7.1-2

What value of q does PARTITION return when all elements in the subarray $A[p:r]$ have the same value? Modify PARTITION so that $q = \lfloor (p+r)/2 \rfloor$ when all elements in the subarray $A[p:r]$ have the same value.

i will run up to r-1-1, PARTITION(A,p,r) will return i+1 = r-1-1+1 = r-1

```
PARTITION(A,p,r)
    x = A[r]
    i = p - 1
    flag = LEFT
    for j = p to r-1
        if A[j] < x or (A[j] == x and flag == LEFT)
            if A[j] == x
                flag = RIGHT
            i = i+1
            exchange A[i] with A[j]
        else if A[j] == x
            flag = LEFT
    exchange A[i+1] with A[r]
    return i+1
```

7.2-2

What is the running time of QUICKSORT when all elements of array A have the same value?

$T(n) = T(n-1) + \Theta(n) \Rightarrow T(n) = \Theta(n^2)$

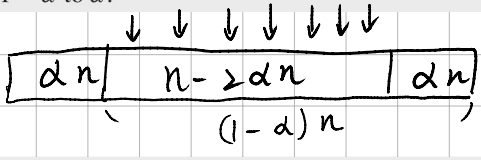
7.2-4

Banks often record transactions on an account in order of the times of the transactions, but many people like to receive their bank statements with checks listed in order by check number. People usually write checks in order by check number, and merchants usually cash them with reasonable dispatch. The problem of converting time-of-transaction ordering to check-number ordering is therefore the problem of sorting almost-sorted input. Explain persuasively why the procedure INSERTION-SORT might tend to beat the procedure QUICKSORT on this problem.

The time complexity of Insertion-Sort could be $O(n)$ and the best case time complexity of QuickSort is $\Theta(n \lg n)$

7.2-6

Consider an array with distinct elements and for which all permutations of the elements are equally likely. Argue that for any constant $0 < \alpha \leq 1/2$, the probability is approximately $1 - 2\alpha$ that PARTITION produces a split at least as balanced as $1 - \alpha$ to α.

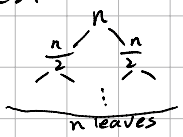


$P = \frac{n-2\alpha n}{n} = 1-2\alpha$

7.3-2

When RANDOMIZED-QUICKSORT runs, how many calls are made to the random-number generator RANDOM in the worst case? How about in the best case? Give your answer in terms of Θ-notation.

Best case :



2n-1 nodes $\Rightarrow \Theta(n)$

worst case :

n-1 internal nodes $\Rightarrow \Theta(n)$

7-1 *Hoare partition correctness*

The version of PARTITION given in this chapter is not the original partitioning algorithm. Here is the original partitioning algorithm, which is due to C. A. R. Hoare.

```
HOARE-PARTITION(A, p, r)
 1  x = A[p]
 2  i = p − 1
 3  j = r + 1
 4  while TRUE
 5      repeat
 6          j = j − 1
 7      until A[j] ≤ x
 8      repeat
 9          i = i + 1
10      until A[i] ≥ x
11      if i < j
12          exchange A[i] with A[j]
13      else return j
```

a. Demonstrate the operation of HOARE-PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$, showing the values of the array and the indices i and j after each iteration of the **while** loop in lines 4–13.

1st :

$x = A[p] = 13$

$\langle 13, \underset{i}{19}, 9, 5, 12, 8, 7, 4, 11, 2, \underset{j}{6}, 21 \rangle$

$\rightarrow \langle 13, 6, 9, 5, 12, 8, 7, 4, 11, 2, 19, 21 \rangle$

2nd :

$\langle 13, 6, 9, 5, 12, 8, 7, 4, 11, \underset{j}{2}, \underset{i}{19}, 21 \rangle$

$j = 9, \ i = 10, \ j < i$

return $j = 9$

b. Describe how the PARTITION procedure in Section 7.1 differs from HOARE-PARTITION when all elements in $A[p:r]$ are equal. Describe a practical advantage of HOARE-PARTITION over PARTITION for use in quicksort.

In PARTITION :

might arbitrarily place all equal elements to one side of the partition. (imbalanced)

In HOARE-PARTITION :

ensures that it returns an index such that one side contains all the elements less than or equal to the pivot, and the other side is empty.

Advantage :

The HOARE-PARTITION ensures balanced partition, and that reduces the risk of worst-case scenario, leading to a better time complexity.

The next three questions ask you to give a careful argument that the procedure HOARE-PARTITION is correct. Assuming that the subarray $A[p:r]$ contains at least two elements, prove the following:

c. The indices i and j are such that the procedure never accesses an element of A outside the subarray $A[p:r]$.

d. When HOARE-PARTITION terminates, it returns a value j such that $p \le j < r$.

e. Every element of $A[p:j]$ is less than or equal to every element of $A[j+1:r]$ when HOARE-PARTITION terminates.

C. When $i \ge j$, the procedure returns j.

when $i < j$, $p \le i < j \le r$, p is min, r is max

so the procedure never accesses an element outside $A[p:r]$.

d.

When all elements of $A[p+1:r]$ are bigger than $A[p]$ it returns the smallest value of j with $i \geq j$, returning $j=p$.
When all elements of $A[p+1:r]$ are smaller than $A[p]$ it returns the biggest value of j.
The first loop : $j=r$, $i=p$, exchange.
The second loop : $j=r+1$, $i=r$, and $i \geq j$
$\Rightarrow$ return $j=r-1$, $p \leq j \leq r-1$

f.

```
QUICKSORT (A, p, r)
    if p < r
        q ← HOARE-PARTITION(A, p, r)
        QUICKSORT (A, p, q)
        QUICKSORT (A, q+1, r)
```

d. Show that

$$\sum_{q=1}^{n-1} q \lg q \leq \frac{n^2}{2} \lg n - \frac{n^2}{8} \tag{7.4}$$

for $n \geq 2$. (*Hint:* Split the summation into two parts, one summation for $q = 1, 2, \ldots, \lceil n/2 \rceil - 1$ and one summation for $q = \lceil n/2 \rceil, \ldots, n-1$.)

$$\sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q \lg q + \sum_{q=\lceil \frac{n}{2} \rceil}^{n-1} q \lg q \leq \lg\left(\frac{n}{2}\right) \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q + \lg n \sum_{q=\lceil \frac{n}{2} \rceil}^{n-1} q$$

$$= \lg n \sum_{q=1}^{n-1} q - \lg 2 \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q \leq \lg n \cdot \left(\frac{(n-1)\cdot n}{2}\right) - \left(\frac{\left(\frac{n}{2}\right)-1}{2}\right) \cdot \left(\frac{n}{2}\right)$$

$$= \frac{n^2 \lg n}{2} - \frac{n^2}{8} - \frac{n}{4} \leq \frac{n^2}{2} \lg n - \frac{n^2}{8} \quad \#$$

8.1-1

What is the smallest possible depth of a leaf in a decision tree for a comparison sort?

compare $n-1$ times $\Rightarrow$ depth = $n-1$

8.1-4

You are given an n-element input sequence, and you know in advance that it is partly sorted in the following sense. Each element initially in position i such that $i \bmod 4 = 0$ is either already in its correct position, or it is one place away from its correct position. For example, you know that after sorting, the element initially in position 12 belongs in position 11, 12, or 13. You have no advance information about the other elements, in positions i where $i \bmod 4 \neq 0$. Show that an $\Omega(n \lg n)$ lower bound on comparison-based sorting still holds in this case.

There are $3^{\frac{n}{4}}$ ways to place $\frac{n}{4}$ elements and $(\frac{3n}{4})!$ to place the others, so the tree height is $\lg(3^{\frac{n}{4}} \cdot (\frac{3n}{4})!) = \frac{n}{4}\lg 3 + \lg((\frac{3n}{4})!)$

$\geq \frac{n}{4}\lg 3 + \frac{3n}{8}\lg(\frac{3}{8}n) = \frac{n}{4}\lg 3 + \frac{3n}{8} \cdot \frac{3}{8} + \frac{3}{8}\lg n \geq \frac{3}{8} n \lg n$

$\Rightarrow \lg(3^{\frac{n}{4}} \cdot (\frac{3n}{4})!) = \Omega(n \lg n)$.

8.2-3

Suppose that we were to rewrite the **for** loop header in line 11 of the COUNTING-SORT as

11 **for** $j = 1$ **to** n

Show that the algorithm still works properly, but that it is not stable. Then rewrite the pseudocode for counting sort so that elements with the same value are written into the output array in order of increasing index and the algorithm is stable.

The sequence of j doesn't influence $B[C[A[j]]] = A[j]$, but it'll change the sequence of $A[j]$ that moved in to $B \Rightarrow$ unstable!

```
COUNTING-SORT (A, n, k)
    for i=0 to k
        C[i] =0
    for j=1 to n
        C[A[j]] = C[A[j]] +1
    D = C
    for i=1 to k
        C[i] = C[i] + C[i+1]
    for j=1 to n
        B[C[A[j]] - D[A[j]]] = A[j]
        D[A[j]] = D[A[j]] - 1
```

8.2-4

Prove the following loop invariant for COUNTING-SORT:

> At the start of each iteration of the **for** loop of lines 11–13, the last element in A with value i that has not yet been copied into B belongs in $B[C[i]]$.

Initialization:

No elements in A are occupied into B.

The last element in A with value i has not yet been copied into B

The last element in A with value i is in B[C[i]]

Maintanance:

A[j] is in B[C[i]], let m be the index of the rightmost element of A with value i before A[j], A[m] goes into B[C[i]-1].

In the next iteration. C[i] = C[i]-1 → B[C[i]-1]

Termination:

Each element of A has been copied into B correctly.

8.2-5

Suppose that the array being sorted contains only integers in the range 0 to k and that there are no satellite data to move with those keys. Modify counting sort to use just the arrays A and C, putting the sorted result back into array A instead of into a new array B.

```
PROCEDURE (A, n, k)
    for i = 0 to k
        C[i] = 0
    for j = 1 to n
        C[A[j]] = C[A[j]] + 1
    index = 1
    for i = 0 to k
        for j = 1 to C[i]
            A[index] = i
            index = index + 1
    return A
```

8.3-1
Using Figure 8.3 as a model, illustrate the operation of RADIX-SORT on the following list of English words: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

8-2 Sorting in place in linear time

You have an array of n data records to sort, each with a key of 0 or 1. An algorithm for sorting such a set of records might possess some subset of the following three desirable characteristics:

1. The algorithm runs in $O(n)$ time.
2. The algorithm is stable.
3. The algorithm sorts in place, using no more than a constant amount of storage space in addition to the original array.

a. Give an algorithm that satisfies criteria 1 and 2 above.

b. Give an algorithm that satisfies criteria 1 and 3 above.

c. Give an algorithm that satisfies criteria 2 and 3 above.

d. Can you use any of your sorting algorithms from parts (a)–(c) as the sorting method used in line 2 of RADIX-SORT, so that RADIX-SORT sorts n records with b-bit keys in $O(bn)$ time? Explain how or why not.

e. Suppose that the n records have keys in the range from 1 to k. Show how to modify counting sort so that it sorts the records in place in $O(n + k)$ time. You may use $O(k)$ storage outside the input array. Is your algorithm stable?

a. counting-sort

b. partition

c. bubble sort

d. counting sort could be since linear time complexity. Sorting n records with b-bit keys in $O(bn)$

e. INPLACE-COUNTING-SORT(A, n, k)

```
for i=1 to k
  C[i] = 0
for j=1 to n
  C[A[j]] = C[A[j]] + 1
for i=2 to k
  C[i] = C[i] + C[i-1]
```

```
for i = k downto 2
    while C[i] > C[i-1]
        A[C[i]] = i
        C[i] = C[i] - 1
while C[1] > 0
    A[C[1]] = 1
    C[1] = C[1] - 1
```

Time complexity = $O(n+k)$, not stable ✱

8-3 *Sorting variable-length items*

a. You are given an array of integers, where different integers may have different numbers of digits, but the total number of digits over *all* the integers in the array is n. Show how to sort the array in $O(n)$ time.

b. You are given an array of strings, where different strings may have different numbers of characters, but the total number of characters over all the strings is n. Show how to sort the strings in $O(n)$ time. (The desired order is the standard alphabetical order: for example, `a` < `ab` < `b`.)

a. $(O(n))$

First, sort the integers by digits of a integer. Then, sort the integers that have the same digits by Radix sort. Let the integers called m_i, where i is the digits of m_i, m_i is the total numbers of integers with i digits.

Time Complexity : $\sum_{i=1}^{n} i \cdot m_i$ since $\sum_{i=1}^{n} i \cdot m_i$ equals the total number of digits over all integers.

$\sum_{i=1}^{n} i \cdot m_i = O(n)$ ✱

b.

First, take the character of every strings to sort. $(O(n))$. Then, sort the strings with the same leading character with the length. $(O(n))$
Lastly, sort the strings with same length.
let the complexity be $f(n)$, $f(n) <$ the total length of all strings $\Rightarrow$ time complexity $= O(n)$ ✱