Sample Solution for Problem Set 4

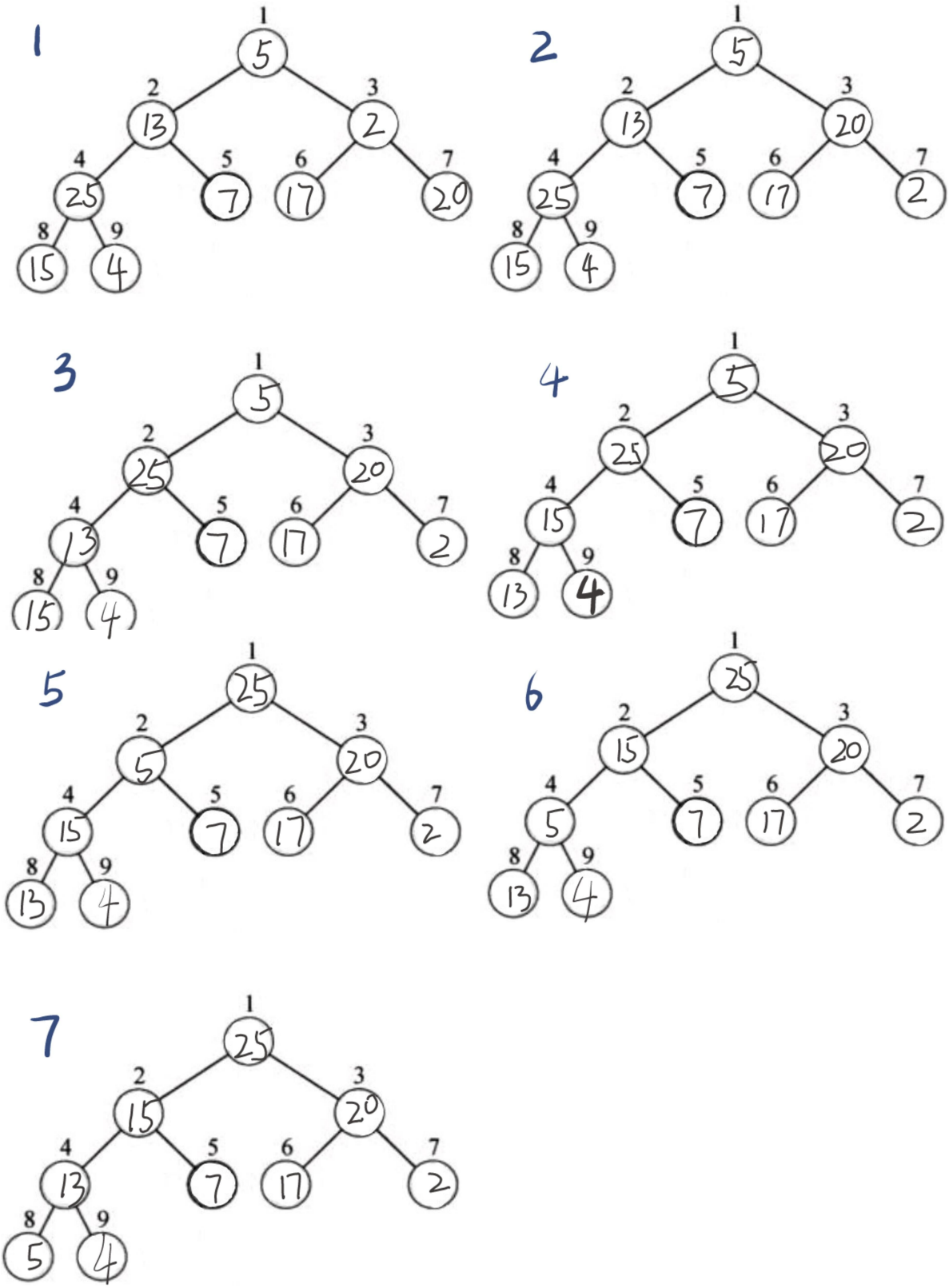
Data Structures and Algorithms, Fall 2021

October 14, 2021

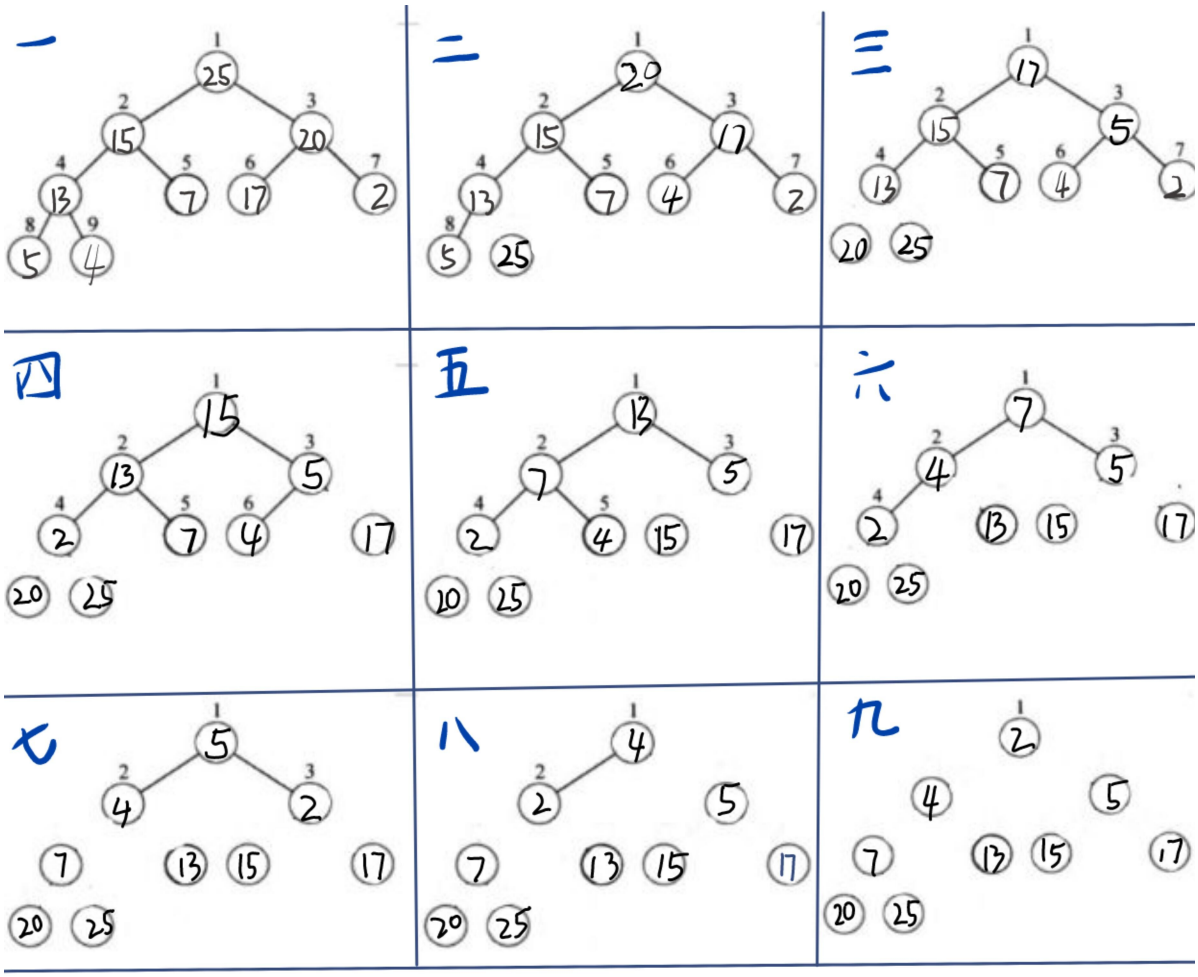
Contents

1 Problem 1

(a)


1
5
2
13
3
2
4
25
5
7
6
17
7
20
8
15
9
4
2
1
5
2
13
3
20
4
25
5
7
6
17
7
2
8
15
9
4
3
1
5
2
25
3
20
4
13
5
7
6
17
7
2
8
15
9
4
4
1
5
2
25
3
20
4
15
5
7
6
17
7
2
8
13
9
4
5
1
25
2
5
3
20
4
15
5
7
6
17
7
2
8
13
9
4
6
1
25
2
15
3
20
4
5
5
7
6
17
7
2
8
13
9
4
7
1
25
2
15
3
20
4
13
5
7
6
17
7
2
8
5
9
4

(b)



The specific process of function Max-Heapify is omited.
It is also acceptable to use sequence or array to show the process.
Thanks for the illustrations from Wensong Sui.

2 Problem 2

We can first build a min-heap from the first elements of each lists, together with its list ID. Then repeat following operations until the min-heap is empty: pop the top element of the min-heap, add it to the final sorted list L and insert the next element from the corresponding list if exists.

Pseudocode

```
1  function Merge(lst[]):
2      list L
3      min_heap q
4      for i from 0 to k - 1 do
5          q.push(make_pair(lst[i].front,i))
6          lst[i].pop_front()
7      end for
8      while not q.empty() do
9          tmp = q.top()
10         key=tmp.first
11         id=tmp.second
12         q.pop()
13         L.push_back(key)
14         if not lst[id].empty() then
15             q.push(make_pair(lst[id].front,i))
16             lst[id].pop_front()
17         end if
18     end while
19     return L
```

Time Complexity

It takes $O(k)$ to build the heap. For every iteration in the n times while loop, it takes $O(\log k)$ to pop and $O(\log k)$ to push. The time complexity in total is $O(k + n \log k)$, which is $O(n \log k)$

3 Problem 3

a If $Cruel$ can sorted correctly, $Unusual(A[1...n])$ should guarantee that: if interval $[1, n/2]$ and $[n/2+1, n]$ are sorted, calling $Unusual(A[1...n])$ can sort interval $[1, n]$ correctly. We prove it using induction.

- (basis) when $n = 2$, interval $[1, 1]$ and $[2, 2]$ are sorted. $Unusual(A[1...n])$ can sort $[1..2]$ correctly.
- (inductive step) Show that for any $k \geq 1$, if it holds for $n = 2^k$, then it also holds for $n = 2^{k+1}$.

  Assume it holds for $n = 2^k$. Consider the case $n = 2^{k+1}$, interval $[1, n/2]$ and $[n/2+1, n]$ are sorted initially. We consider 4 types of elements in A separately.

  - The smallest $n/4$ elements in A. These elements must lies in interval $[1, n/4]$ or interval $[n/2+1, 3n/4]$ initially. The for loop swaps interval $[n/4+1, n/2]$ and $[n/2+1, 3n/4]$ so the smallest $n/4$ elements in A lies in $[1, n/2]$. By the inductive hypothesis, the calling for $Unusual(A[1...n/2])$ sorts interval $[1, n/2]$, which makes the smallest $n/4$ elements in A lies in interval $[1, n/4]$ in sorted order. The rest of operations does not change interval $[1, n/4]$.
  - The largest $n/4$ elements in A. These elements must lies in interval $[n/4+1, n/2]$ or interval $[3n/4+1, n]$ initially. The for loop swaps interval $[n/4+1, /2]$ and $[n/2+1, 3n/4]$ so the largest $n/4$ elements in A lies in interval $[n/2+1, n]$. The call of $Unusual(A[1...n/2])$ does not change interval $[n/2+1, n]$. By the inductive hypothesis, the calling for $Unusual(A[n/2+1...n])$ sorts interval $[n/2+1, n]$, which makes the largest $n/4$ elements in A lies in $[3n/4+1, n]$ in sorted order. The rest of operations does not change interval $[3n/4+1, n]$.
  - Other elements. After the execution of $Unusual(A[1...n/2])$ and $Unusual(A[n/2+1...n])$, these elements are moved out of interval $[3n/4+1, n]$ and $[1, n/4]$ and lies in interval $[n/4+1, 3n/4]$. At this moment, interval $[n/4+1, n/2]$ and $[n/2+1, 3n/4]$ is sorted. By the inductive hypothesis, the calling for $Unusual(A[n/2+1...n])$ sorts interval $[n/4+1, 3n/4]$, which makes these elements lies in $[n/4+1, 3n/4]$ in sorted order.

  Thus, for $n = 2^{k+1}$, $Unusual(A[1...n])$ sorts interval $[1, n]$ correctly.

b If we remove the for loop from $Unusual$, $Cruel$ cannot correctly sort $\{1, 4, 2, 3\}$.

c If we swap the last two lines of $Unusual$, $Cruel$ cannot correctly sort $\{3, 4, 1, 2\}$.

d Runtime analysis:

- $Unusual$: $T_u(n) = 3T_u(\frac{n}{2}) + \Theta(n)$, we have $T_u(n) = \Theta(n^{\log_2 3})$.
- $Cruel$: $T_c(n) = 2T_c(\frac{n}{2}) + \Theta(n^{\log_2 3})$, we have $T_c(n) = \Theta(n^{\log_2 3})$.

4 Problem 4

a We can regard the second and subsequent iterations of the while loop in $TRQuickSort$ algorithm as the second recursive call in QuickSort. Therefore, $TCQuickSort$ must correctly sort the array A.

b If an array of length n is already sorted, $TRQuickSort$'s stack depth is $\Theta(n)$.

c Modify the code to $ModifiedTRQuickSort$ as follow:

```
1  procedure ModifiedTRQuickSort(A[1...n], p, r):
2      while p < r do
3          q = partition(A, p, r)
4          if q < (p + r) / 2 then
5              ModifiedTRQuickSort(A, p, q - 1)
6              p = q + 1
7          elif
8              ModifiedTRQuickSort(A, q + 1, r)
9              r = q - 1
10         end if
11     end while
```

$ModifiedTRQuickSort$ recurses on the interval that is smaller, which is at most half of the length of the current interval. Thus, the length of the interval is reduced by at least half on each recursive call. The stack depth, is $\Theta(\lg n)$ in the worst case.

5 Problem 5

(a)

We can solve this problem step by step.

5.1 Algorithm

Algorithm 1 Sort By SqrtSort

Require: n.

```
for i = 2√n − 1 → 1 do
    for k = 1 → i do
        SqrtSort((k − 1)√n/2)
```

5.2 Number of Call SqrtSort

$$\sum_{i=1}^{2\sqrt{n}-1} i = 2n - \sqrt{n} = \Theta(n)$$

5.3 Correctness

The loop invariant for the inner loop can be stated as follows:

at the end of k_{th} iteration, A[$k\frac{\sqrt{n}}{2}$+1...$(k+1)\frac{\sqrt{n}}{2}$] are sorted largest $\frac{\sqrt{n}}{2}$ elements in A[1..$(k+1)\frac{\sqrt{n}}{2}$].

basis At the end of the first iteration, A[1...$\sqrt{n}$] are sorted, so A[$k\frac{\sqrt{n}}{2}$ + 1...$(k+1)\frac{\sqrt{n}}{2}$] are sorted largest $\frac{\sqrt{n}}{2}$ elements in A[1..$(k+1)\frac{\sqrt{n}}{2}$].

inductive step Assume at the start of the $k+1_{th}$ iteration, A[$k\frac{\sqrt{n}}{2}$ + 1...$(k+1)\frac{\sqrt{n}}{2}$] are sorted largest $\frac{\sqrt{n}}{2}$ elements in A[1..$(k+1)\frac{\sqrt{n}}{2}$], and we call SqrtSort to sort A[$k\frac{\sqrt{n}}{2}$ + 1...$(k+2)\frac{\sqrt{n}}{2}$], so A[$(k+1)\frac{\sqrt{n}}{2}$ + 1...$(k+2)\frac{\sqrt{n}}{2}$] are sorted largest $\frac{\sqrt{n}}{2}$ elements in A[1..$(k+2)\frac{\sqrt{n}}{2}$].

So at the end of inner loop, A[$i\frac{\sqrt{n}}{2}$+1...$(i+1)\frac{\sqrt{n}}{2}$] are sorted largest $\frac{\sqrt{n}}{2}$ elements in A[1..$(i+1)\frac{\sqrt{n}}{2}$].

The loop invariant for the outer loop can be stated as follows:

at the end of $2\sqrt{n} - i_{th}$ iteration, A[$i\frac{\sqrt{n}}{2}$ + 1...n] are sorted largest elements in array A.

basis At the end of the first iteration, A[$n - \frac{\sqrt{n}}{2}$ + 1...n] are sorted largest elements in array $A[1...n]$.

inductive step Assume at the start of the $2\sqrt{n} - i + 1_{th}$ iteration, A[$i\frac{\sqrt{n}}{2}$ + 1...n] are sorted largest elements in array A, and the inner loop make A[$(i-1)\frac{\sqrt{n}}{2}$...$i\frac{\sqrt{n}}{2}$] are sorted largest $\frac{\sqrt{n}}{2}$ elements in A[1..$i\frac{\sqrt{n}}{2}$], so A[$(i-1)\frac{\sqrt{n}}{2}$ + 1...n] are sorted largest elements in array A.

So at the end of outer loop, A[$\frac{\sqrt{n}}{2}$ + 1...n] are sorted, and the last call is SqrtSort(0), so A[1...n] are sorted.

5.4 Remark

We can prove that algorithm with lower complexity on call SqrtSort is impossible.

The number of inversions can be $\Theta(n^2)$, and one call can decrease $\Theta((\sqrt{n})^2 = n)$, so we can say this algorithm has lowest complexity on call SqrtSort.

(b)

$T(n) = (2n - \sqrt{n})T(\sqrt{n}) + \Theta(n)$
We can get $T(n) = \Theta(n^2 logn)$

Proof. We prove it by induction.

Base case: When $n = 1$, $T(n) = \Theta(1)$, meet the conditions.

Induction: When $n = k^2$, $T(n) = (2n - \sqrt{n})T(\sqrt{n}) + \Theta(n) = (2n - \sqrt{n})\Theta(n)\frac{logn}{2} + \Theta(n) = \Theta(n^2 logn)$.

Thus, the induction hypothesis holds. □

6 Problem 6

(a) Suppose *OneInThree* returns 1 with probability p, then

$$p = \frac{1}{2}(1-p)$$

then

$$p = \frac{1}{3}$$

(b) Let X denotes the number of times that this algorithm calls *FairCoin*, then

$$\mathbb{E}(X) = 1 + \frac{1}{2} \times 0 + \frac{1}{2}\mathbb{E}(X)$$

so

$$\mathbb{E}(X) = 2$$

(c)

Algorithm 2 OneInTwo

```
1: a = BiasedCoin()
2: b = BiasedCoin()
3: if a = 1 and b = 0 then
4:     return 0
5: else if a = 0 and b = 1 then
6:     return 1
7: else
8:     return OneInTwo()
```

(d) Let Y denotes the number of times that this algorithm calls *BiasedCoin*, then

$$\mathbb{E}(Y) = 2 + 2p(1-p) \times 0 + \left(p^2 + (1-p)^2\right)\mathbb{E}(Y)$$

so

$$\mathbb{E}(Y) = \frac{1}{p(1-p)}$$

7 Problem 7

7.1 Upper Bound

Suppose we ask questions like 'whether the number is not less than x'. Let the answer be $ge(x)$. Obviously, in the worst case, this algorithm needs $\lceil \log_2 1000000 \rceil = 20$ questions.

Algorithm 3 GuessNumber

```
1: l = 1
2: r = 1000000
3: while l < r do
4:     mid = ⌈(l + r)/2⌉
5:     if ge(mid) = yes then
6:         l = mid
7:     else
8:         r = mid - 1
9:     return l
```

7.2 Lower Bound

Since only yes/no questions are allowed, so we can use a binary decision tree to describe any algorithm on this problem.

- Each internal node denotes a query
- Each internal node has two outgoing edges, corresponding to yes/no

It's obvious that the decision tree has at least 1000000 leaves, so the height of decision tree h satisfies

$$2^h \geq 1000000$$

so h is at least

$$\lceil \log_2 1000000 \rceil = 20$$

7.3 Remark

- To give an upper bound of an algorithm, we only need to give an algorithm and prove its complexity.
- To give a lower bound of a problem, we need to prove any algorithm costs more time than this bound.
- **The lower bound of a problem is different from the best case of an algorithm of this problem**.
- In most cases, it's much harder to prove the lower bound of a problem than the upper bound. Decision tree and adversary argument are useful when proving lower bound of a problem.

8 Bonus

We already know that the runtime of $HeapSort$ is always $O(n \log n)$, and all that remains to be done is to prove that the runtime of $HeapSort$ is always $\Omega(n \lg n)$ if the elements are distinct.

The max heap building operation run once and is $O(n)$ in performance. Then we get a complete binary tree of size n. Find the largest k that satisfies $n/2 < 2^k - 1 \leq n$. After $n - 2^k + 1$ times of swap (the first and the last element) and sift down operation, we get a perfect tree of size $2^k - 1$. We will show that the rest of the runtime of $HeapSort$ is still $\Omega(n \lg n)$.

A perfect tree of size $2^k - 1$ has 2^{k-1} leaves. We will prove that at most half of largest 2^{k-1} elements lies on leaves initially. We prove by contradiction. The ancestors of a largest 2^{k-1} element must be another largest 2^{k-1} element. If there are $2^{k-1} + 1$ elements lies on leaves, there exist at least $(2^{k-2} + 1) + (2^{k-2} + 1) + ... + (1 + 1) + 1 \geq 2^{k-1}$ largest 2^{k-1} elements in the tree, which is a contradiction.

After 2^{k-1} times of swap (the first and the last element) and sift down operation, all the largest 2^{k-1} elements has popped from the max heap. The runtime of a largest 2^{k-1} elements that does not lie on a leaf initially moving to the top is $d - 1$. Here d is its depth. In the best case, half of the largest 2^{k-1} elements does not lie on a leaf. The total runtime of moving them to the top is at least

$$\sum_{i=1}^{k-2}(i-1)2^{i-1} = \Omega(k2^k)$$

Thus, the runtime of $Heapsort$ is always $\Omega(n \lg n)$ if the elements are distinct.