A4

ddl: 2023/10/22 23:59

1 Alternative quicksort analysis

(From CLRS 7-3)

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2      q = RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, q − 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

An alternative analysis of the running time of randomized quicksort focuses on the expected running time of each individual recursive call to `RANDOMIZED-QUICKSORT`, rather than on the number of comparisons performed. As in the analysis of Section 7.4.2, assume that the values of the elements are distinct.

1. Argue that, given an array of size n, the probability that any particular element is chosen as the pivot is $1/n$. Use this probability to define indicator random variables $X_i = I\{i\text{th smallest element is chosen as the pivot}\}$. What is $\mathbb{E}[X_i]$?

2. Let $T(n)$ be a random variable denoting the running time of quicksort on an array of size n. Argue that

$$\mathbb{E}[T(n)] = \mathbb{E}\left[\sum_{q=1}^{n} X_q(T(q-1) + T(n-q) + \Theta(n))\right].$$

3. Show how to rewrite equation(7.2) as

$$\mathbb{E}[T(n)] = \frac{2}{n}\sum_{q=1}^{n-1}\mathbb{E}[T(q)] + \Theta(n). \tag{7.3}$$

4. Show that

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

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

5. Using the bound from equation (7.4), show that the recurrence in equation (7.3) has the solution $\mathbb{E}[T(n)] = O(n\lg n)$. (*Hint:* Show, by substitution, that $\mathbb{E}[T(n)] \le an\lg n$ for sufficiently large n and for some positive constant a.)

2 Stack depth for quicksort

(From CLRS 7-5)

```
QUICKSORT(A, p, r)
1  if p < r
2        // Partition the subarray around the pivot, which ends up in A[q].
3        q = PARTITION(A, p, r)
4        QUICKSORT(A, p, q − 1)  // recursively sort the low side
5        QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

The `QUICKSORT` procedure of Section 7.1 makes two recursive calls to itself. After `QUICKSORT` calls `PARTITION`, it recursively sorts the low side of the partition and then it recursively sorts the high side of the partition. The second recursive call in `QUICKSORT` is not really necessary, because the procedure can instead use an iterative control structure. This transformation technique, called **tail-recursion elimination**, is provided automatically by good compilers. Applying tail-recursion elimination transforms `QUICKSORT` into the `TRE-QUICKSORT` procedure.

```
TRE-QUICKSORT(A, p, r)
1  while p < r
2        // Partition and then sort the low side.
3        q = PARTITION(A, p, r)
4        TRE-QUICKSORT(A, p, q − 1)
5        p = q + 1
```

1. Argue that `TRE-QUICKSORT` correctly sorts the array.

   Compilers usually execute recursive procedures by using a **stack** that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. When a procedure is called, its information is **pushed** onto the stack, and when it terminates, its information is **popped**. Since we assume that array parameters are represented by pointers, the information for each procedure call on the stack requires stack space. The **stack depth** is the maximum amount of stack space used at any time during a computation.

2. Describe a scenario in which `TRE-QUICKSORT`'s stack depth is $\Theta(n)$ on an n-element input array.

3. Modify `TRE-QUICKSORT` so that the worst-case stack depth is $\Theta(\lg n)$. Maintain the $O(n \lg n)$ expected running time of the algorithm.

3 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.

Questions:

1. Give an algorithm that satisfies criteria 1 and 2 above.
2. Give an algorithm that satisfies criteria 1 and 3 above.
3. Give an algorithm that satisfies criteria 2 and 3 above.
4. Can you use any of your sorting algorithms from parts (1)–(3) 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.

```
RADIX-SORT(A, n, d)
1  for i = 1 to d
2      use a stable sort to sort array A[1 : n] on digit i
```