Data Structures & Algorithms 2

Topic 3 – Quicksort

Quicksort

- Quicksort is the most popular sorting algorithm of all
- In the majority of situations it operates in O(n*logn) time (a quicker O(n*logn) than mergesort)
- It also doesn't need additional memory to run (mergesort needed an extra O(n) amount to store the workspace array)
- Like mergesort, quicksort is a recursive sorting algorithm, splitting up an array and calling itself on each half

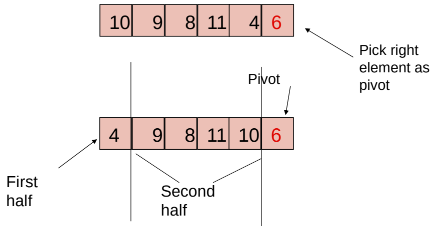
Overview

- Partition array (or subarray) into two halves, putting low values into one half and high values into the other half
- Call quicksort on the left half
- Call quicksort on the right half

Choosing a pivot

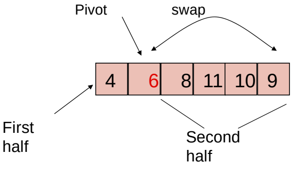
- The idea is that all the elements will be split into two separate lists depending on whether they are bigger or smaller than some pivot value
- Lets start off by just picking a random pivot – the rightmost element in the array to be sorted
- Now split the numbers apart depending on whether they are bigger or smaller than this element

Split into two halves



Put the pivot in place

- Now swap the pivot with the first element in the second half
- The pivot is now in its final resting position!



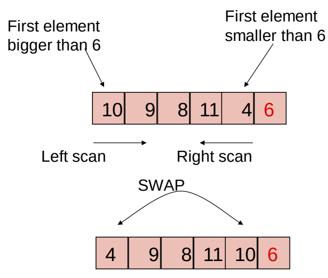
The code for that…

```
public void recQuickSort(int left, int right) {

      if(right-left <= 0)    Base Case       // if size <= 1,
         return;                              //    already sorted
      else{                                   // size is 2 or larger

         long pivot = theArray[right];        // rightmost item
                                              // partition range
         int partition = partitionIt(left, right, pivot);
         recQuickSort(left, partition-1);     // sort left side
         recQuickSort(partition+1, right);    // sort right side
      }
}
```

Sorting into two halves

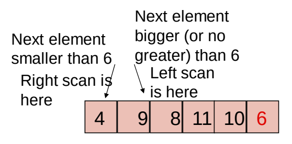
- The sorting works using two ‘scans’ of the array
  - One from left to right →…..
  - One from right to left …..←
- The left scan starts at the beginning and searches for the first element that is bigger than the pivot
- The right scan starts at the end and searches for the first element that is smaller than the pivot
- These elements are then swapped

Scan & swap



Keeping swapping

- The scans keep moving onto the next element and swapping each other’s values
- As soon as one scan has gone past the other the swapping method stops
- Now all the values have been separated into two groups



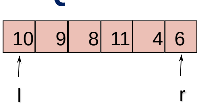
Now swap pivot with left pointer

Quicksort, Notation

10	9	8	11	4	6

```
QS(A,Left array position ,Right array position )

L: Leftmost position
R: Rightmost position
i: L-R scan positions
j: R-L scan positions
pivot: We will pick the rightmost element of the
array as pivot
```

Quicksort




```
QS(A,1,6)

L:      1
R:      6
i:      1
j:      6
pivot:  6
```

Quicksort

10	9	8	11	4	6

l r

```
QS(A,1,6)

L:      1
R:      6
i:      1
j:      6 5
pivot:  6
```

Here the element at position i is > pivot while the element at position j is < pivot so swap the elements in i and j

Quicksort

4	9	8	11	10	6

l r

```
QS(A,1,6)

L:      1
R:      6
i:      1
j:      6 5
pivot:  6
```

Quicksort

4	9	8	11	10	6

r l

```
QS(A,1,6)

L:      1
R:      6
i:      1 2
j:      6 5 4 3 2 1
pivot: 10
```

j<=i so swap i and pivot

Quicksort

4	6	8	11	10	9

```
QS(A,1,6)

L:      1
R:      6
i:      1 2
j:      6 5 4 3 2 1
pivot: 10
```

Now recursively Quicksort the sublist to the left of 6 (4) and the sublist to the right of 6 (8, 11, 10, 9)

Quicksort

4	6	8	11	10	9

l r

```
QS(A,1,6)                 QS(A,2,6)

L:      1                 L:      3
R:      6                 R:      6
i:      1 2 3 4           i:
j:      6 5 4             j:
pivot: 10                 pivot:  9
```

Quicksort

4	6	8	11	10	9

r l

```
QS(A,1,6)                 QS(A,2,6)

L:      1                 L:      3
R:      6                 R:      6
i:      1 2 3 4           i:      4
j:      6 5 4             j:      3
pivot: 10                 pivot:  9
```

i and j have gone past each other again so swap pivot with i

Quicksort

4	6	8	9	10	11

```
QS(A,1,6)

L:      1
R:      6
i:      1 2 3 4
j:      6 5 4
pivot: 10
```

```
QS(A,3,6)

L:       3
R:       6
i:       4
j:       3
pivot:   9
```

Sort the sublist to the left of the pivot (8) and the sublist to the right (10, 11)

- Sort the following numbers showing pivots and swaps:

8	3	9	2	1	6	7	5
1	3	9	2	8	6	7	5
1	3	2	9	8	6	7	5
1	3	2	5	8	6	7	9
1	3	2	5	8	6	7	9
1	2	3	5	8	6	7	9
1	2	3	5	8	6	7	9
1	2	3	5	6	8	7	9
1	2	3	5	6	7	8	9

Code

```
public int partitionIt(int left, int right, long pivot){

       int leftPtr = left-1;            // left    (after ++)
       int rightPtr = right;            // right-1 (after --)
       while(true)
          {
          while( theArray[++leftPtr] < pivot ){}    // scan to the right
          while(rightPtr > 0 && theArray[--rightPtr] > pivot){}
                                                   // scan to the left

          if(leftPtr >= rightPtr)       // if pointers cross,
             break;                     //    partition done
          else                          // not crossed, so
             swap(leftPtr, rightPtr);   //    swap elements
          }
       swap(leftPtr, right);            // swap pivot into correct place
       return leftPtr;                  // return pivot location
}
```

Things to notice

- The left scan will always stop on the pivot (because the pivot is not smaller than the pivot)
- At this point, the two scans are guaranteed to have crossed paths
- However, the right scan might go below 0 off the edge of the range so we need to introduce a check to make sure we don't go too far
- This slows down the performance – we'd like to get rid of this
- Leftscan starts at -1 and rightscan at the pivot because they are incremented /decremented before they're used for the first time

---

Swaps and Comparisons

- Comparisons
  - For each partition there will be at most n+1 or n+2 comparisons
  - Every item will be encountered and compared by one or other of the scans, leading to n comparisons
  - The scans will overshoot each other before they realise it, leading to some additional comparisons

10	9	8	11	4	6

Left scan → ← Right scan

Swaps and Comparisons

- Swaps
  - The number of swaps depends on how the data is arranged
  - If the data is inversely arranged then every pair of values must be swapped, a total of about n/2 (actually (n-1)/2 +1)
  - For random data, half of the elements will already be in the right half position meaning only n/4 swaps will be required

Overall complexity

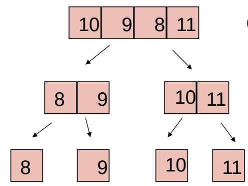
- The order of the algorithm is decided by whichever of swaps or comparisons is greater
- Both are O(n) for one partition
- If each partition halves the array (or subarrays) then the total number of partitionings required is the number of times that n can be halved → $\log_2 n$
- O(n) * $\log_2 n$ gives us O(n*logn)

Complexity

$O(\log_2 n)$ partitioning levels

n-1 levels = 3

$\log_2 n = 2$



O(n) Steps

O(n) Steps

Base Case

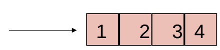
Quicksort performance

- The performance of divide-and-conquer algorithms comes from splitting the problem each time
- If the problem is halved by each split the algorithm will be O(splitting code) * O(logn) since the problem can only be split $\log_2 n$ times before resulting in single units
- However, if the problem isn't being split in half then the efficiency is going to be lower
- The split in quicksort depends on the pivot

Choice of pivot

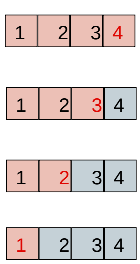
- If the pivot is the middle value in the array, then the array will be split in half perfectly
- However, if the pivot is the highest or lowest element then the split will be completely lop-sided
- In the worst case scenario n number of splits will be required meaning that the performance of quicksort degenerates to $O(n^2)$

Must be split 4 times using rightmost element as a pivot



Worse pivot choice

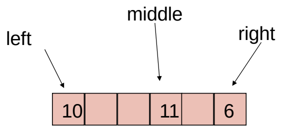
4 Steps for 4 items = O(n)



Choice of pivot

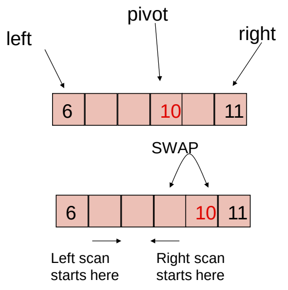
- Ideally we should pick a mid-range value for the pivot (as opposed to just a random number)
- If you were to examine all the numbers and calculate the mid-value this would take longer than the sort itself
- A compromise involves median of three partitioning
- Pick the first, middle and last elements of the array (or subarray) and take the middle of them

Median of Three



- Select middle value as the pivot
- Sort the three values into the correct positions
- Swap the pivot to the right

Median of Three



Bonus

- We can start the partition algorithm at left+1 and right-1 because we have already sorted the left and right elements
- We know they are smaller and bigger than the pivot and are therefore in the right place
- These extreme values act as buffers which stop the left scan from scanning an element below 0
- The right scan will always stop at the leftmost element because it is guaranteed to be less than the pivot

Increases efficiency

- Small increase in code efficiency – no check required for scanning left

```
while( theArray[++leftPtr] < pivot ) {}; // scan right
while( theArray[--rightPtr] > pivot ){}; // scan left
```

- Sort the following numbers showing pivots and swaps using median of 3:

8	3	9	1	6	7	5
1	3	9	5	6	7	8
1	3	9	7	6	5	8
1	3	5	7	6	9	8
1	3	5	7	6	9	8
1	3	5	6	7	9	8
1	3	5	6	9	7	8
1	3	5	6	7	9	8
1	3	5	6	7	8	9

Median of Three

```
public long medianOf3(int left, int right) {

     int center = (left+right)/2;
                                            // order left & center
      if( theArray[left] > theArray[center] )
         swap(left, center);
                                            // order left & right
      if( theArray[left] > theArray[right] )
         swap(left, right);
                                           // order center & right
      if( theArray[center] > theArray[right] )
         swap(center, right);

      swap(center, right-1);                // put pivot on right
      return theArray[right-1];             // return median value
}
```

New problem

- If you use median-of-three partitioning then the quicksort algorithm can't work for partitions of three or fewer items
- You could implement a "manual sort" method to sort three elements
- Another option is to use insertion sort when the subarray to be sorted becomes suitably small (insertion sort works really well for nearly sorted data)
- Rather than a cutoff of 3, you could employ the insertion sort method as soon as the size to be sorted falls below 10 or 20

Switching between sorting algorithms

```
public void recQuickSort(int left, int right) {

     int size = right-left+1;
      if(size < 10)                // insertion sort if small
         insertionSort(left, right);
      else{                        // quicksort if large

         long median = medianOf3(left, right);
         int partition = partitionIt(left, right, median);
         recQuickSort(left, partition-1);
         recQuickSort(partition+1, right);
      }
}
```

Maximizing efficiency

- Knuth recommends a cut-off of 9 for maximal efficiency
- The optimum number depends on the computer, operating system, compiler and of course the data you're sorting
- Another idea is to sort the whole array without bothering to sort small partitions smaller than the cutoff
- Finally, the area is nearly sorted and you can use insertion sort to tidy the whole thing up