Poonam Gupta

COT 5407 – Introduction to Algorithms Homework Assignment #2

Problem 1 (2.5 points) Quicksort based on Tail Recursion

The *Quicksort* algorithm of in the book contains two recursive calls to itself. After *Quicksort* calls *Partition*, it recursively sorts the left subarray and then the right subarray. The second recursive can be avoided using an iterative control structure. This technique, called tail recursion, is provided automatically by good compilers.

(1) Describe a scenario in which *Tail-Recursive-Quicksort*'s *stack depth* is Θ(*n*) on an *n*-element input array. Note: recursive procedure (function) calls are implemented using a *stack* containing parameters and return address etc., where the most recent calls on the top. When a call is finished, its information is popped. The *stack depth* is the maximum amount of stack space used at any time during a computation.

Answer: The stack depth of quick sort is Θ(*n*) on an n-element array if there are equal number of recursive calls to *Tail-Recursive-Quicksort.* This happens if every call to PARTITION (A, p, r) returns q=r. The sequence of recursive calls in this scenario is
T-R-Q (A, p, n), T-R-Q (A, p, n-1), T-R-Q (A, p, n-2) ………………… T-R-Q (A, p, 1)

An array that is already sored in ascending order will cause *Tail-Recursive-Quicksort* to behave this way.

(2) Modify the code for *Tail-Recursive-Quicksort* so that the worst-case stack depth is Θ(lg*n*). Maintain the *O*(*n* lg *n*) expected running time of the algorithm.

Modified-Tail-Recursive-Quicksort (A, p, r)

```
While p < r
        q = Partition (A, p, r)
         if q < ⌊((p + r) /2)⌋
               Modified-Tail-Recursive-Quicksort (A, p, q-1)
                P = q + 1
           else
                 Modified-Tail-Recursive-Quicksort (A, q+1, r)
                  r = q -1
```

Each recursive call reduces the problem size by at least half. Thus, the stack depth is O (logn).

Problem 2 (2.5 points) Linear Time Sorting

(1) You are given an array of strings of different lengths, but the total number of characters over all the strings is *n*. Describe an algorithm to sort the strings in alphabetic order (eg. algorithm < cs < fiu) in *O*(*n*) time.

Answer: In order to sort a set of strings of different lengths, we need to focus on the leftmost character of each string, as this character can help us identify if a string comes before or after another string. The length of a string does not have an implication on the position of the string. We can use this property, along with counting sort algorithm. In order to accomplish this, we can sort the strings by their first character. All the strings with the same character can be grouped together. Then we can apply the same algorithm to the strings in this group excluding the first character. As we can see it would be easier to implement this as a recursive algorithm that sorts each group by the next set of characters.

Let m = No. of strings in the input array
n_i = no. of characters in the *i*th string

Since counting sort is linear on the number of elements being sorted and *i*th strings are considered in at most $n_i + 1$ calls of counting sort(one additional time for the special case when the string does not have the following character), the total cost of the algorithm is

$$O(\sum_{i=1}^{m}(\mathrm{ni}+1)) = O(\sum_{i=1}^{m}(\mathrm{ni}+\mathrm{m})) = O(n+m) = O(n)$$

(2) Show the sorting process with strings – fiu, cs, cot5407, algorithm, spring, 2020 (assume numbers are less than letters).

Array

Fiu	Cs	Cot5407	Algorithm	Spring	2020

index

#	**A**	**B**	**C**	**.....**	**F**	**....**	**S**	**.... Z**
2020	**Algorithm**		**CS** **COT5407**		**Fiu**		**Spring**	
			S **OT5407**					

OT5407	**S**

Explanation: we have an array of strings, where numbers are in range of 0-9and letters are from A to Z. Create a count array to store the counts of strings. Then apply the counting sort recursively where the count has more than one string. In this way, we can sort all the strings in linear time.

Problem 3 (3 points) Hashing

Suppose that we use an open-addressed hash table of size *m* to store $n \leq m/2$ items.

(1) Assuming uniform hashing, show that for i=1,2,…,n, the probability of the ith insertion requiring strictly more than k probes is at most 2^{-k}.

Answer: Since we assume uniform hashing, inserting an element into an open-address hash table with load factor α requires at most 1/1- α probes on average.

As in the proof of theorem for expected number of probes in an unsuccessful search, if we let X be the random variable denoting the number of probes in an unsuccessful search, then

$\Pr(X \geq i) \leq \alpha^{i-1}$ Since $n \leq m/2$, we have $\alpha \leq$ ½.

Lets i =k+1

We have $\Pr(X>k) = \Pr(X \geq k+1) \leq (1/2)^{(k+1)-1} = 2^{-k}$

(2) For i=1,2,…,n, show that the probability of the ith insertion requiring more than 2 *lgn* probes is $O(1/n^2)$.

Answer: Using the result from above part, we get that

$$\Pr(X_i > k) \leq 2^{-k}$$
$$\Pr(X_i > 2\log n) \leq 2^{-2\log n} = n^{-2}$$
$$= 1/n^2$$
$$O(1/n^2)$$

(3) Let the random variable Xii denote the number of probes required by the ith insertion. You have shown in part (2) that $\Pr\{X>2lgn\} = O(1/n^2)$. Let the random variable X=max1≤i≤nXi denote the maximum number of probes required by any of the n insertions. Show that $\Pr\{Xi>2lgn\} = O(1/n)$.

Answer: Using the union bound, we can show that $\Pr(X > 2 \log n) \leq 1/n$

$$\Pr(X > 2 \log n) = \Pr(\vee_i X_i > 2 \log(n))$$
$$\leq \sum_{i=1}^{n} \Pr((Xi > 2\log(n))$$
$$\leq \sum_{i=1}^{n} \frac{1}{n2} = 1/n$$