[redacted]

October 2, 2021

Problem Set 4

[redacted]

Due: 8 am on Thursday, October 07

Problem 4-1 (Reverse Partition) 15+9 points

Recall that the performance of QUICKSORT is entirely dependent on the choice of pivot. For example, if one were to pick the pivot as the median, you will always get equal-sized subproblems leading to an $O(n \log n)$ behavior. However, this is contingent on choosing the median as the pivot. In this question, we try to come up with an input array that will ALWAYS produce equal-sized subproblems (or nearly equal-sized) when we run the standard quicksort algorithm to sort it. In part (a), we look at an algorithm that retraces the path of PARTITION, i.e., it reconstructs the original array on which PARTITION must have been executed to produce the output. In part (c), we will use this REVERSE-PARTITION to produce an input array that always produces equal-sized outputs. To do this, you will write a recursive algorithm FAST-QS. In part (e), we look at a much simpler (and faster) REVERSE-PARTITION algorithm that still produces a valid input to PARTITION to produce the desired output.

Let us define a *pivoted array* $A[p \dots r]$ as an array such that there exists an index q where $p \leq q \leq r$ and all elements in $A[p \dots q-1]$ is less than $A[q]$ and all elements in $A[q+1 \dots r]$ are greater than $A[q]$. In other words A is a valid result of the function PARTITION where the pivot is $A[q]$. Let us assume for simplicity that A consists of distinct elements.

In this question we will study the algorithm REVERSE-PARTITION that takes as input a pivoted array $A[p \dots r]$, the indices $p, q,$ and r and modifies (in place) A to produce an array B such that PARTITION(B, p, r) returns q with B modified to A as result of the exchanges. It is presented below. You may assume that the algorithm is correct and we also welcome you to verify the details.

```
1 REVERSE-PARTITION(A, p, q, r)
2     pivot = A[q]
3     j = r
4     i = q
5     while j ≥ p + 1 and i ≥ p
6         if A[j] > pivot
7             exchange A[j] with A[i]
8             i = i − 1
9         j = j − 1
```

(a) (3 points) In this question we try to observe a structure to the result of REVERSE-PARTITION on a pivoted array. What is the value of A at the end of REVERSE-PARTITION(A, p, q, r) for:

(i) $A = [5, 6, 4, 1, 9, 10]$, $p = 1, q = 5, r = 6$?

(ii) $A = [5, 6, 4, 1, 9, 10, 11, 12, 13]$, $p = 1, q = 5, r = 9$?

(iii) $A = [5, 6, 4, 1, 9, 10, 11, 12, 13, 14]$, $p = 1, q = 5, r = 10$?

(b) (7 points) In this question we construct a *divide and conquer* algorithm FAST-QS whose goal is to modify A, in place, to produce a new array that has the following property: When we run QUICKSORT on this array the recurrence calls create problem sizes which are balanced, i.e differ by at most 1. To simplify the problem and also for ease of visualizing, the algorithm takes as input an array A and integers p, r such that $A[p, \ldots, r] = [p, \ldots, r]$, specifically we have the following driver function:

```
1 DRIVER(n)
2     Initialize A[1...n] = [1...n]
3     FAST-QS(A, 1, n)
```

As you can see, the input to FAST-QS is initially sorted. However, FAST-QS only requires that the input sub-array array $A[p, \ldots, r]$ contains integers between p and r, which is certainly met by the original input array $A = [1, \ldots, n]$.

You will present a pseudocode for the recursive algorithm FAST-QS(A, p, r). Do not forget to use induction to justify the correctness of your algorithm. Do not forget to analyze the running time of your algorithm by deriving a recurrence relation with appropriate base cases and then solving the relation. (**Hint**: Use REVERSE-PARTITION to construct this algorithm.)

(c) (5 points) Notice that the result of FAST-QS will make QUICKSORT run in $\Theta(n \log n)$ time. However, argue that the run time of INSERTION-SORT on an output of FAST-QS is $\Omega(n^2)$. (**Hint**: Let A_1, A_2 be two arrays of size n_1 and n_2 respectively, such that every element of A_1 is less than x and every element of A_2 is greater than x. Consider the following pivoted array A: $A[1 \ldots n_1] = A_1, A[n_1 + 1] = x, A[n_1 + 2, \ldots, n] = A_2$ where $n := n_1 + n_2 + 1$. Let B be the modification of A at the end of REVERSE-PARTITION$(A, 1, n_1 + 1, n)$ Where would A_1, A_2 and x occur in B? You may assume that the running time of INSERTION-SORT is $\Theta(n + I)$ where I is the number of inversions of the input array A of size n.)

(d) **(Extra Credit)** (6 points) Design an algorithm FAST-REVERSE-PARTITION that is faster than the algorithm from part (a) and achieves the sane result as before. This algorithm will achieve the most optimal runtime possible, for any algorithm. Do not forget the briefly justify the running time of your algorithm and its correctness.

Further, how does the recurrence relation for the runtime of FAST-QS from Part(c) change? State the relation and solve it.

(e) **(Extra Credit)** (3 points) How does the runtime of INSERTION-SORT computed from Part(d) change when run on the output of FAST-QS using the new FAST-REVERSE-PARTITION algorithm? (**Hint**: Derive a recurrence relation for the number of inversions in the output of FAST-QS. Solve the same and use it derive the bound for runtime of INSERTION-SORT.)

Problem 4-2 (More on Searching!) 17+5 points

When discussing merge sort, we looked at the MERGE algorithm and how it is used to merge two sorted array A_0, A_1 of lengths n_0, n_1 in time $\Theta(n_0 + n_1)$. It is natural to look at its extension on how to merge k sorted arrays $A_0, \ldots, A_{k-1}$ of lengths $n_0, \ldots, n_{k-1}$ respectively.

Linked Lists. A linked list is a data structure in which objects are arranged in linear order determined by a pointer in each object, as opposed to indices in arrays. For example, an object x in linked list will have two fields: a field $x.next$ which points to the next object in the order or **nil** if that is the last object in the linked list; and $x.key$ which contains the value. Further, a linked list L has two fields $L.head, L.tail$ which points to the first element in that list and the last element in that list respectively. If $L.head = L.tail =$ **nil** , L is empty.

(a) (8 points) Using an appropriately defined heap, present an algorithm to merge lists $A_0, \ldots, A_{k-1}$ and output the sorted list X of length $n = n_0 + \ldots + n_{k-1}$. You may assume that $A_i \cap A_j = \emptyset$ for $1 \leq i < j \leq k$. Analyze the running time of your algorithm. You will also use loop-invariant to prove the correctness of your algorithm. You may assume that $A_0, \ldots, A_{k-1}$ are represented as linked-lists with head pointers $A_0.head, \ldots, A_{k-1}.head$. Please include a brief summary of your logic before you present the pseudocode.

In the previous problem set, we looked at a much simplified problem of searching by using 3 keywords. In this problem, we look at a more general problem of searching by using k key words. As before, we will assume that the document contains n words in total, with each word being one of the k key-words. Let us represent the document by the array $A[1, \ldots, n]$ where $A[i] = j$ for $0 \leq j \leq k-1$. The goal is, as before, to find the minimum span. Recall that a *span* of A is any interval $[start, end]$ such that $\{0, 1, \ldots, k-1\} \subseteq \{A[start], \ldots, A[end]\}$. For example, if $k = 3$ and $A = (0, 1, 1, 0, 2, 1, 0)$, then $[1, 5]$ and $[4, 6]$ are spans of A, while $[1, 4]$ is not (since $2 \notin \{0, 1, 1, 0\}$). The *cardinality* of the span $[start, end]$ is defined to be $(end - start + 1)$. Finally, a span $[start, end]$ is *minimum* if it has minimum cardinality among all other spans. For the example above, $[4, 6]$ is a minimum span (of cardinality 3), while $[1, 5]$ is not a minimum span. Finally, $[\infty, \infty]$ will correspond to the case when no valid span exists for that array.

(b) (4 points) However, for this problem, it would be helpful to have the input as k linked-lists $A_0, \ldots, A_{k-1}$, where A_i has length n_i with $n_0 + \ldots + n_{k-1} = n$ and A_i is the list of all j such that $A[j] = i$, i.e., the set of words in A which is i. For example, when $k = 3$ and $A = (0, 1, 1, 0, 2, 1, 0)$, we have $A_0 = [1, 4, 7]$, $A_1 = [2, 3, 6]$, and $A_2 = [5]$. Present an efficient iterative algorithm GENERATESUBARRAY$(A[1, \ldots, n], n, k)$ and produces the pointers $A_0.head, A_1.head, \ldots, A_{k-1}.head$ to the k linked lists. You may skip the proof of correctness but briefly justify the running time of your algorithm.Please include a brief summary of your logic before you present the pseudocode.

(c) (5 points) For this problem, assume that you are given the lists $A_0, \ldots, A_{k-1}$ in the linked-list representation. Let $[start, end]$ be any minimum span of A. Assume that you run part (a) on the lists $A_0, \ldots, A_{k-1}$. Then prove that there will be a time when the heap will contain both $start$ and end along with $k-2$ other elements that lie in the interval $(start, end)$.

(**Hint**: First prove that if $[start, end]$ were to be a span (let alone a minimum span) and if it were to occur in the heap, then the remaining $k-2$ values need to appear in the interval $(start, end)$. Then, use the correctness of part (a) to argue that $start$ and end need to appear in the heap at the same time.)

(d) (**Extra Credit**)(5 points) Use part (c) to adapt adapt your algorithm from part (a), to find the minimum span. Briefly justify the correctness of your algorithm and argue its running time.

Problem 4-3 (k-ary heap) 11+6 points

Recall that binary heaps are those where each non-leaf node (but with one possible exception) will have 2 children. We can extend the same definition to where each non-leaf node (but with one possible exception) will have k children. This is called a k-ary heap. Assume that we have a *max-heap*. In this question you will be asked to write pseudocode for *efficient* implementations of the standard heap algorithms. Make calls to other algorithms you have defined in the previous parts when you can. You will lose points otherwise. Additionally, ensure that you check boundary cases and throw appropriate error messages.

(a) (2 points) Assume that you have a k-ary heap implemented as a 1-dimensional array similar to what is done for a binary heap with the *root at* $A[1]$. Write the pseudocode for k-Parent(i) to return the index of the parent of node i. Consider the following pseudocode. Further, write a pseudocode k-Child(i, j) to return the index of the j-th child of node i for $j = 1, \ldots, k$.

(b) (3 points) Verify that your algorithm is correct by showing that $\forall j = 1, 2, \ldots, k$,

$$\text{k-Parent}(\text{k-Child}(i, j)) = i$$

(c) (3 point) Write a pseudocode for k-Max-Heapify(A, i, k) that takes as input the array A, an index i into the array and k. Your algorithm will make calls as needed to the algorithms defined above.

(d) (3 points) Write a pseudocode for k-Extract-Max(A).

(e) (**Extra Credit**)(6 points) Derive a Θ bound for the height of the heap in terms of n and k where n is the size of the max-heap. Use this to derive analyze the running time of k-Max-Heapify in terms of k and n. Finally, solve to get the optimal value of k.

(**Hint**: Let $T(h)$ be the running time of k-Max-Heapify on a heap of height h. Recall that a leaf node has height 0.)