Week 4 Applied Sheet

(Solutions)

Useful advice: The following solutions pertain to the theoretical problems given in the applied classes. You are strongly advised to attempt the problems thoroughly before looking at these solutions. Simply reading the solutions without thinking about the problems will rob you of the practice required to be able to solve complicated problems on your own. You will perform poorly on the exam if you simply attempt to memorise solutions to these problems. Thinking about a problem, even if you do not solve it will greatly increase your understanding of the underlying concepts. Solutions are typically not provided for Python implementation questions. In some cases, pseudocode may be provided where it illustrates a particular useful concept.

Problems

Problem 1. In the seminars, a probability argument was given to show that the average-case complexity of Quicksort is $O(n\log(n))$. Use a similar argument to show that for an input array of n elements the average-case complexity of Quickselect with random pivots is $O(n)$.

Solution

In the seminars, we considered the behaviour of Quicksort in the situation where the pivot always fell in the middle 50% of the sorted sequence. Let us call such a pivot a "good pivot". If we always select a good pivot, then the worst outcome is when the pivot lies on the extreme of the good range, either at the 25^{th} percentile or the 75^{th} percentile. If the target element lies in the adjacent 75%, which would be the worst case, Quickselect recurses on a list 75% as large as the original list. Therefore, for some constant c, the total amount of work performed will be

$$T(n) = c\cdot n + 0.75\cdot c\cdot n + 0.75^2\cdot c\cdot n + 0.75^3\cdot c\cdot n + ...$$

This is a geometric series, which we know is bounded by

$$T(n) = (1+0.75+0.75^2+0.75^3+...)\cdot c\cdot n \leq \left(\frac{1}{1-0.75}\right)\cdot c\cdot n = 4\cdot c\cdot n.$$

Therefore, in the worst case when selecting a good pivot every time, Quickselect will take on the order of $4\cdot c\cdot n$ operations. However, we will not likely select a good pivot every single time. Since we have a 50% shot at selecting a good pivot, it will take 2 tries in expectation to select one. Therefore the expected amount of work performed by Quickselect is at most twice the amount of work performed when always selecting a good pivot. Therefore the amount of work we require is at most $2\cdot 4\cdot c\cdot n = 8\cdot c\cdot n = O(n)$. See the course notes for a more rigorous argument involving recurrence relationships.

Problem 2. Devise an algorithm that, given a box with n different locks and n corresponding keys, matches the keys and locks in average-case time complexity $O(n\log n)$. Each lock matches only one key, and each key matches only one lock. You can try a key in a lock to determine whether the key is larger than, smaller than or fits the lock. However, you cannot compare two keys or two locks directly.

Solution

You can use Quicksort to solve this problem, by using the following procedures

1. Pick an arbitrary lock and compare it with every key. Place keys that are too small to the left, and keys that are too large to the right.
2. Take the matching key found in Step 1, and compare to all other locks. Place locks that are too small

to the left, and locks that are too large on the right.

3. Recurse with Steps 1 and 2 on the left and right subsets of keys/locks until all keys and locks are matched.

Step 1 makes n comparisons and Step 2 makes $n-1$ comparisons. So partitioning takes $\Theta(n)$ operations. The analysis of the depth of the recursion tree is similar to Quicksort, i.e., on average-case $\Theta(\log n)$ levels of recursion.

Problem 3. Devise an algorithm that given a sequence of n unique integers and some integer $1 \leq k \leq n$, finds the k closest numbers to the median of the sequence. Your algorithm should run in $O(n)$ time. You may assume that Quickselect runs in $\Theta(n)$ time (achievable by using median of medians to find good pivots).

Solution

First, let's run Quickselect to locate the median of the sequence. The problem is to find the k closest elements to this. Intuitively, let's suppose that we make a copy of the sequence, but subtract the median from every element. The problem is now to find the k numbers closest to zero, or equivalently, the k smallest numbers in absolute value. We could therefore take the absolute value of the elements, run Quickselect again to find the k^{th} element and then take all elements that are less than this (except we take their corresponding equivalent in the unmodified sequence of course). This works in $\Theta(n)$ time since we run Quickselect twice and make a copy of the sequence, but has $\Theta(n)$ auxiliary space complexity.

To remove the space overhead, note that we can simply run a modified Quickselect where we interpret every element of the sequence as being its absolute difference from the median without actually making any copy of the data. This way, we will have an algorithm that runs in $\Theta(n)$ time and uses no additional space except for the output.

Problem 4. One common method of speeding up sorting in practice is to sort using a fast sorting algorithm like Quicksort or Mergesort until the subproblems sizes are small and then to change to using insertion sort since insertion sort is fast for small, nearly sorted lists. Suppose we perform Mergesort until the subproblems have size k, at which point we finish with insertion sort. What is the worst-case running time of this algorithm?

Solution

If we Mergesort until the subproblems are size k, we will perform roughly d levels of recursion, where d satisfies

$$n\left(\frac{1}{2}\right)^d = k.$$

Solving for d reveals $d = \log_2\left(\frac{n}{k}\right)$. Therefore we will expect to perform $\Theta\left(n\log\left(\frac{n}{k}\right)\right)$ work for the Mergesort part of the algorithm. At this stage, there are $\Theta(n/k)$ independent subproblems each of size k. Insertion sort for each subproblem with k elements takes $\Theta(k^2)$ in the worst-case. So the total worst-case cost for the insertion sort part of the algorithm is $\Theta(nk)$. Thus, the worst-case running time for this algorithm will be $\Theta\left(nk + n\log\left(\frac{n}{k}\right)\right)$.

Problem 5. A subroutine used by Quicksort is the partitioning function which takes a list and rearranges the elements such that all elements $\leq p$ come before all elements $> p$ where p is the pivot element. Suppose one instead has $k \leq n$ pivot elements and wishes to rearrange the list such that all elements $\leq p_1$ come before all elements that are $> p_1$ and $\leq p_2$ and so on..., where $p_1, p_2, ..., p_k$ denote the pivots in sorted order. The pivots are not necessarily given in sorted order in the input.

(a) Describe an algorithm for performing k-partitioning in $O(nk)$ time. Write pseudocode for your algorithm.

(b) Describe a better algorithm for performing k-partitioning in $O(n\log(k))$ time. Write pseudocode for your algorithm.

(c) Is it possible to write an algorithm for k-partitioning that runs faster than $O(n\log(k))$?

Solution

First, let's sort the pivots so that we have $p_1 < p_2 < p_3 < ... < p_k$ in sorted order in time $\Theta(k\log(k))$. To perform k-partitioning, we'll reduce it to the problem of ordinary 2-way partitioning. The simplest solution is the following. Let's first perform 2-way partitioning on the first pivot p_1. Then we perform 2-way partitioning on the subarray that comes after p_1, using p_2 as the pivot and so on. There are k partition calls and in the worst-case they take a total time of $\Theta(nk)$. Since, $n \geq k$, the total cost is $\Theta(nk + k\log(k)) = \Theta(nk)$. A possible pseudocode implementation is given below.

```
1: function K_PARTITION(a[1..n], p[1..k])
2:     sort(p[1..k])
3:     Set j = 1
4:     for i = 1 to k do
5:         j = partition(a[j..n], p[i]) + 1    // The ordinary 2-way partitioning algorithm.
6:     end for
7: end function
```

Note that we assume that the ordinary partition function returns the location of the pivot after partitioning. To improve on this, let's use divide and conquer. We will pivot on the middle pivot $p_{k/2}$ first, and then recursively partition the left half with the left $k/2$ pivots and the right half with the right $k/2$ pivots. This strategy will require $\Theta(\log(k))$ levels of recursion, and at each level we will perform partitioning over the sequence of size n, taking $\Theta(n)$ time, adding to $\Theta(n\log(k))$ time. Sorting the pivots takes $\Theta(k\log(k))$, hence the total cost of this algorithm will be $\Theta(n\log(k))$.

```
1: function K_PARTITION(a[1..n], p[1..k])    // Assumes pivots are sorted before calling k_partition
2:     if k > 0 and n > 0 then
3:         Set mid = k/2
4:         j = partition(a[1..n], p[mid])    // The ordinary 2-way partitioning algorithm.
5:         k_partition(a[1..j-1], p[1..mid-1])
6:         k_partition(a[j+1..n], p[mid+1..k])
7:     end if
8: end function
```

We cannot write an algorithm that performs better than this in the comparison model. Suppose we are given a sequence and we select all n elements as pivots. Performing k-partitioning is then equivalent to sorting the sequence, which has an $\Omega(n\log(n))$ lower bound. If we could do k-partitioning faster than $O(n\log(k))$, when $k = n$, this would surpass the lower bound.

Problem 6. Suppose for an array of unique elements Bob implements Quicksort by selecting the average element of the sequence (or the closest element to it) as the pivot. Recall that the average is the sum of all of the elements divided by the number of elements. What is the worst-case time complexity of Bob's implementation? Describe a family of inputs that cause Bob's algorithm to exhibit its worst-case behaviour.

Solution

Since good pivot choices are those that split the list into roughly equal halves and the average is likely to be near the middle, it is tempting to say that this choice will make the worst case time complexity $O(n\log(n))$. However, this is incorrect. There are sequences for which the average is very far away from the median. Consider, for example, the sequence $a_i = (i!)$ for $1 \leq i \leq n$. The average of this sequence is

$$\frac{1}{n}\sum_{i=1}^{n}(i!) \geq \frac{n!}{n} = (n-1)!$$

Hence only the last element is bigger than the average, and therefore this pivot strategy will cause the algorithm to take $\Theta(n^2)$ time.

Problem 7. Consider a generalisation of the median finding problem, the *weighted median*. Given n unique elements $a_1, a_2, ...a_n$ each with a positive weight $w_1, w_2, ..., w_n$ all summing up to 1, the weighted median is the element a_k such that

$$\sum_{a_i<a_k} w_i \leq \frac{1}{2} \qquad \text{and} \qquad \sum_{a_i>a_k} w_i \leq \frac{1}{2}$$

Intuitively, we are seeking the element whose cumulative weight is in the middle (around $\frac{1}{2}$). Explain how to modify the Quickselect algorithm so that it computes the weighted median. Give pseudocode that implements your algorithm.

Solution

When using Quickselect to find the median, we partition around some pivot element p and then recurse on the left half of the array if the final position of p is greater than $\frac{n}{2}$, or on the right half if the final position of p is less than $\frac{n}{2}$. We can easily modify this to find the weighted median by summing the weights of the elements on either side of the pivot. If neither of these is greater than $\frac{1}{2}$ then the pivot is the weighted median. Otherwise we recurse on the side that has total weight greater than $\frac{1}{2}$. The following pseudocode assumes that the `weight` array is permuted in unison with the `array` during partitioning so that the weights always line up correctly. This pseudocode is general and works for any weight; to find the weighted median, call WEIGHTED_QUICKSELECT with a weight of $w = \frac{1}{2}$.

```
1:  function WEIGHTED_QUICKSELECT(array[lo..hi], weight[lo..hi], w)
2:      if hi > lo then
3:          Set pivot = array[lo]
4:          j = partition(array[lo..hi], weight[lo..hi], pivot)
5:          if sum(weight[lo..j-1]) > w then
6:              return weighted_quickselect(array[lo..j-1], weight[lo..j-1], w)
7:          else if sum(weight[lo..j]) ≥ w then
8:              return array[j]
9:          else
10:             return weighted_quickselect(array[j+1..hi], weight[j+1..hi], w - sum(weight[lo..j]))
11:         end if
12:     else
13:         return array[j]
14:     end if
15: end function
```

Note that if there are two elements satisfying the conditions (in which case the border between them splits the weight into halves), then the algorithm will select one of them as the weighted median.

Supplementary Problems

Problem 8. Write pseudocode for a version of Quickselect that is iterative rather than recursive.

Solution

Assuming that we use a three-way partitioning algorithm (see the course notes) that returns `left, right` indicating the position of the first element equal to the pivot, and the first element greater than the pivot respectively, we can write Quickselect iteratively like so.

```
1: function QUICKSELECT(array[1..n], k)
2:     Set lo = 1, hi = n
3:     while lo ≤ hi do
4:         Set pivot = array[lo]
5:         left, right = partition(array[lo..hi], pivot)
6:         if k < left then
7:             hi = left - 1
8:         else if k ≥ right then
9:             lo = right
10:        else
11:            return array[k]
12:        end if
13:    end while
14:    return array[k]
15: end function
```

Problem 9. Write an algorithm that given two sorted sequences $a[1..n]$ and $b[1..m]$ of unique integers finds the k^{th} order statistic of the union of a and b

1. Your algorithm should run in $O(\log(n)\log(m))$ time.
2. **(Advanced)** Your algorithm should run in $O(\log(n)+\log(m))$ time.

Solution

Suppose without loss of generality that the k^{th} order statistic is in the sequence a (if it isn't, we can swap a and b and try again). The order statistic of $a[i]$ is given by

$$\text{order}(a[i]) = i + \text{count}(j : b[j] < a[i]) + 1$$

We can compute `count` in $\log(m)$ time using a binary search on the elements of b. Note that the order function is increasing with respect to i, hence it is binary searchable. Using count as a subroutine, we can binary search the order function over a to find the least element whose order statistic is $\geq k$. This element is then either the k^{th} order statistic, or we deduce that it is not in which case we swap a and b and try again. Some pseudocode is given below.

```
1: // Counts the number of elements of s that are less than x
2: function COUNT(s[1..n], x)
3:     if s[1] ≥ x then return 0
4:     // Invariant: s[lo] < x, s[hi] ≥ x
5:     Set lo = 1, hi = n + 1
6:     while lo < hi - 1 do
7:         Set mid = ⌊(lo + hi)/2⌋
8:         if s[mid] < x then lo = mid
9:         else hi = mid
10:    end while
11:    return lo
12: end function
13:
14: function SELECT_KTH(a[1..n], b[1..m], k)
15:    // Invariant: order(a[lo]) ≤ k, order(a[hi]) > k
16:    Set lo = 1, hi = n + 1
17:    while lo < hi - 1 do
18:        Set mid = ⌊(lo + hi)/2⌋
```

```
        if lo + count(b[1..m], a[lo]) ≤ k then lo = mid
        else hi = mid
    end while
    if lo + count(b[1..m], a[lo]) = k then return lo
    else return select_kth(b[1..m], a[1..n], k)
end function
```

See how if we fail to find a solution in a, we simply swap a and b and try again, which is guaranteed to work since the k^{th} order statistic must be in one of the two. Our solution uses two nested binary searches and hence has time complexity $\Theta(\log(n)\log(m))$.

To do this in just $\Theta(\log(n)+\log(m))$ time, we'll need to eliminate the nested binary searches. Let's see if we can figure out something else to exploit that can be binary searched without needing a nested one. How well can we estimate count and order without doing a binary search?

Suppose that we are looking at $a[i]$ and $b[j]$. If $a[i] > b[j]$, then we can deduce that

$$\text{COUNT}(b[1..m], a[i]) \geq j$$

which means that

$$\text{ORDER}(a[i]) \geq i+j \qquad \text{and} \qquad \text{ORDER}(b[j]) < i+j.$$

If $i+j > k$, then this implies that $\text{ORDER}(a[i]) > k$. Otherwise $i+j \leq k$, and it implies that $\text{ORDER}(b[j]) < k$. The same deductions hold with a and b reversed. Note that although we cannot compute ORDER exactly, these inequalities satisfy the invariants that we were binary searching above, and can be computed in constant time. The trick is therefore not to perform two nested binary searches, but two simultaneous binary searches in which we search both a and b at the same time using these inequalities.

Once one of the binary search ranges reduces down to one element, then either that element is the k^{th} order statistic, or it is the largest element of the other sequence that is less than the k^{th} order statistic, and hence the k^{th} order statistic is the $(k-\text{lo})^{\text{th}}$ element of the other sequence. Pseudocode implementing these ideas is shown below.

```
function SELECT_KTH(a[1..n], b[1..m], k)
    // Corner cases: check whether one sequence is entirely greater than the kth element of the other
    if count(a[1..n], b[1]) ≥ k then return a[k]
    if count(b[1..m], a[1]) ≥ k then return b[k]
    // Two simultaneous binary searches!
    // Invariant: order(a[lo1]) ≤ k, order(a[hi1]) > k, order(b[lo2]) ≤ k, order(b[hi2]) > k
    Set lo1 = 1, hi1 = n + 1, lo2 = 1, hi2 = m + 1
    while lo1 < hi1 - 1 and lo2 < hi2 - 1 do
        Set i = ⌊(lo1 + hi1)/2⌋, j = ⌊(lo2 + hi2)/2⌋
        if a[i] > b[j] then
            if i + j > k then hi1 = i
            else lo2 = j
        else
            if i + j > k then hi2 = j
            else lo1 = i
        end if
    end while
    // Check which binary search converged first and find the answer
    if lo1 = hi1 - 1 then
        if lo1 + count(b[1..m], a[lo1]) = k then return a[lo1]
        else return b[k - lo1]
    else
```

```
23:         if lo2 + count(a[1..n], b[lo2]) = k then return b[lo2]
24:         else return a[k - lo2]
25:     end if
26: end function
```

Since we perform three calls to COUNT, each costing $\Theta(\log(n))$ or $\Theta(\log(m))$ and we perform simultaneous binary searches on a and b each costing at most $\Theta(\log(n)+\log(m))$ in total, the overall time complexity of this solution is $\Theta(\log(n)+\log(m))$.