Due: 15th October 2024 CS 2009: Design and Analysis of Algorithms

20% penalty for 1 day late Assignment 2

40% penalty for 2 days late Total Marks: 100 points

Submission not allowed afterwards

**Question 1: 10 points**

HeapSort is a sorting technique that takes O(nlog⁡n) time. Watch any video to understand how HeapSort works.

You are given an unsorted array A=[4,1,3,2,16,9,10,14,8,7]

1. Apply the **HeapSort** algorithm to sort the array.
2. Display the array at each major iteration of the heap-building and sorting process (both during the heap construction and after each element is extracted from the heap).

**Solution:**

Let's break down the HeapSort algorithm step by step:

1. Heap Construction (Max-Heap Building):

Convert the array into a max-heap by heapifying from the bottom up.

1. Sorting:
   * Repeatedly extract the maximum element from the heap and place it at the end of the array, reducing the heap size at each iteration.

**Initial Array:**

A=[4,1,3,2,16,9,10,14,8,7]

**Step 1: Building Max-Heap**

We start heapifying the array from the middle, moving upwards (from the last non-leaf node). For an array, the parent of a node at index i is at (i-1)/2

1st Iteration (Start heapifying from index 4):

* Sub-array: A[4]=16, left child = A[9]=7, right child = None.
* No action is needed as 16 is larger than its child.

Array remains:  
 A=[4,1,3,2,16,9,10,14,8,7]

2nd Iteration (Heapifying index 3):

* Sub-array: A[3]=2, left child = A[7]=14, right child = A[8]=8.
* Largest is 14, so swap A[3] and A[7].

Array after swapping:  
 A=[4,1,3,14,16,9,10,2,8,7]

3rd Iteration (Heapifying index 2):

* Sub-array: A[2]=3, left child = A[5]=9, right child = A[6]=10.
* Largest is 10, so swap A[2] and A[6].

Array after swapping:  
 A=[4,1,10,14,16,9,3,2,8,7]

4th Iteration (Heapifying index 1):

* Sub-array: A[1]=1, left child = A[3]=14, right child = A[4]=16.
* Largest is 16, so swap A[1] and A[4].

Array after swapping:  
 A=[4,16,10,14,1,9,3,2,8,7]

* Now, heapify at index 4:
  + A[4]=1, left child = A[9]=7, right child = None.
  + Largest is 7, so swap A[4] and A[9].

Array after swapping:  
 A=[4,16,10,14,7,9,3,2,8,1]

5th Iteration (Heapifying index 0):

* Sub-array: A[0]=4, left child = A[1]=16, right child = A[2]=10.
* Largest is 16, so swap A[0] and A[1].

Array after swapping:  
 A=[16,4,10,14,7,9,3,2,8,1]

* Now, heapify at index 1:
  + A[1]=4, left child = A[3]=14, right child = A[4]=7.
  + Largest is 14, so swap A[1] and A[3]

Array after swapping:  
 A=[16,14,10,4,7,9,3,2,8,1]

* Now, heapify at index 3:
  + A[3]=4, left child = A[7]=2, right child = A[8]=8.
  + Largest is 8, so swap A[3] and A[8]

Array after swapping:  
 A=[16,14,10,8,7,9,3,2,4,1]

**Step 2: Sorting**

We now repeatedly extract the largest element (root of the heap) and place it at the end of the array.

1st Extraction (Swap root with last element):

* Swap A[0]=16 and A[9]=1.
* Heap size reduces by 1.

Array after swapping:  
 A=[1,14,10,8,7,9,3,2,4,16]

* Heapify root at index 0:
  + A[0]=1, left child = A[1]=14, right child = A[2]=10.
  + Largest is 14, so swap A[0] and A[1]

Array after swapping:  
 A=[14,1,10,8,7,9,3,2,4,16]

* Heapify index 1:
  + A[1]=1, left child = A[3]=8, right child = A[4]=7.
  + Largest is 8, so swap A[1] and A[3]

Array after swapping:  
 A=[14,8,10,1,7,9,3,2,4,16]

* Heapify index 3:
  + A[3]=1, left child = A[7]=2, right child = A[8]=4.
  + Largest is 4, so swap A[3] and A[8]

Array after swapping:  
 A=[14,8,10,4,7,9,3,2,1,16]

2nd Extraction (Swap root with second last element):

* Swap A[0]=14 and A[8]=1.

Array after swapping:  
 A=[1,8,10,4,7,9,3,2,14,16]

* Follow similar heapifying steps as earlier until the array is fully sorted.

By continuing this process, we eventually get the final sorted array:

Final Sorted Array:

A=[1,2,3,4,7,8,9,10,14,16]

**Question 2: 20 points**

(a) You've learned the binary search algorithm, which is used to search for an element in a sorted array. Another efficient search algorithm is Jump Search, which optimizes search time by skipping multiple elements at once. Learn about the Jump Search algorithm from this [link](https://www.geeksforgeeks.org/jump-search/).

* Using any sorted array of size at least 8, perform a search for a number located at the end of the array using both Binary Search and Jump Search.
* Provide a detailed step-by-step breakdown of each iteration of both algorithms, showcasing the decision-making process at each step.
* After completing both searches, compare their efficiency in terms of time complexity and space complexity.
* Discuss in-depth the advantages and disadvantages of both techniques, including when one might outperform the other based on different scenarios such as array size, structure, and the position of the target element.

(b) Interpolation Search and Exponential Search are two additional advanced search algorithms. Study the concepts of both from this [link](https://www.geeksforgeeks.org/searching-algorithms/) and then:

1. Explain the inner workings of both Interpolation Search and Exponential Search in your own words. Be sure to highlight how they differ from traditional search techniques like Binary Search.
2. Provide a real-world scenario where each algorithm might excel, comparing their performance based on factors like array size, distribution of data, and search element location.
3. For each search algorithm (Interpolation and Exponential), discuss its best-case, average-case, and worst-case time complexity. Include insights into how data distribution (e.g., uniformly distributed or not) can affect their performance.

**Solution:**

### **a) Binary Search vs Jump Search**

We will take the sorted array:  
arr = [3, 8, 15, 17, 23, 35, 42, 56]  
Let the number to search for be 56, which is the last element in the array.

#### **1. Binary Search**

Binary Search follows a divide-and-conquer approach by repeatedly halving the search space.

* Initial array: [3, 8, 15, 17, 23, 35, 42, 56]  
  Target: 56

**Iterations:**

1. **First iteration:**
   * Left index = 0, Right index = 7 (size - 1)
   * Mid index = (0 + 7) // 2 = 3
   * Compare arr[3] = 17 with 56. Since 56 > 17, narrow down the search to the right half: arr[4] to arr[7] = [23, 35, 42, 56].
2. **Second iteration:**
   * Left index = 4, Right index = 7
   * Mid index = (4 + 7) // 2 = 5
   * Compare arr[5] = 35 with 56. Since 56 > 35, narrow down the search to the right half: arr[6] to arr[7] = [42, 56].
3. **Third iteration:**
   * Left index = 6, Right index = 7
   * Mid index = (6 + 7) // 2 = 6
   * Compare arr[6] = 42 with 56. Since 56 > 42, narrow down the search to the last element arr[7] = 56.
4. **Fourth iteration:**
   * Left index = 7, Right index = 7
   * Mid index = (7 + 7) // 2 = 7
   * Compare arr[7] = 56 with 56. The element is found at index 7.

**Time complexity:**

* Best case: O(1) (when the element is found at the middle in the first iteration)
* Average/Worst case: O(log n), where n is the size of the array.

#### **2. Jump Search**

Jump Search involves jumping ahead by fixed steps (block size) and then performing a linear search within the identified block.

Let’s take the block size as sqrt(n) = sqrt(8) ≈ 2.

* Initial array: [3, 8, 15, 17, 23, 35, 42, 56]  
  Target: 56

**Iterations:**

1. **First jump:**
   * Jump 2 steps: Compare arr[1] = 8 with 56. Since 8 < 56, continue jumping.
2. **Second jump:**
   * Jump to arr[3] = 17. Since 17 < 56, continue jumping.
3. **Third jump:**
   * Jump to arr[5] = 35. Since 35 < 56, continue jumping.
4. **Fourth jump:**
   * Jump to arr[7] = 56. The element is found at index 7.

**Time complexity:**

* Best case: O(1) (if the element is the first in the array)
* Average/Worst case: O(sqrt(n)), where n is the size of the array.

### **Comparison: Pros and Cons**

#### **Binary Search:**

* Pros:
  + Efficient with a time complexity of O(log n), especially for large arrays.
  + Works well on sorted arrays, regardless of size.
* Cons:
  + Requires the array to be sorted.
  + Performance degrades on small datasets compared to simpler algorithms (e.g., linear search).
  + May involve more memory due to recursion (depending on the implementation).

#### **Jump Search:**

* Pros:
  + Performs better than linear search when the array is large but relatively small steps (sqrt(n)) suffice.
  + Efficient for uniformly distributed sorted arrays.
* Cons:
  + Works only on sorted arrays.
  + Not as efficient as binary search in larger datasets (O(sqrt(n)) vs O(log n)).
  + The block size (sqrt(n)) is crucial; an incorrect block size can lead to inefficiency.

### **(b) Interpolation Search and Exponential Search**

#### **1. Interpolation Search**

**Working**:  
Interpolation Search improves over binary search by estimating the position of the target value, assuming the values are uniformly distributed. Instead of halving the array like Binary Search, it computes the probable position using the following formula:

pos=low+((target-arr[low]) / arr[high]-arr[low]) x (high-low)

* If the guessed position contains the target element, the search is complete.
* If the target is greater, search in the right portion.
* If the target is smaller, search in the left portion.

**Example Scenario**:  
This search works best when the data is **uniformly distributed**, such as searching for a specific price in a database where prices increase in regular increments.

**Time Complexity**:

* Best case: O(1) (if the guessed position is correct).
* Average case: O(log log n).
* Worst case: O(n) (if the array is not uniformly distributed).

#### **2. Exponential Search**

**Working**:  
Exponential Search is useful for searching in an unbounded or infinite array. It finds the range where the target may be located by growing the search space exponentially. Once a range is found, binary search is applied to find the exact element.

Steps:

* Start by checking the first element.
* Double the index in each iteration until the element at that index is greater than or equal to the target.
* Once a suitable range is found, apply Binary Search within that range.

**Example Scenario**:  
This search is ideal when dealing with **infinite or large unbounded datasets**, such as searching in a large file stream or unknown-length arrays.

**Time Complexity**:

* Best case: O(1) (if the element is found immediately).
* Average/Worst case: O(log n) due to binary search being applied after determining the range.

**Question 3: 20 points**

Find the values of the following variables, using your student ID. The values of these variables will be used, for solving the below problem.

a = first two digits of your student ID

b = last two digits of your student ID

c = (multiply your first two digits by 7) mod 60

d = (your complete four digits student ID) mod 30

e = positive ( (last two digits of your student ID) - (first two digits of your student ID) )

For example, if the student ID is 21k-4531, then the value of the variables will be calculated as follows:

a = 45

b = 31

c = (45\*7) mod 50 = 335 mod 60 = 35

d = 4531 mod 30 = 11

e = positive (31 – 45) = positive (-14) = 14

**Problem to solve**

A Rickshaw driver moves from one road to another to find customers. If a driver does not find any customer on some road, then only fuel is consumed and thus has a loss equal to price of fuel, represented by negative amount (in figure below). If the driver, finds some customer, then some profit is earned. The rickshaw driver made a route map (as shown below), where he used to travel, and also note down the average benefit earned for the specific roads. The rickshaw driver is now eager to know, what is the highest profit he can get while traversing any contiguous subset of route.

45

-11

35

-14

-31

45

-14

31

start

end

a

-d

c

-e

-b

a

-e

b

start

end

(a) (b)

Figure 1. The Route map

1. Redraw the map in Figure 1(a) using the values, calculated using your student ID

(Just for understanding, map is redrawn for the example student ID in the Figure 1(b))

1. Using the maximum sub-array algorithm, dry run on your redrawn graph (that is one, having your own calculated values), to get the highest-profit contiguous subset of route.

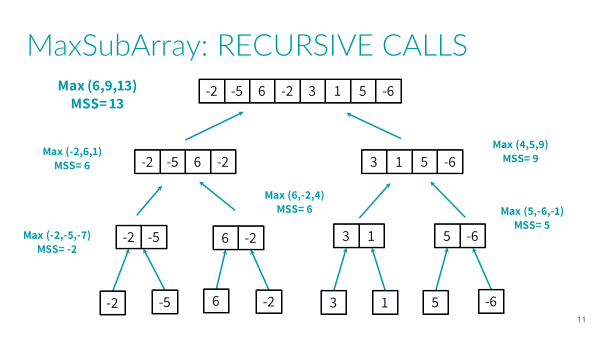
values), to get the highest-profit contiguous subset of route.

**Solution:**

Make an array, and initialize the values to be the profit of the roads, in sequence from start to end.

i.e. R = {45, -11, -14, 35, -31, 45, -14, 31}

Dry run on the above values, as was shown and discussed in the maximum sub-array class lecture. An example is shown below, where we assume R = {-2,-5,6,-2,3,1,5,-6}.

****

**Question 4: 20 points**

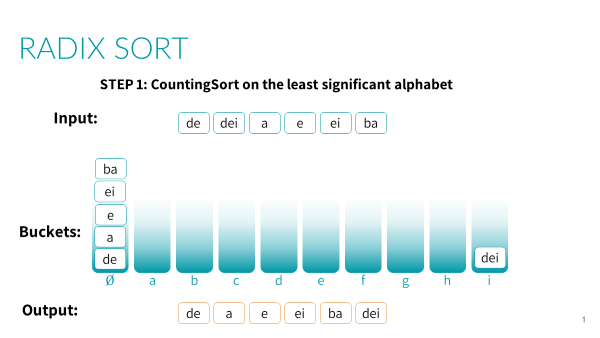
Find the contents of array C[], using your student ID, and then sort it in O(n) time.

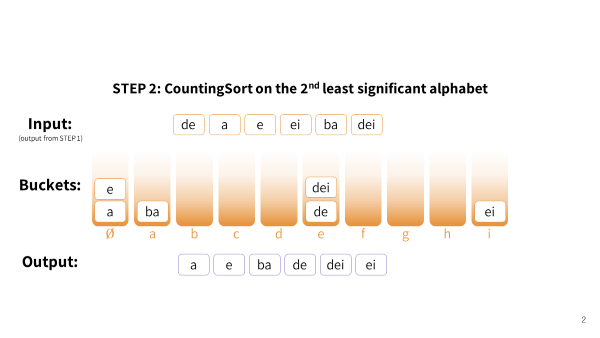
For example, for student ID: 21k-4509, The content of array C = {de, dei, a, e, ei, ba}

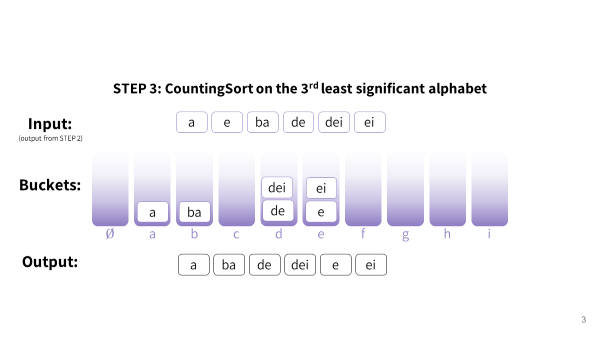
The contents of array have to be derived as follows:

* First find the contents of array S[].
* The contents of array S must be of this given pattern: S = {45, 4509, 09, 50, 509, 21}, which is derived for student ID: 21k-4509.
* Keenly observe the pattern, in which the student ID digits are used.
* That is, the first element of array, is the first two digits of student ID, etc.
* For every digit in the above S array, find the corresponding alphabet for the digit value.
* That is, for a digit value, the respective alternate alphabet value will be: 1=a, 2=b, 3=c, 4=d, 5=e, 6=f, 7=g, 8=h, 9=i, (and 0=ignore).
* Finally: the string value for each content of an array S: 45=de, 4509=dei, 01=a, 50=e, 509=ei, 21=ba
* i.e. C={de, dei, a, e, ei, ba}
* (Note! No respective alternate alphabet is used for digit ‘0’)

**Solution:**







**Question 5: 10 points**

Given an array, you need to shift all zeros to end of array with out changing relative order of other non-zero elements.

For example If A = [1,0,4,0,0,0,2,0] then output = [1,4,2,0,0,0,0,0]

Design algorithm for this that takes O(n) time and O(1) space.

**Solution:**

def pushZerosToEnd(arr, n):

count = 0 # Count of non-zero elements

# Traverse the array. If element encountered is non-zero, then replace the element at index 'count' with this element

for i in range(n):

if arr[i] != 0:

arr[count] = arr[i] # here count is incremented

count+=1

# Now all non-zero elements have been shifted to front and 'count' is set as index of first 0. Make all elements 0 from count to end.

while count < n:

arr[count] = 0

count += 1

# Driver code

arr = [1, 9, 8, 4, 0, 0, 2, 7, 0, 6, 0, 9]

n = len(arr)

pushZerosToEnd(arr, n)

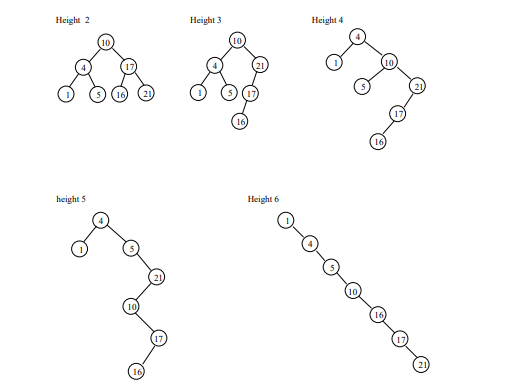
print("Array after pushing all zeros to end of array:")

print(arr)

**Question 6: 10 points**

For the set of keys {1, 4, 5, 10, 16, 17, 21}, draw binary search trees of height 2, 3, 4, 5, and 6.

**Solution:**

****

**Question 7: 10 points**

Find the duplicate element in an array of size 100 in O(n) time.

**Solution:**