Sample Solution for Problem Set 4

Data Structures and Algorithms, Fall 2019

October 18, 2019

7.3-2 (CLRS 3ed). Denote the sorted array as $A[1...n]$ (i.e., the algorithm receives an input array and outputs $A[1...n]$, where $A[1] \leq A[2] \leq \cdots \leq A[n]$). Let X_i be an indicator random variable taking value one iff $A[i]$ has chosen to be pivot during algorithm execution. Obviously the number of times for calling RANDOM equals to $\sum_{i=1}^{n} X_i \leq n$. (Recall each item in the array can be pivot at most once.)

Now we prove the following property: for each $1 \leq i \leq n-1$, X_i and X_{i+1} cannot both be zero. This can be proved by induction on n: for $n = 2$ it is true since one of $A[1]$ and $A[2]$ must be chosen as pivot. Suppose for $n < k$ it is true, we prove it for $n = k$. Recall during execution, the algorithm will chose a pivot—assume $A[i]$ is chosen—and split the input array into two parts, which by the induction hypothesis both fit the property. Meanwhile, we also know that $A[i-1]$ and $A[i]$ are not both zero, and $A[i]$ and $A[i+1]$ are not both zero. This completes the proof of the inductive step.

So each pair of $(X_1, X_2), (X_3, X_4), \cdots$ contributes one to the sum. That means $\sum_{i=1}^{n} X_i \geq \lfloor n/2 \rfloor$. As a result, the upper bound and the lower bound are both $\Theta(n)$.

7.4-2 (CLRS 3ed). Let $T(n)$ be the best-case time for the procedure QUICKSORT on an input of size n, we have

$$T(n) = \min_{0 \leq q \leq n-1} (T(q) + T(n-q-1)) + kn$$

where k is a constant. Suppose that $T(n) \geq cn \lg n$ for some constant c, then for $n > 10$ we have

$$\begin{aligned} T(n) &\geq \min_{0 \leq q \leq n-1} (cq \lg q + c(n-q-1) \lg(n-q-1)) + kn \\ &\geq c(n-2) \lg(n/2-1) + kn \\ &\geq c(n-2)(\lg n - 2) + kn \\ &\geq cn \lg n + 4c - 2c \lg n + (k-2c)n \end{aligned}$$

Notice, the minimum case is half-half split, as $f(x) = n \log x$ is concave. Therefore, when $c < k/2$, for sufficiently large n, we have $T(n) \geq cn \log n$, thus the lower bound is $\Omega(n \log n)$.

7.4-5 (CLRS 3ed). In the proposed algorithm, the recursion stops at level $\lg(n/k)$, which makes the runtime for partitioning sums to $O(n \lg(n/k))$, in expectation. However, this leaves n/k non-sorted, non-intersecting subarrays, each of length at most k. Since we use insertion sort to sort these subarrays, the total runtime for sorting these subarrays will be $(n/k) \cdot O(k^2) = O(nk)$.

In theory, if we ignore the constant factors and want this version of quicksort to beat the classical version, we need to ensure

$$n \lg n \geq nk + n \lg (n/k)$$

$$\lg n \geq k + \lg n - \lg k$$

$$\lg k \geq k$$

which clearly is not possible.

However, if we take the constant factors into consideration, we get

$$c_q n \lg n \geq c_i n k + c_q n \lg(n/k)$$

$$c_q \lg n \geq c_i k + c_q \lg n - c_q \lg k$$

$$\lg k \geq (c_i/c_q)k$$

which indicates that there might be a good candidate. (In fact, the lower-order terms should be taken into consideration too.)

In practice, k should be chosen by experiment.

7-2 (CLRS 3ed). **(a)** Each time the partition process will output one split: only the pivot is removed. That is, we get the recurrence $T(n) = T(n-1) + \Theta(n)$. As a result, the running time is $\Theta(n^2)$.

(b) See Figure 1.[1]

```
1:  procedure PARTITION'(A, p, r)
2:      i = l − 1, j = r, p = l − 1, q = r, v = A[r]
3:      if (l ≤ r) then
4:          return
5:      while (true) do
6:          while (A[i + 1] < v) do
7:              i = i + 1
8:          while (j > l and A[j − 1] > v) do
9:              j = j − 1
10:         if (i ≥ j) then
11:             break
12:         SWAP(A[i], A[j])
13:         if (A[i] == v) then
14:             p = p + 1
15:             SWAP(A[i], A[p])
16:         if (A[j] == v) then
17:             q = q − 1
18:             SWAP(A[j], A[q])
19:     SWAP(A[i], A[r])
20:     j = i − 1, i = i + 1
21:     for (k = l to p) do
22:         SWAP(A[k], A[j])
23:         k = k + 1, j = j − 1
24:     for (k = r − 1 down-to q) do
25:         SWAP(A[k], A[i])
26:         k = k − 1, i = i + 1
27:     return j, i
```

Figure 1

(c) See Figure 2.

(d) Similar to the analysis in Section 7.4.2, we calculate the expected number of comparisons. In particular, we focus on the comparisons occurred at line 6 and 8 in Figure 1, as the number of comparisons occurred in these two lines asymptotically dominate the overall cost of PARTITION'.

[1]The code is due to Robert Sedgewick and Jon Bentley. See `https://www.cs.princeton.edu/~rs/talks/QuicksortIsOptimal.pdf`.

```
1: procedure RANDOMIZED-PARTITION'(A, p, r)
2:     i = RANDOM(p, r)
3:     SWAP(A[r], A[i])
4:     return PARTITION'(A, p, r)
5: procedure QUICKSORT'(A, p, r)
6:     if p < r then
7:         (q, t) = RANDOMIZED-PARTITION'(A, p, r)
8:         QUICKSORT'(A, p, q − 1)
9:         QUICKSORT'(A, t + 1, r)
```

Figure 2

We calculate the number of comparisons between distinct elements and equal elements separately. For comparisons between distinct elements, the analysis in Section 7.4.2 still works. Since there are at most n distinct elements, we get a bound of $O(n \lg n)$ in expectation. As for comparisons between equal elements, notice that for an element with value v appearing $d_v > 1$ times, no comparisons among themselves occur until one of them is chosen as the pivot. (Thus, they also stay in same split during partition process, until one of them is chosen as the pivot.) Moreover, once some element with value v is chosen as pivot in some partition process, $O(d_v)$ comparisons among themselves occur in that partition process, and no further comparisons among themselves will ever occur. Therefore, the total number of comparisons among equal elements is bounded by $O(n)$. At this point, we can conclude that in expectation, the total time consumption happened during partitioning is $O(n \lg n)$.

7-5 (CLRS 3ed). **(a)** p_i is the probability that a randomly selected subset of size three has the $A'[i]$ as its median element. We first compute the number of 3-set satisfying this condition. For each such 3-set, among the two other elements (one element has to be $A'[i]$), one has to be in $\{1, \cdots, i-1\}$, and the other has to be in $\{i+1, \cdots, n\}$. Therefore, there are $(i-1)(n-i)$ such 3-set. On the other hand, there are $\binom{n}{3}$ 3-set in total. As a result, we have

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

(b) If we let $i = \lfloor \frac{n+1}{2} \rfloor$, the previous result gets us an increase of about

$$\frac{6(\lfloor \frac{n-1}{2} \rfloor)(n - \lfloor \frac{n+1}{2} \rfloor)}{n(n-1)(n-2)} - \frac{1}{n}$$

In the limit n goes to infinity, we get

$$\lim_{n\to\infty} \frac{\frac{6(\lfloor \frac{n-1}{2} \rfloor)(n-\lfloor \frac{n+1}{2} \rfloor)}{n(n-1)(n-2)}}{\frac{1}{n}} = \frac{3}{2}$$

(c) For simplicity, suppose n is a multiple of 3. We will approximate the sum as an integral, so

$$\begin{aligned}\sum_{i=n/3}^{2n/3} p_i &\approx \int_{n/3}^{2n/3} \frac{6(-x^2+nx+x-n)}{n(n-1)(n-2)} dx \\ &= \frac{6(-7n^3/81 + 3n^3/18 + 3n^2/18 - n^2/3)}{n(n-1)(n-2)}\end{aligned}$$

which, in the limit n goes to infinity, is $13/27$. In contrast, in the original randomized quicksort implementation, the probability of getting a good split is $1/3$.

(d) Since the new algorithm only boosts the probability of having a good split by a constant factor, there still exists a reasonable chance that the same randomness used in the original algorithm will again lead us into a bad situation, resulting in $\Omega(n \lg n)$ runtime again.

8.1-1 (CLRS 3ed). As a sorted output, any two elements' relative ordering should be known. We could think it as a directed graph whose vertex set is the elements, and each edge's direction indicates the relative ordering between the two endpoints. Notice, to produce a sorted output, the algorithm must be able to construct such a graph. Moreover, the graph must be connected, hence there are at least $n-1$ edges. Since each comparison adds at most one new edge, the smallest possible depth is $n-1$.

8.1-4 (CLRS 3ed). There are n/k sub-sequences and each can be ordered in $k!$ ways. Therefore, there are $(k!)^{n/k}$ possible output permutations. This gives us a bound on the height of the decision tree for solving this problem:

$$(k!)^{n/k} \leq 2^h$$

Taking logarithm of both sides, we get:

$$h \geq \lg(k!)^{n/k} = (n/k) \cdot \lg(k!) = (n/k) \cdot \Theta(k \lg k) = \Theta(n \lg k)$$

8.2-4 (CLRS 3ed). The algorithm will begin by pre-processing exactly as COUNTINGSORT does in lines 1 through 9, so that $C[i]$ contains the number of elements less than or equal to i in the array. When being queried about how many integers fall into a range $[a, \cdots, b]$, simply compute and return $C[b] - C[a-1]$. (Let $C[-1] = 0$.)

8.4-4 (CLRS 3ed). The unit circle has area $\pi \cdot 1^2 = \pi$, and we split it into n regions, each having area π/n. Let $r_0, r_1, r_2, \cdots r_n$ be the radius of the rings that partition the unit circle. Thus $\pi(r_{i+1}^2 - r_i^2) = 1/n$, implying $r_i = \sqrt{i/n}$. As a result, we create buckets as below:

$$c_i = \{d \mid r_{i-1} < d \leq r_i\} \text{ for } i = 1, 2, \cdots, n$$

We can calculate the bucket k for a distance d in constant time, just take

$$k = \begin{cases} \lfloor d^2 n \rfloor + 1 & \text{if } d < 1 \\ n & \text{if } d = 1 \end{cases}$$

Since the points are uniformly distributed on the disk and each region has equal area, it is easy to verify this algorithm will run in expected time $\Theta(n)$.

8-3 (CLRS 3ed). **(a)** First, we create n buckets using $O(n)$ time. Then, we go through each input integer a_i, putting a_i into bucket j if a_i has j bits. This process in total takes $O(\sum_{i=1}^{n}(1+|a_i|)) = O(n)$ time. Assume bucket j now has l_j integers each of length j. We then sort within each bucket using radix sort, this process in total takes $O(\sum_{j=1}^{n}(j \cdot l_j)) = O(n)$ time. Finally, we just concatenate the sorted arrays obtained from each bucket.

(b) We can use the same idea as in part (a). The only difference is that now each 'bit' is a character in the alphabet, and each 'bit' can take b values instead of two. Here, b is the size of the alphabet. So long as b is a constant (i.e., $b = O(1)$), which it indeed is, the whole process will still finish within $O(n)$ time.

8-6 (CRLS 3ed). **(a)** There are $\binom{2n}{n}$ ways to divide $2n$ numbers into two sorted lists, each with n numbers.

(b) Due to part (a), we know any decision tree to merge two sorted size n lists must have at least $\binom{2n}{n}$ leaves, so it has height $h \geq \lg \binom{2n}{n}$. Using the fact $\sqrt{2\pi} \cdot \sqrt{n} \cdot (n/e)^n \leq n! \leq e \cdot \sqrt{n} \cdot (n/e)^n$, we can bound h in the following way:

$$\begin{aligned} h &\geq \lg \binom{2n}{n} = \lg((2n)!) - 2\lg(n!) \\ &\geq 2n\lg(2n) - 2n\lg e + \lg(\sqrt{2\pi n}) - 2n\lg n + 2n\lg e - 2\lg(e\sqrt{n}) \\ &= 2n + \lg(\sqrt{2\pi n}) - 2\lg(e\sqrt{n}) > 2n - 2\lg(e\sqrt{n}) \\ &= 2n - o(n) \end{aligned}$$

(c) Suppose two lists are $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$, further assume that a_t and b_k are consecutive in the sorted output. Consider the lists derived by swapping a_t with b_k, that is, $A' = (a_1, \ldots, a_{t-1}, b_k, a_{t+1}, \ldots, a_n)$ and $B' = (b_1, \ldots, b_{k-1}, a_t, b_{k+1}, \ldots, b_n)$. For any algorithm $\mathcal{A}$ that does not compare a_t and b_k directly, we can construct A and B such that $\mathcal{A}(A, B)$ and $\mathcal{A}(A', B')$ yield the same result. So algorithm $\mathcal{A}$ does not correctly solve the merge problem.

(d) Consider the two lists $(1, 3, \ldots, 2n-1)$ and $(2, 4, \ldots, 2n)$. Apply part (c) and we can obtain the desired $2n - 1$ bound.

Additional Problem One. In the following, all logarithms are 3-based.

(a) The proof is similar to the proof of Theorem 8.1 in CLRS.

Since any comparison between a bolt and a nut yields 3 possible outcomes (too big, too small or the same size), the decision tree of any algorithm is a ternary tree, as compared to a binary tree in a comparison based sorting algorithm.[2] Each leaf of the ternary tree corresponds to a possible matching, and the number of comparisons made by the algorithm is lower bounded by the height h of the tree.

Since there are $n!$ possible matching patterns, we have $3^h \geq n!$, implying $h \geq \lg(n!) = \Theta(n \lg n)$.

(b) Following the previous discussion, to find $n/2$ matching nut-bolt pairs, we need to reach a node in the previous decision tree where, for each leaf in the subtree rooted at this node, there exists a same set of $n/2$ matching nut-bolt pairs.

Let's say such a node is "potentially good". Further, we say a node is "good" if on its path back to the root there are no "potentially good" nodes. By definition, subtrees rooted at good nodes form a set of disjoint subtrees in the original decision tree. Now, for each subtree rooted at a good node, it has at most $(n/2)!$ leaves with distinct matching patterns, because at least half of the nut-bolt pairs are fixed. On the other hand, there are $n!$ leaves with distinct matching patterns in the original tree. Therefore, there are at least $n!/(n/2)! \geq (n/2)!$ such subtrees.

Suppose the largest distance from the root of the decision tree to a good node is l, then any algorithm finding at least $n/2$ matching nut-bolt pairs has a number of comparisons lower bounded by l. Since $3^l \geq (n/2)!$, we have $l \geq \lg((n/2)!) = \Omega(n \lg n)$.

(c) First we use the decision tree method to prove that $\Theta(k \lg n)$ is a lower bound, the idea is similar to that in part (b). Specifically, to find k pairs, we to need to reach a node in the decision tree such that each descent leaf of the node contains k fixed matching pairs. Let's say such a node is "potentially good". Further, we say a node is "good" if on its path back to the root there are no "potentially good" nodes. Once again, these "good" nodes form disjoint subtrees, each with at most $(n-k)!$ distinct leaves. So there are at least $n!/(n-k)!$ such subtrees. Assume the largest distance from the root to a good node is l, we know any algorithm finding k pairs has a runtime lower bound of l. Since $3^l \geq n!/(n-k)!$, we know $l \geq \lg(n!/(n-k)!) = \lg(n!) - \lg((n-k)!)$. Using Stirling's approximation, it is easy to verify $l = \Omega(k \lg n)$.

[2] Any algorithm solving the problem has to be comparison based, so we simply say "any algorithm" instead of "any comparison based algorithm".

Then we use a simple adversary argument to prove that more than $n/2$ tests are required, if we want to find at least one nut-bolt pair. Suppose the sizes of the bolts and nuts are initially not fixed, and there is a referee, who decides the sizes upon our queries. That is, whenever we make a query, the referee can reply any answer, as long as it's consistent with its previous answers.

Suppose our algorithm makes only $q \leq n/2$ queries, and the referee, upon any query we make, always replies that "the nut is too small compared with the bolt". In particular, assume $q' \leq q$ nuts are used in these q comparisons, and these q' nuts are the smallest q' nuts; also assume $q'' \leq q$ bolts are used in these q comparisons, and these q'' bolts are the largest q'' bolts. Since $q \leq n/2$, the referee's answers are legit. In such case, clearly we won't be able to identify a bolt-nut pair.

At this point, we know the time complexity lower bound is $\max\{\alpha k \lg n, n/2\}$, for some constant α. Since $\max\{\alpha k \lg n, n/2\} \geq (\alpha k \lg n + n/2)/2 = \Theta(k \lg n + n)$, the desired bound is obtained.

(d) We first give a recursive algorithm that finds *all* matching pairs. See Figure 3.

```
1:  procedure MATCHALL(N, B, n)
2:      Let Nl, Nr be empty list of nuts.
3:      Let Bl, Br be empty list of bolts.
4:      i ← RANDINT(1, n)          ▷ Gives a random integer from the interval [1, n].
5:      j ← 0
6:      for (k = 1 to n) do
7:          if (N[k] < B[i]) then
8:              Nl.append(N[k])
9:          else if (N[k] > B[i]) then
10:             Nr.append(N[k])
11:         else
12:             j ← k                          ▷ B[i] = N[j]
13:     for (k = 1 to n) do
14:         if (B[k] < N[j]) then
15:             Bl.append(B[k])
16:         else if (B[k] > N[j]) then
17:             Br.append(B[k])
18:     L ←MATCHALL(Nl, Bl, Nl.size)
19:     R ←MATCHALL(Nr, Br, Nr.size)
20:     return (B[i], N[j]) ∪ L ∪ R
```

Figure 3

The algorithm works as follows: Let N be a list of nuts, and B a list of bolts. We first pick a random bolt $B[i]$ as the pivot, and compare all nuts to it. Let the nut that matches $B[i]$ be $N[j]$. For other nuts, if $N[k]$ is smaller than $B[i]$, we put $N[k]$ into a new list Nl, otherwise we put it into Nr. Then we let $N[j]$ be another pivot and compare all bolts to $N[j]$. Since $B[i] = N[j]$, for all other bolts, we put the bolts smaller than $N[j]$ into Bl, and the remaining into Br. It then can be assumed that Nl.size $= Bl$.size, as well as Nr.size $= Br$.size, because the nuts and bolts are one-to-one matched originally. We then apply the algorithm recursively on (Nl, Bl) and (Nr, Br).

The expected time complexity of the above algorithm is $O(n \lg n)$, which follows from an analysis similar to that of randomized quicksort.

Now we give an algorithm that finds *at least* k pairs. Without loss of generality, we assume $k = o(n)$ (In case $k = \Theta(n)$, we simply invoke MATCHALL). Our algorithm has expected runtime $O(n + k \lg k)$. See Figure 4 for the pseudocode.

The code works as follows: If the number of input pairs is at most $2k$, then we directly find and output all matching pairs. Otherwise, we use the same process as in the previous algorithm, to find a pair $B[i] = N[j]$, and split the other bolts and nuts into Bl, Br, Nl, Nr. We then apply our algorithm

```
1: procedure MATCHK(N, B, n, k)
2:     if n ≤ 2k then
3:         return MATCHALL(N, B, n)
4:     Let Nl, Nr be empty list of nuts.
5:     Let Bl, Br be empty list of bolts.
6:     repeat
7:         Nl.clear(), Nr.clear(), Bl.clear(), Br.clear()
8:         i ← RANDINT(1, n)
9:         j ← 0
10:        for (k = 1 to n) do
11:            if (N[k] < B[i]) then
12:                Nl.append(N[k])
13:            else if (N[k] > B[i]) then
14:                Nr.append(N[k])
15:            else
16:                j ← k                                   ▷ B[i] = N[j]
17:        for (k = 1 to n) do
18:            if (B[k] < N[j]) then
19:                Bl.append(B[k])
20:            else if (B[k] > N[j]) then
21:                Br.append(B[k])
22:    until (Nl.size > k or Nl.size > k)
23:    if (Nl.size < Nr.size) then
24:        return MATCHK(Nl, Bl, Nl.size, k)
25:    else
26:        return MATCHK(Nr, Br, Nr.size, k)
```

Figure 4

recursively on the *smaller* list.

Let $T(n)$ be the expected runtime of the algorithm with n input pairs, we have:

$$T(n) \leq \begin{cases} \Theta(n \lg n) & \text{if } n \leq 2k \\ \Theta(n) + T(n/2) & \text{otherwise} \end{cases}$$

which implies $T(n) = O(n + k \log k) = O(n + k \log n)$.