Algorithm Assignment6

Xinyu Jiang

October 2018

1

(15 pts) Suppose that we modify the `Partition` algorithm in QuickSort in such a way that on alternating levels of the recursion tree, `Partition` either chooses the second best possible pivot or the second worst possible pivot. Write down a recurrence relation for this version of QuickSort and give its asymptotic solution. Then, give a verbal explanation of how this `Partition` algorithm changes the running time of QuickSort.

Best case: For the best case, the pivot is in the middle of the array. Therefore, for the second best case, the pivot is in either $\frac{n}{2}-1$ or $\frac{n}{2}+1$, because the best case and second best case is basically the same, therefore, the recurrence relation is $T(n) = 2T(n)+O(n)$ ($O(n)$ is the cost of partition)=$O(nlogn)$, therefore the second best case is $T(n) = O(nlogn)$
Worst case: For the worst case, the pivot is in either in the start of the array or the end of the array, which is 1 or n-2, then for the second worst case, the pivot is either in the 2 or n-2. Because the second worst case is basically the same with worst case, the recurrence relation is $T(n) = T(n-1)+n$
$T(n-1) = T(n-2)+(n-1)+n$
$T(n-2) = T(n-3)+(n-2)+(n-1)+n$
$T(1) = T(0)+1+2+3+...+n$
$T(1) = T(0)+\Sigma_1^i i$=n^2
therefore, the second worst case is $T(n) = O(n^2)$
Average case: For each iteration, suppose there is one second best case and one second worst case, because the pivot for the best case is either $\frac{n}{2}-1$ or $\frac{n}{2}+1$ and the pivot for the second worst case is either 2 or $n-2$, by looking the 2 iteration together(two second best cases and two second worst cases) the second best case is $T_1(n) = T_2(\frac{n}{2}-1)+T_2(\frac{n}{2}+1)+O(n)$
, the second worst case is $T_2(n) = T_1(2)+T_1(n-2)+O(n)$
By plugging in T_2 to T_1, the run time is $O(n*logn)$
Verbal explanation:
Because the pivot for second best case either $\frac{n}{2}+1$ or $\frac{n}{2}$-1, and the second worst case is either 1 or n-2. If the array have big amount of elements, then the best case have significant bigger effect than the worst case, because the best case is dividing the array to half, and iteration from it, and the worst case does not

affect the iteration so much (suppose there are 10000 elements in an array, and in the process of choosing pivots, there are 999 worst case and 1 best case, the best case is cutting the array to half, but the worst case is just choosing pivot from either start or end), so the average case is the same with the best case, which is $O(nlogn)$

2

(10 pts) You are given n metal balls $B_1, \ldots, B_n$, each having a different weight. You can compare the weights of any two balls by comparing their weights using a balance to find which one is heavier.

1. Consider the following algorithm to find the heaviest ball:

   (a) Divide the n balls into $\frac{n}{2}$ pairs of balls.

   (b) Compare each ball with its pair, and retain the heavier of the two.

   (c) Repeat this process until just one ball remains.

   Illustrate the comparisons that the algorithm will do for the following $n = 8$ input:

   $$B_1 : 3, \quad B_2 : 5, \quad B_3 : 1, \quad B_6 : 2, \quad B_5 : 4, \quad B_6 : \tfrac{7}{2}, \quad B_7 : \tfrac{3}{2}, \quad B_8 : \tfrac{9}{2}$$

2. Show that for n balls, the algorithm balls:a uses at most n comparisons.

3. Describe an algorithm that uses the results of balls:a to find the *second* heaviest ball, using at most $\log_2 n$ additional comparisons. There is no need for pseudocode; just write out the steps of the algorithm like we have written in balls:a.
   Hint: if you follow sports, especially wrestling, read about the *repechage*.

4. Show the additional comparisons that your algorithm in balls:c will perform for the input given in balls:a.

1).The process of finding the heaviest ball is by cutting the balls to many pairs of two and compare each of them, then return the heavier one, and compare with the heavier one from other pairs until there is only one ball left, then that is the heaviest one.
For the example above, the iteration process is:

first iteration: $(B_1\ ,\ B_2)\ (B_3\ ,\ B_4)\ \ (B_5\ ,\ B_6)\ (B_7\ ,\ B_8)$
second iteration: $(B_2\ ,\ B_4)\ (B_5\ ,\ B8)$
third iteration: $(B_2\ , B_8)$
forth iteration: (B_2)

which B_2 is the heaviest ball.
2). The algorithm do n times comparison because the algorithm is dividing the array into two pairs and compare the value, so the first iteration does comparison $\frac{n}{2}$ times, then the second iteration does comparison $\frac{n}{4}$ times, and the third iteration does $\frac{n}{8}$ times and n iteration does comparison $\frac{n}{2^n}$ times, by adding all iteration up, it is $\Sigma_1^i \frac{1}{2^n} n$=n, so the algorithm above the does comparison n times.
3). Divide the n balls into $\frac{n}{2}$ pairs of balls. Compare each ball with its pair, and retain the heavier of the two, then record the ball that "lost" in an array. Repeat this process until just one ball remains. After finding the heaviest ball, check all the balls that "lost" to the heaviest ball, then the "heaviest ball" that in the array that lost to the heaviest ball is the second heaviest.
4).

first iteration: $(B_1\ ,\ B_2)\ (B_3\ ,\ B_4)\ \ (B_5\ ,\ B_6)\ (B_7\ ,\ B_8)$
second iteration: $(B_2\ ,\ B_4)\ (B_5\ ,\ B8)$
third iteration: $(B_2\ , B_8)$
forth iteration: (B_2)
After we find the heaviest ball, check all the balls that "lost" to the heaviest ball, which is
B_1, B_4, B_8
And find the heaviest ball in B_1, B_4, B_8, which is B_8.

By the value given B_8 is the second heaviest.

3

(10 pts) An array is *almost k sorted* if every element is no more than k positions away from where it would be if the array were actually sorted in ascending order.

As an example, here is an almost 2-sorted array:

1. Write down pseudocode for an algorithm that sorts the original array in place in time $n\,k \log k$. Your algorithm can use a function `sort`(A, ℓ, r) that sorts the subarray $A[\ell], \ldots, A[r]$.

a).
Algorithm description: The algorithm is making a new array and by each time of iteration sort it use the function given. For example, the first iteration sorts from 0 to k, and second iteration sorts from 1 to k+1, then after all iteration finishes, the new array fulfill the k sort requirement. Because the run time for sorting is *nlogn*, and the algorithm runs k times, so the run time is $O(nklogk)$

```
kStort(A,k){
    for(int i=0;i<A.length()-1;i++){
        sort(A,i,i+k)
    }
}
```

4

(10 pts) Consider an n-node complete binary tree T, where $n = 2^d - 1$ for some d. Each node v of T is labeled with a real number x_v. You may assume that the real numbers labeling the nodes are distinct. A node v of T is a local minimum if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge. Show how to find a local minimum of T evaluating only O(lg n) nodes of T.

```
localMin(A){
    if(A.value< A.left.value && A.value< A.right.value){
        return A.value;
    }else if(A.value>A.left.value && A.value<A.right.value){
        localMin(A.right);
    }else if(A.value<A.left.value && A.value>A.right.value){
        localMin(A.left)
    }else if(A.value> A.left.value && A.value> A.right.value){
    //which means there are two local min, so by going any direction(left or right),
        localMin(A.left)
        //choose to go to the left
    }
}
```

Base on the algorithm above, the recurrence relation is $T(n) = T(\frac{n}{2}) + c$, so that the run time $T(n) = O(logn)$

5

(20 pts) Consider the following strategy for choosing a pivot element for the `Partition` subroutine of QuickSort, applied to an array A.

- Let n be the number of elements of the array A.
- If $n \leq 15$, perform an Insertion Sort of A and return.
- Otherwise:
  - Choose $2\lfloor\sqrt{n}\rfloor$ elements at random from n; let S be the new list with the chosen elements.
  - Sort the list S using Insertion Sort and use the median m of S as a pivot element.
  - Partition using m as a pivot.
  - Carry out QuickSort recursively on the two parts.

1. If the element m obtained as the median of S is used as the pivot, what can we say about the sizes of the two partitions of the array A?

2. How much time does it take to sort S and find its median? Give a Θ bound.

3. Write a recurrence relation for the worst case running time of QuickSort with this pivoting strategy.

a). For the best case find of obtaining the median of S, because there are $2\sqrt{n}$, then the best case is when it divide evenly, which is one side has $\sqrt{n}$ elements, and other side has $\sqrt{n}$ elements as well.
The worst case occur when the pivot is at the start point, which one side is $\sqrt{n}$, and other side is $n - \sqrt{n}$
b)Because we know the run time for insertion sort is $\Theta(n^2)$, then for $2\sqrt{n}$, the run time is $\Theta(2\sqrt{n})^2$, which is $\Theta(4n)$, so the run time is $\Theta(n)$
c).For the worst case, because it occurs when the pivot is at the start point, which one side is $\sqrt{n}$, and other side is $n - \sqrt{n}$, and base on the answer in b, the time takes to sort S and find the median is n so the recurrence relation is $T(n) = T(\sqrt{n}) + T(n - \sqrt{n})$+n