Homework 02 — Solutions

CS 624, 2024 Spring

1. Exercise 6.5-8 (page 166) on HEAP-DELETE.

   The operation HeapDelete(A, i) deletes the item in node i from heap A. Give an implementation of HeapDelete that runs in $O(\lg n)$ time for an n-element max-heap.

   Include a brief explanation of the running time.

   HeapExtractMaximum already implements removal; let's re-use that.

   Use HeapIncreaseKey to move the element to delete to the root (either set its key to something greater than the current maximum, or ∞). Then remove it using HeapExtractMaximum.

   The call to HeapMaximum takes $O(1)$ time; the call to HeapIncreaseKey takes $O(\lg n)$ time; and the call to HeapExtractMaximum takes $O(\lg n)$ time. So HeapDelete takes $O(1) + 2O(\lg n) = O(\lg n)$ time.

   More details:

   Algorithm 1 HeapDelete(A, i)

   $k \leftarrow$ HeapMaximum(A)
   HeapIncreaseKey$(A, i, k + 1)$
   HeapExtractMaximum(A)

2. Exercise 6.5-9 (page 166) on merging k sorted lists.

   Give an $O(n \lg k)$-time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. (Hint: Use a min-heap for k-way merging.)

   Include a brief explanation of the running time.

   Use a min-heap to determine which array to take an element from next. Loop until all input arrays are exhausted and the merged array is full.

   The min-heap contains objects ("nodes") with three attributes (fields):

   - key — used for comparison, equal to array(*node*)[next(*node*)]
   - array — a reference to one of the input arrays
   - next — the index of the next available element in the array

   The min-heap is initialized by adding an object for each non-empty input array with the `key` field set to the first element of the array, the `array` field set to the array itself, and the `next` field set to `1`.

   The root (minimum) of the min-heap points to the array with the minimum not-yet-merged element.

   The merged array is filled by repeatedly extracting the input array with the minimum next element from the priority queue, adding that element to the merged array, then re-adding the input array with its priority updated to the next element (unless the input array is empty, in which case it is not re-added).

The heap always contains at most k nodes, where k is the number of input arrays to merge. The initialization loop takes time $O(k \lg k)$ (we could do better, $O(k)$ by using BUILD-HEAP, but it doesn't matter here). The merging loop takes time $n(O(\lg k) + O(\lg k))$, where n is the total number of elements (that is, the sum of the input array lengths). Each iteration of the merging loop performs at most two heap operations, both of which run in $O(\lg k)$ time.

So the total time is $O(k \lg k + n \lg k)$. If none of the input arrays are empty then $k \leq n$, so the time simplifies to $O(n \lg k)$.

More details: Here is pseudocode for the algorithm:

Algorithm 2 MergeK(*arrays*)

```
len ← 0
minheap ← new min-heap with size = length(arrays)
for array in arrays do
  if length(array) > 0 then
    HeapInsert(minheap, new heapnode(key = array[1], array = array, next = 1))
    len ← len + length(array)
  end if
end for
merged ← new array of length len
for i ← 1 to len do
  node ← HeapExtractMinimum(minheap)
  merged[i] ← key(node)
  if next(node) < length(array(node)) then
    next(node) ← next(node) + 1
    key(node) ← array(node)[next(node)]
    HeapInsert(minheap, node)
  end if
end for
return merged
```

3. Exercise 6.1 in the Lecture 3 auxiliary handout on selecting k smallest elements.

Build a min-heap from the array, then call HEAP-EXTRACT-MIN k times to get the k smallest elements.

The loop invariant is the following: at the beginning of the i iteration, $r[1 \,..\, i-1]$ contains the $i-1$ smallest elements of the original array, and A is a heap containing the rest of the original elements. So the call to HeapExtractMin gives us the ith-smallest element, which we add at $r[i]$. When the loop exits, $i = k+1$, so $r[1 \,..\, k]$ contains the k smallest elements of the original array, as we wanted.

The running time is $O(n + k \lg n)$. The $O(n)$ cost is the construction of the heap, and then there are k extractions, each of which costs $O(\lg n)$. Without knowing more about how k compares to n, we can't simplify the bound further.

More details:

Algorithm 3 SelectKMin(A, k)

```
r ← new array of length k
BuildMinHeap(A)
for i ← 1 to k do
  r[i] ← HeapExtractMin(A)
end for
return r
```

4. Problem 7-2 (page 186) on quicksort with equal element values.

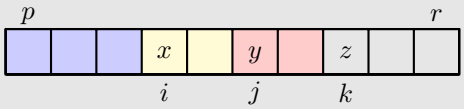
   Do parts (a) and (b) from the textbook.

   (a)

   If all of the elements are equal, then each call to Partition results in an empty $>$ range and a $\leq$ range of size $n-1$, so the recursion tree is as unbalanced as possible.
   This is the worst case, and the running time is $O(n^2)$.

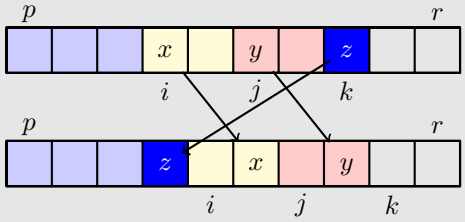
   (b)

   Here is one way to do it. First, read the *pivot*, then consider the entire array (including the pivot) as consisting of four parts: $<$ range, $=$ range, $>$ range, and the unexamined range. The first three ranges are initially empty. The pivot is considered part of the unexamined range. (It is also possible to put the unexamined range between the $<$ and $=$ ranges or between $=$ and $>$ ranges.) I'll use p, i, j, k for the starting index of the first four ranges and $i-1, j-1, k-1, r-1$ for the ending indexes. (This is different from the book's convention.) I'll use x, y, z fo the values at the beginning of the $=$, $>$, and unexamined ranges in the illustrations below.

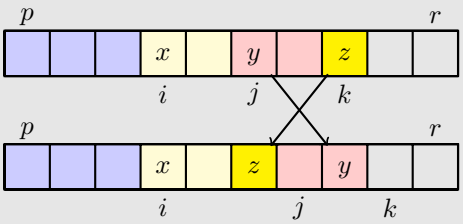


   While the unexamined range is not empty, perform the following steps. (Each step results in index k being incremented by one, so this can be done as a **for** loop on k that also updates i and j when needed.) In each iteration, compare the element at k (labeled z below) against the pivot. There are three cases:

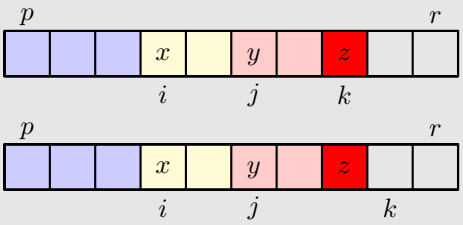
   - If $z <$ *pivot*: Both the $=$ and $>$ ranges need to be rotated right one step, so that z can be added to the end of the $<$ range. All three of i, j, k are incremented.



   - If $z =$ *pivot*: The $>$ range must be rotated right by one step, so that z can be added to the end of the $=$ range. The $<$ range is unchanged. Both j and k are incremented.



- If $z > pivot$: No moves are necessary, just increment k to include z in the $>$ range.



This glosses over some subtleties related to empty ranges, but they are not overly tricky.

Do not do (c) or (d). Instead:

(c′) Show the steps of your Partition′ algorithm on the input array $[1, 6, 5, 8, 5, 4, 5]$ with $p = 1$ and $r = 7$. That is, partition the entire array. (Similar to Figure 7.1 on page 172.)

Here's one way to write it:

The brackets represent the four ranges: $<$, $=$, $>$, and unexamined.

```
[][][][1,6,5,8,5,4,5]
[1][][][6,5,8,5,4,5]
[1][][6][5,8,5,4,5]
[1][5][6][8,5,4,5]
[1][5][6,8][5,4,5]
[1][5,5][8,6][4,5]
[1,4][5,5][6,8][5]
[1,4][5,5,5][8,6][]
```

Here's another way to write it:

$<$	$=$	$>$	unexamined	A
			1,6,5,8,5,4,5	1,6,5,8,5,4,5
1			6,5,8,5,4,5	1 ,6,5,8,5,4,5
1		6	5,8,5,4,5	1 , 6 ,5,8,5,4,5
1	5	6	8,5,4,5	1 , 5 , 6 ,8,5,4,5
1	5	6,8	5,4,5	1 , 5 , 6,8 ,5,4,5
1	5,5	8,6	4,5	1 , 5,5 , 8,6 ,4,5
1,4	5,5	6,8	5	1,4 , 5,5 , 6,8 ,5
1,4	5,5,5	8,6		1,4 , 5,5,5 , 8,6

Important: Note that the $6, 8$ swap orders in each of the last three steps.

(d′) State the loop invariant for your Partition′ algorithm. You are *not* required to write the proof of correctness, but the loop invariant you state must be correct and it must be strong enough to prove the correctness of your algorithm.

The loop invariant for my choice of variables:

- $A[p \,..\, i-1] < pivot$, and

- $A[i \,..\, j-1] = pivot$, and
- $A[j \,..\, k-1] > pivot$, and
- A is a permutation of the original A

Your Partition′ procedure must not be randomized; it should use the final element of the array range as the pivot, like Partition does. It should make a single pass over the array range. I recommend using a **while** loop instead of a **for** loop, but it can be solved either way.

5. Problem 7-4 (page 188) on Tail-Recursive-Quicksort.

Part a: This algorithm is correct because it corresponds to the original Quicksort algorithm. Specifically, entering line 1 with particular values of A, p, and r correspond to calls to Quicksort with A, p, and r as arguments. The modified version calls Partition to get q, it recursively calls Tail-Recursive-Quicksort on the left part, and then it continues the loop after setting p to $q+1$, which corresponds to the original version's second recursive call with A, $q+1$, r.

Part b: If the array is sorted, then at each partitioning the left partition gets all of the elements except the pivot, and we handle the left partition through a recursive call.

Part c: Choose the smaller partition to handle via recursive call. This requires adjusting either p or r before continuing the `while` loop to handle the other partition. The correspondence described in the answer to (a) is still maintained, which guarantees the correctness and same running time. The recursive call is made with at most $n/2$ elements, so the stack depth is $O(\lg n)$, where n is the number of elements in the array segment to be sorted.

More details for (c): Here is the pseudocode:

Algorithm 4 BetterTailRecursiveQuicksort(A, p, r)

```
while p < r do
  q ← Partition(A, p, r)
  if (q − p) < (r − q) then
    // Left partition is smaller
    BetterTailRecursiveQuicksort(A, p, q − 1)
    p ← q + 1
  else
    // Right partition is smaller (or same)
    BetterTailRecursiveQuicksort(A, q + 1, r)
    r ← q − 1
  end if
end while
```

6. Problem 7-6 (page 188) on fuzzy sorting of intervals.

Part (a)

Define FuzzyPartition by modifying the Partition′ algorithm from problem 7-4 (problem 4(b) on this assignment) as follows:

- The *pivot* is an *interval*, not a number. It is initialized from a randomly-selected element of the array range. The pivot is updated during the course of partitioning the array.
- The comparison of each element to the pivot is replaced by the following three cases:
  - The element is added to the = range if its interval *overlaps* with the current pivot. (That is, their intersection is not empty.)
    In this case, the pivot is *updated* to be the intersection of the old pivot and the current element's interval.
  - The element is added to the < range if its interval's upper endpoint is less than the

pivot's lower endpoint. The pivot is not modified.
  - The element is added to the $>$ range if its interval's lower endpoint is less than the pivot's upper endpoint. The pivot is not modified.

Note the following about FuzzyPartition:

- The pivot's lower endpoint never decreases, and its upper endpoint never increases. So if an intermediate pivot places an element in the $<$ or $>$ range, the final pivot value would make the same decision. And if the final pivot places an element in the $=$ range, then every intermediate pivot would make the same decision.
- Consequently, the final pivot interval is strictly greater than every interval in the $<$ range and strictly less than every interval in the $>$ range.
- So pick some point in the final pivot (the lower endpoint, say) and set that as the representative value (the "c_j" value) of every element in the $=$ range. This value is greater than any possible representative value chosen for the $<$ elements and less than any possible representative value chosen for the $>$ elements.

Define FuzzyQuicksort by modifying Quicksort to use FuzzyPartition.

Part (b)

The greater the overlap, the more elements will be included in the $=$ range and excluded from the sub-problems and the cost of the recursive calls.

In particular, if all of the elements overlap, then their intersection is non-empty, and the first call to FuzzyPartition will place all of them in the $=$ range. The recursive calls to the empty $<$ and $>$ ranges will immediately terminate. So the running time is the cost of a single call to FuzzyPartition on the array of length n, so $\Theta(n)$ time.

Alternative approach:

Take the pivot as a number, not an interval. Pick a random element of the array, then pick a point within its interval as the pivot. (Could be lower limit, upper limit, midpoint, or random point in the interval.)

The element comparison process has three cases: the element's interval contains the point ($=$), the element's interval is entirely to the left of the pivot ($<$), or the element's interval is entirely to the right ($>$).

Recur on the $<$ and $>$ ranges normally.

For every element j in the $=$ range, the pivot itself can be considered its c_j value.

Part (b)

Unlike in the first solution, we are not guaranteed to pick a pivot that is actually in the intersection of all of the intervals. But it is "likely enough" (and the possibilities of overlapping intervals are constrained enough), that I think the expected case is still linear. (I haven't proved it, though.)

A partly-WRONG answer:

Take the pivot to be an interval, as in the first answer, but don't update it on each $=$ (ie, overlapping) comparison.

Then the $=$ range is not guaranteed to have a non-empty intersection, so there may be no single point that can act as the "c_j" value of all of its intervals. So the $=$ range must also be sorted, and care must be taken that the sorting is consistent with c_j values that are greater than c_j values chosen for the $<$ range and less than the c_j values chosen for the $>$ range.

Because of the need to sort the $=$ range, the $\Theta(n)$ bound is not achieved.