Data Structures – Brett Bernstein

Lecture 15

Review Exercises

1. Given 2 sorted lists, return the number of elements they have in common.

   public static int numShared(**int**[] a, **int**[] b)

2. Given 2 sorted lists, return a new sorted list that contains the elements in both lists.

   public static int[] merge(**int**[] a, **int**[] b)

3. Given an array arr of length n consider all pairs of distinct indices We consider the pair i, j an inversion if $i < j$ and arr[i]>arr[j]. Note there are $\binom{n}{2}$ pairs i, j with $i < j$.

   (a) What is the minimum possible number of inversions? How do you achieve this value?

   (b) What is the maximum possible number of inversions? How do you achieve this value?

   (c) ($\star$) One of the sorts we learned has runtime that is $\Theta(m + n)$ where m is the number of inversions in the array. Which algorithm is it?

   (d) ($\star\star$) Suppose you randomly shuffle an array of n distinct integers. How many inversions do you expect it to have?

4. For each of the following, determine whether the given sort is stable. Recall that a sort is stable if elements that are equal with respect to the ordering are left in their original relative order.

   (a) Selection sort.

   (b) Insertion sort.

   (c) Bubble sort.

5. Show how to force any sort to be stable by changing the data type.

Review Solutions

1. Code follows:

```
public static int numShared(int[] a, int[] b) {
  int i = 0, j = 0, cnt = 0;
  while (i < a.length && j < b.length) {
    if (a[i] == b[j]) {
```

```
      cnt++;
      i++;
      j++;
    }
    else if (a[i] < b[j]) i++;
    else j++;
  }
}
```

The runtime is $\Theta(m+n)$ where m is the length of the first list and n is the length of the second.

2. Code follows:

```
public static int[] merge(int[] a, int[] b) {
  int m = a.length, n = b.length;
  int[] ret = new int[m+n];
  int i = 0, j = 0;
  for (int k = 0; k < m+n; ++k) {
    if (i >= m) ret[k] = a[j++];
    else if (j >= n || a[i] <= b[j]) ret[k]=a[i++];
    else ret[k] = a[j++];
  }
  return ret;
}
```

The runtime is $\Theta(m+n)$ where m is the length of the first list and n is the length of the second.

3. (a) 0 if the array is sorted.
   (b) $\binom{n}{2}$ if the array is reverse sorted.
   (c) Insertion sort.
   (d) Using concepts from probability, each pair is equally likely to be an inversion or not, so the result is $\frac{1}{2}\binom{n}{2}$. More precisely, let X_{ij} be the indicator variable that the pair i, j is an inversion. Then apply linearity of expectation.

4. (a) Not stable: $10, 1, 1$. The first step swaps the 10 and the second 1 breaking stability.
   (b) Yes, by never swapping equal items (or by never inserting before an item of equal value).
   (c) Yes, by never swapping equal items.

5. Make each element a pair of the initial value, and the initial index. Then make the Comparator use the initial ordering, but defer to the index for equal items.

Merge Sort

Earlier we saw that we can merge two sorted lists of length m and n into a sorted list of length $m + n$ in $\Theta(m + n)$ time. We can use this procedure to design a sorting algorithm called merge sort. The idea is as follows:

1. Split list evenly (or off by 1) into two halves: left and right.
2. Recursively call merge sort on each half.
3. Merge the halves together to produce a full sorted array.

This style of algorithm is called divide-and-conquer. To analyze the runtime, we can consider the implicit binary tree lurking in the background (called a recursion tree). Let each call to merge sort denote a node of the tree. Each call splits the array into a left and right half, and thus creates two children for a given node. Excluding the recursive call, each call to merge sort does $\Theta(n)$ work in the merging step. Thus $\Theta(n)$ work is done by each level of the tree. The tree will have $\Theta(\lg n)$ levels giving a runtime of $\Theta(n \lg n)$.

Below we give an implementation that uses $\Theta(n)$ extra space. To do this we try to manage memory a bit more efficiently.

MergeSort.java

```
import java.util.Arrays;
import java.util.Random;

public class MergeSort
{
	//Merge sorted lists in data[L..M] and data[M+1..R]
	//using tmp as temporary storage
	public static void merge(int[] data, int L, int M, int R, int[] tmp)
	{
		int N = R-L+1;
		int a = L, b = M+1;
		for (int i = 0; i < N; ++i)
		{
			if (a > M) tmp[i] = data[b++];
			else if (b > R || data[a] <= data[b]) tmp[i] = data[a++];
			else tmp[i] = data[b++];
		}
		for (int i = 0; i < N; ++i) data[L+i] = tmp[i];
	}
	public static void mergeSort(int[] data)
	{
		mergeSort(data,0,data.length-1, new int[data.length]);
	}
	//Merge sort data[L..R], tmp is auxiliary storage
```

```
    public static void mergeSort(int[] data, int L, int R, int[] tmp)
    {
        if (L == R) return;
        int M = (L+R)/2; // M<R
        mergeSort(data,L,M,tmp);
        mergeSort(data,M+1,R,tmp);
        merge(data,L,M,R,tmp);
    }

    public static void main(String[] args)
    {
        Random ran = new Random();
        int[] arr = {1,2,3,4,5,6,7,8,9,10};
        ArrayUtils.shuffle(arr, ran);
        System.out.println(Arrays.toString(arr));
        mergeSort(arr);
        System.out.println(Arrays.toString(arr));
    }
}
```

Note that unlike selection, insertion, bubble and heap sort, merge sort requires extra memory to do the sorting. We say that the other sorts are in-place but merge sort isn't. To see that heap sort can be done in-place, note that we can just treat the original array as our heap's data array.

Often, once the lists get small enough, some implementations of merge sort will change to insertion sort which can run faster on small lists (like smaller than 7).

Merge Sort Exercises

1. Show step-by-step how merge sort orders the elements of $8, 7, \ldots, 2, 1$.

2. Is merge sort stable?

3. ($\star$) Explain how to augment the merge sort algorithm to also compute the number of inversions in the array.

4. ($\star\star$) Give an algorithm that merges k sorted lists of length n into a combined sorted list of length kn in $\Theta(kn \log k)$ time.

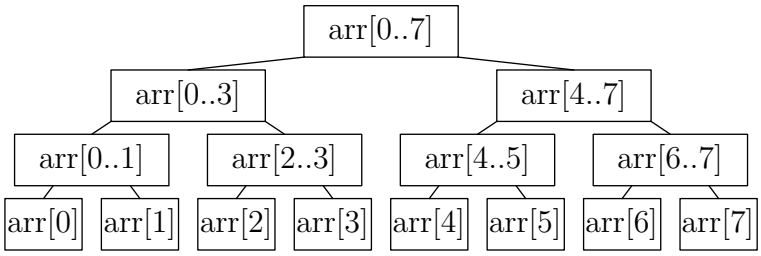
5. Given an array arr and an index (called the pivot) order the array so all elements less than or equal to the pivot are on its left side, and all elements greater than the pivot are on its right side. Return the new location of the pivot (which will likely need to move to obtain the above property). Your runtime should be $\Theta(n)$ but you may use extra space.

   For example, if the list begins as $5, 4, 3, 2, 1$ and index 3 (the value 2) is the pivot then you can reorder the array to be $1, 2, 5, 4, 3$ with the pivot now at index 1.

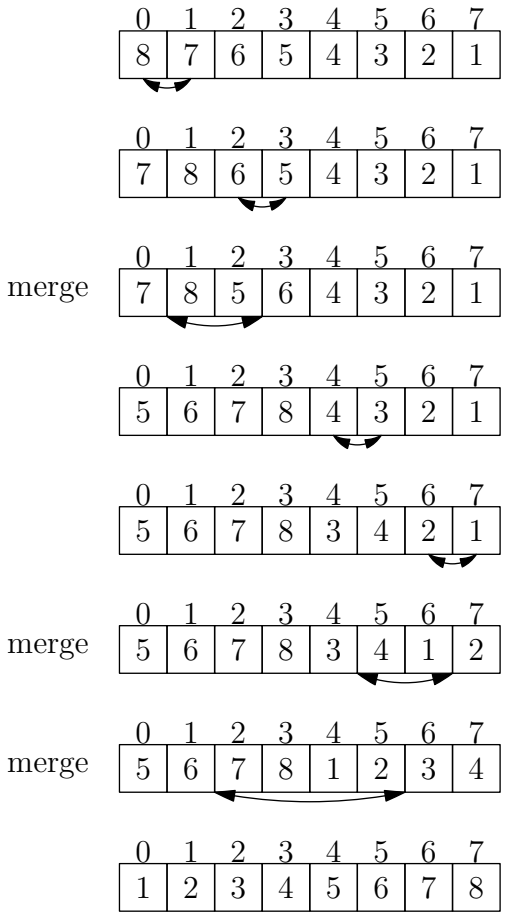
public static int partition(**int**[] arr, **int** pivot)

Merge Sort Solutions

1. Depicted below is the implicit tree existing in the background of merge sort.



Merge sort does a postorder traversal of this tree. Here are the steps.



2. Yes. The key step that ensures stability is that we prefer the left list to the right when we merge two sorted lists.

3. At a high level, each inversion in the list occurs in the left half, the right half, or crosses the middle. Recursively we compute the inversions in each half. The middle crossing inversions are computed in the call to merge.

   Here we describe the code more precisely. Inversions are detected when merge chooses an element of the right list over the left list. Each time an element of the right list is chosen we add k to the total number of inversions, where k is the number of remaining elements in the left list. We then augment merge to return the total number of inversions it finds. We augment mergeSort to sum the inversions from its two recursive calls, and the call to merge, and then returns the sum.

4. Build a min heap of the lists that uses the first unmerged element for the comparisons (you can store an index/iterator with each list of what element it is up to). Each step of the merge you call removeMin to obtain the list containing the smallest unmerged item. After using the item, if there are still unused elements in that list, re-add the list to the min heap.

   Another solution simply pairs up the lists and merges them, and repeats this process trying to merge lists of equal length (as if you were midway through the merge sort algorithm on your way up the implicit tree).

5. A simple implementation constructs two new lists containing the elements below the pivot and greater the pivot and then copies them back into the original array. Here we give an implementation using $\Theta(1)$ memory.

```
public static int partition(int[] arr, int pivot) {
  int pv = arr[pivot];
  swap(arr,0,pivot);
  int i = 1, j = arr.length−1;
  while (i <= j) {
    if (arr[i] <= pivot) i++;
    else if (arr[j] > pivot) j−−;
    else swap(arr,i++,j−−);
  }
  swap(arr,0,i−1);
}
```

   The code above partitions in-place and has a $\Theta(n)$ runtime.

Quick Sort

The idea of quick sort is to repeatedly call the partition function above. The process is as follows:

1. Choose a pivot index (somehow) and partition the array around it.

2. Recursively call quick sort on the left and right parts of the partition.

If the pivots are chosen well (close to the median) then the left and right parts will be roughly half of the remaining values, and we can get a $\Theta(n \lg n)$ runtime as we did in merge sort. If the partitions are chosen poorly then the runtime can deteriorate to $\Theta(n^2)$. The issue is that apriori we don't know where the median is. Here are some potential choices for the pivot:

1. Choose the first element in the list.
2. Take the first, middle, and last element and choose their median (called median-of-three).
3. Randomly choose an element from the list.
4. Find the actual median and use it.

The first element is quick and easy to choose, but it also can deteriorate on some simple cases. Median-of-three does better on many arrays, but can still deteriorate, especially if the list to sort is supplied by an adversary. Random choice can defeat an adversary, and has an expected runtime of $\Theta(n \lg n)$, but this is only an average case. Sometimes it will deteriorate to $\Theta(n^2)$. There is a somewhat complex algorithm that will find the median of a list of numbers in linear time. If used, this will guarantee a worst-case runtime of $\Theta(n \lg n)$ for quick sort. That said, this method has a very large constant associated with it, and is rarely if ever used.

Below we give an implementation of quick sort:

QuickSort.java

```
import java.util.Arrays;
import java.util.Random;

public class QuickSort
{
	public static void swap(int[] arr, int i, int j)
	{
		int tmp = arr[i];
		arr[i] = arr[j];
		arr[j] = tmp;
	}
	//Partitions arr[L..R] around pivot in location p
	public static int partition(int[] arr, int L, int R, int p)
	{
		int pivot = arr[p];
		swap(arr,p,L); //Put pivot in first spot
		int i = L+1, j = R;
		while (i <= j)
		{
```

```
            if (arr[i] <= pivot) i++;
            else if (arr[j] > pivot) j--;
            else swap(arr,i++,j--);
        }
        //Now i points after last element <= pivot
        swap(arr,L,i-1);
        return i-1;
    }
    public static void quickSort(int[] arr, Random ran)
    {
        quickSort(arr, 0, arr.length-1, ran);
    }
    public static void quickSort(int[] arr, int L, int R, Random ran)
    {
        if (L>=R) return;
        int p = ran.nextInt(R-L+1)+L;
        p = partition(arr, L, R, p);
        quickSort(arr,L,p-1,ran);
        quickSort(arr,p+1,R,ran);
    }
    public static void main(String[] args)
    {
        Random ran = new Random();
        int[] arr = {1,2,3,4,5,6,7,8,9,10};
        ArrayUtils.shuffle(arr, ran);
        System.out.println(Arrays.toString(arr));
        quickSort(arr,ran);
        System.out.println(Arrays.toString(arr));
    }
}
```

Note that the implementation above is in-place (since partition is in-place), but is unstable (since partition is unstable). As with merge sort, quick sort implementations will often change to insertion sort once the list size gets small enough.

The Java API uses variant of quicksort with 2 pivots (dual-pivot quicksort) for sorting primitive arrays and a variant of merge sort for objects (called Timsort). Some C++ STL sort implementations use introsort, a variant of quicksort that will change to heapsort if there are too many poor pivot choices in a row (indicating a weird adversarial case).

Quick Sort Exercises

1. Run quick sort on $5, 1, 7, 4, 3, 2, 6$ where you always choose the first element of each sublist as the pivot. You can partition using whichever implementation you want.

2. Suppose the first element is your choice of pivot. Give an example of a list of length n that gives $\Theta(n^2)$ runtime.

3. (⋆⋆) Show how to use partition to find the kth smallest element of a list in expected linear time (called quick select). Your code will modify the underlying array.

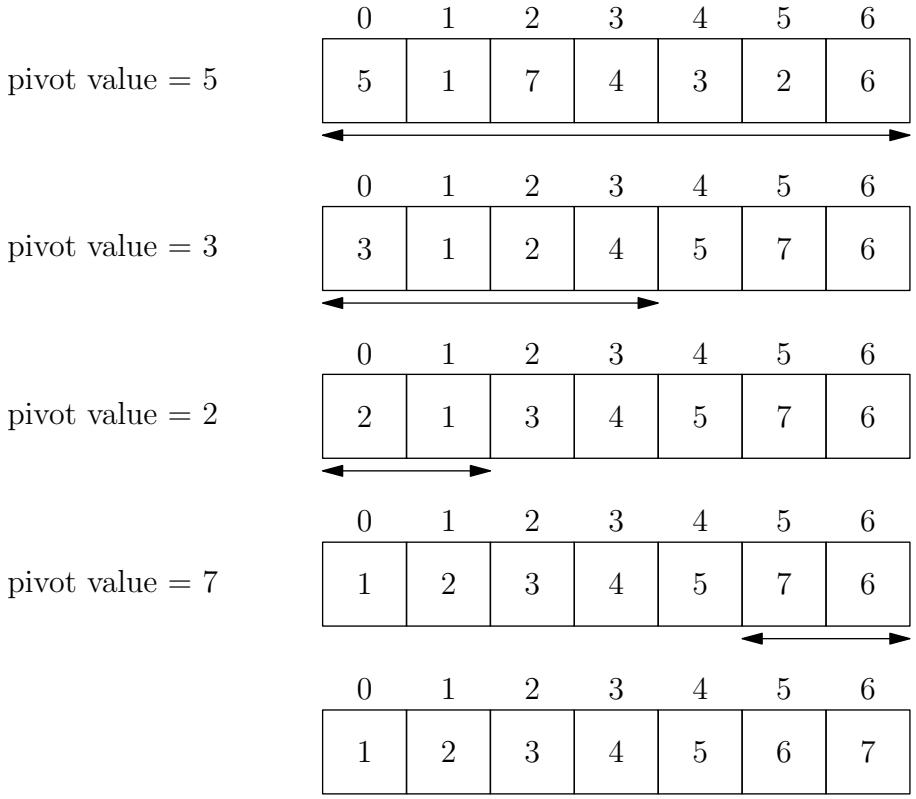
4. (⋆⋆) Bogosort is an algorithm that keeps randomly shuffling the array until it is sorted:

   while(!isSorted(arr)) shuffle(arr);

   Given an array that isn't sorted, what is the expected runtime of Bogosort?

Quick Sort Solutions

1. The partitions performed are shown below.



2. For the partition code above (and most implementations of partition), a sorted list or reverse sorted list will give $\Theta(n^2)$ runtime.

3. The idea is that after partition is called, the pivot is in its correct place in the sorted list. Then, similar to binary search, you can choose which side to continue to search on. Code follows:

   QuickSelect.java

   import java.util.Arrays;
   import java.util.Random;

```
public class QuickSelect {
	//Selects element k (in sorted order)
	//Modifies arr
	public static int quickSelect(int[] arr, int k,
			Random ran)
	{
		return quickSelect(arr,k,0,arr.length−1,ran);
	}
	public static int quickSelect(int[] arr, int k,
			int a, int b, Random ran)
	{
		int n = b−a+1;
		if (n == 1) return arr[a];
		int p = a+ran.nextInt(n);
		p = QuickSort.partition(arr, a, b, p);
		if (p−a == k) return arr[p];
		if (p−a > k)
			return quickSelect(arr,k,a,p−1,ran);
		else
			return quickSelect(arr,k−(p−a+1),p+1,b,ran);
	}
	public static void main(String[] args)
	{
		Random ran = new Random();
		int[] arr = {1,2,3,4,5,6,7,8,9,10};
		ArrayUtils.shuffle(arr, ran);
		System.out.println(Arrays.toString(arr));
		for (int k = 0; k < arr.length; ++k)
		{
			if (k > 0) System.out.print(", ");
			int[] tmp = Arrays.copyOf(arr, arr.length);
			System.out.print(quickSelect(tmp,k,ran));
		}
		System.out.println();
	}
}
```

Similar to quick sort, this is $\Theta(n^2)$ in the worst-case.

4. Essentially we keep flipping a coin that has a $1/n!$ chance of coming up heads. The expected number of flips is $n!$. Each flip requires $\Theta(n)$ runtime, so the total expected runtime is $\Theta(n \cdot n!)$. To see why $n!$ iterations are expected, let X denote the number of iterations (a random variable). Then we have

$$E[X] = 1 + \frac{n! - 1}{n!} E[X]$$

giving $E[X] = n!$.

Lower Bounds on Sorting

Here we sketch the argument that if all you have is an array of Comparables (or a Comparator) you cannot do better than $n \lg n$ in the worst case. Suppose we have an array of n distinct items. It can have any one of $n!$ different orderings. Each time we perform a comparison of two items a, b in the list, we divide the orderings into two piles: orderings consistent with $a < b$ and orderings consistent with $b < a$. In the worst case, we will end up with the larger pile. Thus to minimize our worst-case runtime we will try to make the piles as close to even as possible. This creates a balanced binary tree with $n!$ leaves. The height is thus $\Theta(\lg(n!))$ which is the same as $\Theta(n \lg n)$. As the height determines the worst-case runtime, we are done.

Counting Sort

If we know more about the values than just how to compare them we can do better than $n \lg n$. For instance, supppose we have a list of n integers between 0 and $k - 1$, inclusive, where k is not prohibitively large. Then we can sort in $\Theta(n)$ time by using an auxiliary array of size $\Theta(k)$ that simply counts the number of occurrences of each element. Below we see that we can also sort objects with integer keys stably using a similar idea.

Lower Bounds and Counting Sort Exercises

1. Suppose you have a list of n integers between 1000000900 and 1000010000. How can you sort them in $\Theta(n)$ time?

2. Suppose you have an array of Student objects given below.

Student.java

```
public class Student
{
	private int ID;
	private String name;
	public Student(int ID, String name)
	{
		this.ID = ID;
		this.name = name;
	}
	public int getID() { return ID; }
	public String getName() { return name; }
	public String toString() { return "("+ID+","+name+")"; }
}
```

Suppose further that you have an array of n Students and all IDs are between 0 and $k-1$, where k is not enormous. How can you stably sort the Students by ID.

3. ($\star$) Suppose you have n integers between 0 and n^2-1, inclusive. How can you sort them in $\Theta(n)$ time with $\Theta(n)$ memory? Will it work up to n^3-1?

4. ($\star\star$) Show how to sort an array n of Java ints in $\Theta(n)$ time with $\Theta(n)$ memory.

5. ($\star\star$) Show that $\lg(n!) = \Theta(n \lg n)$.

Counting Sort Solutions

1. Subtract 1000000900 from all values and use counting sort.

2. Proceed like counting sort, but the array is an array of lists (called buckets). Each time you put an element in the appropriate slot it gets appended to the end of the list. This algorithm is called *bucket sort*. Note that we cannot simply count the number of elements with each ID since we lose the name information.

   Alternatively, we can use counting sort directly, but after we count how many of each student ID occurs, we use it to compute the positions in the final list. Code follows:

CountingSortStudents.java

```
import java.util.Arrays;

public class CountingSortStudents
{
	public static int getMaxID(Student[] arr)
	{
		int max = arr[0].getID();
		for (int i = 1; i < arr.length; ++i)
			if (arr[i].getID() > max)
				max = arr[i].getID();
		return max;
	}
	//Assume IDs are non−negative and max isn't too big
	public static void countingSort(Student[] arr)
	{
		int max = getMaxID(arr);
		int[] counts = new int[max+1];
		for (Student s : arr) counts[s.getID()]++;
		//Fill counts with starting positions
		int total = 0;
		for (int i = 0; i < counts.length; ++i)
		{
			int old = total;
```

```
                total += counts[i];
                counts[i] = old;
            }
            Student[] copy = Arrays.copyOf(arr, arr.length);
            for (Student s: copy)
            {
                arr[counts[s.getID()]] = s;
                counts[s.getID()]++;
            }
        }

        public static void main(String[] args)
        {
            Student a = new Student(1,"A");
            Student b = new Student(1,"B");
            Student c = new Student(2,"C");
            Student d = new Student(2,"D");
            Student e = new Student(3,"E");
            Student f = new Student(3,"F");
            Student[] arr = {f,d,c,e,b,a};
            countingSort(arr);
            System.out.println(Arrays.toString(arr));
        }
    }
```

3. Think of each integer as having the form $x = an + b$. That is, we are representing it in base n. Using counting sort, sort all of the integers by their b-values (that is, mod n). Then use counting sort again (as in the previous part) to stabily sort all of the integers by their a-values (that is, sort by x/n). The same type of algorithm will work for $n^3 - 1$ as well. We simply write $x = an^2 + bn + c$ and sort 3 times: first by c, then by b, then by a.

4. Java ints have 32 bits. First sort by the lowest order bit. Then stably by the next bit, and so forth. After 32 rounds of counting sort all elements will be sorted. This is called *(binary) radix sort.*

5. Note that
$$\lg(n!) = \lg(1) + \lg(2) + \lg(3) + \cdots + \lg(n-1) + \lg(n).$$
Then we have
$$\frac{n}{2}\lg(n/2) \leq \lg(1) + \lg(2) + \lg(3) + \cdots + \lg(n-1) + \lg(n) \leq n\lg(n).$$
Thus $\lg(n!) = \Theta(n \lg n)$.