Assignment 13, Data-Structures

Oleg Sivokon

<2016-04-09 Sat>

Contents

1 Problems

1.1 Problem 1

1. How many comparisons does the `partition` algorithm have to make before it completes on an input of size n, provided the input is sorted in ascending order?
2. What if the input is sorted in descending order?

1.1.1 Answer 1

It doesn't matter if input is sorted, or that it has duplicates. `partition` will do $n - 1$ comparisons (provided it selects the first element to be the pivot element). This is so because it has to compare each element to the pivot element, but there is no need to compare them more than once.

1.2 Problem 2

Array is said to be almost sorted with error of size k when for any two elements it holds that if there is more than k elements between them, then they are in correct order.

1. Modify `quicksort` algorithm to produce almost sorted arrays.
2. What is the running time of this new algorithm?

1.2.1 Answer 2

1. The modification is as follows: before we proceed to partition a slice of array we check that its length is greater than k. This gives us the running time of $O(\lg(n - k)n)$. Partitioning of larger slices of array

ensures that $\forall e \in A[x \dots x+k] : e \leq A[x+k+1 \dots]$, but this is exactly our requirement.

2. As suggested above, the running time is $O(\lg(n-k)n)$ since we skip the processing of the subtrees at the depth greater than $n-k$.

Reference implementation is given below:

```
pair three_way_partition(array partitioned, comparison_fn_t cmp) {
    time_t t;
    size_t lidx, ridx, i, mid = 1;
    printable* left;
    printable* right;
    pair result = make_pair();
    int default_val = 0;

    if (partitioned->length <= 1) {
        result->first = int_element_generator(&default_val);
        result->last = int_element_generator(&default_val);
        return result;
    }
    srand((unsigned)time(&t));
    lidx = rand() % (partitioned->length / 2);
    ridx = (partitioned->length / 2) +
           rand() % (partitioned->length / 2);
    left = partitioned->elements[lidx];
    right = partitioned->elements[ridx];
    if (cmp(&left, &right) > 0) {
        swap(partitioned, lidx, ridx);
        left = partitioned->elements[lidx];
        right = partitioned->elements[ridx];
    }
    swap(partitioned, 0, lidx);
    swap(partitioned, partitioned->length - 1, ridx);
    lidx = 1;
    ridx = partitioned->length - 1;
    for (i = 0; i < partitioned->length - 2; i++) {
        if (cmp(&partitioned->elements[mid], &left) < 0) {
            swap(partitioned, lidx, mid);
            lidx++;
            mid++;
        } else if ((cmp(&partitioned->elements[mid], &right) < 0)) {
            mid++;
        } else {
            swap(partitioned, ridx - 1, mid);
            ridx--;
        }
    }
    int lidx_int = (int)lidx;
    int ridx_int = (int)ridx;
    result->first = int_element_generator(&lidx_int);
    result->last = int_element_generator(&ridx_int);
    return result;
}
```

```c
void quicksort(array unsorted, comparison_fn_t cmp, size_t error) {
    if (unsorted->length <= error) return;
    switch (unsorted->length) {
        case 1: break;
        case 2:
            if (cmp(&unsorted->elements[0],
                    &unsorted->elements[1]) > 0)
                swap(unsorted, 0, 1);
            break;
        default: {
            pair walls = three_way_partition(unsorted, cmp);
            int low = *(int*)walls->first->val;
            int high = *(int*)walls->last->val;
            if (low > 1)
                quicksort(slice(unsorted, 0, (size_t)low),
                          cmp, error);
            if (low + 1 < high)
                quicksort(
                    slice(unsorted, (size_t)low, (size_t)high),
                    cmp, error);
            if (high + 1 < unsorted->length)
                quicksort(slice(unsorted, (size_t)high,
                                unsorted->length), cmp, error);
        }
    }
}
```

1.3 Problem 3

Show the steps taken by the algorithm `randomized-select` on the sequence $S = (60, 70, 80, 90, 100, 1, 5, 9, 11, 15, 19, 21, 25, 29, 30)$ for $k = 5$.

1.3.1 Answer 3

This algorithm will use the same `random-partition` procedure used in `quicksort`:

1. *random-partition*(S, S_1, S_2), randomly selected 19, hence: $S_1 = (1, 5, 9, 11, 15, 19)$ and $S_2 = (60, 70, 80, 90, 100, 21, 25, 29, 30)$.
2. Since $|S_1| = 6 > k$ the kth element must be in S_1, recure on it.

3. *random-partition*(S_1, S_3, S_4), randomly select 5, hence: $S_3 = (1, 5)$ and $S_4 = (9, 11, 15, 19)$.
4. Since $|S_3| = 2 < k$, decrease k by 2 and recurse on S_4:
5. $k = k - 2$.
6. *random-partition*(S_4, S_5, S_6), randomly select 11, hence: $S_5 = (9, 11)$ and $S_6 = (15, 19)$.
7. Since $|S_5| = 2 = k$ we are done, the kth element therefore is 11.

1.4 Problem 4

1. Prove that a sequence S of length n has at most three elements that repeat more than $\lfloor \frac{n}{4} \rfloor$ times.
2. Write an algorithm finding all elements which repeat more than $\lfloor \frac{n}{4} \rfloor$ times. Running time $O(n)$.

1.4.1 Answer 4

Assume for contradiction that there are four elements in S that repeat $\lfloor \frac{n}{4} \rfloor$ times. That is exist fourth element in the sequence that repeats $\lfloor \frac{n}{4} + 1 \rfloor$ times. Note that $3\lfloor \frac{n}{4} \rfloor + \lfloor \frac{n}{4} + 1 \rfloor > n$ contrary to assumed. Hence, by contracition, there are at most three such elements.

1.4.2 Answer 5

Simple solution involves using hash-table: loop over the sequence, using elements as keys in the hash-table. Increment the key whenever you encounter the element. Finally, loop over the hash-table to collect all those for which the condition holds.

This can be improved by using $\lfloor \frac{n}{4} \rfloor$ hash-tables, first for the elements which have been seen just once, second—for the elements seen twice and

so on. Thus, instead of icnrementing the counter, the elements would be promoted to the last hash-table. This removes the requirement of the final loop, but doesn't change the running time significantly.

A more complicated way to do this, without using hash-tables is to do the following:

1. Partition in three parts using the same partition procedure used in `quick-sort` algorithm. If a partition is smaller than $\lfloor \frac{n}{4} \rfloor$, throw it away and repeat.
2. At this point, the following could happen:
   - You have three partitions each containing the same element—you are done.
   - Further partitioning is impossible: no such elements exist—you are done.
   - Some partitions only contain same elements (store the element, throw away the partition), others contain different element: throw them away (they aren't candidates).

1.5 Problem 5

1. Given a sequence of real numbers: $(a_0, a_1, a_2, \ldots, a_n)$ define:

$$m = \min\{a_i \mid 0 \leq i \leq n\}$$
$$M = \max\{a_i \mid 0 \leq i \leq n\}$$

   Show that there exist a_i, a_j s.t.:

$$\left|a_i - a_j\right| \leq \frac{M - m}{n}$$

2. Write the algorithm for finding the a_i, a_j from the question above.

1.5.1 Answer 5

Note that the sequence with the given M and m will have $M - m = k \geq n$ elements. Suppose now we arrange the elements in increasing order. Suppose, for contradiction, that none of the elements of the sequence satisfies $\left|a_i - a_j\right| \leq \frac{M-m}{n}$, then it also means that the difference between every two adjacent elements must be at least $\frac{M-m+\epsilon}{n}$ for some positive ϵ. since there are k pairs of adjoint elements, we get that:

$$\begin{aligned}\sum_{i=1}^{k}(a_i - a_{i-1}) &= \sum_{i=1}^{k}\left(\frac{M-m+\epsilon}{n}\right) \\ &= \frac{k}{n}(M-m) + k\epsilon \geq M - m\end{aligned}$$

contrary to assumed. Hence, by contradiction, there must be at least one pair a_i, a_j s.t. $\left|a_i - a_j\right| \leq \frac{M-m}{n}$.

1.5.2 Answer 6

The general idea for the algorithm is to normalize all members of the given sequence by subtracting the minimum and mutliplying by the ratio of one less than the length of the sequence and the difference between the maximum and the minumum. Once done, do the insertion sort: two elements which fall in the same cell will have a distance between them less than the one between the minimum and the maximum divided into one less than the number of elements.

Algorithm 1 Find $x, y \in Elts$ s.t. $|x - y| \leq \frac{\max(Elts) - \min(Elts)}{|Elts| - 1}$

```
procedure min-pair(elements)
    max ← max(elements)
    min ← min(elements)
    size ← size(elements)
    copy ← make-vector(size, nil)
    for element ∈ elements do
        index ← ⌊((elt−min)×(size−1)) / (max−min)⌋
        if copy_index = nil then
            copy_index = element
        else
            return element, copy_index
        end if
    end for
end procedure
```