COSC 302: Analysis of Algorithms Lecture — Spring 2018
Prof. Darren Strash
Colgate University

Worksheet 5 — Heaps and Non-Comparison Sorting

1. Iterated functions and the tower of twos.

   Problem: Let $2 \uparrow\uparrow i$ denote the *tower of twos*:

$$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{i \text{ times}}.$$

   Write the tower of twos as an iterated function.

   Solution #1: As a first try, let's make a recursive function:

$$P(i) = \begin{cases} 2^{P(i-1)} & i > 1, \\ 2 & i = 1. \end{cases}$$

   However, this is not an iterated function per se. So instead, let's give a function that gives us the result when iterated.

   Solution #2: Define our function to be $f(n) = 2^n$. Then following the framework of iterated functions, we get

$$f^{(i)}(n) = \begin{cases} f^{(i-1)}(f(n)) & i > 0, \\ n & i = 0, \end{cases}$$

   which solves to $\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}}_{i \text{ times}}$. Thus, for $n = 1$, $f^{(i)}(n)$ is the tower of twos.

2. Let $A[1..n]$ store a heap of n distinct elements, and suppose we execute HeapSort(A). Show that if, at the end of each call to *Max-Heapify*$(A, 1)$, A[heap-size(A)] always remains the minimum element in A, then HeapSort(A) requires $\Omega(n \lg n)$ time. Use the following template for guidance.

   (a) One way to prove this result is to focus on a single operation, and show that this operation occurs $\Omega(n \lg n)$ times. For this problem, the number of *swaps* made by the algorithm is a useful operation to bound. Explain why.

   Solution: The number of swaps is equivalent to the number of times Max-Heapify is called. This seems to be the most expensive operation. We call it n times, so if a constant fraction of these calls make $\Omega(\lg n)$ swaps, then we have shown the result!

   (b) Where is the minimum value element in a Max-Heap? What about in the Max-Heap with our extra constraint?

Solution: The minimum element in a MAX-HEAP of distinct elements is a leaf. Otherwise it would have a child with larger value and the max-heap property would be violated. With our extra constraint, it is the last leaf: the rightmost node on the bottommost level.

(c) Call all the nodes on the path from a node v to the root of the heap v's *ancestors.* Describe the how the values of v's ancestors relate compare to each other and to v. How many ancestors does the minimum value have in our constrained MAX-HEAP? How many ancestors does the value have after calling EXTRACT-MAX(A)?

Solution: Let an ancestor with height h be called v_h. Then by the max-heap property $v_h > v_{h-1}$. Our minimum value has $\lfloor \lg n \rfloor$ ancestors, since it is on the bottommost level and $\lfloor \lg n \rfloor$ is the height of a max-heap. After calling EXTRACT-MAX(A), then the minimum value either still has $\lfloor \lg n \rfloor$ ancestors (if the level does not become empty) or else $\lfloor \lg n \rfloor - 1$ if the number of levels decreased by one.

(d) How many calls are made to MAX-HEAPIFY during a call to EXTRACT-MAX(A) for the general MAX-HEAP? What about for our constrained version? What is the minimum number of calls? The maximum number? *The minimum number is what we need to compute a lower bound for this case.* How many swaps are done?

Solution: MAX-HEAPIFY is called $O(\lg n)$ times during a call to EXTRACT-MAX(A). However, it is possible that it will make only one call, and therefore is only $\Omega(1)$. For our constrained version, the same upper bound $O(\lg n)$ holds. However, the element must be moved to the bottommost level, which is at least $\lfloor \lg n \rfloor - 1 = \Omega(\lg n)$ calls to MAX-HEAPIFY. The number of swaps equals the number of calls to MAX-HEAPIFY.

(e) Now compute the total number of swaps performed by HEAPSORT(A) and use it to show that HEAPSORT(A) takes $\Omega(n \lg n)$ time in this case.

Solution: During the course of the HEAPSORT algorithm, the minimum element will be in index n, $n-1$, …, $\lfloor n/2 \rfloor$ (the parent of n) of A, among others. For each one of these indices, EXTRACT-MAX calls MAX-HEAPIFY at least $\lfloor \lg n \rfloor - 2$ times. (When swapping $A[\lfloor n/2 \rfloor]$ and $A[1]$, removing $A[n/2]$ could empty level $\lfloor \lg n \rfloor - 1$. And then $\lfloor \lg n \rfloor - 2$ swaps would be performed.) Thus, at least

$$(n - \lfloor n/2 \rfloor + 1)(\lfloor \lg n \rfloor - 2) = \Omega(n \lg n)$$

swaps are performed. Since the algorithm performs this many swap operations, the entire algorithm uses $\Omega(n \lg n)$ operations.

3. Suppose there is an operation called SqrtSort(k), which sorts the subarray $A[k+1..k+\sqrt{n}]$ in place, given an arbitrary integer k between 0 and $n-\sqrt{n}$ as input. (To simplify the problem, assume that $\sqrt{n}$ is an integer.)[1]

   (a) Describe an algorithm that sorts an input array $A[1..n]$. 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. How many times does your algorithm call SqrtSort in the worst case?

   (b) Prove that your algorithm is correct; that is, it sorts the input $A[1..n]$.

   Inefficient Solution: If iteratively we call SqrtSort(k) for $k = 0, 1, 2, \ldots, n-\sqrt{n}$, we move at least one element into correct position at the end. We can see this by the following invariant.

   Invariant: Before calling SqrtSort(k), $A[k+\sqrt{n}-1]$ contains the maximum element in $A[1..k+\sqrt{n}-1]$. To briefly justify: when we call SqrtSort(k), it moves the maximum element in $A[k+1..k+\sqrt{n}]$ to $A[k+\sqrt{n}]$, which since, $k+1\sqrt{n}-1 \in \{k, \ldots, k+\sqrt{n}\}$is the maximum of $A[k+1\sqrt{n}-1]$ (max in $A[1..k+\sqrt{n}-1]$), and $A[k+1..k+\sqrt{n}]$, which is the maximum in $A[1..k+\sqrt{n}]$.

   We can therefore run SqrtSort this way n times, which moves all elements into correct position.

Algorithm 1 A simple sorting algorithm with SqrtSort.

```
proc Simple-SqrtSort(A[1..n])
  for i ← 1 to n
      for k ← 0 to n − √n
          SqrtSort(k)
```

This takes $n(n-\sqrt{n}+1)$ calls to SqrtSort in the worst case.

More Efficient Solution: We iteratively call SqrtSort(k) for $k = 0, \sqrt{n}/2, \sqrt{n}, \ldots, n-\sqrt{n}$. This moves $\sqrt{n}/2$ elements into correct position.

Invariant: Before calling SqrtSort(k), $A[k+1..k+\sqrt{n}/2]$ contains the $\sqrt{n}/2$ largest elements in $A[1..k+\sqrt{n}/2]$. To briefly justify, when we call SqrtSort(k), it moves the largest $\sqrt{n}/2$ elements in $A[k+1..k+\sqrt{n}]$ to $A[k+\sqrt{n}/2+1..k+\sqrt{n}]$, which includes the largest $\sqrt{n}/2$ elements from $A[1..k+\sqrt{n}/2]$.

See next page for analysis →

[1] Parts of this problem are taken from Jeff Erickson's *Algorithms and Models of Computation* (`http://www.cs.illinois.edu/~jeffe/teaching/algorithms`); Chapter 1: Recursion.

Since the ranges overlap by $\sqrt{n}/2$ elements, we call SqrtSort at most $\frac{n}{\sqrt{n}/2} - 1 = 2\sqrt{n} - 1$ times. We then repeat this for $k = 0, \sqrt{n}/2, \sqrt{n}, \ldots, n - \sqrt{n} - \sqrt{n}/2$ and continue this pattern, as illustrated with the following pseudocode.

Algorithm 2 A more efficient algorithm with SqrtSort.

proc Efficient-SqrtSort($A[1..n]$)

for $i \leftarrow 0$ **to** $2\sqrt{n} - 1$
  for $k \leftarrow 0$ **to** $2\sqrt{n} - (i + 1)$
    SqrtSort($\sqrt{n}/2k$)

In this algorithm, SqrtSort is called

$$\sum_{i=0}^{2\sqrt{n}-1} \sum_{j=0}^{2\sqrt{n}-i-1} 1 = \sum_{i=0}^{2\sqrt{n}-1} (2\sqrt{n} - i) = \Theta(n)$$

times. Or, more precisely:

$$\begin{aligned}
\sum_{i=0}^{2\sqrt{n}-1} \sum_{j=0}^{2\sqrt{n}-i-1} 1 &= \sum_{i=0}^{2\sqrt{n}-1} (2\sqrt{n} - i) \\
&= \sum_{i=0}^{2\sqrt{n}-1} 2\sqrt{n} - \sum_{i=0}^{2\sqrt{n}-1} i \\
&= 2\sqrt{n}(2\sqrt{n}) - \frac{2\sqrt{n}(2\sqrt{n} - 1)}{2} \\
&= 2\sqrt{n}(2\sqrt{n}) - \sqrt{n}(2\sqrt{n} - 1) \\
&= 4n - 2n + \sqrt{n} \\
&= 2n + \sqrt{n}
\end{aligned}$$

times.

4. Can the value 2^n be represented in the RAM model? Describe your answer.
   (to be discussed in a future recitation)

5. Suppose you are given an array $A[1..n]$ of integers between 0 and $n^2 - 1$. Describe how to sort A in $\Theta(n)$ time.
   (to be discussed in a future recitation)

6. It is generally not reasonable to expect to sort non-integers with Radix-Sort. Explain why.
   (to be discussed in a future recitation)

7. Of the comparison sorts that we've learned, which ones are stable? Choose a non-stable comparison sort. Propose a change to make it stable without changing its asymptotic running time.
   (to be discussed in a future recitation)

8. Suppose we don't use a stable sort to sort by keys in Radix-Sort. Give an example input with 2 values, where Radix-Sort fails to sort the input.
(to be discussed in a future recitation)

9. In Radix-Sort, suppose we no longer sort values from the least- to most-significant keys, but now sort from most- to least-significant. Give an example with 2 values where Radix-Sort fails to sort the input.
(to be discussed in a future recitation)

10. Describe how to modify Counting-Sort to actually sort items with keys not equal to their values.
(to be discussed in a future recitation)

11. Describe how to modify BUCKET-SORT to have worst-case $\Theta(n \lg n)$ time.
(to be discussed in a future recitation)

12. Let $a, b \in \mathbb{R}$. Describe how to modify BUCKET-SORT to sort elements drawn from $[a, b)$ uniformly at random.
(to be discussed in a future recitation)