Homework 5

Tushar Jain, N12753339
CSCI-GA 1170-003 - Fundamental Algorithms

October 12, 2017

Problem 1.a. What is the smallest possible depth of a leaf in a decision tree for a comparison sort?

Solution:

The absolute best case happens when we just check every element and see that the data's already sorted.

This will result in $n-1$ comparisons and thus the leaf will have a depth of $n-1$. Thus, it's $\Theta(n)$, or more precisely, $n-1$. This is the minimal number of comparisons we need to perform in order to check if an array is sorted and return it. It's what insertion sort does. □

Problem 1.b. Show that there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n. What about a fraction of $1/n$ of the inputs of length n? What about a fraction $1/2^n$?

Solution:

For depth h, the maximum number of leaves is 2^h and according to the question, the comparison sort reaches at least half of $n!$ in linear time, it must hold that:

$$\begin{aligned}
&\frac{n!}{2} \leq 2^h \\
&h \geq \log(\frac{n!}{2}) && \text{(Taking log on both sides)} \\
&\quad = \log(n!) - 1 \\
&\quad = \Omega(n \log n) - 1 \\
&\quad = \Omega(n \log n)
\end{aligned}$$

Thus, the lower bound is $\Omega(n \log n)$, which is clearly more than linear.

Similarly, for fraction of $1/n$ of the inputs of length n:

$$\begin{aligned}
&\frac{n!}{n} \leq 2^h \\
&h \geq \log(\frac{n!}{n}) && \text{(Taking log on both sides)} \\
&\quad = \log(n!) - logn \\
&\quad = \Omega(n \log n) - \log n \\
&\quad = \Omega(n \log n)
\end{aligned}$$

And now similarly, for fraction of $1/2^n$ of the inputs of length n:

$$\begin{aligned}
\frac{n!}{2^n} &\leq 2^h \\
h &\geq \log(\frac{n!}{2^n}) && \text{(Taking log on both sides)} \\
&= \log(n!) - n \\
&= \Omega(n \log n) - n \\
&= \Omega(n \log n)
\end{aligned}$$

For all these cases, we have the lower bound $\Omega(n \log n)$ and not linear. □

Problem 2.a. Prove that `COUNTING-SORT` is stable.

Solution:
Pseudo-Code:

```
1  def COUNTING-SORT(A, B, k):
2    let C[0...k]  be a new array
3    for i = 0 to k:
4      C[i]  = 0
5    for j = 1 to A.length:
6      C[A[j]] = C[A[j]] + 1
7      # C[i] now contains the number of elements equal to i.
8    for i=1 to k:
9      C[i]  = C[i] + C[i - 1]
10     # C[i] now contains the number of elements less than or equal to i.
11   for j = A.length down to 1:
12     B[C[A[j]]] = A[j]
13     C[A[j]] = C[A[j]] - 1
```

Proof:

Suppose positions i and j with $i < j$ both contain some element k. We consider lines 11 through 13 of `COUNTING-SORT`, where we construct the output array. Since $j > i$, the loop will examine $A[j]$ before examining $A[i]$. When it does so, the algorithm correctly places $A[j]$ in position $m = C[k]$ of B. Since $C[k]$ is decremented in line 13, and is never again incremented, we are guaranteed that when the for loop examines $A[i]$ we will have $C[k] < m$. Therefore, $A[i]$ will be placed in an earlier position of the output array, proving stability.

□

Problem 2.b. Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, and quicksort? Justify your answers with short proofs or counter-examples.

Solution:

Insertion Sort: Stable
Pseudo-Code:

```
1  def INSERTION-SORT(A):
2    for j = 2 to A.length:
3      key = A[j]
```

```
4       # Insert A[j]    into the sorted sequence A[1...j-1]
5       i=j-1
6       while i > 0 and A[i] >key:
7         A[i + 1]  = A[i]
8         i=i-1
9       A[i+1] = key
```

Proof:

Suppose positions l and m with $l < m$ both contain some element k. We consider line 6 in the `INSERTION-SORT` algorithm above, where we check conditions before modifying the output array. Since $l < m$, the for loop starting at line 2 will first find position for the element placed at l position before examining $A[m]$. Then after it is placed at the correct position when the situation arises such that the *key* is equal to $A[i]$, the while loop condition will not be satisfied and the later element will be placed after the previously encountered occurrences of the same value.

Merge Sort: Stable

Proof:

During the merge step of 2 sub-arrays (say, $Left$ and $Right$) in the merge sort, it's ensured that when 2 elements are equal, the $Left$ is returned by the conditional statement: if(Left[i] $\leq$ Right[j]). Thus, the order of elements with same values is preserved and hence, merge-sort is stable.

Heap Sort: Not Stable

Counter example:

Consider an array with all the elements having the same value (say, k). For example, A = [k,k,...k,k] or A = [7,7,...,7,7]. As array A is already a maxheap according to the $BUILD - MAXHEAP$ method, it will return the same array. Thus, the $HEAPSORT$ algorithm will place root element (k or 7) as the last element in the sorted array. Therefore, the order of elements with same value will be changed and hence, heap sort is not stable.

Quick Sort: Not Stable

Counter example:

Partitions are created and elements are swapped depending on their relative value to the pivot (could be the last element or random element), this makes the behavior unpredictable and thus, quick sort does not ensure stability. For a thorough look, let's consider the following example:

$$A = [6, 2_1, 7_1, 4, 9, 1, 8, \mathbf{5}, 7_2, 3_1, 3_2, 3_3, 10, 2_2, 11]$$

Since the index pointers used to sort a sub-array move from the ends of the sub-array towards the pivot element so outer elements get swapped before the middle elements. Consider 5 to be the pivot in our case. After the sweep (and before the recursive calls) the sub-array looks like this:

$$A = [2_2, 2_1, 3_3, 4, 3_2, 1, 3_1, \mathbf{5}, 7_2, 8, 9, 7_1, 10, 6, 11]$$

All elements sharing the same key have had their orders reversed.

□

Problem 2.c. Give a simple scheme that makes any sorting algorithm stable, by treating the algorithm as a black-box (i.e. you cant change the code of the algorithm, but you can add some code that calls the algorithm on whatever input you like). How much additional time and space does your scheme entail?

Solution:

To make any sorting algorithm stable we can preprocess, replacing each element of an array with an ordered pair. The first entry will be the value of the element, and the second value will be the index of the element. For example, the array [2, 1, 1, 3, 4, 4, 4] would become [(2, 1),(1, 2),(1, 3),(3, 4),(4, 5),(4, 6),(4, 7)]. We now interpret $(i, j) < (k, m)$ if $i < k$ or, $i = k$ and $j < m$ (lexicographically). Under this definition of less-than, the algorithm is guaranteed to be stable because each of our new elements is distinct and the index comparison ensures that if a repeat element appeared later in the original array, it must appear later in the sorted array. This doubles the space requirement, but the running time will be asymptotically unchanged.

That's it takes an additional $\Theta(n)$ space but without changing the time complexity.

□

Problem 3. Describe an algorithm that, given n integers in the range 0 to k (inclusive), preprocesses its input and then answers any query about how many of the n integers fall into a range $[a...b]$ in $O(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time. Briefly prove correctness and running time.

Solution:

For preprocessing, we can simply count the frequency (let's say in array C with 0-indexing) of each number in the given range of 0 to k in $O(n)$ and then find the cumulative frequency by iterating over the frequency array C and summing up in $O(k)$ time complexity exactly like the part of `COUNTING-SORT` which build up array C with the property that $C[i]$ contains the number of elements less than or equal to i in the original array. For all queries, using direct access to elements in array C we can respond with $C[b] - C[a-1]$ (where $C[-1] = 0$), that is in $O(1)$ time.

Pseudo-Code:

```
1  def PREPROCESS(A, k):
2    # let C[0...k]  be a new array initialized with 0
3    C = [0]*(k+1)
4
5    for j = 1 to A.length:
6      C[A[j]] = C[A[j]] + 1
7      # C[i] now contains the number of elements equal to i.
8    for i=1 to k:
9      C[i] = C[i] + C[i - 1]
10     # C[i] now contains the number of elements less than or equal to i.
11   return C
12
13 def QUERY(a, b, C):
14   if a == 0:
15     return C[b]
16   return (C[b] - C[a-1])
```

Correctness:

After preprocessing, $C[i]$ array C contains the number of elements less than or equal to i in the original array A. Therefore, number of elements in the range $[a, b]$ is given by (C[b] - C[a-1]) which is implemented in the `QUERY` method.

Running Time:

On considering lines 5-6 in the `PREPROCESS` method, we see it takes $O(n)$ time, while the for loop in lines 8-9 take $O(k)$ time. Thus, the `PREPROCESS` method has a time complexity of order $\Theta(n + k)$. However, the `QUERY` method does 2 simple direct access of elements in an array and thus, takes only $O(1)$ (or constant) time.

□

Problem 4. Suppose that you have a *black-box* worst-case linear-time median subroutine. Give a simple, linear-time algorithm that solves the selection problem for an arbitrary order statistic. Briefly prove correctness and running time.

Solution:

To use it, we just find the median, partition the array based on that median. If i is less than half the length of the original array, recurse on the first half, if i is half the length of the array, return the element coming from the median finding black box. Lastly, if i is more than half the length of the array, subtract half the length of the array, and then recurse on the second half of the array.

Pseudo-Code:

```
1  def select(A, p, r, i):
2      if p + 1 == r:
3          return A[p]
4      x = black_box_median(A, p, r)
5      q = partition(A, p, r, x)
6      k = q - p + 1
7      if i == k:
8          return A[q]
9      if i < k:
10         return select(A, p, q, i)
11     return select(A, q + 1, r, i - k)
12
13 # Pythonic implementation of the same
14 def select(A, k):
15     med     = black_box_median(items)
16     smaller = [item for item in A if item < med]
17     larger  = [item for item in A if item > med]
18
19     if len(smaller) == k:
20         return med
21     elif len(smaller) > k:
22         return select(smaller, k)
23     else:
24         return select(list(larger), k - len(smaller) - 1)
```

Correctness:

Assuming that the $black_box_median$ works correctly and partition splits the array correctly around the median.

- Thus, if $i = k$, the current element is returned which is correct.
- if $i < k$, it looks in subarray A[p...q] which is correct as the i-th smallest element will be in this left subarray.
- else it looks in subarray A[q+1...r] which is correct as the $i-k$-th smallest element will be in this right subarray because k smaller elements were in the left subarray.

Running Time:

We find the median in linear time partition the array around it (again, in linear time). If the median index (always $\lceil n/2 \rceil$) equals n we return the median. Otherwise, we recurse either in the lower or upper part of the array, adjusting n accordingly.
This yields the following recurrence:

$$T(n) = T(n/2) + O(n)$$

Applying the master theorem, we get an upper bound of $O(n)$.

□

Problem 5. Describe a linear time algorithm which, given an n elements array A and a number $k < n$, returns k elements of A which are closest to the median of A (excluding the median itself). For example, if $A = (10, 5, 11, 1, 6, 7, 25)$ and $k = 2$, the median of A is 7, and 2 closest numbers to 7 are 6 and 5. Briefly prove correctness and running time.

Solution:

The following is overview of an algorithm that satisfies the given constraints in the question:

- We find the median of the array in linear time.
- We find the absolute distance of each element to the median in linear time.
- We find the k-th order statistic (i.e. the k-th smallest) of the distance, again, in linear time.
- We now select only the elements that have distance lower than or equal to the k-th order statistic. This provides us with the desired values.

Pseudo-Code:

```
1  def k_closest(A, k):
2      median = black_box_median(A)
3      B = [abs(item - median) for item in A]
4      kth = select(B, k) # from previous question
5      median_excluded = False
6      for item in B:
7          if item == 0:
8              if not median_excluded:
9                  median_excluded = True
10             else:
```

```
11                  closest.append(item)
12          elif (item < kth):
13              closest.append(item)
14      for item in B:
15          if len(closest) >= k:
16              break
17          if item == kth:
18              closest.append(item)
19      return closest
```

Correctness:

The above algorithm is correct as it returns a list of k closest elements to the median.

Running Time:
Running time is O(n) as the median takes O(n) and so do the individual loops starting at lines 3,6 and 14. □

Problem 6. Suppose you are given two sorted lists A, B of size n and m, respectively. Give an $O(logk)$ algorithm to find the k-th smallest element in AB, where $k \leq min(m, n)$. Briefly prove correctness and running time.

Solution:
Pseudo-Code:

```
1
2  def kth(A, B, k):
3    m, n = len(A), len(B)
4
5    if (k > (m+n) or k < 1):
6      return None
7
8    # Force m <= n
9    if (m > n):
10     return kth(B, A, k)
11
12   # if A is empty returning k-th element of B
13   if (m == 0):
14     return B[k - 1]
15
16   # if k = 1 return minimum of first two elements of both arrays
17   if (k == 1):
18     return min(A[0], B[0])
19
20   # now the divide and conquer part
21   i, j = min(m, k // 2), min(n, k // 2) # python3 division
22
23   if (A[i - 1] > B[j - 1] ) :
24     #Now we need to find only k-j th element since we have found out the lowest j
25     return kth(A, B[j:], k - j)
26   else:
```

```
27      # Now we need to find only k-i th element since we have found out the lowest i
28      return kth(A[i:], B, k - i)
```

Correctness: The algorithm works on the basis of dividing the arrays into segments of $k/2$ and finding appropriate elements in the correct segments.
Base Step: $k = 1$, algorithm returns min(A[0], B[0]).

Induction Step: Let us suppose, the algorithm is correct for $k = l$. Then for $k = l+1$, in lines 21-28 it selects a segment in which the $(l+1)$-th largest exists and then recursively calls the algorithm for smaller size for which we know it's true. Thus, the algorithm is correct.

Running Time:

$$T(k) = T(k/2) + C$$

Using Master Theorem:

$$T(k) = \log(k)$$

□

Problem 7. Show that the second smallest element of an n element (distinct) array can be found with $n + \lceil \log n \rceil - 2$ comparisons in the worst case.

Hint: Also find the smallest element.

Solution:

The first step consists of finding the smallest element. The standard algorithm simply compares the second and later elements with the smallest. We can thus find the smallest in n - 1 comparisons. Unfortunately, this does not solve the problem. The first pass is modified so that the n elements are first compared in pairs (n/2 comparisons), the winners are compared (n/4 comparisons) all the way to the last two. Total number of comparisons are still n - 1. Note that the second smallest element must have been compared to the smallest, and lost. The trick consists in maintaining a list of losers for every comparison. We then enter the list of losers attached to the smallest element (i.e. skipping the smallest element) and find the smallest in this list. The list has $\lceil \log n \rceil$ elements and finding the smallest element involves one fewer comparisons: $\lceil \log n \rceil 1$ for a total of $\lceil \log n \rceil - 2$. Note that maintaining the list adds a constant cost to each comparison.
Pseudo-Code:

```
1   def minTournament(A):
2     ''' return : (smallest, [candidate list for 2nd smallest ])
3     '''
4     n = len(A)
5     if n ==1:
6       return (A[0], [])
7     if n == 2:
8       if A[1] > A[0]:
9         return (A[0],A[1:])
10      else:
11        return (A[1],A[0:1])
12
13    # We know n >=3
```

```
14    mid = n//2
15    l1, l2 = minTournament(A[:mid])
16    r1, r2 = minTournament(A[mid:])
17
18    if r1 < l1:
19      min1 = r1
20      min2 = r2 + [l1]
21    else:
22      min1 = l1
23      min2 = l2 + [r1]
24
25    return (min1,min2)
26
27  def second_smallest(A):
28    '''
29    Description: with $n + \lceil \log n \rceil - 2$ comparisons in the worst case.
30    return: (min1, min2)
31    '''
32    min1, min2_candidates = minTournament(A)
33    min2, min3_candidates = minTournament(min2_candidates)
34    #For finding min2 from min2_candidates, any algorithm using n-1 comparisons can be used.
35    return (min1, min2)
```

The above algorithm returns the `smallest` element in the whole array A and a list of candidate elements for the second smallest element.

The number of comparisons is characterized by the following recurrence relation:

$$T(1) = 0$$
$$T(2) = 1$$
$$T(n) = 2 \cdot T(\frac{n}{2}) + 1$$

This recurrence is easily solved to get $T(n) = n - 1$.

The size of the candidate list for second smallest element can be characterized by the following recurrence:

$$G(1) = 0$$
$$G(2) = 1$$
$$G(n) = 1 + G(\frac{n}{2})$$

$G(n)$ is easily solved to be $\log_2 n$. We can find the smallest element in the candidate list using at most $\log_2 n - 1$ comparisons. Thus, in total, the second smallest element can be found in at most $n + \log_2 n - 2$ comparisons. □