Devoir 8 IFT2125-A-H19

Student: Qiang Ye (20139927)

Date: 15 Mar 2019

mailto: samuel.ducharme@umontreal.ca (mailto:samuel.ducharme@umontreal.ca)

Question

1. Trouvez la solution exacte de la récurrence Soit

$$T(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \sqrt{T^2(n-1) + 2T^2(n-2) + n} & n > 1 \end{cases}$$

2. Soit $T = \{c_1, \cdots, c_n\}$ un ensemble de clés; vous pouvez supposer qu'elle sont données dans un tableau, également appelé T, de manière évidente: $T[i] = c_i$.

   (1) Donnez un algorithme qui trouve les deux plus grandes clés en moins de $\frac{3}{2}n$ comparaisons.

   (2) Donnez un algorithme qui trouve la plus grande et la plus petite clé en moins de $\frac{3}{2}n$ comparaisons.

   (3) Donnez un algorithme qui trouve les deux plus grandes clés en moins de $n + \lg n$ comparaisons.

   Dans chacun des cas, prouvez, ou au moins justifiez, vos dires.

Answer 1

For $n > 1$,

$$T(n) = \sqrt{T^2(n-1) + 2T^2(n-2) + n}$$
$$\Longrightarrow T^2(n) = T^2(n-1) + 2T^2(n-2) + n$$

Let $S(n) = T^2(n)$, then:

$$\begin{aligned} S(n) &= S(n-1) + 2S(n-2) + n \qquad (1) \\ S(n) - S(n-1) - 2S(n-2) &= n \end{aligned}$$

The characteristic polynomial is:

$$(x+1)(x-2)(x-1)^2 \qquad (2)$$

Therefore, $S(n)$ can be written as:

$$\begin{aligned} S(n) &= c_1(-1)^n + c_2 2^n + c_3 1^n + c_4 n 1^n \\ &= c_1(-1)^n + c_2 2^n + c_3 + c_4 n \end{aligned}$$

We know:

$$\begin{aligned} S(0) &= T^2(0) = 0 \\ S(1) &= T^2(1) = 1 \\ S(2) &= S(1) + 2S(0) + 2 = 3 \\ S(3) &= S(2) + 2S(1) + 3 = 8 \end{aligned}$$

This gives us 4 linear equations for solving unknown constants:

$$\begin{aligned} c_1 + c_2 + c_3 + 0 &= 0 \\ -c_1 + 2c_2 + c_3 + c_4 &= 1 \\ c_1 + 4c_2 + c_3 + 2c_4 &= 3 \\ -c_1 + 8c_2 + c_3 + 3c_4 &= 8 \end{aligned}$$

We obtain the solution:

$$c_1 = -\frac{1}{12},\ c_2 = \frac{4}{3},\ c_3 = -\frac{5}{4},\ c_4 = -\frac{1}{2} \tag{3}$$

and therefore

$$S(n) = -\frac{1}{12}(-1)^n + \frac{4}{3}2^n - \frac{5}{4} - \frac{1}{2}n \tag{4}$$

and therefore

$$T(n) = \sqrt{-\frac{1}{12}(-1)^n + \frac{4}{3}2^n - \frac{5}{4} - \frac{1}{2}n} \tag{5}$$

Answer2

We use DIVIDE-AND-CONQUER algorithms to solve the problems.

First, we give two similar algorithms which find the largest or smallest key in a Table T both respectively by $n-1$ comparisons:

In [1]:
```python
def find_largest(T):
    largest = T[0] # set the largest be the first element
    index = 0 # keep its index
    for i in range(1, len(T)):
        if largest < T[i]: # n-1 comparisons altogether
            largest = T[i]
            index = i
    return largest, index

def find_smallest(T):
    smallest = T[0] # set the smallest be the first element
    index = 0
    for i in range(1, len(T)): # from the second to last
        if smallest > T[i]: # n-1 comparisons altogether
            smallest = T[i]
            index = i
    return smallest, index
```

(1) To find the largest two keys in a table T, we do the following steps:

- step1: Group every two adjacent elements in table T. If the number of elements in T is odd, the last element itself forms a group. So there are altogether $\lceil \frac{n}{2} \rceil$ groups.
- step2: For each group, select the larger element by one comparison; the group which has only one element doesn't need comparison, the only element will considered larger one. So there are at most $\lfloor \frac{n}{2} \rfloor$ comparisons. All larger elements form a new table(T') in which the largest 2 elements must be, whereas T' only has $\lceil \frac{n}{2} \rceil$ elements at most.
- step3: Use the Algorithm(function) `find_largest(T)` to find the largest. We need $\lceil \frac{n}{2} \rceil - 1$ comparisons. Once the largest is found, remove the largest from the T' or just assign a value small enough to its position.
- step4: Use again the same Algorithm(function) to find the largest element in T' with the largest element removed. the output of the algorithm(function) this time will give us the second largest element in the original table T. In this step, we need $\lceil \frac{n}{2} \rceil - 2$ comparisons.

In total, the number of comparison will be:

$$\lfloor \frac{n}{2} \rfloor + (\lceil \frac{n}{2} \rceil - 1) + (\lceil \frac{n}{2} \rceil - 2) < \frac{3}{2}n \tag{6}$$

Here is an implementation by Python:

In [2]:
```python
def find_two_largest(T):
    n = len(T)
    possible_largests = [] # possible largest
    for i in range(n//2):
        if T[2*i] < T[i*2+1]:
            possible_largests.append(T[i*2+1])
        else:
            possible_largests.append(T[i*2])

    if n % 2 == 1: # need to compare the last element of T
        possible_largests.append(T[-1]) # add last element of T

    largest, i = find_largest(possible_largests)
    possible_largests[i] = float('-inf') # set to a possible minimal value
    second_largest, _ = find_largest(possible_largests)
    return largest, second_largest
```

Here is an example:

In [3]:
```python
import numpy as np
T = np.arange(0, 21) # smallest is 0, largest is 19, second largest is 18
np.random.shuffle(T) # shuffle T

print("T: ", T)
l, s = find_two_largest(T)
print("largest:", l, ", second largest:", s)
```

```
T:  [14  8  2 17  7 20  4 12 18  3 13 15  5  9 11 10  6 16  0 19  1]
largest: 20 , second largest: 19
```

(2) To find the largest and the smallest keys in a table T, we use the similar algorithm:

```
function find_largest_smallest(T):
input: T a table with different elements(keys), where key index start
       from 0 to n-1 where n is the length of T
output: the largest and smallest element(key)

n = length of T
m = n // 2
largers = []
smallers = []
for i from 0 to m-1: # every two adjacent elements
    if T[2*i] < T[2*i+1]: # n/2 comparisons at most
        largers.append(T[2*i+1])
        smallers.append(T[2*i])
    else:
        largers.append(T[2*i])
        smallers.append(T[2*i+1])

if n is odd: # the last element, one more comparison,
    if T[n-1] < smallers[0]:  couldn't be the largest
        smallers.append(T[n-1])
    else: # couldn't be the smallest
        largers.append(T[n-1])

largest = find_largest(largers) # n/2 - 1 comparisons at most
smallest = find_smallest(smallers) # n/2 - 1 comparisons at most
return largest, smallest
```

In total, the number of comparison will be at most:

$$\lfloor\frac{n}{2}\rfloor + (\lceil\frac{n}{2}\rceil - 1) + (\lceil\frac{n}{2}\rceil - 1) + 1 < \frac{3}{2}n \tag{7}$$

Here is an implementation by Python:

In [4]:
```
def find_largest_smallest(T):
    n = len(T)
    m = n // 2
    largers, smallers = [], []
    for i in range(m):
        if T[2*i] < T[2*i+1]:
            largers.append(T[2*i+1])
            smallers.append(T[2*i])
        else:
            largers.append(T[2*i])
            smallers.append(T[2*i+1])

    if n%2 == 1:
        if T[-1] < smallers[0]:
            smallers.append(T[-1])
        else:
            largers.append(T[-1])

    largest, _ = find_largest(largers)
    smallest, _ = find_smallest(smallers)
    return largest, smallest
```

An example:

In [5]:
```
np.random.shuffle(T) # shuffle T
print(T)
largest, smallest = find_largest_smallest(T)
print("largest:", largest, ", smallest:", smallest)
```

```
[15  3 17  1 13 18 10 19  6 14  9  8 11 20  5  0  4  7 16  2 12]
largest: 20 , smallest: 0
```

(3) To find the two largest keys by using less than $n + \lg n$ comparisons, we first give another algorithm to find the largest key in table T:

```
def find_largest2(T, his):
    """find the largest key in a table
    inputs:
        T     a table with different keys, list
        his   a dict keeps the comparing history where his[key] is a list
              meaning 'key' once compared with all elments in that list
    outputs:
        the largest key in T
    """
    n = len(T) # length of elements in current group
    if n == 1: # only one lement in T, it is the largest
        return T[0]
    elif n == 2: # 2 elements in T, compare them, output the largest
        his.setdefault(T[0], []).append(T[1]) # record compare history
        his.setdefault(T[1], []).append(T[0]) # each record twice
        return T[0] if T[0] > T[1] else T[1]  # output the larger one
    else: # more than 2 elements in T
        m = n//2 # divide T into two sub list equally
        l1 = find_largest2(T[:m], his) # largest key in left half part T
        l2 = find_largest2(T[m:], his) # largest key in right half part T
        his.setdefault(l1, []).append(l2) # record compare history
        his.setdefault(l2, []).append(l1) # twice for each
        return l1 if l1 > l2 else l2 # output the larger one
```

Let $T(n)$ be the compared times needed to find the largest key in a table with the length n, according to the above algorithm:

$$T(n) = \begin{cases} 0 & n = 0, 1 \\ 1 & n = 2 \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 1 & n > 2 \end{cases}$$

Solving this recurrence equation, we obtain that:

$$T(n) = n - 1 \tag{8}$$

which means it also need $n - 1$ comparisons to find the largest key in table T.

By using this divide and conquer algorithm, every largest key will be selected out after at most $\lg n$ times comparison with other keys, and the second largest key must be the largest one of these keys once compared with the largest key.

By keep an history of all comparison, we have access to the keys compared with the largest key, and it only needs $\lg n - 1$ comparisons to find the second largest key.

Therefore, total comparison times will be no more than

$$(n - 1) + (\lg n - 1) = n + \lg n - 2 < n + \lg n \tag{9}$$

times.

Here is the implementation and the examples by python: By counting the length of the `his` dict, we have an intuition of the comparison times during the procedure of finding the largest key from a given list.

In [6]:

```python
def find_largest2(T, his):
    """find the largest key in a table
    inputs:
        T     a table with different keys, list
        his   a dict keeps the comparing history where his[key] is a list
              meaning 'key' once compared with all elments in that list
    outputs:
        the largest key in T
    """
    def record_his(i, j): # helper function
        """recording comparison history"""
        his.setdefault(i, []).append(j)
        his.setdefault(j, []).append(i)


    n = len(T) # length of elements in current group
    if n == 1: # only one lement in T, it is the largest
        return T[0]
    elif n == 2: # 2 elements in T, compare them, output the largest
        record_his(T[0], T[1])
        return T[0] if T[0] > T[1] else T[1]  # output the larger one
    else: # more than 2 elements in T
        m = n//2 # divide T into two sub list equally
        l1 = find_largest2(T[:m], his) # largest key in left half part T
        l2 = find_largest2(T[m:], his) # largest key in right half part T
        record_his(l1, l2)
        return l1 if l1 > l2 else l2 # output the larger one

def compared_times(his):
    n = 0
    for num in his:
        n += len(his[num]) # the value of a key in dict is a list
    return n/2 # each comparison are recorded 2 times, so divided by 2

def find_two_largest2(T):
    his = {}
    n = len(T)
    largest = find_largest2(T, his)
    print("compared {} times for largest key".format(compared_times(his)))
    print("these keys once compared with the largest key:")
    print(his[largest])
    T_prime = [e for e in his[largest]]
    his2 = {}
    second_largest = find_largest2(T_prime, his2)
    print("compared {} times for second largest key".format(compared_times(his2)))
    return largest, second_largest
```

In [7]:

```
T = np.arange(0, 32)
np.random.shuffle(T) # shuffle T
print(T)
largest, second_largest = find_two_largest2(T)
print("largest:", largest, ", second largest:", second_largest)
```

```
[11 17  5  7  0  2 27 28 12  4 14 29 10 19 13 26 22  3  1  9 31  6 18 23
 20 24 16 30 25 15 21  8]
compared 31.0 times for largest key
these keys once compared with the largest key:
[6, 23, 22, 30, 29]
compared 4.0 times for second largest key
largest: 31 , second largest: 30
```

Appendix: Verification for Question1

In [8]:

```
# find the values of c_1, c_2, c_3, c_4
import numpy as np
X = np.array([[ 1, 1, 1, 0 ],
              [-1, 2, 1, 1 ],
              [ 1, 4, 1, 2 ],
              [-1, 8, 1, 3 ]], dtype = np.float64)

b = np.array([0, 1, 3, 8], dtype = np.float64).reshape(-1, 1)
c = np.dot(np.linalg.inv(X), b)
print(c)
```

```
[[-0.08333333]
 [ 1.33333333]
 [-1.25      ]
 [-0.5       ]]
```

In [9]:

```
# verify T
import math
def T(n):
    t = math.pow(-1, n) * (-1.0)/12.0
    t += math.pow(2, n) * 4.0 / 3.0
    t += -5.0/4.0
    t += -n/2.0
    return math.sqrt(t)

def verify_T(n, epsilon = 1e-10):
    """check if the solution for T(n) is equal to the recurrence definiiton
    of T"""
    if n == 0:
        return T(n) == 0
    elif n == 1:
        return T(n) == 1
    else:
        t_n_solution = T(n)
        t_n_recurrence = math.sqrt(math.pow(T(n-1),2) + 2*math.pow(T(n-2),2) + n)
        return abs(t_n_solution - t_n_recurrence) < epsilon

for n in range(30): # verify n from 1 to 29
    if verify_T(n) is not True:
        print("Wrong")
        break

print("if no 'Wrong' printed, then all True")
```

```
if no 'Wrong' printed, then all True
```