Problem 1 Let A and B be arrays of n integers each (do not assume they are sorted). Given an integer x, describe a O(n log n)-time algorithm for determining if there is an integer a in A and an integer b in B such that x = a + b.

Present pseudocode and analyze the running time.

Solution

```
"""
   If Integer in A and B present such that a + b = x then we will
return a, b else will return None, None.

   1. Here sorting of a list and b list will take O(n log n) time
both
   2. While loop will take O(n) time to traverse from start to end
where start and end will both move towards one
   another
   3. Else will be constant time complexity c1

   So total time complexity of the algorithm will be O(2 n log n)
+ O(n) + O(c1) only bottleneck is
   O(n log n) so total time complexity will be O(n log n)
"""


def is_sum_present_in_2_lists(a: [], b: [], x: int):
   # Sorting here will take O(n log n) time where n = number of
elements in both list
    a.sort()
   # Sorting here will take O(n log n) time where n = number of
elements in both list
    b.sort()
    start = 0
    end = len(b) - 1

   # O(n) time complexity for the while loop till be go in between
where start >= end
    while start < len(a) and end > -1:
        if a[start] + b[end] == x:
            return a[start], b[end]
        elif a[start] + b[end] < x:
            start += 1
        else:
            end -= 1
    return None, None
```

Problem 2 Characterize each of the following recurrence equations using the master method (assuming that T(n) = c for n < d, for constants d ≥ 1).

Solution

According to master's Theorem

The Master Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b as $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$, and all sufficiently large n, then $T(n) = \Theta(f(n))$.

a. $T(n) = 2T(n/2) + (n \log n)^4$

Solution

f(n) = $(n \log n)^4$
a = 2, b= 2
$\log_b a = \log_2 2 = 1$
n^($\log_b a$ + e) = n < n^4 and e = 3 > 0 here
T(n) = theta(f(n)) = theta($(n \log n)^4$)

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

Solution

f(n) = $\log^2 n$
a = 2, b= 2
$\log_b a = \log_2 2 = 1$
n^($\log_b a$) > n^0
n^($\log_b a$ - e) = n^0 and e = 1 >0 here
T(n) = theta(n^$\log_b a$) = theta(n)

c. $T(n) = 9T(n/3) + n^2$

Solution

f(n) = n^2
a = 9, b= 3
$\log_b a = \log_3 9 = 2$
n^($\log_b a$) = n^2 = f(n)
T(n) = theta(n^($\log_b a$) logn) = theta(n^2 logn)

d. $T(n) = 9T(n/3) + n^3$

Solution

$f(n) = n^3$

$a = 9$, $b= 3$

$\log_b a = \log_3 9 = 2$

$n^\wedge(\log_b a) = n^2 < n^3 = n^\wedge(\log_b a + e)$, where $e = 1$

$T(n) = theta(f(n)) = theta(n^3)$

e. $T(n) = 7T(n/2) + n^2$

Solution

$f(n) = n^2$

$a = 7$, $b= 2$

$n^\wedge\log_b a = n^\wedge\log_2 7 > n^2 = n^\wedge(\log_b a - e)$, where $e > 0$

$n^\wedge\log_b a > n^3$

$T(n) = theta(n^\wedge\log_b a) = theta(n^\wedge\log_2 7)$

Problem 3 Show that the running time of QUICKSORT is $\Theta(n^2)$ when array A contains distinct elements and is sorted in decreasing order.
Solution

Let's say we have array of following numbers x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10}
here numbers are in decreasing order so
$x_1 > x_2 > x_3 > x_4 > x_5 > x_6 > x_7 > x_8 > x_9 > x_{10}$

Quick Sort Algorithm will be like the following:

```
class QuickSort:
   def __init__(self, elements):
       self.elements = elements

   def quick_sort(self):
       self.__quick_sort(0, len(self.elements) - 1)

   def __quick_sort(self, left: int, right: int):
       if left < right:
           pivot = self.__partition(left, right)
           self.__quick_sort(left, pivot - 1)
           self.__quick_sort(pivot + 1, right)

   def __partition(self, left: int, right: int):
       # we are taking all small elements to pivot to the left
       # of it and all big element to pivot to the right of it
       pivot = right
       divider = left

       for iterator in range(left, right):
           if self.elements[iterator] <= self.elements[pivot]:
               self.elements[divider], self.elements[iterator] =
self.elements[iterator], self.elements[divider]
               divider += 1
       self.elements[divider], self.elements[pivot] =
self.elements[pivot], self.elements[divider]
       print(self.elements[left:right])
       pivot = divider
       return pivot
```

Here in partition, the left-hand side of the pivot will be less than the pivot and the right-hand side of the pivot will be greater than the pivot.

First Pass
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10}
pivot = x_{10}

partition = $\mathbf{x_{10}}$ **(pivot)** x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_1

Here we can see that because the pivot is the smallest element in the list after partition it does not contain left elements which are lesser than it. So there will be no elements to sort on the left side of the pivot.

2nd Pass
left no element
right
x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_1
pivot = x_1
partition = x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 $\mathbf{x_1}$ **(Pivot)**

Here we can see that because the pivot is the largest element in the list after partition it does not contain the right elements which are greater than it. So there will be no elements to sort on the right side of the pivot.

3rd Pass
left
x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9
pivot = x_9
partition = $\mathbf{x_9}$ **(pivot)** x_3 x_4 x_5 x_6 x_7 x_8 x_2

right no element

4th Pass
left no element

right
x_3 x_4 x_5 x_6 x_7 x_8 x_2
pivot = x_2
partition = x_3 x_4 x_5 x_6 x_7 x_8 $\mathbf{x_2}$ **(pivot)**

5th Pass
left
x_3 x_4 x_5 x_6 x_7 x_8
pivot = x_8
partition = $\mathbf{x_8}$ **(pivot)** x_4 x_5 x_6 x_7 x_3

right no element

6th Pass
left no element

right
x_4 x_5 x_6 x_7 x_3
pivot = x_3
partition = x_4 x_5 x_6 x_7 **x_3(pivot)**

7th Pass
left
x_4 x_5 x_6 x_7
pivot = x7
partition = **x_7(pivot)** x_5 x_6 x_4
right no element

8th Pass
left no element

right
x_5 x_6 x_4
pivot = x_4
partition = x_5 x_6 **x_4(pivot)**

9th Pass
left
x_5 x_6
pivot = x_6
partition = **x_6(pivot)** x_5

right no element

10th Pass
left no element

right
x_5
pivot = x_5
partition = **x_5(pivot)**

Here time complexity of every pass of the partition will be as following:
T(n) = T(n-1) + O(n)

Because the pivot will get sorted to its position and remaining n - 1 element will be on the right side of the pivot or left side of the partition according to our partition algorithm. Here O(n) time complexity will be of the partitioning algorithm.
So T(n) = theta(n + n-1 + n-2 + ... + 1)
T(n) = theta(n^2)

Problem 4 Describe an O(n)-time algorithm that, given a set S of n distinct numbers and a positive integer k ≤ n, determines the k numbers in S that are closest to the median of S.

Assume n is odd and the set S is given as an **unsorted** array of size n. You cannot assume the input array is sorted.

Example: if S = {1, 3, 5, 9, 13, 21, 101} and k = 4, the solution is {3, 5, 9, 13}. That is, the median itself is included. The answer {5, 9, 13, 21} is not correct since 3 is closer to the median (which is 9) than 21. The algorithm should write the output in a separate array, and the numbers **do not** have to be sorted.

You can use the selection algorithm as a subroutine. Precisely, assume that the following procedure is given: SELECT(A,p,q,i) returns (finds) the index j such that A[j] is the i^{th} smallest 1 number among A[p], A[p+1], . . . , A[q]. SELECT correctly runs in time O(q−p) even if the elements of A are not distinct. SELECT here is an extension of Quick-select(A,1,n,i) as in the notes, and only requires two extra tricks for implementation.

Partial credit will be given to correct algorithms, but with larger running time.

Solution

Here I have used Selection in the worst linear time algorithm found in the book chapter 9 Medians and Order Statistics and implemented **median_kth_smallest_element**, which gives k_th smallest element of the unsorted array in linear time O(n).

Main Algorithm is below:

```
"""
   first we will find median_value which will take O(n) time
   To find distances to the median will take O(n) time
   kth_distance from distances will take O(n) time
   k_nearest to the k_th distance will take O(n) time

   So total time complexity will O(4*n) = O(n)
"""
```

```
def find_k_closet_to_median(elements: [], k: int):
    mid = len(elements) // 2

    median_index, median_value =
median_kth_smallest_element(elements, 0, len(elements) - 1, mid)

    distances_to_median = []
```

```
        for index in range(0, len(elements)):
            distances_to_median.append(abs(elements[index] -
median_value))

        kth_distance_index, kth_distance_value =
median_kth_smallest_element(distances_to_median, 0,
len(distances_to_median) - 1, k - 1)

        k_nearest = []
        for index in range(0, len(elements)):
            distances_to_median = abs(elements[index] -
median_value)
            if distances_to_median <= kth_distance_value:
                k_nearest.append(elements[index])

        return k_nearest
```

Supporting Algorithms is as follows:

```
"""
    Here sorting will take O(5 log 5) time so time complexity is
O(1) and for loop will take O(n) time bcz
    all all inside it is of constant time so total time complexity
of this algo will be O(n)
"""


def find_medians(elements: [], left: int, right: int):
    medians = []
    for index in range(left, right, 5):
        if index + 5 < right:
            lis = elements[index: index + 5]
        else:
            lis = elements[index: right]
        lis.sort()
        mid = len(lis) // 2
        medians.append(lis[mid])
    return medians


"""
    Algo will be O(n) time complexity by definition in the book
"""
```

```
def median_kth_smallest_element(elements: [], left: int, right:
int, k: int):
   if left <= right:

       medians = find_medians(elements, left, right)
       number_of_median = len(medians)
       if number_of_median == 1:
           median_of_median = medians[0]
       else:
           median_of_median = median_kth_smallest_element(medians,
left, number_of_median - 1,

number_of_median // 2)

       divider = element_partition(elements, left, right,
median_of_median)
       if divider == k:
           return divider, elements[divider]
       elif k < divider:
           return median_kth_smallest_element(elements, left,
divider - 1, k)
       else:
           return median_kth_smallest_element(elements, divider +
1, right, k)


"""
    Here first for loop will take O(n) time + last for loop will
take O(n) time rest will take constant time
   complexity so total time complexity will be O(n)
"""


def element_partition(elements: [], left: int, right: int,
element: int):
   # we are taking all small elements to pivot to the left
   # of it and all big element to pivot to right of it
   index_position = left
   for index in range(left, right):
       if elements[index] == element:
           index_position = index
           break

     elements[index_position], elements[right] = elements[right],
elements[index_position]
   pivot = right
```

```
    divider = left

    for iterator in range(left, right):
        if elements[iterator] <= elements[pivot]:
            elements[divider], elements[iterator] = elements[iterator], elements[divider]
            divider += 1
    elements[divider], elements[pivot] = elements[pivot], elements[divider]
    return divider
```

Problem 5

1. Draw the 11-item hash table resulting from hashing the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5, using the hash function h(i) = (2i + 5) mod 13 and assuming collisions are handled by chaining. Here the hash table has 13 slots.

Solution

h(k) = (2k + 5) mod m

k = key to be inserted

m = number of slots = 13

```
Hashing
slot: 0 --> None
slot: 1 --> 11
slot: 2 --> 5 <--> 44
slot: 3 --> 12
slot: 4 --> None
slot: 5 --> 39 <--> 13
slot: 6 --> 20
slot: 7 --> None
slot: 8 --> None
slot: 9 --> None
slot: 10 --> None
slot: 11 --> 16 <--> 94
slot: 12 --> 23 <--> 88
```

2. What is the result of the previous exercise, assuming collisions are handled by linear probing? In the notation from the textbook, use h as defined above for the auxiliary function h′.

Solution

Here collisions are handled by hashfunction h(k, i) which is as following:-

h(k, i) = (h′(k) + i) mod m

h′(k) = (2k + 5)

k = key to be inserted

m = number of slots = 13

```
Linear Probing
slot:  0  key  23
slot:  1  key  11
slot:  2  key  44
slot:  3  key  12
slot:  4  key  16
slot:  5  key  13
slot:  6  key  39
slot:  7  key  20
slot:  8  key  5
slot:  9  key  None
slot:  10  key  None
slot:  11  key  94
slot:  12  key  88
```

3. Show the result of Exercise 1 above in this problem, assuming collisions are handled by quadratic probing, up to the point where the method fails because no empty slot is found. In the notation from the textbook, use h as defined above for the auxiliary function h′ and c1 = 0, c2 = 1. Also, the key will not be inserted if no empty slot is discovered with at most 10 probes.

Solution

Here collisions are handled by hashfunction h(k, i) which is as following:-

$h(k\ ,\ i) = (h'(k) + c1 * i + c2 * i^2) \bmod m$

$h'(k) = (2k + 5)$

k = key to be inserted

m = number of slots = 13

```
Quadratic Probing
slot:  0  key  23
slot:  1  key  11
slot:  2  key  44
slot:  3  key  12
slot:  4  key  None
slot:  5  key  13
slot:  6  key  39
slot:  7  key  20
slot:  8  key  None
slot:  9  key  None
slot:  10  key  16
slot:  11  key  94
slot:  12  key  88
```

4. What is the result of Exercise 1 above in this problem, assuming collisions are handled by double hashing using a secondary hash function h2(k) = 7− (k mod 7)? In the notation from the textbook, use h1 = h.

Solution

Here collisions are handled by hashfunction h(k, i) which is as following:-

h(k, i) = (h1(k) + i h2(k)) mod m

h1(k) = (2k + 5)

h2(k) = 7− (k mod 7)

m = number of slots = 13

k = key to be inserted

```
Double Hashing
slot:  0  key  16
slot:  1  key  11
slot:  2  key  44
slot:  3  key  12
slot:  4  key  23
slot:  5  key  13
slot:  6  key  20
slot:  7  key  None
slot:  8  key  39
slot:  9  key  None
slot:  10  key  5
slot:  11  key  94
slot:  12  key  88
```