2023학년도 1학기

Computer Algorithms

Hands-On Assignment 1 Report

과 목 명	ComputerAlgorithms(01)
담 당 교 수	Se Eun Oh
학 과	컴퓨터공학과
학 번	2071035
이 름	이소민 (LeeSomin)
제 출 일	2023.03.07

ha1_1.py

1. Code:

```
1
2  # Quick_Sort
3
4  def pivotSelect(arr, low, high, tcom):
5      pivot = low
6      if tcom == 1:
7          # Time Complexity n^2
8          # Select pivot as the max value
9          for i in range(low,high+1):
10             if arr[pivot] < arr[i]:
11                 pivot=i
12     elif tcom == 2:
13         # Time Complexity n(log n)
14         # Select pivot as the median value
15         for i in range(low+1,high,2):
16             if (arr[i] <= arr[i+1] <= arr[pivot]) or (arr[pivot] <= arr[i+1] <= arr[i]):
17                 pivot = i+1
18             elif (arr[i+1] <= arr[i] <= arr[pivot]) or (arr[pivot] <= arr[i] <= arr[i+1]):
19                 pivot = i
20     else:
21         print("pivot select error")
22     # return the selected pivot
23     print("p_index:",pivot,"#tpivot: ",arr[pivot])
24     return pivot
25
26 def partition(arr,low,high,tcom):
27     # choose the pivot
28     p_index=pivotSelect(arr,low,high,tcom)
29     (arr[p_index],arr[high])=(arr[high],arr[p_index])
30     pivot=arr[high]
31     i = low-1
32
33     # compare each element with pivot
34     for j in range(low,high):
35         if arr[j]<=pivot:
36             i += 1
37             (arr[i],arr[j])= (arr[j],arr[i])
38             print(i,j,arr)
39     (arr[i+1],arr[high])=(arr[high],arr[i+1])
40     print(i,j,arr)
41
42     # return the position where partition is done
43     return i+1
44
45 def quickSort(arr,low,high,tcom):
46     if low<high:
47         # Find pivot element
48         # element smaller than pivot are on the left
49         # element greater than pivot are on the right
50         c_pivot = partition(arr,low,high,tcom)
51         # recursive call on the left array of pivot
52         quickSort(arr,low,c_pivot-1,tcom)
53         # recursive call on the right array of pivot
54         quickSort(arr,c_pivot+1,high,tcom)
55
56 if __name__=="__main__":
57     array1 = [21,3,12,15,7,32,4,25,9,18]
58     print("***pivot: MAX***#nOriginal Array: ", array1)
59     quickSort(array1,0,len(array1)-1,1)
60     print ("#nSort Result: ", array1)
61
62     array2 = [21,3,12,15,7,32,4,25,9,18]
63     print("#n***pivot: MID***#nOriginal Array: ", array2)
64     quickSort(array2,0,len(array2)-1,2)
65     print ("#nSort Result: ", array2)
```

2. Result:

```
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39
tel)] on win32
Type "help", "copyright", "credits" or "license()" fo
>>>
=== RESTART: C:₩Users₩소민₩Desktop₩ComputerAlgorithm₩
==
***pivot: MAX***
Original Array:  [21, 3, 12, 15, 7, 32, 4, 25, 9, 18]
p_index: 5      pivot:  32
0 0 [21, 3, 12, 15, 7, 18, 4, 25, 9, 32]
1 1 [21, 3, 12, 15, 7, 18, 4, 25, 9, 32]
2 2 [21, 3, 12, 15, 7, 18, 4, 25, 9, 32]
3 3 [21, 3, 12, 15, 7, 18, 4, 25, 9, 32]
4 4 [21, 3, 12, 15, 7, 18, 4, 25, 9, 32]
5 5 [21, 3, 12, 15, 7, 18, 4, 25, 9, 32]
6 6 [21, 3, 12, 15, 7, 18, 4, 25, 9, 32]
7 7 [21, 3, 12, 15, 7, 18, 4, 25, 9, 32]
8 8 [21, 3, 12, 15, 7, 18, 4, 25, 9, 32]
8 8 [21, 3, 12, 15, 7, 18, 4, 25, 9, 32]
p_index: 7      pivot:  25
0 0 [21, 3, 12, 15, 7, 18, 4, 9, 25, 32]
1 1 [21, 3, 12, 15, 7, 18, 4, 9, 25, 32]
2 2 [21, 3, 12, 15, 7, 18, 4, 9, 25, 32]
3 3 [21, 3, 12, 15, 7, 18, 4, 9, 25, 32]
4 4 [21, 3, 12, 15, 7, 18, 4, 9, 25, 32]
5 5 [21, 3, 12, 15, 7, 18, 4, 9, 25, 32]
6 6 [21, 3, 12, 15, 7, 18, 4, 9, 25, 32]
7 7 [21, 3, 12, 15, 7, 18, 4, 9, 25, 32]
7 7 [21, 3, 12, 15, 7, 18, 4, 9, 25, 32]
p_index: 0      pivot:  21
0 0 [9, 3, 12, 15, 7, 18, 4, 21, 25, 32]
1 1 [9, 3, 12, 15, 7, 18, 4, 21, 25, 32]
2 2 [9, 3, 12, 15, 7, 18, 4, 21, 25, 32]
3 3 [9, 3, 12, 15, 7, 18, 4, 21, 25, 32]
4 4 [9, 3, 12, 15, 7, 18, 4, 21, 25, 32]
5 5 [9, 3, 12, 15, 7, 18, 4, 21, 25, 32]
6 6 [9, 3, 12, 15, 7, 18, 4, 21, 25, 32]
6 6 [9, 3, 12, 15, 7, 18, 4, 21, 25, 32]
p_index: 5      pivot:  18
0 0 [9, 3, 12, 15, 7, 4, 18, 21, 25, 32]
1 1 [9, 3, 12, 15, 7, 4, 18, 21, 25, 32]
2 2 [9, 3, 12, 15, 7, 4, 18, 21, 25, 32]
3 3 [9, 3, 12, 15, 7, 4, 18, 21, 25, 32]
4 4 [9, 3, 12, 15, 7, 4, 18, 21, 25, 32]
5 5 [9, 3, 12, 15, 7, 4, 18, 21, 25, 32]
5 5 [9, 3, 12, 15, 7, 4, 18, 21, 25, 32]
p_index: 3      pivot:  15
0 0 [9, 3, 12, 4, 7, 15, 18, 21, 25, 32]
1 1 [9, 3, 12, 4, 7, 15, 18, 21, 25, 32]
2 2 [9, 3, 12, 4, 7, 15, 18, 21, 25, 32]
3 3 [9, 3, 12, 4, 7, 15, 18, 21, 25, 32]
4 4 [9, 3, 12, 4, 7, 15, 18, 21, 25, 32]
4 4 [9, 3, 12, 4, 7, 15, 18, 21, 25, 32]
p_index: 2      pivot:  12
0 0 [9, 3, 7, 4, 12, 15, 18, 21, 25, 32]
1 1 [9, 3, 7, 4, 12, 15, 18, 21, 25, 32]
2 2 [9, 3, 7, 4, 12, 15, 18, 21, 25, 32]
3 3 [9, 3, 7, 4, 12, 15, 18, 21, 25, 32]
3 3 [9, 3, 7, 4, 12, 15, 18, 21, 25, 32]
p_index: 0      pivot:  9
0 0 [4, 3, 7, 9, 12, 15, 18, 21, 25, 32]
1 1 [4, 3, 7, 9, 12, 15, 18, 21, 25, 32]
2 2 [4, 3, 7, 9, 12, 15, 18, 21, 25, 32]
2 2 [4, 3, 7, 9, 12, 15, 18, 21, 25, 32]
p_index: 2      pivot:  7
0 0 [4, 3, 7, 9, 12, 15, 18, 21, 25, 32]
1 1 [4, 3, 7, 9, 12, 15, 18, 21, 25, 32]
1 1 [4, 3, 7, 9, 12, 15, 18, 21, 25, 32]
p_index: 0      pivot:  4
0 0 [3, 4, 7, 9, 12, 15, 18, 21, 25, 32]
0 0 [3, 4, 7, 9, 12, 15, 18, 21, 25, 32]

Sort Result:  [3, 4, 7, 9, 12, 15, 18, 21, 25, 32]
```

```
***pivot: MID***
Original Array:  [21, 3, 12, 15, 7, 32, 4, 25, 9, 18]
p_index: 2      pivot:  12
0 1 [3, 21, 18, 15, 7, 32, 4, 25, 9, 12]
1 4 [3, 7, 18, 15, 21, 32, 4, 25, 9, 12]
2 6 [3, 7, 4, 15, 21, 32, 18, 25, 9, 12]
3 8 [3, 7, 4, 9, 21, 32, 18, 25, 15, 12]
3 8 [3, 7, 4, 9, 12, 32, 18, 25, 15, 21]
p_index: 2      pivot:  4
0 0 [3, 7, 9, 4, 12, 32, 18, 25, 15, 21]
0 2 [3, 4, 9, 7, 12, 32, 18, 25, 15, 21]
p_index: 2      pivot:  9
2 2 [3, 4, 7, 9, 12, 32, 18, 25, 15, 21]
2 2 [3, 4, 7, 9, 12, 32, 18, 25, 15, 21]
p_index: 9      pivot:  21
5 6 [3, 4, 7, 9, 12, 18, 32, 25, 15, 21]
6 8 [3, 4, 7, 9, 12, 18, 15, 25, 32, 21]
6 8 [3, 4, 7, 9, 12, 18, 15, 21, 32, 25]
p_index: 5      pivot:  18
5 5 [3, 4, 7, 9, 12, 15, 18, 21, 32, 25]
5 5 [3, 4, 7, 9, 12, 15, 18, 21, 32, 25]
p_index: 8      pivot:  32
8 8 [3, 4, 7, 9, 12, 15, 18, 21, 25, 32]
8 8 [3, 4, 7, 9, 12, 15, 18, 21, 25, 32]

Sort Result:  [3, 4, 7, 9, 12, 15, 18, 21, 25, 32]
>>>
```

3. Analysis:

(*각 함수의 tcom 매개변수에 1을 넣으면 n^2의 time complexity를 가지는 quick sort가, 2를 넣으면 nlogn의 time complexity를 가지는 quick sort가 실행된다.)

1) pivotSelect(arr, low, high, tcom)

parameter: int[] arr, int low, int high, int tcom
return: int pivot

이 함수는 partition() 함수의 실행을 위한 pivot을 뽑는 함수이다. 매개변수 low와 high는 배열의 시작지점과 끝 지점의 index를 나타낸다. 매개변수 tcom의 값이 1이 들어오면 주어진 배열의 최댓값의 index를 리턴하고, tcom의 값이 2가 들어오면 주어진 배열의 중앙값의 index를 리턴한다.

2) partition(arr, low, high, tcom)

parameter: int[] arr, int low, int high, int tcom
return: int i+1

이 함수는 pivot값보다 작은 값은 pivot의 왼쪽으로, 큰 값은 pivot의 오른쪽으로 옮기고 pivot의 값이 최종적으로 자리한 index 위치인 i+1의 값을 리턴한다. pivotSelect()을 호출하여 pivot값을 정하고 pivot값을 배열의 맨 오른쪽 값과 swap한다. 그리고 왼쪽부터 각 원소의 값을 pivot값과 비교하며 해당 값이 pivot보다 작으면 i의 index에 있는 원소와 비교중인 원소를 swap하고 i를 한 칸 오른쪽으로 옮긴다. 여기서 i는 pivot보다 작은 값이 끝나고 큰 값이 시작되는 경계의 index를 가리키는 포인터이다. 주어진 배열의 모든 원소에 대한 비교와 재배열이 끝나면, i+1의 위치에 있는 pivot보다 큰 첫 번째 값과 맨 오른쪽에 위치한 pivot의 값의 위치를 바꿔준다. 이 과정이 끝나면 함수는 pivot값이 위치한 index 값인 i+1을 반환한다.

3) quickSort(arr, low, high, tcom)

parameter: int[] arr, int low, int high, int tcom
return: -

이 함수는 주어진 정수 배열 arr의 low부터 high까지에 대해 quick sort를 수행한다. 배열의 원소가 한 개 이상일 때 함수는 partiton()을 호출하여 pivot값을 정해 pivot과의 크기 비교를 통해 왼쪽과 오른쪽 배열을 만든다. 이후, 함수는 pivot의 왼쪽 배열과 오른쪽 배열에 대하여 각각 quickSort()를 재귀적으로 호출하여 정렬을 실행한다.

4) main

parameter: -
return: -

main함수에서는 원본 배열인 array1과 array2를 정의하고 배열에 대해 pivot을 최댓값으로 정할 때와 중앙값으로 정할 때 각각 quickSort()를 호출하여 그 과정과 정렬 결과를 출력한다.

4. Time Complexity:

1) pivotSelect():

line 5: Θ(1)
line 6-19 (for문): Θ(n)

$T(n) = T(n-1) + 2$
$T(n-1) = T(n-2) + 2$
⋮
+) $T(1) = 2$

$T(n) = 2n \Rightarrow \Theta(n)$

2) partition():

line 28 (pivotSelect() call): Θ(n)

line 29-31: Θ(1)
line 34-38 (for문): Θ(n)

```
T(n) = T(n-1) + 3
T(n-1) = T(n-2) + 3
        ⋮
+) T(1) = 3
-------------------
T(n) = 3n        => Θ(n)
```

line 39-43: Θ(1)

3) quickSort():

line 50-54 (partition() call and recurrsive call):

I. pivot == MAX: Θ(n^2)

```
T(n) = T(n-1) + 3n
T(n-1) = T(n-2) + 3n
        ⋮
+) T(1) = 3n
-------------------
T(n) = 3n^2      => Θ(n^2)
```

II. pivot == MID: Θ(nlogn)

```
T(n) = 2*T(1/2*n) + 3n
2*T(n/2) = (2^2)*T(n/(2^2)) + 3n
(2^2)*T(n/(2^2)) = (2^3)*T(n/(2^3)) + 3n
                ⋮
+) (2^(k-1))*T(n/(2^(k-1))) = (2^k)*T(n/(2^k))+3n
-------------------------------------------
T(n) = (2^k)T(n/(2^k))+3kn
suppose 1/(2^k) = 1
T(n) = nT(1)+h*n*log2n == Θ(nlogn)
```

4) *Total Time Complexity:*

I) pivot == MAX: Θ(n^2)

II) pivot == MID: Θ(nlogn)

5. Step3: Explain why one type performs more efficiently than the other.

pivot 값을 최댓값이나 최솟값으로 정할 경우, pivot을 기준으로 작은 원소들을 왼쪽에, 큰 원소들을 오른쪽에 배치하였을 경우 배열의 크기가 각각 1과 n이 되므로 subproblem의 크기가 줄어들지 않고 그대로 n이 되며 한 번의 재귀 호출만이 의미 있는 시간복잡도 값을 가진다. partition의 time complexity는 Θ(n)이므로 이 함수가 n번 수행되게 된다. 따라서 이때 quick sort는 Θ(n^2)이라는 비교적 큰 값의 time complexity를 갖게 된다. 반면 중앙값을 pivot으로 정하면 재귀 호출에서 problem size는 반이 되고 왼쪽과 오른쪽의 배열에 대해 재귀 호출을 하므로 T(n) = 2*T(1/2*n) + 3n가 되어 Θ(nlogn)의 time complexity를 갖는다.

ha1_2.py

1. Code:

```
1
2  # Find minimum number of US postage stamps_Greedy Algorithm
3
4  def getStampQuantity(p_stamp,cost):
5      q_stamp = [0]*len(p_stamp)
6      s_stamp=0
7      # get the maximum number of stamp that fit the cost for each stamp
8      # update the cost with the remainder
9      # after paying with the maximum number of stamp
10     # check from most expensive one to cheapest one
11     for i in range(len(p_stamp)-1,-1,-1):
12         q_stamp[i]=cost//p_stamp[i]
13         cost%=p_stamp[i]
14         s_stamp+=q_stamp[i]
15         #print(p_stamp[i],"cent post:",q_stamp[i],"₩tcost: ",cost)
16     print("Cost of stamp = ",p_stamp)
17     print("Quantity of stamp = ",q_stamp)
18     print("total stamp: ",s_stamp)
19
20 if __name__=="__main__":
21     p_stamp=[1,10,21,34,70,100,350,1225,1500]
22     cost = 140
23     getStampQuantity(p_stamp,cost)
24
```

2. Result:

```
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1916 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=== RESTART: C:₩Users₩소민₩Desktop₩ComputerAlgorithm₩ComputerAlgorithm₩ha1_2.py
==
Cost of stamp =  [1, 10, 21, 34, 70, 100, 350, 1225, 1500]
Quantity of stamp =  [6, 0, 0, 1, 0, 1, 0, 0, 0]
total stamp:  8
>>>
```

3. Analysis:

1) getStampQuantitiy(p_stamp, cost)

parameter: int[] p_stamp, int cost

return: -

이 함수는 Greedy Algorithm을 이용해 찾을 수 있는 우표의 최소 수량을 구한다. (단, greedy algoritm은 항상 optimal한 답을 반환하지는 않는다.) p_stamp는 우표의 가격을 오름차순으로 담고있는 int형 배열이며, cost는 총 내야 할 비용이다. q_stamp는 p_stamp와 크기가 같은 int형 배열로 각 우표가 사용된 수를 나타낸다. s_stamp는 사용된 우표의 총 개수를 나타낸다. 이 함수에서는 for문을 돌며 제일 비싼 우표부터 최대로 사용하며 cost값을 줄여나간다. 이때, cost를 우표의 가격으로 나눈 몫이 해당 가격의 우표를 사용할 수 있는 최대 개수이고, 나머지가 우표를 사용하고 남은 비용이다. 따라서 몫은 q_stamp에 넣고 나머지는 cost에 넣는다. s_stamp에 해당 for 루프에서 q_stamp에 넣은 값을 더해준다. 모든 가격의 우표에 대한 계산이 끝나고 cost가 0이 되면 p_stamp, q_stamp, s_stamp를 화면에 출력한다.

2) main

parameter: -

return: -

main 함수에서는 p_stamp에 우표의 가격을 오름차순으로 저장하고 cost에 우편물의 총 가격을 저장한다. getStampQuantitiy()를 호출하여 Greedy Algorithm을 사용하여 사용할 수 있는 우표의 최솟값을 구한다.

4. Time Complexity

1) getStampQuantitiy():
   line 5-6: Θ(1)
   line 11-14 (for문): Θ(n)
   T(n) = T(n-1) + 3
   T(n-1) = T(n-2) + 3
   ⋮
   +) T(1) = 3

   T(n) = 3n => Θ(n)
   line 16-18: Θ(1)
2) main:
   line 21-22: Θ(1)
   line 23 (getStampQuantity() call): Θ(n)
3) *Total Time Complexity: Θ(n)*

5. Step3: Show that your implementation does NOT provide an optimal solution by comparing the solution returned by the implementation with the optimal one that can be manually found.

- Minimum number of stamp found by Greedy: 8 stamps (100cent*1, 34cent*1, 1cent*6)
- Minimum number of stamp found manually: 2 stamps (70cent*2)

(3번 문제 이어짐)

ha1_3.py

1. Code:

```
1
2  # Improved in-place merge sort algorithm
3
4  def findMax(arr,start,end):
5      # Find and return the max value in array
6      max=0
7      for k in range(start,end+1):
8          if arr[k]>max:
9              max=arr[k]
10     return max
11
12 def merge(arr,start,mid,end):
13     # Set key as the number that is greater than the max value of the array
14     key=findMax(arr,start,end)+1
15     # index is the position which merge data goes in to
16     # start1, start2 are the positions where two arrays start
17     index=start
18     start1=start
19     start2 = mid +1
20
21     # If first element of second array is greater than the last element of
22     # first array, the array is already sorted. return sorted array.
23     if(arr[mid]<= arr[start2]):
24         return
25
26     # Check the data from two sorted array and merge until data of both arrays
27     # are merged in to result array
28     # Multiply the key and add the value to merge datum to the array
29     # This key is used to save data on top of original array
30     while (start1 <=mid or start2<=end):
31         # When one array has merged all elements to the result array,
32         # Put another array's leftover to the result array
33         if(start1 > mid):
34             arr[index]+= key*(arr[start2]%key)
35             start2 += 1
36
37         elif(start2 > end):
38             arr[index]+=arr[start1]%key*key
39             start1 += 1
40
41         # Merge smaller element between the first element of two arrays
42         # that hasn't been merged yet and shift the start point of source array
43         elif(arr[start1]%key<=arr[start2]%key):
44             arr[index]+= key*(arr[start1]%key)
45             start1 += 1
46
47         else:
48             arr[index]+= key*(arr[start2]%key)
49             start2 += 1
50         # Increase index after merging one element to result array
51         index+=1
52
53     # Get Quotients of data divided by the key to get the sorted array of original data
54     for i in range(start,end+1):
55         arr[i]//=key
56
57
58
59 def mergeSort(arr,l,r):
60     if(l<r):
61         m=l+(r-l)//2
62         # Sort fist and second halves
63         mergeSort(arr,l,m)
64         mergeSort(arr,m+1,r)
65         # Merge the sorted half-arrays
66         merge(arr,l,m,r)
67
68 if __name__=='__main__':
69     arr=[27, 10, 12, 20, 25, 13, 15, 22]
70     mergeSort(arr,0,len(arr)-1)
71     print("sorted array:",arr)
```

2. Result:

```
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24) [MSC v.1916 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=== RESTART: C:₩Users₩소민₩Desktop₩ComputerAlgorithm₩ComputerAlgorithm₩ha1_3.py
==
sorted array: [10, 12, 13, 15, 20, 22, 25, 27]
>>>
```

3. Analysis:

1) findMax(arr, start, end)

parameter: int[] arr, int start, int end

return: int max

이 함수는 배열의 주어진 범위 내에서 최댓값을 찾아 리턴한다.

2) merge(arr, start, mid, end)

parameter: int[] arr, int satrt, int mid, int end

return: -

이 함수는 매개변수로 전달받은 arr에서 배열의 두 부분 start와 mid+1에서 시작하는 두 배열을 오름차순으로 합병한다. findMax()를 이용해 key를 주어진 배열의 최댓값보다 큰 값으로 설정하고, 두 배열의 시작점을 start1, start2로 지정해준다. 왼쪽 배열의 끝 값보다 오른쪽 배열의 시작 값이 더 클 경우, 배열은 이미 정렬되어 있으므로 return한다. 아닐 시, 두 배열의 맨 앞 원소를 비교하여 더 작은 값을 결과 배열로 합병한다. 이때, 합병되는 값은 key값이 곱해진 상태로 배열의 원래 원소에 더해진다. 원 배열의 원소 비교 또한 배열의 값을 key 값으로 나눈 나머지로 비교된다. 이는 배열을 이중으로 사용하는 방법이다. 만약 두 배열 중 한 배열의 모든 원소가 결과에 더해졌다면, 다른 배열의 나머지 원소를 순서대로 결과 배열에 합병해준다. 합병이 끝나면, 배열을 key값으로 나눠 합병하기 전의 배열 정보를 지워준다.

3) mergeSort(arr, l, r)

parameter: int[] arr, int l, int r

return: -

이 함수는 매개변수로 전달받은 배열을 합병정렬 하는 함수이다. 배열의 원소가 한 개가 될 때까지 배열을 반으로 쪼개 각각의 반절 크기의 배열에 대하여 mergeSort()함수를 재귀적으로 호출한다. 부분 배열들이 정렬되면 merge() 함수를 통해 다시 두 배열을 합병해준다.

4) main

parameter: -

return: -

return: -

메인함수에서 배열 arr가 정의되며 이는 정렬해야 할 raw-data를 담고 있다. arr에 대해 mergeSort()를 호출하여 배열을 합병정렬하고 정렬된 배열을 출력한다.

4. Time Complexity:

1) findMax():

line 6: Θ(1)

line 7-9:

T(n) = T(n-1) + 2
T(n-1) = T(n-2) + 2
⋮
+) T(1) = 2

T(n) = 2n => Θ(n)

2) merge

line 17-24: Θ(1)

line 30-51 (for문):

(k는 조건문에 따른 실행되는 명령의 수, n의 값이 변화해도 변하지 않는 상수이다.)

T(n) = T(n-1) + k
T(n-1) = T(n-2) + k
⋮
+) T(1) = k

T(n) = kn => Θ(n)

line 51: Θ(1)

line 54-55 (for문):

T(n) = T(n-1) + 1
T(n-1) = T(n-2) + 1
⋮
+) T(1) = 1

T(n) = n => Θ(n)

3) mergeSort():

line 61: Θ(1)

line 63-66 (recurrsive call, merge()):

(h는 상수)
T(n) = 2*T(1/2*n) + hn
2*T(n/2) = (2^2)*T(n/(2^2)) + hn
(2^2)*T(n/(2^2)) = (2^3)*T(n/(2^3)) + hn
⋮
+) (2^(k-1))*T(n/(2^(k-1))) = (2^k)*T(n/(2^k))+hn

T(n) = (2^k)T(n/(2^k))+hkn
suppose 1/(2^k) = 1
$T(n) = nT(1)+h*n*\log_2 n == \Theta(n\log n)$

4) main:

line 69: Θ(1)

line 70 (mergeSort() call): Θ(nlogn)

line 71: Θ(1)

5) *Total Time Complexity: Θ(nlogn)*

5. Briefly explain your strategies about how to improve the time complexity.

강의에서 다루었던 in-place merge sort algorithm은 한 개의 배열을 쓰는 대신 원소 한 개의 정렬을 한 후 남은 원소들을 한칸씩 shift 하여 한번의 loop에 n번의 연산이 추가되어 총 Θ(n^2)의 시간 복잡도를 가진다. 그 반면, 현재의 알고리즘은 in-place를 배열의 원소의 최댓값보다 큰 값을 곱하여 원래의 원소에 더하는 방식으로 한 배열을 이중으로 사용하는 방식을 택하였다. 배열에는 (결과 원소)*key+(정렬 전 원소)의 형태로 값이 저장되고, arr[i]//key를 하게 되면 정렬 결과에 해당하는 원소가, arr[i]%key를 하게 되면 정렬 전의 원소가 도출된다. (단, key는 data의 최댓값보다 큰 값이어야 한다.) 이때, 추가된 key값 분리 과정의 time comlexity는 Θ(n)으로 최종 time complexity에 큰 영향을 주지 않는다.