ESO207A/211: Data Structures and Algorithms

Homework *3 Solutions* **Due: September 10 2014**

Problem 1. Consider the following code for non-recursive quicksort. Here *Partition* is the standard procedure from quicksort and may or may not be randomized.

```
NR-Qsort(A, n) {
    Stack S;  // S is a stack of pairs of indices
    Push(S, (1, n))
    while S is not empty {
        (p, r) = Pop(S)
        q = Partition(A, p, r)
        if(q − 1 > p)
            Push (p, q − 1)
        if(r > q + 1)
            Push (q + 1, r)
    }
}
```

1. Give a (worst-case) scenario for the execution of **NR-Qsort** where the depth of the stack S is $\Omega(n)$.
2. Modify the loop so that the worst-case depth of the stack S is $O(\log n)$. Maintain the time complexity bound of standard quicksort.

Answer

1. Worst-case occurs if, in every iteration, the position of randomly chosen pivot turns to be 3^{rd} from beginning of the part of array being partitioned. Thus if k elements are being partitioned, left sub-array has 2 elements and right one has $k-3$. Left sub-array is pushed first onto the stack followed by right counterpart which gets popped first and undergoes the same procedure. Due to accumulation of left sub-array index pairs stack depth grows till $\lfloor\frac{n}{3}\rfloor + c$ ($c \in \{0, 1\}$), which is $\Omega(n)$.
2. **Modified pseudo-code**

```
NR-Qsort(A, n) {
    Stack S;
    Push(S, (1, n))
    while S is not empty {
        (p, r) = Pop(S)
        q = Partition(A, p, r)
        if(r − q > q − p)
            if(r > q + 1)
                Push (S, (q + 1, r))
```

```
            if(q − 1 > p)
                Push (S, (p, q − 1))
        else
            if(q − 1 > p)
                Push (S, (p, q − 1))
            if(r > q + 1)
                Push (S, (q + 1, r))
    }
}
```

Above pseudo-code ensures that after partition larger sub-array's indices are pushed first onto the stack followed by the smaller sub-array indices.

O(log n) worst-case stack depth

Preliminaries: Let us call the index pairs which can be pushed onto the stack as eligible stack pairs. It is trivial to note that length of sub-array corresponding to an eligible stack pair is atleast two. Also in each iteration, either one or two eligible stack pairs are pushed onto the stack. In next iteration smaller index pair always gets popped. Thus a net increase in stack depth takes place only if two pairs are pushed.

Proof: To achieve maximum stack depth there should be a net increase in every iteration, i.e., both index pairs should be eligible stack pairs. So for every element in the stack there is a *Partition* call to the smaller counter-part. After partitioning of a popped sub-array index pair(of size k), larger sub-array has size $\geq k/2$ and smaller one is of size $\leq k/2$.

Thus if a stack pair of size k is popped, corresponding index pair in the stack has size $k_{1,1} \geq k/2$ and its counterpart partition call is on a sub-array of size $k_{1,2} \leq k/2$. In the next iteration, index pair in the stack has size $k_{2,1} \geq k_{1,2}/2$ and its counterpart partition call is on a sub-array of size $k_{2,2} \leq k_{1,2}/2$.

Suppose m elements on the stack are:
index-pair($k_{1,1}$), index-pair($k_{2,1}$), index-pair($k_{3,1}$),....,index-pair($k_{m,1}$)
Replacing these by counter part Partition(size of sub-array) calls:
Partition($k_{1,2}$), Partition($k_{2,2}$),Partition($k_{3,2}$),.....,Partition($k_{m,2}$)
or, Partition($\leq k/2$), Partition($\leq k_{1,2}/2$),Partition($\leq k_{2,2}/2$),.....,Partition($k_{m-1,2}/2$)
or, Partition($\leq k/2^1$), Partition($\leq k/2^2$),Partition($\leq k/2^3$),.....,Partition($k/2^m$)

Since any partition calls takes sub-array of size atleast 2, $k/2^m \geq 2 \implies m \leq \log_2 k - 1 \implies m < \log_2 k$. Maximum value of $k = n$. Thus $m < \log_2 n$. In other words, stack cannot have more than m or $O(\log_2 n)$ elements.

Note: When only one eligible stack pair is generated after partition, the iteration can be safely ignored since no corresponding index pair on stack is available.

Time Complexity

Recursive and iterative versions of quicksort are equivalent. Instead of recursive call on the sub-arrays, their index pairs are pushed onto the stack and same operations are perfomed on the sub-array when index pair is popped in the iterative version. And these operations are performed in the same order. In the modified version same operations are performed but they may not be in same order. Since recursive calls on the partitioned left and right sub-arrays

are independent, they can be perfomed in any order producing the same result. Thus the time complexity of modified iterative quicksort is maintained.

Problem 2. Kendall tau distance between permutations. A permutation (or ranking) is an array of n integers where each of the integers between 1 and n appears exactly once. The Kendall tau distance between the two rankings is the number of pairs that are in different order in the two rankings. For example, the Kendall tau distance between 0 3 1 6 2 5 4 and 1 0 3 6 4 2 5 is 4 because the pairs $(0,1),(3,1),(2,4),(5,4)$ are in different relative order in the two rankings but all other pairs are in the same relative order.

1. Design an efficient $O(n \log n)$ time algorithm for computing the Kendall tau distance between a given permutation and the identity permutation (that is, 1 2 $\ldots n$). (***Hint:*** Modify Merge-sort.)
2. Extend the above algorithm to give an $O(n \log n)$ time algorithm for computing the Kendall tau distance between two permutations.

Answer

1. The first part of the problem is equivalent to finding the number of inversions of a given permutations. This can be achieved by modifying merge sort as follows:

Inversions-count-Merge-sort$(A, right, left)$**{**//Initially $right$=n-1, $left$=0 and A is given permuatation
    $inv\_count = 0$; // inv_count will have the total number of inversions
    if$(right > left)${
        $mid = (right + left)/2$;
        $inv\_count =$ **Inversions-count-Merge-sort**$(A, left, mid)$;
        $inv\_count \mathrel{+}=$ **Inversions-count-Merge-sort**$(A, mid+1, right)$;
        $inv\_count \mathrel{+}=$ **Merge**$(A, left, mid+1, right)$;
    }
    return inv_count ;
}

Merge$(A, right, mid, left)$ {
    $inv\_count = 0$, p1=$left$, p2=mid, p3=$left$;
    while$((p1 \leq mid - 1)$ && $(p2 \leq right))${
        if$(A[p1] \leq A[p2])$ {
            $temp[p3{+}{+}] = A[p1{+}{+}]$; //Initially $temp$ is an empty array of size as that of A
        }
        else{
            $temp[p3{+}{+}] = A[p2{+}{+}]$;
            $inv\_count \mathrel{+}= (mid - p1)$;
        }
    }
    while$(p1 \leq mid - 1)$ { //Copy remaining elemnets of left sub-array(if remaining) to $temp$
        $temp[p3{+}{+}] = A[p1{+}{+}]$;

}
while($p1 \leq mid - 1$) { //Copy remaining elemnets of right sub-array(if remaining) to $temp$
$temp[p3{+}{+}] = A[p2{+}{+}]$;
}
for($i = left$; $i \leq$ $right$; $i{+}{+}$) { // Copy back the merged elements to original array
$A[i] = tmp[i]$;
}
return inv_count ;
}

Since the complexity of merge-sort is $O(n \log n)$ the Kendall tau distance between a given permutation and the identity permutation, i.e., count of the number of inversions can be done in $O(n \log n)$.

2. For computing the Kendall tau distance between two given permutations, the above algorithm needs to be modified slighly, more precisely we need to do some more steps before calling the above defined function.
All we need to assume the first permuation permuatation to be the identity and make change to the other permutation accordingly. After this we could just find the number of inverions of the second permuation using the above technique.

Relative-Kendall-Tau-Distance(A, B, N){//A and B are the first and second permuations respectively
for($i = 0$; $i \leq$ N ; $i{+}{+}$) { // N is size of the permutation
$Id\_new[A[i]] = i$; // Id_new denotes the new identity permutatuion
}
for($i = 0$; $i \leq$ N ; $i{+}{+}$) {
$B\_new[i] = id\_new[B[i]]$; //B_new is the permuation B if Id_new is the identity permutation
}
return Inversions-count-Merge-sort($B\_new, 0, N - 1$) ; //Find number of inversions for B_new
}

Here the modification is simply done in $O(n)$ and we already know finding number of inversions can be done in $O(n \log n)$. So the complexity of this algorithm is also $O(n \log n)$.

Problem 3. Lower bound for sorting with equal keys. Suppose that there are k distinct key values, with the ith key value occurring f_i times in the array A. There are a total of $n = f_1 + f_2 + \ldots + f_n$ keys. Show that any comparison sorting algorithm must make at least $nH - n$ comparisons, where, H is the Shannon entropy defined as

$$H = -\left(p_1 \log_2 p_1 + p_2 \log_2 p_2 + \ldots + p_k \log_2 p_k\right)$$

and $p_i = f_i/n$. (***Hint:*** You can use the fact that the number of arrangements with k distinct keys and with the ith key value occurring f_i times is $\frac{n!}{f_1! f_2! \ldots f_k!}$. Now revisit the comparison sort lower bound done in class.)

Answer The number of arrangements with k distinct keys and with the ith key value occurring f_i times is $\frac{n!}{f_1!f_2!\ldots f_k!}$. This means the number of comparisons are lowerbounded by $\log(\frac{n!}{f_1!f_2!\ldots f_k!})$

$$= \log\ n! - \sum_{i=1}^{k} \log f_i!$$

Using Stirling's approximation, we have $n \log n > \log n! > n \log n - n$. So, the no. of comparisons,

$$> n \log n - n - \sum_{i=1}^{k} (f_i \log f_i)$$

Now, we have given $H = -\left(p_1 \log_2 p_1 + p_2 \log_2 p_2 + \ldots + p_k \log_2 p_k\right)$ and $p_i = f_i/n$.

So, $H = -\sum p_i \log_2 p_i = -\sum \frac{f_i}{n} \log_2 \frac{f_i}{n} = -\sum \frac{f_i}{n} (\log_2 f_i - \log_2 n)$

$$\implies nH = -\sum f_i \log_2 f_i + \sum f_i \log_2 n = -\sum f_i \log f_i + n \log n$$

Substituting this value, no of comparisons $> nH - n$