反馈与讨论

5.29

7-4 Stack depth for quicksort

The QUICKSORT algorithm of Section 7.1 contains two recursive calls to itself. After QUICKSORT calls PARTITION, it recursively sorts the left subarray and then it recursively sorts the right subarray. The second recursive call in QUICKSORT is not really necessary; we can avoid it by using an iterative control structure. This technique, called *tail recursion*, is provided automatically by good compilers. Consider the following version of quicksort, which simulates tail recursion:

```
TAIL-RECURSIVE-QUICKSORT(A, p, r)
1  while p < r
2        // Partition and sort left subarray.
3        q = PARTITION(A, p, r)
4        TAIL-RECURSIVE-QUICKSORT(A, p, q − 1)
5        p = q + 1
```

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

a. Argue that TAIL-RECURSIVE-QUICKSORT$(A, 1, A.length)$ correctly sorts the array A.

QuickSort的正确性利用循环不变式已得到证明，这里我们利用QuickSort证明Tail-Recursive-Quicksort。
Tail-Recursive-Quicksort调用与QuickSort相同的Partition，分为左右两部分：即参数(A, p, q-1)和参数(A, q+1, r)

1.对于参数(A, p, q-1)，QuickSort与Tail-Recursive-Quicksort相同，都是进行递归调用。
2.对于参数(A, q+1, r)，QuickSort直接进行递归调用；而Tail-Recursive-Quicksort则是让p=q+1后执行了while循环的另一次迭代，然后递归调用(A, q+1, r)。

Tail-Recursive-Quicksort与QuickSort递归调用的效果相同。因此， Tail-Recursive-Quicksort (A, 1, Length[A])能正确对数组进行排序。

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

This will happen whenever partition returns r.

c. Modify the code for TAIL-RECURSIVE-QUICKSORT so that the worst-case stack depth is $\Theta(\lg n)$. Maintain the $O(n \lg n)$ expected running time of the algorithm.

如果最坏情况下堆栈的深度为Θ(lgn)，那么调用PARTITION函数后左侧的子数组大小大约为原数组的一半，这样递归深度最多为Θ(lgn)

改进的算法是：首先求得(A, p, r)的中位数，作为PARTITION的枢轴元素，这样可以保证左右两边的元素的个数尽可能的均衡。因为求中位数的过程MEDIAN的时间复杂度为Θ(n)，因此可以保证算法的期望的时间复杂度O(nlgn)不变。

```
QUICKSORT''(A, p, r)
while p < r
    do ▸ Partition and sort left subarray.
       m ← MEDIAN(A, p, r)
       exchange A[m] ↔ A[r]
       q ← PARTITION(A, p, r)
       QUICKSORT''(A, p, q - 1)
       p ← q + 1
```

7-5 Median-of-3 partition

One way to improve the RANDOMIZED-QUICKSORT procedure is to partition around a pivot that is chosen more carefully than by picking a random element from the subarray. One common approach is the *median-of-3* method: choose the pivot as the median (middle element) of a set of 3 elements randomly selected from the subarray. (See Exercise 7.4-6.) For this problem, let us assume that the elements in the input array $A[1..n]$ are distinct and that $n \geq 3$. We denote the sorted output array by $A'[1..n]$. Using the median-of-3 method to choose the pivot element x, define $p_i = \Pr\{x = A'[i]\}$.

a. Give an exact formula for p_i as a function of n and i for $i = 2, 3, \ldots, n-1$. (Note that $p_1 = p_n = 0$.)

- There are n!/(n−3)! permutations of all possible picks. In order to have the i th element, we need to pick one smaller, the i th element and one larger. There are i−1 ways to pick a smaller one and n−i ways to pick the larger. There are 3! ways to arrange how the three elements are picked. Thus:

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

b. By what amount have we increased the likelihood of choosing the pivot as $x = A'[\lfloor (n+1)/2 \rfloor]$, the median of $A[1..n]$, compared with the ordinary implementation? Assume that $n \to \infty$, and give the limiting ratio of these probabilities.

$$\lim_{n\to\infty} \frac{6(i-1)(n-i)}{n(n-1)(n-2)} \Big/ \frac{1}{n}$$

$$= \lim_{n\to\infty} \frac{6n(n/2-1)(n/2)}{(n-1)(n-2)}$$

$$= \lim_{n\to\infty} \frac{6(n^2-2n)}{4(n^2-3n+2)}$$

$$= \frac{6}{4}$$

- 因为是求极限，所以取**i=n/2**和**i=(n+1)/2**，结果是一样的。

c. If we define a "good" split to mean choosing the pivot as $x = A'[i]$, where $n/3 \le i \le 2n/3$, by what amount have we increased the likelihood of getting a good split compared with the ordinary implementation? (*Hint:* Approximate the sum by an integral.)

$$\lim_{n\to\infty} \sum_{i=n/3}^{2n/3} \frac{6(i-1)(n-i)}{n(n-1)(n-2)}$$

$$= \lim_{n\to\infty} \frac{6}{n(n-1)(n-2)} \sum_{i=n/3}^{2n/3} (i-1)(n-i)$$

$$= \lim_{n\to\infty} \binom{n}{3} \int_{n/3}^{2n/3} (i-1)(n-i)\,di$$

$$\left(\int (i-1)(n-i)\,di = \frac{1}{6}(3ni^2 - 6ni - 2i^3 + 3i^2) \right)$$

$$= \lim_{n\to\infty} \binom{n}{3} \frac{1}{6} \left[\frac{36}{27}n^3 - \frac{16}{27}n^3 + o(n^3) - \frac{9}{27}n^3 + \frac{2}{27}n^3 + o(n^3) \right]$$

$$= \lim_{n\to\infty} \frac{1}{n(n-1)(n-2)} \frac{13}{27}(n^3 + o(n^3))$$

$$= \lim_{n\to\infty} \frac{13}{27} \frac{n^3 + o(n^3)}{n^3 + o(n^3)}$$

$$= \frac{13}{27}$$

- 一般情况下，得到一个好的划分的概率是 **1/3**

$$\frac{13}{27} \div \frac{1}{3} = \frac{39}{27}$$

d. Argue that in the $\Omega(n \lg n)$ running time of quicksort, the median-of-3 method affects only the constant factor.

- The running time would improve if the new approach can always pick a good split. Unfortunately, it can't. It makes it impossible for one of the splits to be empty, **but it can still pick a 1 to (n–2) split**. It improves the probability of a good split and adds some overhead to pick the pivot, but it makes no hard guarantees on the quality of the split. Thus, the algorithm remains $\Omega(n \lg n)$ and $\mathcal{O}(n^2)$

8-2 *Sorting in place in linear time*

Suppose that we have an array of n data records to sort and that the key of each record has the value 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.

1. The algorithm runs in $O(n)$ time.
2. The algorithm is stable.

- 计数排序（非原地排序、稳定）

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

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

- 从数组两头向中间扫描，发现逆序的一对元素后，即进行交换，直到扫描结束（原地排序、不稳定）

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

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.

- 插入排序（原地排序、稳定）

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.

- 基数排序要求所使用的排序方法是稳定的；如果在O(bn)时间内完成，排序时间应该是线性时间O(n)。所以(a)、(b)、(c)中只有(a)满足条件。

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? (*Hint:* How would you do it for $k = 3$?)

In place counting sort

We build an array of counts as in `COUNTING-SORT`, but we perform the sorting differently. We start with `i = 0` and then.

```
while i ≤ A.length
    if A[i] is correctly placed
        i = i + 1
    else
        put A[i] in place, exchanging with the element there
```

On each step we're either (1) incrementing `i` or (2) putting an element in its place. The algorithm terminates because eventually we run out of misplaced elements and have to increment `i`.

There are some details about checking whether `A[i]` is correctly placed that are in the C code.

```c
void in_place_counting_sort(item *A, int size, int range) {
    int counts[range + 1];
    int positions[range + 1];

    for (int i = 0; i <= range; i++) {
        counts[i] = 0;
    }

    for (int i = 0; i < size; i++) {
        counts[A[i].key]++;
    }

    for (int i = 2; i <= range; i++) {
        counts[i] += counts[i-1];
    }

    for (int i = 0; i <= range; i++) {
        positions[i] = counts[i];
    }

    int i = 0;
    while (i < size) {
        int key = A[i].key;
        bool placed = (positions[key - 1] <= i && i < positions[key]);

        if (placed) {
            i++;
        } else {
            EXCHANGE(A[i], A[counts[key] - 1]);
            counts[key]--;
        }
    }
}
```

9.3-7

Describe an $O(n)$-time algorithm that, given a set S of n distinct numbers and a positive integer $k \leq n$, determines the k numbers in S that are closest to the median of S.

1. We find the median of the array in linear time
2. We find the distance of each element to the median in linear time
3. We find the *k*-th order statistic of the distance, again, in linear time
4. We select only the elements that have distance lower than or equal to the *k*-th order statistic