# 论题 2-7 作业

姓名：陈劭源　　学号：161240004

## 1 [TC] Problem 7.1-2

The procedure returns $r$ when all elements in the array have the same value.

```
MODIFIED-PARTITION(A, p, r)
 1  x = A[r]
 2  i = p − 1
 3  for j = p to r − 1
 4      if A[j] ≤ x
 5          i = i + 1
 6          exchange A[i] with A[j]
 7  exchange A[i + 1] with A[r]
 8  j = i
 9  while j ≥ p and A[j] == A[i]
10      j = j − 1
11  return ⌊(i + j)/2⌋
```

## 2 [TC] Problem 7.2-4

We have already proved that the asymptotically running time of INSERTION-SORT is $\Theta(n+I)$, where $n$ is the length of the input array and $I$ is the number of inversions in the input array. There are not too many inversions in an almost-sorted array, so the procedure INSERTION-SORT performs well for an almost-sorted array. However, in QUICKSORT, if we use the first element as a pivot to partition an almost-sorted array, the split is quite unbalanced, so it performs bad for an almost-sorted array. Even if we use the randomized version of quicksort, the best running time is $\Theta(n \lg n)$. Therefore, the procedure INSERTION-SORT would tend to beat the procedure QUICKSORT on this problem.

## 3 [TC] Problem 7.3-2

The number of calls to the number generator equals to the number of calls to the procedure PARTITION.

Best case: $\Theta(n)$

Worst case: $\Theta(n)$

## 4 [TC] Problem 7.4-2

Let $T(n)$ denote the best-case time for the procedure QUICKSORT on an input of size $n$. We have the recurrence

$$T(n) = \min_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

We guess that $T(n) \ge cn \lg n$ for some constant $c$. Substituting it into the recurrence, we obtain

$$\begin{aligned}
T(n) &\ge \min_{0 \le q \le n-1} c(q \lg q + (n-q-1)\lg(n-q-1)) + \Theta(n) \\
&\ge 2c\frac{n-1}{2}\lg\frac{n-1}{2} + \Theta(n) \\
&= c\frac{n-1}{\lg}\frac{n-1}{2} + \Theta(n) \\
&= cn\lg n + cn\lg\frac{n-1}{n} - c\lg(n-1) - c(n-1) + \Theta(n) \\
&\ge cn\lg n - cn - cn - cn + \Theta(n) \\
&= cn\lg n - 3cn + \Theta(n) \\
&\ge cn\lg n
\end{aligned}$$

The second inequality is obtained by differentiating with respect to $q$. The third inequality holds when $n > 2$. Assuming that $\Theta(n) \ge c_0 n$ for sufficiently large $n$, the last inequality holds if we choose $c < c_0/3$.

Therefore, the best-case running time of quicksort is $\Omega(n \log n)$.

## 5 [TC] Problem 7.4

***a.*** We use mathematical induction to prove the correctness of the algorithm.

For the base step, the length of array is 0 or 1, i.e. $p \ge r$, the procedure does nothing and the array is sorted.

For the induction step, assume that the process sorts any array of length $k < n$ correctly. For array of length $n$, the procedure partitions the array into two parts, and every element in the left subarray is less than or equal to every element in the right subarray. Then the procedure recursively calls it self, making the left subarray sorted, by induction hypothesis. The getsment $p = q + 1$ updates the parameters of the procedure, and then it jumps to line 1, in order to sort the right subarray, and by induction hypothesis, the right subarray can be sorted correctly. Since every element in the left subarray is less than or equal to every element in the right subarray, the whole array is sorted.

By mathematical induction, TAIL-RECURSIVE-QUICKSORT$(A, 1, A.length)$ correctly sorts the array $A$ when it terminates. For each iteration of **while** and each recursive call to TAIL-RECURSIVE-QUICKSORT, $r - p$ decreases, so the procedure terminates for any valid input. Therefore, this algorithm is totally correct.

***b.*** When the procedure PARTITION always produces a left subarray of length $n-1$ and a right subarray of length 0, the procedure TAIL-RECURSIVE-QUICKSORT will be recursively called for $n$ times, and the stack depth is $\Theta(n)$.

***c.*** Always let the recursive call sort the smaller subarray. Since the length of the smaller subarray is less than half of the length of the original array, the worst-case stack depth is $\Theta(\lg n)$.

```
MODIFIED-RECURSIVE-QUICKSORT(A, p, r)
1  while p < r
2      q = PARTITION(A, p, r)
3      if q < (p + r)/2
4          TAIL-RECURSIVE-QUICKSORT(A, p, q − 1)
5          p = q + 1
6      else
7          TAIL-RECURSIVE-QUICKSORT(A, q + 1, r)
8          r = q − 1
```

# 6 [TC] Problem 7.5

***a.***

$$p_i = (i-1)(n-i)/\binom{n}{3} = \frac{6(i-1)(n-i)}{n(n-1)(n-2)}$$

***b.***

$$\begin{aligned}\lim_{n\to\infty} p_{\lfloor (n+1)/2\rfloor}/(1/n) &= \lim_{n\to\infty} \frac{6(\lfloor (n+1)/2\rfloor - 1)(n - \lfloor (n+1)/2\rfloor)}{(n-1)(n-2)} \\ &= \lim_{n\to\infty} \frac{6(n-1)^2}{4(n-1)(n-2)} \\ &= 3/2\end{aligned}$$

So the likelihood is improved by 50% compared with the ordinary implementation.

***c.***

$$\begin{aligned}\sum_{i=n/3}^{2n/3} p_i/(n/3) &= (3/n)\sum_{i=n/3}^{2n/3} \frac{6(i-1)(n-i)}{n(n-1)(n-2)} \\ &\approx \frac{18}{n(n-1)(n-2)} \int_{n/3}^{2n/3} (-i^2 + (1+n)i - n)\mathrm{d}i \\ &= \frac{(13n-27)n}{9(n-1)(n-2)} \\ \lim_{n\to\infty} \sum_{i=n/3}^{2n/3} p_i/(n/3) &= \frac{13}{9} \approx 1.44\end{aligned}$$

So the likelihood of getting a good split is improved by 44.4% compared with the ordinary implementation when $n \to \infty$.

***d.*** For median-of-3 method, the best case is still that PARTITION produces two subarray of the same size, and it takes a running time of $\Theta(n \lg n)$. Therefore, the running time of the median-of-3 method is still $\Omega(n \lg n)$, i.e. it only affects the constant factor.

## 7 [TC] Problem 8.1-3

Assume there exists such algorithm. Consider a decision tree of this algorithm, operating on half of the $n!$ inputs of length $n$, and for these inputs, the running time of the algorithm is linear. This decision tree has $n!/2$ leaves, and its height is at least $\lg(n!/2)$, i.e. $\Theta(n \log n)$, by the property of a binary tree. However, the algorithm sorts the $n!/2$ inputs in a linear time. That means, there exists $N$, for every $n > N$, the number of the comparisons the algorithm make is less than the height of the decision tree, which leads to a contradiction.

For a fraction of $1/n$ and $1/2^n$, the corresponding minimum heights of the decision trees are $\lg((n-1)!/2)$ and $\lg(n!/2^n)$, i.e. $\Theta(n \log n)$ and $\Theta(n \log n)$, which are still impossible. Therefore, such algorithm does not exist.

## 8 [TC] Problem 8.1-4

In this problem, there are $(k!)^{n/k}$ possible inputs, so the height of the decision tree is at least $\lg(k!)^{n/k}$, i.e. $\Theta(n \lg k)$, as $n$ and $k$ grow large. That means, at least $\Theta(n \lg k)$ comparisons are needed to solve this problem. Therefore, the lower bound on the number of comparisons is $\Omega(n \lg k)$.

## 9 [TC] Problem 8.2-4

Let $C[0..k]$ be a new array

PREPROCESS$(A)$

```
1  for i = 0 to k
2      C[i] = 0
3  for i = 1 to A.length
4      C[A[i]] = C[A[i]] + 1
5  for i = 1 to k
6      C[i] = C[i] + C[i − 1]
```

QUERY$(a, b)$

```
1  if a == 0
2      return C[b]
3  else
4      return C[b] − C[a − 1]
```

## 10 [TC] Problem 8.3-4

First, convert these integers to base-$n$ notation. Then, use RADIX-SORT to sort these integers. By Lemma 8.3, the running time of RADIX-SORT is $\Theta(d(n+k))$, where $d = \log_n n^3 = 3$ and $k = n$. Therefore, we sort these $n$ integers in $O(n)$ time.

## 11 [TC] Problem 8.4-2

When all the elements are distributed to the same bucket, it takes $\Theta(n^2)$ running time for insertion sort, which is the worst-case running time.

Using a sorting algorithm with worst-case time complexity $O(n \lg n)$, such as merge sort, instead of insertion sort, will make its worst running time $O(n \lg n)$.

## 12 [TC] Problem 8.2

***a.*** Counting sort.

***b.*** Partitioning with the pivot $1/2$ .

***c.*** Insertion sort.

***d.*** Counting sort can be used, because its stable, and it takes a running time of $O(n+2) = O(n)$ for each bit, so RADIX-SORT sorts $n$ records with $b$-bit keys in $O(bn)$ time.

For partitioning, it is not stable. For insertion sort, it can't ensure a running time of $O(n)$ for each iteration of RADIX-SORT.

***e.*** The modified version is shown below.

MODIFIED-COUNTING-SORT$(A, B, k)$

```
1  let C[0..k] be a new array
2  for i = 0 to k
3      C[i] = 0
4  for j = 1 to A.length
5      C[A[j]] = C[A[j]] + 1
6  for i = 1 to k
7      C[i] = C[i] + C[i − 1]
8  i = 1
9  while i < A.length
10     if C[A[i] − 1] < i and i ≤ C[A[i]]
11         i = i + 1
12     else
13         C[A[i]] = C[A[i]] − 1
14         exchange A[i] with A[C[A[i]] + 1]
```

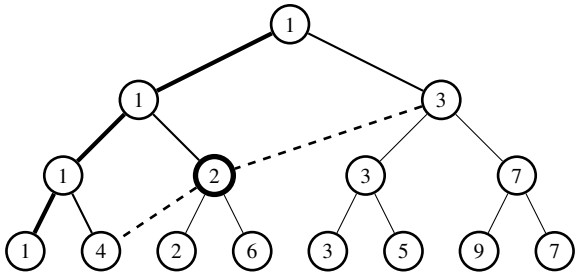
This algorithm is not stable.

## 13 [TC] Problem 9.1-1

Compare all the elements in pair. Then compare the winners in pair. Repeating this until there is only one element left. This is the smallest element. Now we have built a tree of comparisons (this is a complete

binary tree). Note that the second smallest element must have been compared to the smallest element and lost. Now we can design an algorithm to find the second smallest element. We just need to find the minimum of the elements who were once compared to the smallest element, i.e. the offsprings of the smallest element in the tree, except the smallest element itself.

To find the smallest element, the number of comparisons we made satisfied the recurrence $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$ with $T(1) = 0$, whose solution is $T(n) = n - 1$. To find the second smallest element, we have to make at most $h - 2$ comparisons, where $h$ is the height of the tree. For a complete binary tree with $n$ leaves, the height is $\lceil \lg n \rceil + 1$. Therefore, we have to make $\lceil \lg n \rceil - 1$ comparisons in this step. In all, we have made at most $n + \lceil \lg n \rceil - 2$ comparisons to find the second smallest element of $n$ elements.




## 14 [TC] Problem 9.3-5

Find the median of the array, and use the median as the pivot to partition the array. Determine whether the element you want to find is in the left or right subarray, and calculate its position in the subarray. Recursively doing this, we will find the element we want.

The running time satisfies the recurrence:

$$T(n) = T(n/2) + \Theta(n)$$

By the master theorem we get $T(n) = \Theta(n)$.

## 15 [TC] Problem 9.3-7

First, calculate the range of the $k$ numbers. For example, the numbers we want can be ranging from the $\lfloor (n-k)/2 \rfloor$-th smallest to the $\lfloor (n-k)/2 \rfloor$ smallest elements. Find the $\lfloor (n-k)/2 \rfloor$-th smallest element in $O(n)$ time, and use it as the pivot to partition the array. Then find the $k$-th smallest element in the right subarray and partition, and the left subarray of the right subarray, along with two pivots, contains the $k$ elements we want. The total running time is $O(n)$.