Chapter 8

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 8.1-1

We can construct the graph whose vertex set is the indices, and we place an edge between any two indices that are compared on the shortest path. We need this graph to be connected, because otherwise we could run the algorithm twice, once with everything in one component less than the other component, and a second time with the everything in the second component larger. As long as we maintain the same relative ordering of the elements in each component, the algorithm will take exactly the same path, and so produce the same result. This means that there will be no difference in the output, even though there should be. For a graph on n vertices, it is a well known that at least $n-1$ edges are necessary for it to be connected, as the addition of an edge can reduce the number of connected components by at least one, and the graph with no edges has n connected components.

So, it will have depth at least $n-1$.

Exercise 8.1-2

Since $\lg(k)$ is monotonically increasing, we use formula $A.11$ to approximate the sum:

$$\int_0^n \lg(x)dx \leq \sum_{k=1}^{n} \lg(k) \leq \int_1^{n+1} \lg(x)dx.$$

From this we obtain the inequality

$$\frac{n\ln(n)-n}{\ln 2} \leq \sum_{k=1}^{n} \lg(k) \leq \frac{(n+1)\ln(n+1)-n}{\ln 2}$$

which is $\Theta(n \lg n)$.

Exercise 8.1-3

Suppose to a contradiction that there is a c_1 so that for every $n \geq k$, at least half of the inputs of length n have depth at most $c_1 n$. However, there are less than $2^{c_1 n+1}$ elements in the tree of depth at most $c_1 n$. However, $1/2n! > 1/2(n/e)^n > 2^{c_1 n+1}$ so long as $n > e2^{c_1}$. This is a contradiction.

To have a $1/n$ fraction of them with small depth, similarly, we get a contradiction because $1/nn! > 2^{c_1 n+1}$ for large enough n.

To make an algorithm that is linear for a $1/2^n$ fraction of inputs, we yet again get a contradiction because $2^{-n}n! > (n/2e)^n > 2^{c_1 n+1}$ for large enough n.

The moral of the story is that $n!$ grows very quickly.

Exercise 8.1-4

We assume as in the section that we need to construct a binary decision tree to represent comparisons. Since each subsequence is of length k, there are $k!^{n/k}$ possible output permutations. To compute the height h of the decision tree we must have $k!^{n/k} \leq 2^h$. Taking logs on both sides and using exercise 2 this gives

$$h \geq (n/k)\lg(k!) \geq (n/k)\left(\frac{k\ln k - k}{\ln 2}\right) = \frac{n\ln(k) - n}{\ln 2} = \Omega(n \lg k).$$

Exercise 8.2-1

We have that $C = \langle 2, 4, 6, 8, 9, 9, 11\rangle$. Then, after successive iterations of the loop on lines 10-12, we have $B = \langle \, , \, , \, , \, , 2, \, , \, , \, , \, \rangle$,$B = \langle \, , \, , \, , \, , 2, \, , 3, \, , \, \rangle$,$B = \langle \, , \, , 1, \, , 2, \, , 3, \, , \, , \, \rangle$, and at the end, $B = \langle 0, 0, 1, 1, 2, 2, 3, 3, 4, 6, 6\rangle$

Exercise 8.2-2

Suppose positions i and j with $i < j$ both contain some element k. We consider lines 10 through 12 of COUNTING-SORT, where we construct the output array. Since $j > i$, the loop will examine $A[j]$ before examining $A[i]$. When it does so, the algorithm correctly places $A[j]$ in position $m = C[k]$ of B. Since $C[k]$ is decremented in line 12, and is never again incremented, we are guaranteed that when the for loop examines $A[i]$ we will have $C[k] < m$. Therefore $A[i]$ will be placed in an earlier position of the output array, proving stability.

Exercise 8.2-3

The algorithm still works correctly. The order that elements are taken out of C and put into B doesn't affect the placement of elements with the same key. It will still fill the interval $(C[k-1], C[k]]$ with elements of key k. The question of whether it is stable or not is not well phrased. In order for stability to make sense, we would need to be sorting items which have information other than their key, and the sort as written is just for integers, which don't. We could think of extending this algorithm by placing the elements of A into a collection of elements for each cell in array C. Then, if we use a FIFO collection, the modification of line 10 will make it stable, if we use LILO, it will be anti-stable.

Exercise 8.2-4

The algorithm will begin by preprocessing exactly as COUNTING-SORT does in lines 1 through 9, so that $C[i]$ contains the number of elements less than or equal to i in the array. When queried about how many integers fall into a range $[a..b]$, simply compute $C[b] - C[a-1]$. This takes $O(1)$ times and yields the desired output.

Exercise 8.3-1

Starting with the unsorted words on the left, and stable sorting by progressively more important positions.

COW	SEA	TAB	BAR
DOG	TEA	BAR	BIG
SEA	MOB	EAR	BOX
RUG	TAB	TAR	COW
ROW	RUG	SEA	DIG
MOB	DOG	TEA	DOG
BOX	DIG	DIG	EAR
TAB	BIG	BIG	FOX
BAR	BAR	MOB	MOB
EAR	EAR	DOG	NOW
TAR	TAR	COW	ROW
DIG	COW	ROW	RUG
BIG	ROW	NOW	SEA
TEA	NOW	BOX	TAB
NOW	BOX	FOX	TAR
FOX	FOX	RUG	TEA

Exercise 8.3-2

Insertion sort and merge sort are stable. Heapsort and quicksort are not. To make any sorting algorithm stable we can preprocess, replacing each element of an array with an ordered pair. The first entry will be the value of the element, and the second value will be the index of the element. For example, the array $[2, 1, 1, 3, 4, 4, 4]$ would become $[(2, 1), (1, 2), (1, 3), (3, 4), (4, 5), (4, 6), (4, 7)]$. We now interpret $(i, j) < (k, m)$ if $i < k$ or $i = k$ and $j < m$. Under this definition of less-than, the algorithm is guaranteed to be stable because each of our new elements is distinct and the index comparison ensures that if a repeat element appeared later in the original array, it must appear later in the sorted array. This doubles the space requirement, but the running time will be asymptotically unchanged.

Exercise 8.3-3

After sorting on digit i, we will show that if we restrict to just the last i digits, the list is in sorted order. This is trivial for $i = 1$, because it is just claiming that the digits we just sorted were in sorted order. Now, suppose it's

true for $i-1$, we show it for i. Suppose there are two elements, who, when restricted to the i last digits, are not in sorted order after the i'th step. Then, we must have that they have the same i'th digit because otherwise the sort of digit i would put them in the right order. Since they have the same first digit, their relative order is determined by their restrictions to their last $i-1$ digits. However, these were placed in the correct order by the $i-1$'st step. Since the sort on the i'th digit was stable, their relative order is unchanged from the previous step. This means that they are in the correct order still. We use stability to show that being in the correct order prior to doing the sort is preserved.

Exercise 8.3-4

First run through the list of integers and convert each one to base n, then radix sort them. Each number will have at most $\log_n(n^3) = 3$ digits so there will only need to be 3 passes. For each pass, there are n possible values which can be taken on, so we can use counting sort to sort each digit in $O(n)$ time.

Exercise 8.3-5

Since a pass consists of one iteration of the loop on line $1-2$, only d passes are needed. Since each of the digits can be one of ten decimal numbers, the most number of piles that would be needed to be kept track of is 10.

Exercise 8.4-1

The sublists formed are $\langle .13, .16\rangle$, $\langle .20\rangle$,$\langle .39\rangle$, $\langle .42\rangle$, $\langle .53\rangle$, $\langle .64\rangle$, $\langle .71, .79\rangle$, $\langle .89\rangle$. Putting them together, we get $\langle .13, .16, .20, .39, .42, .53, .64, .71, .79, .89\rangle$

Exercise 8.4-2

In the worst case, we could have a bucket which contains all n values of the array. Since insertion sort has worst case running time $O(n^2)$, so does Bucket sort. We can avoid this by using merge sort to sort each bucket instead, which has worst case running time $O(n \lg n)$.

Exercise 8.4-3

X is 0 or 2 with probability a quarter each, and 1 with probability 2. Note that $E[X] = 1$, so, $E^2[x] = 1$. Also, X^2 takes 0 or 4 with probability a quarter each, and 1 with probability a half. So, $E[X^2] = 1.5$.

Exercise 8.4-4

Define $r_i = \sqrt{\frac{i}{n}}$ and $c_i = \{(x, y) | r_{i-1} \leq x^2 + y^2 \leq r_i\}$ for $i = 1, 2, \ldots, n$. The c_i regions partition the unit disk into n parts of equal area, which we will

use as the buckets. Since the points are uniformly distributed on the disk and each region has equal area, bucket sort will run in expected time $\Theta(n)$.

Exercise 8.4-5

We have to pick our bounds for our buckets in bucket sort in such a way that there is approximately equal probability that an element drawn from the distribution will be any one of the buckets. To do this, we can perform a binary search to find the Dyadic rationals with denominator at least a constant fraction of n. Finding each of these takes only $\lg(n)$ time. Then, these form the bounds for a bucket sort. There are an expected constant number of elements in any one of these buckets, so just use any sort you want at this point.

Problem 8-1

a. There are $n!$ possible permutations of the input array because the input elements are all distinct. Since each is equally likely, the distribution is uniformly supported on this set. So, each occurs with probability $\frac{1}{n!}$ and corresponds to a different leaf because the program needs to be able to distinguish between them.

b. The depths of particular elements of LT(resp. RT) are all one less than their depths when considered elements of T. In particular, this is true for the leaves of the two subtrees. Also, $\{LT, RT\}$ form a partition of all the leaves of T. So, if we let $L(T)$ denote the leaves of T,

$$D(T) = \sum_{\ell \in L(T)} D_T(\ell) = \sum_{\ell \in L(LT)} D_T(\ell) + \sum_{\ell \in L(RT)} D_T(\ell) =$$

$$\sum_{\ell \in L(LT)} (D_{LT}(\ell) + 1) + \sum_{\ell \in L(RT)} (D_{RT}(\ell) + 1) =$$

$$\sum_{\ell \in L(LT)} D_{LT}(\ell) + \sum_{\ell \in L(RT)} D_{RT}(\ell) + k = D(LT) + D(RT) + k$$

c. Suppose we have a T with k leaves so that $D(T) = d(k)$. Let i_0 be the number of leaves in LT. Then, $d(k) = D(T) = D(LT) + D(RT) + k$ but, we can pick LT and RT to minimize the external path length

d. We treat i as a continuous variable, and take a derivative to find critical points. The given expression has the following as a derivative with respect to i

$$\frac{1}{\ln(2)} + \lg(i) + \frac{1}{\ln(2)} - \lg(k-i) = \frac{2}{\ln(2)} + \lg\left(\frac{i}{k-i}\right)$$

which is zero when we have $\frac{i}{k-i} = 2^{-\frac{2}{\ln(2)}} = 2^{-\lg(e^2)} = e^{-2}$. So, $(1+e^{-2})i = k$, so, $i = \frac{k}{1+e^{-2}}$.

Since we are picking the two subtrees to be roughly equal size, the total depth will be order $\lg(k)$, with each level contributing k, so the total external path length is at least $k \lg(k)$.

e. Since before we that a tree with k leaves needs to have external length $k \lg(k)$, and that a sorting tree needs at least $n!$ trees, a sorting tree must have external tree length at least $n! \lg(n!)$. Since the average case run time is the depth of a leaf weighted by the probability of that leaf being the one that occurs, we have that the run time is at least $\frac{n! \lg(n!)}{n!} = \lg(n!) \in \Omega(n \lg(n))$

f. Since the expected runtime is the average over all possible results from the random bits, if every possible fixing of the randomness resulted in a higher runtime, the average would have to be higher as well.

Problem 8-2

Algorithm 1 O(n) and Stable(A)

a.
```
Create a new array C
index = 1
for i = 1 to n do
    if A[i] == 0 then
        C[index] = A[i]
        index = index + 1
    end if
end for
for i = 1 to n do
    if A[i] == 1 then
        C[index] = A[i]
        index = index + 1
    end if
end for
```

This algorithm first selects all the elements with key 0 and puts them in the new array C in the order in which they appeared in A, then selects all the elements with key 1 and puts them in C in the order in which they appeared in A. Thus, it is stable. Since it only makes two passes through the array, it is $O(n)$.

The algorithm maintains the 0's seen so far at the start of the array. Each time an element with key 0 is encountered, the algorithm swaps it with the position following the last 0 already seen. This is in-place and $O(n)$, but not stable.

c. Simply run BubbleSort on the array.

Algorithm 2 O(n) and in place(A)

b.
```
index = 1
for i = 1 to n do
    if A[i] == 0 then
        Swap A[i] with A[index]
        index = index + 1
    end if
end for
```

d. Use the algorithm given in part a. For each of the b-bit keys it takes $O(n)$ and is stable, as is required by Radix-Sort. Thus, the total running time will be $O(bn)$.

e. Create the array C as done in lines 1 through 5 of counting sort. Create an array B which is a copy of C, then run lines 7 through 9 on B. In other words, $C[i]$ gives the number of elements in A which are equal to i, and $B[i]$ gives the number of elements in A which are less than or equal to i, so given an element, we can correctly identify where it belongs in the array. While the element in position i doesn't belong there, swap it with the element in the place it belongs. Once an element which belongs in position i appears, increment i. The preprocessing takes $O(n + k)$ time, the total number of swaps is at most n, and we iterate through the whole array once, so the overall runtime is $O(n + k)$. This algorithm has the advantage of sorting in place, however it is no longer stable like counting sort.

Problem 8-3

a. First, sort the integer by their lengths. This can be done efficiently using a bucket sort, where we make a bucket for each possible number of digits. We sort each these uniform length sets of integers using radix sort. Then, we just concatenate the sorted lists obtained from each bucket.

b. Make a bucket for every letter in the alphabet, each containing the words that start with that letter. Then, forget about the first letter of each of the words in the bucket, concatenate the empty word (if it's in this new set of words) with the result of recursing on these words of length one less. Since each word is processed a number of times equal to it's length, the runtime will be linear in the total number of letters.

Problem 8-4

a. Select a red jug. Compare it to blue jugs until you find one which matches. Set that pair aside, and repeat for the next red jug. This will use at most $\sum_{i=1}^{n-1} i = n(n-1)/2 = O(n^2)$ comparisons.

b. We can imagine first lining up the red jugs in some order. Then a solution to this problem becomes a permutation of the blue jugs such that the i^{th} blue jug is the same size as the i^{th} red jug. As in section 8.1, we can make a decision tree which represents comparisons made between blue jugs and red jugs. An internal node represents a comparison between a specific pair of red and blue jugs, and a leaf node represents a permutation of the blue jugs based on the results of the comparison. We are interested in when one jug is greater than, less than, or equal in size to another jug, so the tree should have 3 children per nod. Since there must be at least $n!$ leaf nodes, the decision tree must have height at least $log_3(n!)$. Since a solution corresponds to a simple path from root to leaf, an algorithm must make at least $\Omega(n \lg n)$ comparisons to reach any leaf.

c. We use an algorithm analogous to randomized quicksort. Select a blue jug at random. Partition the red jugs into those which are smaller than the blue jug, and those which are larger. At some point in the comparisons, you will find the red jug which is of equal size. Once the red jugs have been divided by size, use the red jug of equal size to partition the blue jugs into those which are smaller and those which are larger. If k red jugs are smaller than the originally chosen jug, we need to solve the original problem on input of size $k-1$ and size $n-k$, which we will do in the same manner. A subproblem of size 1 is trivially solved because if there is only one red jug and one blue jug, they must be the same size. The analysis of expected number of comparisons is exactly the same as that of randomized-quicksort given on pages 181-184. We are running the procedure twice so the expected number of comparisons is doubled, but this is absorbed by the big-O notation. In the worst case, we pick the largest jug each time, which results in $\sum_{i=2}^{n} i + i - 1 = n^2$ comparisons.

Problem 8-5

a. Since each of the averages will be of sets of a single element, so, each element will be $\leq$ than the next element. So, this is the usual definition of sorted

b. $\langle 1, 6, 2, 7, 3, 8, 4, 9, 10 \rangle$

c. Suppose we have $A[i] \leq A[i+k]$ for all appropriate i. Then, every element in the sum $\sum_{j=i}^{i+k-1} A[j]$ has a corresponding element in the sum $\sum_{j=i}^{i+k-1} A[j]$ that it is less than. This means the sums must have the desired inequality.

Now, suppose that we had the array was k-sorted. This means that

$$\sum_{j=i}^{i+k-1} A[j] \leq \sum_{j=i+1}^{i+k}$$

but, if we subtract off all the terms that both sums have in common, we get:

$$A[i] \leq A[i+k]$$

d. We can do quicksort until we are down to subarrays of size at most k. In exercise 7.4-5, this was shown to take time $O(n \lg(n/k)$. Note that this much work is all that is needed because we only need show that elements separated by k positions are in different blocks which is true because they are $\leq k$ positions wide.

e. Consider the lists, which, for every $i \in \{0, \ldots, k-1\}$, are $L_i = \langle A[i], A[i+k], \ldots, A[i + \lfloor n/k \rfloor k] \rangle$. Note that by part c, if A is $k-$ sorted, then we must have each L_i is sorted. Then, by 6.5-9, we can merge them all into a single sorted array in the desired time.

f. Let $t(k, n)$ be the time required to k-sort arrays of length n. Then, we have that, for every k, we can sort an array by first k sorting it and then applying the previous part in time $t(n, k) + n \lg(k)$. If $t(n, k)$ were $o(n \lg(n))$, then we would have a contradiction to the comparison based sorting lower bounds.

Problem 8-6

a. There are $\binom{2n}{n}$ ways to divide $2n$ numbers into two sorted lists, each with n numbers.

b. Any decision tree to merge two sorted lists must have at least $\binom{2n}{n}$ leaf nodes, so it has height at least $\lg\left(\frac{2n!}{n!n!}\right)$. We'll use the inequalities derived in Exercise 8.1-2 to bound this:

$$\begin{aligned}
\lg\left(\frac{2n!}{n!n!}\right) &= \lg((2n)!) - 2\lg(n!) \\
&= \sum_{k=1}^{2n} \lg(k) - 2\sum_{k=1}^{n} \lg(k) \\
&\geq \frac{2n\ln(2n) - 2n}{\ln 2} - 2\frac{(n+1)\ln(n+1) - n}{\ln 2} \\
&= 2n + 2n\lg(n) - 2n\lg(n+1) - 2\lg(n+1) \\
&= 2n + 2n\lg(n/(n+1)) - 2\lg(n+1) \\
&= 2n - o(n).
\end{aligned}$$

c. If we don't compare them, then there is no way to distinguish between the original unmerged lists and the unmerged lists obtained by swapping those two items between lists. In the case where the two elements are different we require a different action to be taken in order to correctly merge the lists, so the two must be compared.

d. Let list $A = 1, 3, 5, ..., 2n-1$ and $B = 2, 4, 6, \ldots, 2n$. By part c, we must compare 1 with 2, 2 with 3, 3 with 4, and so on up until we compare $2n-1$ with $2n$. This amounts to a total of $2n-1$ comparisons.

Problem 8-7

a. Since we claim that $A[p]$ is the smallest value placed at a wrong location, everything that has value less than or equal to $A[p]$ is in it's correct spot, and so, cannot be placed where $A[p]$ should of been. This means that $A[q] \geq A[p]$. Also, if we had that $A[q] = A[p]$. Then it wouldn't be an error to have $A[q]$ in the spot that should of had $A[q]$. This gets us $A[q] > A[p]$. Since we are only considering arrays with 0-1 values, this means that we must have $0 = A[p]$ and $1 = A[q]$.

b. It fails to sort B correctly because there is some position that $A[p]$ was moved to that was greater than where it should of been, this means that $A[q]$ is to the left of it, which shows the array is unsorted.

c. All of the even steps in the algorithm do not even look at the values of the cells, they just push them around. The odd steps are also oblivious because we can just use the oblivious compare exchange version of insertion sort givin at the beginning of the problem.

d. After step 1, we know that each column looks like some number of zeroes followed by some number of ones. Suppose that column i had z_i zeroes in it. Then, after the reshaping step, we know that each of the columns will contribute $\lceil z_i/(r/s) \rceil$ zeroes to the first $z_i \mod s$ columns, and one less to the rest. In particular, since the number of zeros contributed to each of the columns after step 2 only has a single jump by one, the sum over each must only change by at most s. Then, after sorting, this means that there will only be s dirty rows.

e. From the prior part, we know that before step 4, there are at most s dirty rows, with a total of s^2 elements in them. So, after step 4, these dirty elements will map to some bottom part of a column, some of another column, and then some top part of a column. with everything to the left being zero, and everything to the right being one.

f. From the previous part, we have that there are at most s^2 values that may be bad. Also, $r \geq 2s^2$. Then, there are two cases. The first is that the dirty region spans two columns, in this case, we have that after step 6, the dirty region is entirely in one column. So, after doing step 7, we have that the column that was dirty has a clean top half, and the array is sorted, so, it remains sorted when applying step 8. The second case is that the dirty region is entirely in one column. If this is the case, then, after sorting in step 5, the array is already sorted, and the only dirty column has a clean first half. So, it will maintain it's being sorted after performing steps 6-8.

g. The proof is very similar to the way that part e was shown. There must be some care taken with how the transformation in steps 2 and 4 are changed. We would need that $r \geq 2s^2 - 2s$, since a dirty region of $2s - 1$ rows corresponds to a dirty region (when read column major) of size $2s^2 - s$. we

could simply pad the array with a number of rows with all zero, to bring the number of rows up to a multiple of s. Then, take away that many zeroes from the final answer.