CS 624 - HW 3

Max Weiss

17 February 2020

1. Suppose that S is a comparison sort such that for all n, there are at least $n!/2^n$ input arrays, of length n, in the sorting of which S makes at most cn comparison calls. Any sort which is correct on $n!/2^n$ inputs must be able to execute at least $n!/2^n$ reorderings, so in its decision tree, at least $n!/2^n$ leaves correspond to those inputs. However, we assumed that S makes at most cn comparison calls on those inputs. Since nodes on a branch correspond to comparison calls in a sort process, the lengths of those branches cannot exceed cn. In other words, it follows from our hypothesis that (where "depth" of a node is its distance from the root)

(*) for all n, there is a binary tree with at least $n!/2^n$ leaves of depth $\leq cn$.

Toward a refutation of (*), first suppose there is a tree satisfying this description. By trimming away any descendant nodes of depth $> cn$, we obtain a tree with depth $\leq cn$ and with at least $n!/2^n$ leaves. Thus, (*) implies the following:

(**) for all n, some tree of height $\leq cn$ has at least $n!/2^n$ leaves.

We know that any tree with at least p leaves must have depth at least $\lg p$ (since any tree with depth at least p has at most 2^p leaves). So, the hypothesis implies that there is a c such that

$$cn \geq \lg\left(\frac{n!}{2^n}\right) \tag{1}$$

for all n. By a simple version of Stirling's formula,

$$\begin{aligned} \lg\left(\frac{n!}{2^n}\right) &= \lg n! - \lg 2^n \\ &= \lg n! - n \\ &\geq c' n \lg n - n \end{aligned} \tag{2}$$

for some constant c'. It follows from (1) and (2) that

$$\frac{c+1}{c'} \geq \lg n \tag{3}$$

for all n, and this contradicts the fact that lg is unbounded.

2 (8.5, p207). Note: Since k-sortedness is vacuous for $k \leq 0$, we can assume that $k > 0$. Also, division by a positive integer preserves inequalities. So, let's write

$$\phi(i,k) = \sum_{j=i}^{i+k-1} A[j]$$

whenever $i \in [1, \ldots, n-k]$. Then the definition of k-sortedness comes to this: that

$$\phi(i,k) \leq \phi(i+1,k) \tag{4}$$

for all $i \in [1, \ldots, n-k]$.

2a (8.5a, p207). Note that $\phi(i,1) = A[i]$. So, it follows from (4) that an algorithm is 1-sorted iff $A[i] \leq A[i+1]$ for all $i \in [1, ..., n-1]$. Hence, 1-sortedness is equivalent to sortedness.

2b (8.5b, p207). Let $A = [2, 1, 4, 3, 6, 5, 8, 7, 10, 9]$. Then

$$A[k] = \begin{cases} k+1 \text{ if } k \text{ is odd} \\ k-1 \text{ otherwise} \end{cases}$$

Since A is not sorted, by part (a) it cannot be 1-sorted. However,

$$\begin{aligned} \phi(i,2) &= A[i] + A[i+1] \\ &= 2i+1 \end{aligned}$$

whenever $1 \leq i \leq n$. Therefore, $\phi(i,2) \leq \phi(i+1,2)$ whenever $1 \leq i \leq n-2$, so (4) holds for $k=2$. So A is 2-sorted.

2c (8-5c, p207). Note that

$$\phi(i,k) \ = \ A[i] + \phi(i+1,\ k-1)$$

and

$$\phi(i+1,k) \ = \ \phi(i+1,\ k-1) + A[i+k].$$

Therefore,

$$\begin{aligned} &\phi(i,k) \leq \phi(i+1,k) \\ &\leftrightarrow \phi(i+1,k) - \phi(i,k) \geq 0 \\ &\leftrightarrow A[i+k] - A[i] \geq 0 \\ &\leftrightarrow A[i] \leq A[i+k] \end{aligned}$$

as desired.

2d (8-5d, p207). By 2c, it is sufficient to arrange that $A[i] \le A[i+k]$ whenever $i + k \le \text{len}(A)$. Consider, for example, the algorithm KSort:

```
for i in (1 .. k):
  stepped_sort(A, i, k)
```

where SteppedSort in turn is a procedure which sorts, in-place, the stepped subarray $[A[i], A[i+k], A[i+2k], \ldots]$. It is clear that SteppedSort can be implemented in time $O(\frac{n}{k} \lg \frac{n}{k})$, by adapting an ordinary $O(n \lg n)$ sorting algorithm.

Alternatively, the implementation might simply copy the stepped subarray into an ordinary array, sort that as usual, and then copy it back into the original stepped subarray:

```
  def stepped_sort(A, start, step):
    B = []
    for i in (1 .. len(A):
      if i % step == start:
        B.push(A[i])
    sort(B)
    for i in (1 .. len(B)):
      A[start + i * step] = B[i]
```

This version copies the $\text{len}(A)/\text{step} = \frac{n}{k}$ entries, sorts the resulting array, and copies the entries back: so, its running time is $O(\frac{n}{k} \lg \frac{n}{k})$.

Within KSort, SteppedSort is called k times. Hence, the running time of KSort is $O(n \lg \frac{n}{k})$.

3a. *A priori*, a sorting algorithm which might maximally exploit an accelerated merge procedure is this:

```
def super_merge_sort(A, lo, hi):
  if lo + 1 < hi:
    mid = lo + floor ((hi - lo) / 2)
    super_merge_sort(A, lo, mid)
    super_merge_sort(A, mid, hi)
    super_merge(A, lo, mid+1, hi)
```

3b. A recurrence to describe the runtime of SuperMergeSort is

$$T(n) = 2T(n/2) + n^c + 1 \tag{5}$$

3c. I will use the master theorem to evaluate (5). Let $a = 2$, $b = 2$, and $f(n) = n^c + 1$. Then, $\log_b a = 1$, so that $f(n) = n^{\log_b a - (1-c)} + 1$. Therefore, we are in case (a), (b), or (c) of the master theorem according to whether we have $c < 1$, $c = 1$, or $c > 1$ in the running time n^c of the merge subroutine. Of course, that subroutine can be implemented without magic in $O(n)$ time. So, only the case $c < 1$ could yield some asymptotic improvement over standard mergesort. And in that case, part (a) of the master theorem tells us that $T(n) \in \Theta(n^{\log_b a}) = \Theta(n)$. Since SuperMergeSort does not assume anything about the input array, this could very well be an improvement over existing sort algorithms!

Unfortunately, the existence of a sub-linear merge procedure is implausible. Unless it assumes something about the input lists, a merge must at least look at all of the list entries, requiring at least $O(n)$ read operations.

4. I will argue by induction on n. For the basis step, if $n = 4$, then $2^n = 16 \leq 24 = n!$. Now, suppose that $2^n \leq n!$ with $n \geq 4$. Then

$$\begin{aligned} 2^{n+1} &= 2 * 2^n \\ &\leq 2n! \\ &\leq (n+1)n! \\ &= (n+1)! \end{aligned}$$

as desired.

5. I will argue by induction on n that $\lg b^n = n \lg b$. For $n = 1$, we have $\lg b^n = \lg b = n \lg b$. Now, suppose that $\lg b^n = b \lg n$. Using $\lg cd = \lg c + \lg d$, it follows that

$$\begin{aligned} \lg b^{n+1} &= \lg(b^n b) \\ &= \lg b + \lg b^n \\ &= \lg b + n \lg b \\ &= (n+1) \lg b \end{aligned}$$

as desired.

6. Let me first prove the suggested lemma

$$n = \left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil. \tag{6}$$

Since n must be an integer, it has one of the forms $2m$ or $2m+1$ for some integer m. I will handle these cases separately. First suppose that $n = 2m$. Then (6) follows from

$$\left\lfloor \frac{n}{2} \right\rfloor = \lfloor m \rfloor = m = \lceil m \rceil = \left\lceil \frac{n}{2} \right\rceil$$

On the other hand, if $n = 2m+1$, then (6) follows from

$$\begin{aligned} \left\lfloor \frac{n}{2} \right\rfloor &= \left\lfloor m + \frac{1}{2} \right\rfloor = m + \left\lfloor \frac{1}{2} \right\rfloor = m \\ \left\lceil \frac{n}{2} \right\rceil &= \left\lceil m + \frac{1}{2} \right\rceil = m + \left\lceil \frac{1}{2} \right\rceil = m + 1. \end{aligned}$$

Now, I will argue (elaborating on the lecture notes) for the bound

$$\lg n! \geq \frac{1}{2} n \lg n - \frac{n}{2} - \lg n + 1 \tag{7}$$

First of all, note that if $j > \lceil \frac{n}{2} \rceil$, then $j \geq \frac{n}{2}$, and also that $k! \geq 1$ for all $k \geq 0$. Consequently,

$$\begin{aligned} n! &= \prod_{j=1}^{n} j \\ &\geq \left(\left\lceil \frac{n}{2} \right\rceil - 1 \right)! \prod_{j=\lceil \frac{n}{2} \rceil}^{n} \frac{n}{2} \\ &\geq \prod_{j=\lceil \frac{n}{2} \rceil}^{n} \frac{n}{2} \\ &= \left(\frac{n}{2} \right)^{n - \lceil \frac{n}{2} \rceil} \end{aligned} \tag{8}$$

Lemma (6) now gives

$$\left(\frac{n}{2} \right)^{n - \lceil \frac{n}{2} \rceil} = \left(\frac{n}{2} \right)^{\lfloor \frac{n}{2} \rfloor} \tag{9}$$

while $\lfloor x \rfloor \geq x - 1$ implies

$$\begin{aligned} \left(\frac{n}{2} \right)^{\lfloor \frac{n}{2} \rfloor} &\geq \left(\frac{n}{2} \right)^{\frac{n}{2} - 1} \\ &= \frac{2}{n} \left(\frac{n}{2} \right)^{\frac{n}{2}}. \end{aligned} \tag{10}$$

Finally, note that

$$\begin{aligned} \lg \left(\frac{2}{n} \left(\frac{n}{2} \right)^{\frac{n}{2}} \right) &= \lg \left(\frac{n}{2} \right)^{\frac{n}{2}} + \lg \frac{2}{n} \\ &= \frac{n}{2} (\lg n - 1) - \lg n + 1 \\ &= \frac{1}{2} n \lg n - \frac{n}{2} - \lg n + 1. \end{aligned} \tag{11}$$

The desired result (7) follows from (8) - (11).

7. In a previous assigment, it was to be proved that a tree with depth d has at most $2^d - 1$ leaves. That result implies that if a tree has at least 2^d leaves, then it has depth $> d$. So, a tree with $L = 2^{\lg L}$ leaves must have depth $> \lg L$. Because $\lg L$ is a lower bound on the depth of trees with L leaves, that depth is $\Omega(\lg L)$.

8a (9-1, p224). Consider the sort-based algorithm

```
def top_few_a(A, k):
  quick_sort(A)
  return A[n-k+1 .. ]
```

QuickSort is $O(n \log n)$; this wipes out the constant required for the slicing operation; so the sort-based algorithm is $O(n \log n)$.

8b (9-1, p224). This is similar to exercise 4 of Homework 3. With minor adjustments, the algorithm I gave there becomes

```
def top_few_b(A, k)
  build_max_heap(A)
  R = []
  for _ in 1 .. k:
    R.push(maxheap_extract_max(S))
  return R
```

The runtime analysis is the same as in Homework 3. Build-MaxHeap costs $O(n)$, while each of the k calls to MaxHeap-Extract-Max (and push of the result onto a list) costs $\lg n$. So, the overall running time is $O(n + k \lg n)$.

8c (9-1, p224). The exercise more or less already gives the algorithm. Here is a bit more detail:

```
def top_few_c(A, k):
 lo = len(A) - k + 1
 pivot = get_order_statistic(A, lo)
 partition_around(A, pivot)
 B = A[lo .. ]
 sort(B)
 return B
```

Ordinary Partition will not quite work here: if the array contains entries which are "equal" to the pivot, then the pivot index may end up fewer than $i - 1$ places from the end of A, the difference being occupied by possibly smaller entries. So instead, I use PartitionAround, which divides the array into the entries less than, equal to, or greater than the pivot:

```
  def partition_around(A, pivot)
    last_smaller, last_equal, frontier = 0, 0, 1
    while frontier <= len(A):
      if A[frontier] <= pivot:
        last_equal += 1
        swap A[frontier], A[last_equal]
        if A[last_equal] < pivot:
          last_smaller += 1
          swap A[last_equal], A[last_smaller]
```

The main algorithm splits into several separate tasks of greater than constant time.

- call some implementation of GetOrderStatistic, which we can assume to be $O(n)$
- call PartitionAround, which is $O(n)$
- copy k entries into B, which is $O(k)$
- sort B, which is $k \lg k$

The running time of the whole thing, then, is $O(n + \lg k)$. If k is constant, this amounts to $O(n)$. If k is proportional to n, then it amounts to $O(n \lg n)$.