CS 225: Data Structures
Homework 3

Group D1

Last Modified on March 18, 2022

Members	
Name	**Student ID**
Li Rong	3200110523
Zhong Tiantian	3200110643
Zhou Ruidi	3200111303
Jiang Wenhan	3200111016

Please turn over for our answers.

Ex. 1 **Our Answer.**

(i) In order to gain the smallest number in the n numbers, At first, suppose we know the exact number which is the smallest number, we should compare it with the other $(n-1)$ numbers which means we need $(n-1)$ comparisons to ensure it is the smallest number. For the specific method, we could select two numbers randomly and then compare them. Then we choose the smaller number and compare it with another number in the left $(n-2)$ numbers. Then repeat this step. We need $(n-1)$ steps (comparisons) to know the smallest number.

(ii) Actually, when we know from the first question, we should use $(n-1)$ comparisons to know the smallest numbers, and then we could know the second smallest number after this step. In fact, in the first question, we have already compared the smallest number with the second smallest number, we can make use of this hidden condition to decrease the comparison times to find the second smallest number after having found the smallest number in the list. We can only find the second smallest number in the number list which has been compared with the smallest number in the first step.

From Figure **??** we know that: At first, we separate the numbers in pairs and compare them (no matter the number is odd or even), Then we can get the smaller number in each pair and we will get the smallest number at last which needs $(n-1)$ comparisons in total. Then we just find the number of elements which have been compared with the smallest number, we can know that there is only one number in each layer(The number which are yellow). For n numbers, there are $\log n$ levels in total, so there are $\log n$ elements which have been compared with the smallest number. Through the first question, we know we need $(\log n - 1)$ comparisons to know the smallest number in the $\log n$ elements (the smallest number in these elements is the second smallest number in the whole list), So we need $(n-2+\log n)$ comparisons[1] to know the smallest number and the second smallest number.

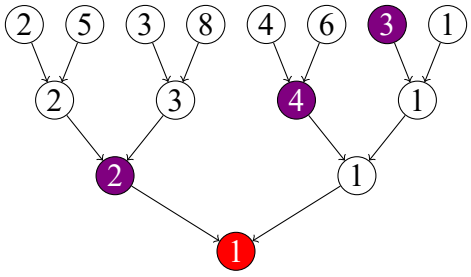

Figure 1: An example structure for the algorithm.

[1] We suspect that there's a typo in the assignment sheet, which said the number of comparisons should be $(n-1+\log n)$, one comparison larger than ours.

(iii) Here is a Python implement of our algorithm. For detailed explanations, see the comments.

```
1  import numpy as np
2  compareDic = {} # Record a set of elements which have been
       compared with their keys
3
4  def find(myList):   # Find the smallest and second smallest
       element
5      if len(myList) == 1:    # The list contains only one
           element
6          return myList[0]
7      nextSublist = []
8      for i in range(len(myList)):
9          if i % 2 == 1:  # Pairing process always happens on
               even-index element
10             continue
11         else:
12             if i+1==len(myList):    # Reaching the end of a
                   list whose length is an odd number
13                 nextSublist.append(myList[i])
14                 continue
15
16             # Record that the `$(i+1)$`-th element have been
                   compared with the `$i$`-th
17             # To reduce the memory space used by compareDic,
                   we assume that a key is always larger than its
                   entries.
18             if myList[i] > myList[i+1]:
19                 nextSublist.append(myList[i+1])
20                 if myList[i+1] in compareDic.keys():
21                     compareDic[myList[i+1]].append(myList[i])
22                 else:
23                     compareDic[myList[i+1]] = [myList[i]]
24             else:
25                 nextSublist.append(myList[i])
26                 if myList[i] in compareDic.keys():
27                     compareDic[myList[i]].append(myList[i+1])
28                 else:
29                     compareDic[myList[i]] = [myList[i+1]]
30     return find(nextSublist)
```

```
31
32 # The main function only provides a testing environment
33 def main():
34     np.random.seed(114514)
35     listsize = 100
36     random_list = np.random.randint(low=0, high=1000, size=
           listsize).tolist()
37
38     smallest = find(random_list)
39
40     se_sm = 1002
41     for log_n in compareDic[smallest]:
42         if log_n < se_sm:
43             se_sm=log_n
44
45     print(random_list, smallest,se_sm)
46
47 if __name__ == "__main__":
48     main()
```

Ex. 2 **OUR ANSWER.**

The algorithm is to keep two memory positions for the current-largest element and current-smallest element. For the first element, there is no comparison. For the second element, there is only one comparison with the first element.

Then, for the left $(n-2)$ elements including the third element. There are only two comparisons for one element, comparison with the current-largest element and current-smallest element. If it is larger than the current-largest, it will take its place and then the algorithm goes for next element. If it is smaller than the current-smallest, then it will take its place and then the algorithm goes for next element. Otherwise it is ignored.

Therefore, the count for total comparison is $2(n-2)+1$,which is $2n-3$.

(i) For the best situation, all the numbers are arranged in the sequence from small to large. Then for every element, there is only one comparison needed. The overall comparison is $n-1$. For the worst situation, all the numbers are arranged in the sequence from large to small. The overall comparison is $2n-3$.

(ii) The number of comparison will not change whether n is a power of 2 or not.

Ex. 3 **OUR ANSWER.**

The first step is to create a library that contains n tags that represent every element in the totally ordered set. Besides, each tag also contains a counter that records the times each time element has been found.

The next thing we need to do is to traverse the list, each time we find a elements, we add one to the corresponding counter. After the list traverse, we traverse the library to find which element is majority.

The structured algorithm is described in Algorithm **??**

Algorithm 1: Majority Element

Input : Two lists T, L

Output: isMajor, denoting whether L contains any major elements.

```
1  isMajor ← False
   /* First create libraries, or "buckets", for each unique element in
      T. */
2  foreach t ∈ T do
3      if t ∉ library then
4          CreateLibrary(t_i)
5      end
6  end
7  foreach l ∈ L do
8      library[l] ← library[l] +1
9  end
10 foreach library[i] do
11     if library[i] > n/2 then
12         isMajor ← True
13     end
14 end
15 return isMajor
```