+1+1+1)+1+1$

$5n+5$

$5(n+1)$

$T(n) = O(n)$

3. Write the equation T(n, m) modelling the run time in terms of n and m.

```
1 for(int i = 0; i < n; i++) {
2       for(int j = 0; j < m; j++) {
3             int result = i * j;
4             System.out.println(i + " x " + j + " = " + result);
5       }
6 }
```

$1+n(1+m(1+1+1+1)+1)+1$

$2+n(4m+3)$

$4mn+3n+2$

$T(n)=O(n*m)$

4. Write the equation T(n) modelling the run time in terms of the size n of the input array.

```
 public boolean contains(int[] array, int key) {
1       boolean found = false;
2       for(int i = 0; i < array.length; i++) {
3             if(array[i] == key) {
4                   found = true;
5             }
6       }
7       return found;
 }
```

$1+1+n(1+1+1+1)+1+1$

$4n+4$

$4(n+1)$

$O(n)=T(n)$

5. Write the equation T(n) modelling the run time in terms of the size n of the linked list.

```
public boolean contains(Node<Integer> head, Integer key) {
1     boolean found = false;
2     Node<Integer> current = head;
3     while(current != null) {
4           if(current.datum.equals(key)) {
5                 found = true;
6           }
7           current = current.next;
8     }
9     return found;
}
```

$1+1+n(1+1+1+1)+1+1$

$4n+4$

$4(n+1)$

$O(n) = T(n)$

6. Write the equation T(n) modelling the run time in terms of the index n.

```
public String get(String[] array, int index) {
1     if(index >= array.length || index < 0) {
2           return null;
3     }
4     return array[index];
}
```

Worst: $1+1+1 = 3 = O(3)$

$= O(1) = T(1)$

7. Write the equation T(n) modelling the run time in terms of the size n of the linked list.

```
 public String get(Node<String> head, int index) {
1      int currentIndex = 0;
2      Node<Integer> current = head;
3      while(current != null && currentIndex <= index) {
4            if(currentIndex == index) {
5                  return current.datum;
6            }
7            current = current.next;
8            currentIndex++;
9      }
8      return null;
 }
```

$1+1+n(1+1+1+1+1)+1+1$

$6n+4$

$2(3n+2)$

$O(n) = T(n)$ → If LL is not empty

Multiple returns, $O(1)$ if LL is empty

Problem Set 2 **Name:** Hudson Arney

Submission instructions: Your written answers should be submitted to Canvas as a PDF.

1. For each expression below, give two valid examples of big-o notation for that expression. One example should be consistent with convention (e.g., a tight bound) and one should violate convention (e.g., an unnecessarily loose bound).

	Tight Bounds	Unnecessarily Loose Bound
a) $64n^2 + 3n - 2$	$O(n^2)$	$O(n^3)$
b) $6n + 2$	$O(n)$	$O(n^2)$
c) $n + \log n + 2$	$O(n)$	$O(n^2)$

2. Illustrate the steps of insertion sort on the following lists. You should show the key, which elements are shifted, and the list after the insertion.

a) [5, 4, 3, 2, 1]

Step 1: k=4, [4, 5, 3, 2, 1]

Step 2: k=3, [3, 4, 5, 2, 1]

Step 3: k=2, [2, 3, 4, 5, 1]

Step 4: k=1, [1, 2, 3, 4, 5]

b) [5, 2, 3, 4, 8, 1, 2]

Step 1: k=2, [2, 5, 3, 4, 8, 1, 2]

Step 2: k=3, [2, 3, 5, 4, 8, 1, 2]

Step 3: k=4, [2, 3, 4, 5, 8, 1, 2]

Step 4: k=8, [2, 3, 4, 5, 8, 1, 2]

Step 5: k=1, [1, 2, 3, 4, 5, 8, 2]

Step 6: k=2, [1, 2, 2, 3, 4, 5, 8]

3. For each function below, answer the following questions.

```
public static int findMinimum(int[] input) {
     int minimum = Integer.MAX_VALUE;
     for(int i = 0; i < input.length; i++) {
          if(input[i] < minimum) {
               minimum = input[i];
          }
     }
     return minimum;
}
```

a) Trace the code for following input lists: [1, 2, 3, 4, 5], [4, 2, 3, 8, 1], [5, 4, 3, 2, 1]. Write the values of **`minimum`** and **`i`** at each step.

b) Describe the general idea behind the function in words.

c) What loop invariant is maintained?

d) Write the equation for the number of instructions executed in terms of the array size using exact analysis.

e) Express your answer from (d) in big-o notation.

a) [1,2,3,4,5]
1: i=0, min=1
2: i=1, min=1
3: i=2, min=1
4: i=3, min=1
5: i=4, min=1

[4,2,3,8,1]
1: i=0, min=4
2: i=1, min=2
3: i=2, min=2
4: i=3, min=2
5: i=4, min=1

[5,4,3,2,1]
1: i=0, min=5
2: i=1, min=4
3: i=2, min=3
4: i=3, min=2
5: i=4, min=1

b) Look through all elements in the array to achieve the minimum

c) Minimum always holds the minimum value in the array from index 0 to i during the iteration

d) $C_1 + C_2 + n(C_3 + C_4 + C_5 + C_6) + C_7 = 4n+3$ ⇒ worst case

e) O(n), the number of instructions executed because it grows with the size of the input array.

4.

```
public static int slowMaximumIndex(int[] input) {
    int maximum = -1;
    for(int i = 0; i < input.length; i++) {
        boolean larger = true;
        for(int j = 0; j < input.length; j++) {
            if(input[i] < input[j]) {
                larger = false;
            }
        }
        if(larger) {
            maximum = i;
            break;
        }
    }
    return maximum;
}
```

a) Trace the code for following input lists: [1, 2, 3, 4, 5], [4, 2, 3, 8, 1], [5, 4, 3, 2, 1]. Write the values of **maximum**, **i, and larger** after each iteration of the outer loop.

b) Describe the general idea behind the function in words.

c) What loop invariant is maintained?

d) Write the equation for the number of instructions executed in terms of the array size using exact analysis.

e) What is the run time in the best case? What is the run time in the worst case?

f) Express your answers from (e) in big-o notation.

a) [1,2,3,4,5] 1: i=0, max=0, lar=false 2: i=1, max=0, lar=false 3: i=2, max=0, lar=false 4: i=3, max=0, lar=false 5: i=4, max=4, lar=true

[4,2,3,8,1] 1: i=0, max=0, lar=false 2: i=1, max=0, lar=false 3: i=2, max=0, lar=false 4: i=3, max=0, lar=false 5: i=4, max=4, lar=true

[5,4,3,2,1] 1: i=0, max=0, lar=false 2: i=1, max=0, lar=false 3: i=2, max=0, lar=false 4: i=3, max=0, lar=false 5: i=4, max=0, lar=false

b) Find the index of the max element by using nested for loops

c) "maximum" contains the index of the max element

d) Worst: $4n^2+7n+4$

e) Best: larger is true in the first iteration
Worst: larger is true in the last iteration

f) Worst: $O(n^2)$, Best: $O(n)$

5.

```
public static void partitionBySign(int[] input) {
    int i = 0;
    for(int j = 0; j < input.length; j++) {
        if(input[j] < 0) {
            int tmp = input[i];
            input[i] = input[j];
            input[j] = tmp;
            i++;
        }
        System.out.println(i + " " + j);
        System.out.println(Arrays.toString(input));
        System.out.println();
    }
}
```

Worst case is if input[i] < 0, $O(n^2)$

Best case is $O(n^2)$, if input[i] ≥ 0 ∀i ⇒ x = 0 ⇒ $T(n) \in O(n^2)$

a) Trace the code for following input lists: [-1, -2, 0, 4, 5], [4, -2, 0, 8, -1], [5, 4, 0, -2, -1]. Write the values of **i**, **j**, and **input** at each step.

b) Describe the general idea behind the function in words.

c) What loop invariant is maintained?

d) Write the equation for the number of instructions executed in terms of the array size using exact analysis.

e) Express your answer from (d) in big-o notation.

a) [-1,-2,0,4,5] 1) j=1, i=0, input=[-2,-1,0,4,5] → 3) j=2, i=1, input=[-2,-1,0,4,5]

1) j=0, i=0, input=[-1,-2,0,4,5]

4) j=3, i=1, input=[-2,-1,0,4,5] → 5) j=4, i=2, input=[-2,-1,0,4,5]

[4,-2,0,8,-1]

1) j=0, i=0, input=[4,-2,0,8,-1] → 2) j=1, i=0, input=[-2,4,0,8,-1] → 3) j=2, i=1, input=[-2,4,0,8,-1]

4) j=3, i=1, input=[-2,4,0,8,-1] → 5) j=4, i=2, input=[-2,4,-1,8,0]

[5,4,0,-2,-1]

1) i=0, j=0, input=[5,4,0,-2,-1] → 2) j=1, i=0, input=[5,4,0,-2,-1] → 3) j=2, i=0, input=[5,4,0,-2,-1]

4) i=0, j=3, input=[-2,4,0,5,-1] → 5) i=1, j=4, input=[-2,-1,0,5,4]

b) To partition the input array into a negative numbers part and non-negative numbers part

c) all elements from 0 to i-1 are negative or zero, all from i to j-1 are not

d) 10n+2

e) O(n)

6.

```
SELECTION-SORT(A)
n = A.length
for j=1 to n - 1
    smallest = j
    for i in j + 1 to n
        if A[i] < A[smallest]:
            smallest = i
    tmp = A[j]
    A[j] = A[smallest]
    A[smallest] = tmp
```

$1+1+n(1+n(1+1+1+1+1)+1+1+1+1)+1$

$3+6n+5n^2$

a) Trace the code for following input lists: [1, 2, 3, 4, 5], [4, 2, 3, 8, 1], [5, 4, 3, 2, 1]. Write the values of **j** and **A** after each iteration of the outer loop.

b) Describe the general idea behind the function in words.

c) What loop invariant is maintained?

d) Write the equation for the number of instructions executed in terms of the array size using exact analysis.

e) Express your answer from (d) in big-o notation.

f) Will selection sort execute fewer instructions on a sorted list than a reverse sorted list? How does this compare with insertion sort?

7. Identify and describe real-world problems that can be solved using the insertion sort algorithm.

Insertion sort can be used for sorting names in a database, if new names get added they can be automatically sorted.

6) [1,2,3,4,5] ↦ 1) j=1, A=[1,2,3,4,5] → 2) j=2, A=[1,2,3,4,5] → 3) j=3, A=[1,2,3,4,5]
→ 4) j=4, A=[1,2,3,4,5]

[4,2,3,8,1] → 1) j=1, A=[1,2,3,8,4] 2) j=2, A=[1,2,3,8,4] 3) j=3, A=[1,2,3,8,4]
4) j=4, A=[1,2,3,8,4]

[5,4,3,2,1] ↦ 1) j=1, A=[1,4,3,2,5] → 2) j=2, A=[1,2,3,4,5] → 3) j=3, A=[1,2,3,4,5]
→ 4) j=4, A=[1,2,3,4,5]

b) It finds the smallest element in the unsorted portion of the array and swaps it with the element at the begining of the unsorted portion.

c) A[0:j] is sorted, all elements in A[j+1:n] are greater than A[0:j]

d) $5n^2+6n+3$

e) $O(n^2)$

f) The same # of instructions on a sorted list, which is not as efficient as insertion sort

Problem Set 3 **CS 3851** **Name** Hudson Arney

Submission instructions: Your written answers should be submitted to Canvas as a PDF.

1. Describe in your own words the difference between $\Theta(\cdot)$, $O(\cdot)$, and $\Omega(\cdot)$.

Big Theta = Represents the tight bound of a function. It is used to describe an upper and lower bound that grow at the same rate as the function.

Big O = Represents an upper bound of a function. It is the upper limit of a functions behavior. Can have best and worst case scenarios.

Big Omega = Signifies the lower bound of a function. It represents the lower limit on a functions behavior.

2. Prove the following:

a. $n + 3 \in \Theta(n)$

$n+3$ is an upper bound for $n \leq n+3$ for all $n \geq 0$

$n+3$ is a lower bound for $n+3 \geq c*n$ for $c>0$ and $n \geq 0$

$c = \frac{1}{4} \Rightarrow n+3 \geq \frac{1}{4} * n$

$n+3 \geq \frac{1}{4} n + n$

$n+3 \geq n$ Therefore $n+3 \in \Theta(n)$

b. $n \notin \Theta(n \log n)$

Prove $c_1 * n \log n \leq n \leq c_2 * n \log n$ for $n \geq n_0$

$c_1 \log n \leq 1 \leq c_2 \log n \Rightarrow c_1 \leq \frac{1}{\log n} \leq c_2$

This cannot exist because there are no constants c_1 or c_2 that complete this $\lim_{n \to \infty} c_1 \leq \alpha \leq c_2$

Therefore $n \notin (n \log n)$

c. $n^2 + n - 25 \in \Theta(n^2)$

$0 \le c_1 n^2 \le n^2 + n - 25 \le c_2 n^2$

$0 \le c_1 \le 1 + \frac{1}{n} - \frac{25}{n^2} \le c_2$

$\lim_{n \to \infty} [0 \le c_1 \le 1 \le c_2]$

There are constants such that c_1 is less than 1 but more than 0, and an infinite # of values greater than 1 for c_2 that satisfy $n^2 + n - 25 \in \Theta(n^2)$

d. $n^3 - n + 5 \notin \Theta(n)$

$0 \le c_1 n \le n^3 - n + 5 \le c_2 n$

$0 \le c_1 \le n^2 - 1 + \frac{5}{n} \le c_2$

$\lim_{n \to \infty} [0 \le c_1 \le \infty \le c_2]$

c_2 does not have a valid constant greater than ∞ therefore, $n^3 - n + 5 \notin \Theta(n)$

e. $n + 3 \in O(n)$

$0 \le n + 3 \le cn$

$0 \le 1 + \frac{3}{n} \le c$

$\lim_{n \to \infty} \left[1 + \frac{3}{n}\right] = 0 \le 1 \le c$

Any number over 1 is a constant to satisfy, $n + 3 \in O(n)$

f(n) g(n)

f. $n^2 \notin O(n \log n)$

$0 \leq (c_1 \log n) \leq c_1 n^2$

$0 \leq \log n \leq n * c_1$

There is no value for c_1 that can satisfy this set therefor $n^2 \notin O(n \log n)$

g. $n^2 + n - 25 \in \Omega(n)$

$0 \leq c * n \leq n^2 + n - 25$

$0 \leq c \leq n + 1 - \frac{25}{n}$

$\lim_{n \to \infty} \left[0 \leq c \leq n + 1 \right]$ There is an infinite # of constants for c to fullfill this statement; therefore, $n^2 + n - 25 \in \Omega(n)$

h. $n^3 - n + 5 \notin \Omega(n^4)$

$0 \leq cn^4 \leq n^3 - n + 5$

$0 \leq c \leq \frac{1}{n} - \frac{1}{n^3} + \frac{5}{n^4}$

$\lim_{n \to \infty} \left[0 \leq c \leq \frac{1}{n} - \frac{1}{n^3} + \frac{5}{n^4} \right]$

$0 \leq c \leq 0$, there is no such value for c to exist in this problem; therefore, $n^3 - n + 5 \notin \Omega(n^4)$

3. Give an example of a function that is in $O(n^2)$ but not in $\Theta(n^2)$ and then prove it.

$n \in O(n^2)$

$0 \leq n \leq cn^2$

$0 \leq 1 \leq cn$

$\lim_{n\to\infty}[0 \leq 1 \leq c\cdot\infty]$

c can be any constant that is non-negative and complete this

$n \notin \Theta(n^2)$

$0 \leq c_1 n^2 \leq n \leq c_2 n^2$

$0 \leq c_1 n \leq 1 \leq c_2 n$

$\lim_{n\to\infty}[0 \leq \infty \leq 1 \leq \infty] \Rightarrow$ Not possible

$f(n) = n \in O(n^2)$ but $\notin \Theta(n^2)$

4. Give an example of a function that is in $\Omega(n^2)$ but not in $\Theta(n^2)$ and then prove it.

$f(n) = n$

$\Omega(n^2)$

$0 \leq c \cdot n \leq n^2$

$\lim_{n\to\infty} 0 \leq c \leq n$

$0 \leq c \leq \infty$

If c is positive this is true

$f(n) = n \in \Omega(n^2)$ but $\notin \Theta(n^2)$

$\Theta(n^2)$

$0 \leq c_1 n \leq n \leq c_2 n^2$

$0 \leq c_1 n \leq 1 \leq c_2 n$

$\lim_{n\to\infty}[0 \leq \infty \leq 1 \leq \infty] \Rightarrow$ Not possible

5. Prove that $\Theta(g(n)) \subset \Omega(g(n))$.

If there exists some element $x \in \Theta(g(n))$ it would satisfy

$0 \leq c_1 g(n) \leq x \leq c_2 g(n)$ where $c_1, c_2,$ and $n_0 \geq 0$

Thus x will also satisfy

$0 \leq c\, g(n) \leq x$

Therefore $\Theta(g(n))$ is a subset of $\Omega(g(n))$

Definitions

$\Theta(g(n)) = \{ f(n)$: there exists positive constants c_1, c_2, and n_0 such that

$0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for all $n \geq n_0 \}$

$O(g(n)) = \{ f(n)$: there exists positive constants c and n_0 such that

$0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0 \}$

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that

$0 \leq cg(n) \leq f(n)$ for all $n \geq n_0 \}$

Problem Set 4 **CS 3851** **Name** Hudson Arney

Submission instructions: Your answers should be submitted to Canvas as a PDF.

1. Practice the "guess and check" substitution method by solving the following recurrence relation:

$T(n) - O(n) = T(n/2)$ $T(n) \le cO(n)$

T(n) = 2T(n/2) + O(n)

a. Check if $T(n) \in O(n^3)$ a) $T(n) \le cn^3$

$2c(n/2)^3 + O(n) \le cn^3$

$(n^3/4)c + O(n) \le cn^3$

$O(n) \le \frac{3cn^3}{4}$

This holds true

b. Check if $T(n) \in O(n^2)$ b) $T(n) \le cn^2$

$2c(n/2)^2 + O(n) \le cn^2$

$\frac{cn^2}{2} + O(n) \le cn^2$

$O(n) \le \frac{cn^2}{2}$

This holds true

c. Check if $T(n) \in O(n)$

c) $2c(n/2) + O(n) \le cn$

$2cn + O(n) \le cn$

$O(n) \le -cn$ This is not possible

d. Based on your results, what is the tightest upper bound you found for T(n)?

d) $T(n) \in O(n^2)$ is the tightest upper bound from the sub. method

a) $T(n) = 2T(\frac{n}{2}) + O(n)$

$\le 2 \cdot c \cdot \frac{n^3}{8} + c_1 \cdot n$

$\frac{cn^3}{4} + c_1 n \le cn^3/n^3$

$\frac{c}{4} + c_1 \cdot \frac{1}{n^2} \le c$

$\frac{c_1}{n^2} \le \frac{3}{4}c$

Problem Set 4 **CS 3851** **Name** Hudson Arney

Submission instructions: Your answers should be submitted to Canvas as a PDF.

1. Practice the "guess and check" substitution method by solving the following recurrence relation:

$T(n) = 2T(n/2) + O(n)$

a. Check if $T(n) \in O(n^3)$

b. Check if $T(n) \in O(n^2)$

c. Check if $T(n) \in O(n)$

d. Based on your results, what is the tightest upper bound you found for T(n)?

Induction Goal: $T(k) \in O(k^3)$; $k < n$

$T(n/2) \le \left(\frac{n}{2}\right)^3$

$T(n/2) \le c \cdot \frac{n^3}{8}$

Sub into eq:

$T(n) \le 2 \cdot c \cdot \frac{n^3}{8} + \underbrace{c_0 n}_{O(n) \text{ term}}$

$2c \frac{n^3}{8} + c_0 n \le cn^3$ for all $n \ge n_0$

$\frac{1}{4} c + c_0 \cdot \frac{1}{n^2} \le c$

$c_0 \frac{1}{n^2} \le \frac{3}{4} c$

$\frac{4}{3} \cdot c_0 \cdot \frac{1}{n^2} \le c$

let $n_0 = 2$:

$c \ge \frac{4}{3} c_0 \cdot \frac{1}{2^2} = \frac{c_0}{3}$

For any $c \ge \frac{c_0}{3}$ where c_0 is the bounding constant for the $O(n)$ term, the induction goal holds for $n_0 = 2$

IG: $T(n) \le cn^2$

IH: $T(\frac{n}{2}) \in O(n^2)$

b) $T(n) = 2T(\frac{n}{2}) + O(n)$

$\le 2(c(\frac{n}{2})^2) + c_1 n \rightarrow \text{plug } T(\frac{n}{2}) \text{ for } T(n)$

$= \frac{2(c\frac{n^2}{4}) + c_1 n}{n^2} \le \frac{cn^2}{n^2}$

$= \frac{c_1}{n} \le \frac{c}{2} \Rightarrow \frac{1}{4} \le \frac{1}{2}$

$n_0 = 4$

$c = 1$

$c_0 = 1$

$T(n) \in O(n^2)$

c) $k = \frac{n}{2}$, Induction Hypothesis: $T(\frac{n}{2}) \le c(\frac{n}{2})$ for all $k < n$

$cn + c_1 n \le cn$

$c_1 n \le 0$

but $c_1 > 0$ ∴ breaks rule and $T(n) \notin O(n)$

2. For each of the following recurrence relations, determine which of the three cases of the Master's theorem applies.

a. $T(n) = 2T(n/2) + \Theta(n)$

b. $T(n) = 8T(n/2) + \Theta(n^2)$

c. $T(n) = 3T(n/4) + n \log n$

a) $\Theta(n) \equiv n^{\log_2 2} = n$ = Case 2 of master theorem

↑ $f(n) = O(n)$, $T(n) = \Theta(n \log n)$

b) $\Theta(n^2) \equiv n^{\log_2 8} = n^3 \rightarrow n^3$ dominates $f(n) = \Theta(n^2)$ therefore it is Case 1 of the master theorem

↳ $\Theta(n^2) \in O(n^{3-1}) \Rightarrow \Theta(n^2) \in O(n^2) \Rightarrow T(n) = \Theta(n^3)$

c) $n \log n \equiv n^{\log_4 3}$

$\log_4 3 \log(n)$

$f(n) = n \log n$ dominates $\log_4 3 \log(n)$, therefore it is case 3 of the master theorem

3. For each of the following recurrence relations: (1) find a reasonably tight bound in terms of big-o using the substitution method and (2) check your solution by also solving the recurrence relation with the Master method. Show all work.

a) T(n) = 2T(n/2) + n

b) T(n) = 4T(n/4) + 1

c) T (n) = 9T(n/3) + n

d) T(n) = 3T(n/4) + n log n

4. The pseudocode for binary search is given below. The function takes an array A, a query item key, a starting index i, and an ending index j. (Assume that it is initially called with i = 1 and j = A.length.)

```
BINARY-SEARCH(A, key, i, j)
    if i > j
        return NIL
    mid=(i+j)/2
    if A[mid] == key
        return mid
    elif key < A[mid]
        return BINARY-SEARCH(A, key, i, mid - 1)
    elif key > A[mid]
        return BINARY-SEARCH(A, key, mid + 1, j)
```

Base case O(1)

Divide O(1)

Conquer O(log n)

a) Specify which steps correspond to the divide and conquer steps in the divide and conquer framework. Note that binary search does not have a combine step.

b) Give the run time for each step in the divide and conquer framework.

c) Write the recurrence relation. $T(n) = T(n/2) + O(1)$ (a, b, $f(n)$)

d) Solve the recurrence relation.

e) Why can we describe the run time of binary search using O notation but not Θ notation?

Binary Search can be described using O notation because it provides an upper bound on the runtime of the algorithm. BS doesn't provide a tight lower bound, which Θ needs.

d) $k \leq \log_2(n)$ $\quad 1 \equiv n^{\log_2 1}$ using the M.M., Binary Search falls under Case 2.

$a=1, b=2, f(n)=1$

$1 = n^0$

$1=1$ ✓

Definitions

$\Theta(g(n)) = \{ f(n)$: there exists positive constants c_1, c_2, and n_0 such that

$0 \le c_1\, g(n) \le f(n) \le c_2\, g(n)$ for all $n \ge n_0 \}$

$O(g(n)) = \{ f(n)$: there exists positive constants c and n_0 such that

$0 \le f(n) \le c\, g(n)$ for all $n \ge n_0 \}$

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that

$0 \le cg(n) \le f(n)$ for all $n \ge n_0 \}$

Master theorem: Let $a \le 1$ and $b > 1$ be constants, let f(n) be an asymptotically positive function and let T(n) be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n)$$

Then T(n) has the following asymptotic bounds:

If $n^{\log_b a}$ dominates $f(n)$

1. If $f(n) \in O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) \in \Theta(n^{\log_b a})$.

2. If $f(n) \in \Theta(n^{\log_b a})$, then $T(n) \in \Theta(n^{\log_b a} \log n)$.

3. If $f(n) \in \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and if $a\, f(n/b) \le c\, f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) \in \Theta(f(n))$.

$f(n)$ dominates $n^{\log_b a}$