Problem Set 4

Data Structures and Algorithms, Fall 2021

Due: October 9 23:59:59 (UTC+8), mail to `dsalg21ps@chaodong.me`.

Problem 1

(a) Given input array $\langle 5, 13, 2, 25, 7, 17, 20, 15, 4\rangle$, show step-by-step how the linear time heap building algorithm discussed in class `BuildMaxHeap` builds a max-heap out of it.

(b) Given the max-heap part (a) constructs, show step-by-step how the `HeapSort` algorithm sorts the input array.

Problem 2

Suppose you are given k sorted lists, of total length n. Devise an algorithm with runtime $O(n \lg k)$ that merges these lists into one single sorted list. You should first briefly describe your algorithm, then provide pseudocode, and finally analyze its runtime. *(Hint: Using a heap might help.)*

Problem 3

The following cruel and unusual sorting algorithm was proposed by Gary Miller. Assume for this problem that the input size n is always a power of 2.

```
Cruel($A[1 \cdots n]$)
1: if ($n > 1$) then
2:     Cruel($A[1 \cdots (n/2)]$).
3:     Cruel($A[(n/2+1) \cdots n]$).
4:     Unusual($A[1 \cdots n]$).
```

```
Unusual($A[1 \cdots n]$)
1: if ($n == 2$) then
2:     if ($A[1] > A[2]$) then
3:         Swap($A[1], A[2]$).
4: else
5:     for ($i = 1$ to $n/4$) do                          ▷ Swap 2nd and 3rd quarters.
6:         Swap($A[i + n/4], A[i + n/2]$).
7:     Unusual($A[1 \cdots (n/2)]$).
8:     Unusual($A[(n/2+1) \cdots n]$).
9:     Unusual($A[(n/4+1) \cdots (3n/4)]$).              ▷ Recurse on middle half.
```

(a) Prove by induction that `Cruel` correctly sorts any input array. *(Hint: Examine the pseudocode of `Cruel`, if it can do sorting correctly, what guarantees `Unusual` should provide? Does `Unusual` indeed enforce those properties?)*

(b) Prove that `Cruel` would *not* correctly sort if we remove the **for** loop from `Unusual`.
(c) Prove that `Cruel` would *not* correctly sort if we swap the last two lines of `Unusual`.
(d) What is the running time of `Unusual` and `Cruel`? Justify your answers.

Problem 4

The `QuickSort` algorithm introduced in class contains two recursive calls to itself. Specifically, after `QuickSort` calls `Partition`, it recursively sorts the left subarray and then it recursively sorts the right subarray. However, the second recursive call in `QuickSort` is not really necessary; we can avoid it by using an iterative control structure. In particular, consider the following version of quicksort:

```
TRQuickSort($A[1 \cdots n], p, r$)
while ($p < r$) do
    $q \leftarrow$ Partition($A, p, r$).
    TRQuickSort($A, p, q-1$).
    $p \leftarrow q+1$.
```

(a) Prove that `TRQuickSort`$(A[1 \cdots n], 1, n)$ correctly sorts the array A.

As we have discussed in class previously, compilers usually execute recursive procedures by using a stack that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. Upon calling a procedure, its information is pushed onto the stack; when it terminates, its information is popped. Since we assume that array parameters are represented by pointers, the information for each procedure call on the stack requires $O(1)$ stack space. The stack depth is the maximum amount of stack space used at any time during a computation.

(b) Describe a scenario in which `TRQuickSort`'s stack depth is $\Theta(n)$ on an n-element input array.

(c) Modify the code for `TRQuickSort` so that the worst-case stack depth is $\Theta(\lg n)$. Your modification should maintain the $O(n \lg n)$ expected running time of the algorithm. You also need to explain why your modification can guarantee $\Theta(\lg n)$ worst-case stack depth.

Problem 5

In this problem, to simplify presentation and analysis, you may assume square root of n and repeated square roots of n are always integers.

(a) Describe an algorithm that sorts an input array $A[1 \cdots n]$ by calling a subroutine `SqrtSort`(k), which sorts the subarray $A[(k+1) \cdots (k+\sqrt{n})]$ in place, given an arbitrary integer k between 0 and $n - \sqrt{n}$ as input. (As mentioned above, to simplify the problem, you may assume that $\sqrt{n}$ is an integer.) Notice, your algorithm is *only* allowed to inspect or modify the input array by calling `SqrtSort`. In particular, your algorithm must not directly compare, move, or copy array elements. You need to give the pseudocode of your algorithm, and prove its correctness. You also need to analyze how many times does your algorithm call `SqrtSort`, in the worst case.

(b) Now suppose `SqrtSort` is implemented recursively, by calling your sorting algorithm from part (a). For example, at the second level of recursion, the algorithm is sorting arrays roughly of size $n^{1/4}$. Analyze what is the worst-case running time of the resulting sorting algorithm? (Again, to simplify the analysis, recall that we have assumed repeated square roots of n are always integers.) *(Hint: The last several paragraphs of Subsection 4.3 of CLRS might help.)*

Problem 6

Suppose you have access to a function FairCoin that returns a single random bit, chosen uniformly and independently from the set $\{0, 1\}$, in $O(1)$ time. (That is, each time you invokes FairCoin, it returns 0 or 1 each with probability $1/2$.) Consider the following randomized algorithm for generating biased random bits.

```
OneInThree()
1: if (FairCoin() == 0) then
2:     return 0.
3: else
4:     return 1 − OneInThree().
```

(a) Prove that OneInThree returns 1 with probability $1/3$.

(b) What is the exact expected number of times that this algorithm calls FairCoin? You need to prove your answer.

(c) Now suppose instead of FairCoin you are given a subroutine BiasedCoin that returns an independent random bit equal to 1 with some *fixed but unknown* probability p, in $O(1)$ time. Describe an algorithm OneInTwo that returns 0 or 1 each with probability $1/2$, using BiasedCoin as its only source of randomness. You must give the pseudocode of your OneInTwo algorithm. *(Hint: In each execution of OneInTwo, it might be necessary that BiasedCoin are invoked multiple times, though not all returned results are necessarily helpful.)*

(d) What is the exact expected number of times that your OneInTwo algorithm calls BiasedCoin? You need to prove your answer.

Problem 7

Eve thinks of an integer between 1 and $1,000,000$. You are trying to determine this number by asking as few yes/no questions as possible. How many yes/no questions are required to determine Eve's number in the worst case? Give both an upper bound (i.e., devise an algorithm) and a lower bound. You also need to briefly explain why your upper bound and lower bound are correct.

Bonus Problem

Show that when all n input elements are distinct, the runtime of HeapSort is *always* $\Theta(n \lg n)$.