STUDENT OUTLINE

Lesson 25 – Quicksort

INTRODUCTION: Quicksort is another recursive sorting algorithm that works by dividing lists in half. Whereas mergesort divided lists in half and then sorts each sublist, quicksort will roughly sort the entire list, and then split the list in half. The order of these two sorts falls into the category O(N * log N), which was introduced in Lesson 23. When the lists become large, either of these sorts will do an excellent job.

The key topics for this lesson are:

A. Quicksort
B. Order of Quicksort

VOCABULARY: QUICKSORT

DISCUSSION:

A. Quicksort

1. Here is the overall strategy of quicksort:
   a. Quicksort chooses an arbitrary value from somewhere in the list. A common location is the middle value of the list.
   b. This value becomes a decision point for roughly sorting the list into two sublists, which we call the "left" and the "right" sublists. All the values smaller than the dividing value are placed in the left sublist, while all the values greater than the dividing value are placed in the right sublist.
   c. Each sublist is then recursively sorted with quicksort.
   d. The termination of quicksort occurs when a list of one value is obtained, which by definition is sorted.

2. This is the code for quicksort:

See Handout, H.A.25.1, *quickSort Method.*

```
void quickSort (int[] list, int first, int last)
{
 int g = first, h = last;
 int midIndex = (first + last) / 2;
 int dividingValue = list[midIndex];
 do
 {
  while (list[g] < dividingValue) g++;
  while (list[h] > dividingValue) h--;
  if (g <= h)
  {
   //swap(list[g], list[h]);
   int temp = list[g];
   list[g] = list[h];
   list[h] = temp;
   g++;
   h--;
  }
 }
 while (g < h);
 if (h > first) quickSort (list,first,h);
 if (g < last) quickSort (list,g,last);
}
```

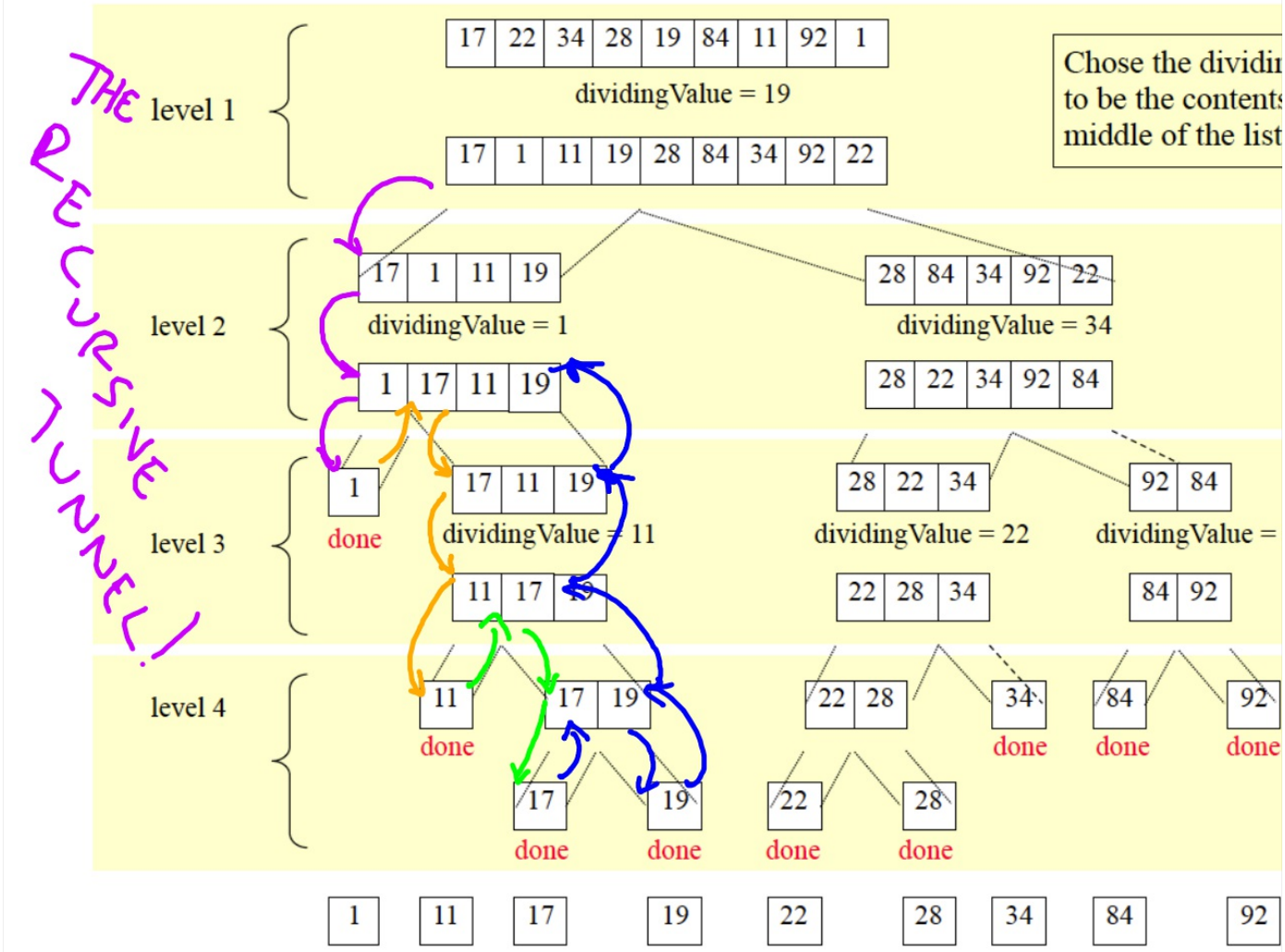
FIND THE DECIDING VALUE

LOOK FOR 2 VALUES OUT OF ORDER

SWAP IF NEEDED

ADVANCE TRACKER

BASE CASES


THE RECURSIVE TUNNEL!
level 1
17 22 34 28 19 84 11 92 1
dividingValue = 19
17 1 11 19 28 84 34 92 22
Chose the dividi
to be the content
middle of the list
level 2
17 1 11 19
dividingValue = 1
28 84 34 92 22
dividingValue = 34
1 17 11 19
28 22 34 92 84
level 3
1
done
17 11 19
dividingValue = 11
28 22 34
dividingValue = 22
92 84
dividingValue =
11 17 19
22 28 34
84 92
level 4
11
done
17 19
22 28
done
34
done
84
done
92
done
17
done
19
done
22
done
28
done
1 11 17 19 22 28 34 84 92

See Transparency T.A.25.1, *Action of Quicksort.*

READ ....
TRACE....
EXPERIMENT

3. Suppose we have the following list of 9 unsorted values in an array:

17 22 34 28 19 84 11 92 1

a. The values received by the formal parameter `list` in the first call of `quicksort` are a reference to an array, and the first and last cells of the array, 0 and 8. Variable `first` is initialized with the value 0, and `last` is initialized with the value 8.

b. The `midIndex` value is calculated as (0+8) / 2, which equals 4.

c. The `dividingValue` is the value in location 4, `list[4] = 19`.

d. The value of 19 will be our decision point for sorting values into two lists. The list to the left will contain all the values less than or equal to 19. The list to the right will contain the values larger than 19. Or simply put, small values go to the left and large values go to the right.

e. The identifiers `g` and `h` will be indices to locations in the list. The **`while`** loops will move `g` and `h` until a value is found to be on the wrong side of the dividing value of 19. The `g` index is initialized with the value of `first` and the `h` index is initialized with the value of `last`.

f. The `g` index starts at position 0 and moves until it "sees" that 22 is on the wrong side. Index `g` stops at location 1.

g. The `h` index starts at position 8. Immediately it "sees" that the value 1 is on the wrong side. Index `h` never moves and stays at position 8 of the array.

h. Since `g <= h` (1 <= 8), the values in `list[1]` and `list[8]` are swapped. After the values are swapped, index `g` moves one position to the right and index `h` moves one position to the left.

i. The values of the pointers are now: `g = 2, h = 7`. We continue the **`do-while`** loop until `g` and `h` have passed each other, that is when `g > h`. At this point, the lists will be roughly sorted, with values smaller than 19 on the left, and values greater than 19 on the right.

j. If the left sublist has more than one value, (which is determined by the `h > first` expression), then a recursive call of `quicksort` is made. This call of `quicksort` will send the index positions that define that smaller sublist.

k. Likewise, if the right sublist has more than one value, `quicksort` is called again and the index positions that define that sublist are passed.

B. Order of Quicksort

See Transparency T.A. 25.2, *Log_2N Steps.*

1. Determining the order of quicksort, O (N* $\log_2$N), is a difficult process. The best way to understand it is to imagine a hypothetical situation in which each call of `quicksort` results in sublists of the same size. Such a size is 128, because it is a power of 2.

2. If a list has 128 elements, the number of calls of `quicksort` required to move a value into its correct spot is $\log_2$128, which equals 7 steps. Dividing the list in half gives us the $\log_2$N aspect of quicksort.

3. But we need to do this to 128 numbers, so the approximate number of steps to sort 128 numbers will be 128 * $\log_2$ 128. A general expression of the order of quicksort will be O(N * $\log_2$N). An O(N * $\log_2$N) algorithm is a more specific designation of the broader category called O(N * log N).

4. A graph of an O(N* $\log_2$N) algorithm is close to a linear algorithm, for large values of N. The $\log_2$N number of steps grows very slowly, making quicksort a dramatic improvement over the O(N^2) sorts.

CUTTING IN HALF

HOW MANY TIMES TO CUT LISTS IN HALF

SUMMARY/ REVIEW: Quicksort is generally the fastest and therefore most widely used sorting algorithm. There is a variation of quicksort named "quickersort" but it is still in the same class of algorithms. Once again, recursion makes fast work of a difficult task.

ASSIGNMENT: Lab Exercise L.A.25.1, *Quicksort*

ADDING CODE INTO ORIGINAL QUADRATICS LAB