Week 4 Studio Sheet

(Solutions)

Useful advice: The following solutions pertain to the theoretical problems given in the tutorial 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 the tutorial 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, psuedocode may be provided where it illustrates a particular useful concept.

Implementation checklist

By the end of week 4, write Python code for:

1. **Quicksort** (see Problem 6)
2. **Quickselect** (see Problem 7)

Assessed Preparation

Problem 1. What are the worst-case time complexities of Quicksort assuming the following pivot choices:

(a) Select the first element of the sequence

(b) Select the minimum element of the sequence

(c) Select the median element of the sequence

(d) Select an element that is greater than exactly 10% of the others

Describe a family of inputs that cause Quicksort to exhibit its worst case behaviour for each of these pivot choices.

Solution

If we select the first element of the sequence, the worst-case complexity will be $O(n^2)$. To see this, consider applying Quicksort to a list that is already sorted. In this case, the pivot will always have no elements to its left, and every other element to its right. This means that we recurse on lists of size n, $n-1$, $n-2$, ... which will add up to $O(n^2)$ time.

Selecting the minimum element of the sequence is even worse. Not only will this have worst-case complexity $O(n^2)$, it actually has **best-case** complexity $O(n^2)$ since we will always have every element on the right side of the pivot. This means that any sequence will cause the worst-case behaviour.

Selecting the median is the optimal choice. The median is the element that splits the sequence into two equal halves, hence we will only require $O(\log(n))$ levels of recursion, and the worst-case complexity will be $O(n\log(n))$. This behaviour will occur for any input sequence.

Selecting the 10^{th} percentile element sounds bad since we split the list into sublists of size 10% and 90% which is rather unbalanced, but this actually still has good performance. After recursing to a depth of k, the list will be of size at most $0.9^k n$, hence we hit the base case after

$$0.9^k n = 1.$$

Solving this, we find

$$k = \log_{\frac{10}{9}}(n) = O(\log(n)).$$

Even though the base of the logarithm is worse, we still achieve $O(n\log(n))$ performance since the subproblem sizes are decreasing by a constant factor each time. Indeed, replace 10% with any percent, even 1% and you will still achieve $O(n\log(n))$ performance (although the constant factor will be rather large). This performance occurs for any input sequence.

Studio Problems

Problem 2. Suppose that 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. The average of a random sequence is likely to be close to its median, but we can construct sequences for which the average is very far away. Such a sequence would be one that grows extremely fast, so let's try one of the fastest growing functions that we know! Consider the sequence $a_i = (i!)$ for $1 \le i \le n$. The average of this sequence is

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

hence the average is closest to one of the last two elements, and the pivot will divide the list into sublists of size $n-2$ and 1, which will cause the algorithm to take $O(n^2)$ time.

Problem 3. In the lectures, 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 the average-case complexity of Quickselect is $O(n)$.

Solution

In the lectures, 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 target element lies in the adjacent 75%, then in the worst case, Quickselect recurses on a list 75% as large as the original list. Therefore, the total amount of work performed will be

$$T(n) = n + 0.75n + 0.75^2 n + 0.75^3 n + \ldots$$

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

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

Therefore, in the worst case when selecting a good pivot every time, Quickselect will take on the order of $4n$ 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 $2 \times 4n = 8n = O(n)$. See the lecture notes for a more rigorous argument involving recurrence relationships.

Problem 4. 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 $O(n)$ time.

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 $O(n)$ time since we run Quickselect twice and make a copy of the sequence, but imposes $O(n)$ extra space overhead.

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 $O(n)$ time and uses no additional space except for the output.

Problem 5. 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 $d = O\left(\log_2\left(\frac{n}{k}\right)\right)$. Therefore we will expect to perform $O\left(n \log\left(\frac{n}{k}\right)\right)$ work for the Mergesort part of the algorithm. At this stage, there are $O(n/k)$ independent subproblems each of size k. Insertion sort for each subproblem takes $O(k^2)$. So the total cost for the insertion sort is $O(nk)$. Thus, the worst-case running time for this algorithm will be $O\left(nk + n \log\left(\frac{n}{k}\right)\right)$.

Problem 6. Write a Python function that implements the Randomised Quicksort algorithm (Quicksort with random pivot selection).

Problem 7. Implement Quickselect with random pivot choice. Modify your implementation from problem 6 so that it uses Quickselect to choose a median pivot.

Problem 8. Compare your solutions to problem 6 and 7, and see which one performs better for randomly generated lists.

Problem 9. Consider an application of radix sort to sorting a sequence of nonempty strings of lowercase letters a to z. Each character of the strings is interpreted as a digit, hence we can understand this as radix sort operating in base-26. Radix sort is traditionally applied to a sequence of equal length elements, but we can modify it to work on variable length strings by simply padding the shorter strings with empty characters at the end.

(a) What is the time complexity of this algorithm? In what situation is this algorithm very inefficient?

(b) Describe how the algorithm can be improved to overcome the problem mentioned in (a). The improved algorithm should have worst-case time complexity $O(N)$, where N is the sum of all of the string lengths, i.e. it should be optimal

Solution

Since we scan each string k times, the complexity of this algorithm is $O(nk)$, where k is the length of the longest string and n is the number of strings. This is inefficient if there are many short strings in the list and only a few long ones. For example, if there were one million strings of length 10 and a single string of length one million, the algorithm would perform one trillion operations, which would take far too long.

To improve this to $O(N)$ where N is the total length of all strings, we want to avoid looking at the short strings before they actually matter. Note that a string of length k only needs to be looked at in the final k iterations. To achieve this, let's first sort the strings by length. This can be done in $O(n+k)$ using counting sort with the string's length as the key. We need to sort them shortest to longest, since we want shorter strings to come before longer strings if they share a prefix.
Then let's keep a pointer m such that the strings `S[1..m]` have length $\geq j$. Each iteration when we decrement j, we decrement m while `length(S[m+1])` $\geq j$ to account for strings that have just become worth considering. By doing so, we will only perform the inner sort on the strings that actually have characters in position j, hence we do not waste any time sorting on non-existent characters. This improves the complexity to $O(N)$ as required. An example implementation in psuedocode is shown below.

```
1:  function RADIX_SORT(S[1..n])
2:      Set k = max(length(S[1..n]))    // k is the length of the longest string
3:      sort S[1..n] by ascending length using counting sort    // Takes O(n + k) time
4:      Set m = n
5:      for j = k to 1 do
6:          while m > 1 and length(S[m − 1]) ≥ j do
7:              m -= 1
8:          end while
9:          Set count[a..z] = 0
10:         for i = m to n do
11:             count[S[i][j]] += 1
12:         end for
13:         Set pos['a'] = 1
14:         for char = 'b' to 'z' do
15:             Set pos[char] = pos[char-1] + count[char-1]
16:         end for
17:         Set temp[1..n − m + 1] = null
18:         for i = m to n do
19:             temp[pos[count[S[i][j]]]] = S[i]
20:             pos[S[i][j]] += 1
21:         end for
22:         swap(S[n..m], temp)
23:     end for
24: end function
```

Supplementary Problems

Problem 10. 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 I instead

have $k \leq n$ pivot elements and wish 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 psuedocode for your algorithm

(b) Describe a better algorithm for performing k-partitioning in $O(n \log(k))$ time. Write psuedocode 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. 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. Each call to partition takes $O(n)$ time and we perform it k times, hence this algorithm takes $O(nk)$ time plus $O(k \log(k))$ time to sort the pivots, which gives a total of $O(nk)$ time since $n \geq k$. A possible psuedocode 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 = 0 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 $O(\log(k))$ levels of recursion, and at each level we will perform partitioning over the sequence of size n, taking $O(n)$ time, adding to $O(n \log(k))$ time. Sorting the pivots takes $O(k \log(k))$, hence the total cost of this algorithm will be $O(n \log(k))$.

```
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 can not 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 11. Write psuedocode for a version of Quickselect that is iterative rather than recursive.

Solution

Assuming that we use a three-way partitioning algorithm (see the lecture 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 12. (Advanced) 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

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

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 psuedocode 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 $\frac{1}{2}$.

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

This code assumes that the `weight` array is permuted in unison with the `array` during partitioning so

that the weights always line up correctly. To find the weighted median, call WEIGHTED_QUICKSELECT with a weight of $w = \frac{1}{2}$. In fact, this code is general and works for any weight.

Problem 13. (Advanced) Consider the problem of sorting one million 64-bit integers using radix sort.

(a) Write down a formula in terms of b for the number of operations performed by radix sort when sorting one million 64-bit integers in base b

(b) Using your preferred program (for example, Wolfram Alpha), plot a graph of this formula against b and find the best value of b, the one that minimises the number of operations required. How many passes of radix sort will be performed for this value of b?

(c) Implement radix sort and use it to sort one million randomly generated 64-bit integers. Compare various choices for the base b and see whether or not the one that you found in Part (b) is in fact the best

Solution

Recall that the complexity of radix sort for n integers with k digits in base b is

$$O(k(n+b)).$$

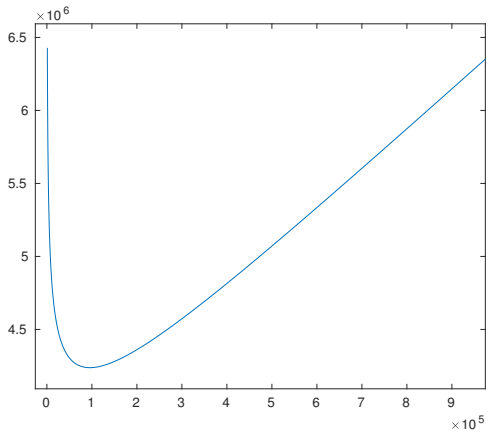
Since our integers are 64 bits, the number of digits k will be equal to

$$k = \frac{64}{\log_2(b)},$$

and hence the number of operations will be proportional to

$$\frac{64}{\log_2(b)}\left(10^6 + b\right).$$

A plot of this function looks like:



Your favourite mathematics software will then tell you that the minimum of this function occurs for $b = 95536$. Since $\log_2(95536) \approx 16.5$, and the closest divisor of 64 to that is 16, a base of $2^{16} = 65536$ should perform optimally.

My test implementation found the following results for various bases. Your results should hopefully look similar.

Base (b)	In base-2	Time (seconds)
4	2^2	23.52
16	2^4	11.77
256	2^8	6.19
65536	2^{16}	3.38
1048576	2^{20}	4.63
16777216	2^{24}	14.87

This seems to confirm our prediction that $b = 2^{16}$ would be the optimal base. Of course, the bases 2^{20} and 2^{24} are naturally disadvantaged since 20 and 24 do not divide 64, but attempting to test base 2^{32} caused my computer to run out of RAM.

Problem 14. (Advanced) Implement the median-of-medians algorithm for Quickselect as described in the lecture notes. Use this algorithm to select a pivot for Quicksort and compare its performance against the previous pivot-selection methods.

Problem 15. (Advanced) 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. 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 psuedocode 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⌋
19:         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 $O(\log(n)\log(m))$.

To do this in just $O(\log(n)+\log(m))$ 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 I am 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 $\text{ORDER}(a[i]) > k$ and $\text{ORDER}(b[j]) \leq k-1 \leq k$. The same deductions hold with a and b reversed. Note that although we can not 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. Psuedocode 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
    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
        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 $O(\log(n))$ or $O(\log(m))$ and we perform simultaneous binary searches on a and b each costing at most $O(\log(n)+\log(m))$ in total, the overall time complexity of this solution is $O(\log(n)+\log(m))$.