Homework 02 — Solution

CS 624, 2022 Fall

Review the course homework policies before you start!

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

   Let `HeapDelete` be defined as follows:

   ```
   HeapDelete(A,i) :=

   k = HeapMaximum(A)
   HeapIncreaseKey(A, i, k+1)
   HeapExtractMaximum(A)
   ```

   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.

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

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

   - `key` — used for comparison, equal to `array[next]`
   - `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`.

   Then the merged array is filled by repeatedly selecting the input array with the minimum next element from the priority queue, 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).

   Here is pseudocode for the algorithm:

   ```
   struct heapnode {
     int key
     int[] array
     int next
   }

   MergeK(int[][] arrays) :=

   len ← 0
   minheap ← new MinHeap with max size = arrays.length
   for array in arrays
     if (array.length > 0)
   ```

```
        HeapInsert(minheap, new heapnode(key = array[1], array = array, next = 1))
        len ← len + array.length
      end if
    end for

    merged ← new int[len]
    next ← 1
    while (minheap.size > 0)
      node ← HeapExtractMinimum(minheap)
      merged[next] ← node.key
      next ← next + 1
      if (node.next < node.array.length)
        node.next ← node.next + 1
        node.key ← node.array[node.next]
        HeapInsert(minheap, node)
      end if
    end while
    return merged
```

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 $O(n \lg k)$.

3. Exercise 6.1 in Lecture 3 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:

```
SelectKMin(A, k) :=

results = new array of size k
BuildMinHeap(A)
for i ← 1 to k
  results[i] ← HeapExtractMin(A)
end for
return results
```

The loop invariant is the following: at the beginning of the i iteration, `results[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 `results[i]`. When the loop exits, $i = k+1$, so `results[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.

4. Exercise 7.3-2 (page 180) on the number of calls to Random.

Let n be the size of the array to sort. In the worst-case situation, there are $\Theta(n)$ calls to RANDOM. In the best-case situation, there are also $\Theta(n)$ calls to random. Let $R(n)$ be the number of calls to RANDOM made for an array segment of length n. The base cases are $R(1) = R(0) = 0$. For the worst-case and best-case situations, we have the following recurrences:

$$R(n) = R(n-1) + 1 \qquad \text{(worst-case situation)}$$
$$R(n) = 2R(n/2) + 1 \qquad \text{(best-case situation)}$$

Both recurrences yield $R(n) = \Theta(n)$.

In fact, in *any* case, there are $\Theta(n)$ calls to RANDOM.

There is one call to RANDOM per call to PARTITION, and PARTITION puts the pivot in its final sorted place. So the number of calls is bounded above by the number of potential pivots, n. That gives us $R(n) = O(n)$.

But RANDOMIZED-QUICKSORT doesn't call RANDOMIZED-PARTITION if the array segment is "trivial" (0 or 1 elements), so there will actually be fewer than n calls. But each pivot has at most two trivial immediate neighbors, so at least $n/3$ of the final sorted array elements are pivots. That gives us $R(n) = \Omega(n)$.

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 reverse-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.

```
Better-Tail-Recursive-Quicksort(A,p,r) :=

while p < r
  q ← Partition(A,p,r)
  if ((q-p) < (r-q)
    // Left partition is smaller
    Better-Tail-Recursive-Quicksort(A,p,q-1)
    p ← q+1
  else
    // Right partition is smaller (or same)
    Better-Tail-Recursive-Quicksort(A,q+1,r)
    r ← q-1
  end if
end while
```

6. Assume that $c \geq 0$, and assume you had some kind of super-hardware that, when given two lists of length n that are sorted, merges them into one sorted list, and takes only n^c steps.

(a) Write down a recursive algorithm that uses this hardware to sort lists of length n.

It's the same as MERGE-SORT, except it uses the hypothetical hardware implementation of MERGE.

(b) Write down a recurrence to describe the run time.

$$T(n) = 2T(n/2) + n^c$$

(c) For what values of c does this algorithm perform substantially better than $O(n \log n)$? Why is it highly implausible that this kind of super-hardware could exist for these values of c?

Our previous implementation of MERGE is $\Theta(n)$, where n is the length of each input to merge (roughly). So $c = 1$ is not exciting, and $c > 1$ is worse than what we can already do in software.

If $c < 1$, then the master theorem gives us a running time $T(n) = \Theta(n)$. That would be exciting, but we know that no comparison-based sorting algorithm can do better than $\Omega(n \lg n)$, so we should disbelieve the hardware manufacturer's claims.

7. In a binary tree, a *leaf node* is a node whose left and right children are both NIL. The *depth* of the tree is the maximum number of edges between the root node and any leaf node.

Show that if a binary tree has depth n, then it has at most 2^n leaf nodes.

Restatement of the goal: for all n, for all binary trees t, if t has depth n, then t has at most 2^n leaf nodes.

By strong induction on n.

Case $n = 0$**:** If the depth of t is 0, it must be NIL (0 leaves) or a leaf node (1 leaf). In either case, the number of leaves is less than or equal to $2^0 = 1$.

Case $n = k + 1$**:** The inductive hypothesis is:

for all $j \leq k$, for all binary trees t, if t has depth j, then t has at most 2^j leaf nodes

The goal: for all binary trees t, if t has depth $k + 1$, then t has at most 2^{k+1} leaf nodes.

Let t be an arbitrary binary tree, and assume that t has depth $k + 1$. Since the depth is not 0, the tree is not NIL, and the depth of one of its subtrees is exactly k and the depth of the other is some $j \leq k$. So I can apply the IH to the subtrees to discover that they have 2^k and $2^j \leq 2^k$ leaf nodes, respectively. The total number of leaf nodes is then

$$\text{total leaf nodes} = 2^k + 2^j \leq 2^k + 2^k = 2^{k+1}$$

Done.