Data Structures - Advanced Sorting 2

Dr. TGI Fernando [1] [2]

[1] Email: tgi.fernando@gmail.com
[2] URL: http://tgifernando.wordpress.com/

Partitioning

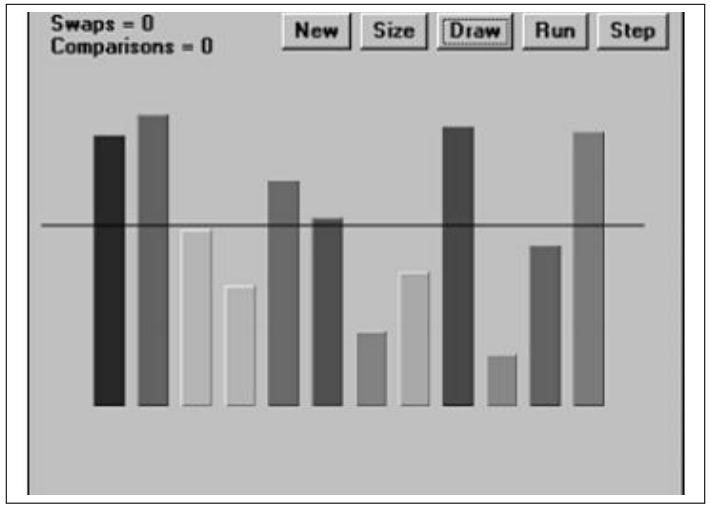
Divide data into two groups, so that all the items with a key value higher than a specified amount are in one group, and all the items with a lower key value are in another.

E.g.

- Employees who live within 15 miles of the office and those who live farther away.
- Divide students into those with GPA higher and lower than 3.5.

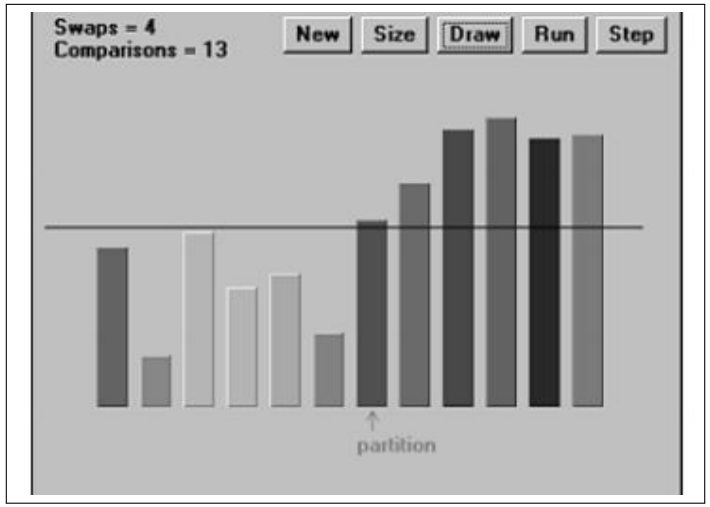
Pivot - value used to determine into which of the group an item is placed.

Before Partitioning



Note: The horizontal line represents the pivot value.

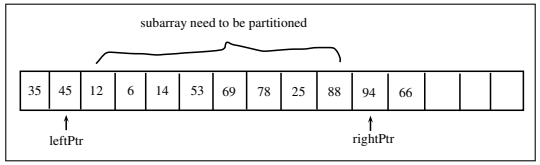
After Partitioning



The arrow labelled **partition** points to the leftmost item in the right (higher) subarray. This value is returned from the partitioning method.

The Partition Algorithm

- Needs two pointers (array indices),
  - One is pointing to the left end (`leftPtr`), and
  - The other is pointing to the right end (`rightPtr`).
- `leftPtr` moves toward the right, and `rightPtr` moves towards left.
- When implementing this on an array,
  - `leftPtr` is initialized to one position to the left of cell, and
  - `rightPtr` to the right of last cell.
- This will help the algorithm to increment (`leftPtr`) or decrement(`rightPtr`) before they are used.



The Partition Algorithm (Contd.)

Stopping and swapping

- When `leftPtr` encounters a date item smaller than the `pivot` value, it keeps going because the data item is already in correct side of the subarray.

```
while(arr[++leftPtr] < pivot) // find bigger item
  ; // (no operation)
```

- However, if it finds a data larger than the `pivot`, it stops keep going.
- Similarly, when `rightPtr` encounters an item larger than the `pivot`, it keeps going.

```
while(arr[--rightPtr] > pivot ) // find smaller item
  ; // (no operation)
```

- But, when it finds a smaller item, it also stops.

The Partition Algorithm (Contd.)

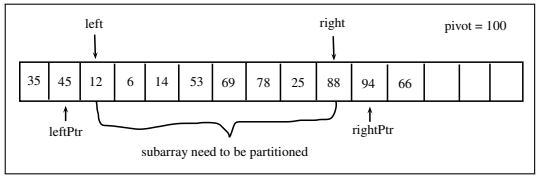
Stopping and swapping (Contd.)

- When both of these pointers stop moving, both `leftPtr` and `rightPtr` point to items that are in the wrong sides of the array.
- At this time, these two items are swapped.
- After the swap, the two pointers moving again and stopping at items that are in the wrong side of the array and swapping them.
- When the two pointers meet each other, the partitioning process is completed.

The Partition Algorithm (Contd.)

Handling unusual data

- If all the data is smaller than the `pivot` value, the `leftPtr` variable will go indefinitely looking for a larger item.



- A similar fate will occur if all the data is larger than the `pivot` value.
- To avoid these problems, extra test must be placed in the `while` loops to check the ends of the subarray:
  1. `leftPtr < right` in the first loop in slide 6
  2. `rightPtr > left` in the first loop in slide 6

The Partition Algorithm (Contd.)

Handling unusual data (Contd.)

```
while(leftPtr <right && arr[++leftPtr] < pivot) // find
bigger item
  ; // (no operation)

while(rightPtr> left && arr[--rightPtr] > pivot ) // find
smaller item
  ; // (no operation)
```

After being partitioned, the data is by no means sorted; it has simply been divided into two groups. However, it's more sorted than it was before.

The Partition Algorithm (Contd.)

```
public int partition (int theArray[], int left, int right,
long pivot)
{
  int leftPtr = left - 1; // right of first elem
  int rightPtr = right + 1; // left of pivot
  while(true)
  {
    while(leftPtr < right &&  // find bigger item
    theArray[++leftPtr] < pivot)
      ;  // (nop)

    while(rightPtr > left &&  // find smaller item
    theArray[--rightPtr] > pivot)
      ;  // (nop)

    if(leftPtr >= rightPtr) // if pointers cross,
      break;  // partition done
    else  // not crossed, so
      swap(leftPtr, rightPtr);  // swap elements
  } // end while(true)

  return leftPtr; // return partition
} // end partition()
```

Efficiency of the Partition Algorithm

No of comparisons

- Every item will encountered and used in a comparison by `leftPtr` or `rightPtr`, leading to N comparisons.
- But `leftPtr` and `rightPtr` overshoot each other before they find out they have gone beyond each other.
- So there are one or two extra comparisons before the partition is completed.
- Thus, for each partition there will be $N+1$ or $N+2$ comparisons.
- Does not depend on how the data is arranged.

Efficiency of the Partition Algorithm (Contd.)

No of swaps

- Depend on how the data is arranged.
- If it is inversely ordered, and the pivot value divides the items in half, then every pair of values must be swapped, which is $N/2$ swaps.
- For random data, there will be fewer than $N/2$ swaps in a partition.
- On average, for random data, about half the maximum number of swaps take place.

Although there are fewer swaps than comparisons, they are both proportional to N. Thus, the partitioning process runs in $O(N)$ time.

Quicksort

- Discovered by `C.A.R. Hoare` in 1962.
- $O(N \log N)$
- Recursive algorithm
- Partition an array into two subarrays and then calling itself recursively to quick sort each of these subarrays.

Quicksort (Contd.)

```
public void quicksort(int theArray[], int left, int right)
{
  if(right-left <= 0) // if size <= 1,
    return; // already sorted
  else // size is 2 or larger
  {
    long pivot = ?; // rightmost item
    // partition range
    int partitionIndex = partition(theArray, left, right,
    pivot);
    quicksort(theArray, left, partitionIndex-1); // sort
     left side
    quicksort(theArray, partitionIndex+1, right); // sort
     right side
  }
} // end quicksort()
```

1. Partition the array or subarray into left (smaller keys) and right (larger keys) groups.
2. Call ourselves to sort the left group.
3. Call ourselves again to sort the right group.

Quicksort (Contd.)

Recursive step

- After a partition, all the item in the left subarray are smaller than all those on the right.
- Next we sort the left and the right subarrays separately.
- After this, the entire array is sorted.
- How do we sort these subarrays?
- By calling the `quicksort` algorithm recursively to the left and right subarrays separately.

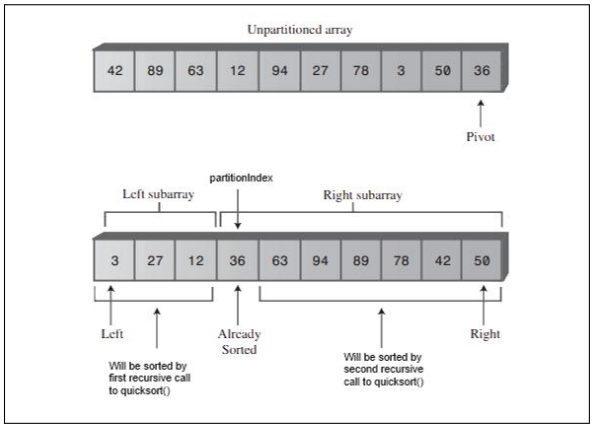
Basic step

The method checks if the subarray consists of only one element. In this case the subarray is already sorted, and the method returns immediately.

Quicksort (Contd.)

Partition method

- This method returns the index of the partition: the left element in the right (larger keys) subarray.



Quicksort (Contd.)

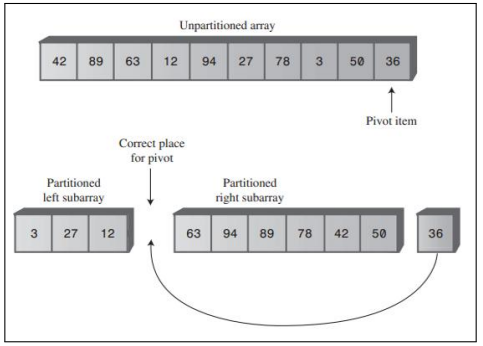
Recursive calls

- After the array is partitioned, `quicksort` calls itself recursively.
- Once for the left part of its array, from `left` to `partitionIndex-1`.
- Once for the right, from `partitionIndex+1` to `right`.
- The item at `partitionIndex` is not included in either of the recursive calls. **WHY?**
- The answer to this question depends on how the `pivot` value is selected.

Quicksort (Contd.)

Choosing a pivot value

1. The pivot value should be a key value of an actual data item.
2. We can pick a data item randomly. For simplicity, let's pick the item on the right of the subarray being partitioned (fixed position).
3. After the partition, the `pivot` must be at the boundary between the left and right subarrays (that's its final position in the array).



Quicksort (Contd.)

How could we move the `pivot` to the proper place?

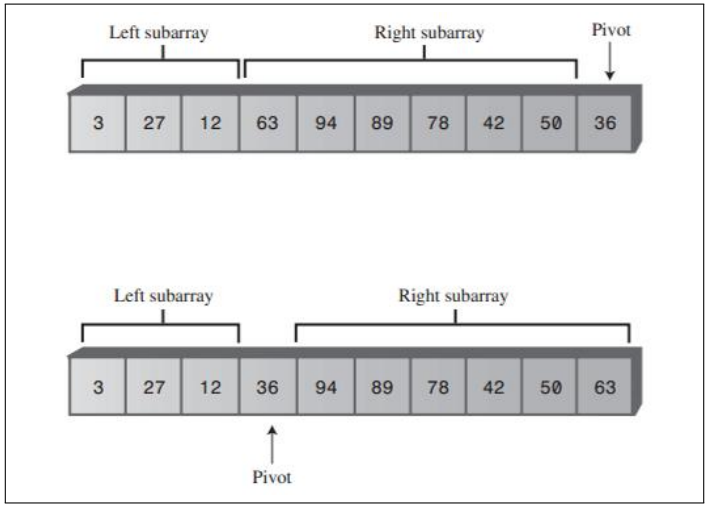
- One option is to shift all items in the right subarray one cell to make room for the `pivot`.
- This is inefficient and unnecessary.
- **Efficient method**
  Since the elements in the right subarray are not sorted, they can be moved around the right subarray.

  Simply swap the `pivot` (36) and the left item in the right subarray (63).
- **The `partition()` method**
  Exclude the rightmost item (`pivot`) from the partitioning process.

  After the partitioning process, swap the left item in the right subarray and the `pivot`.

Quicksort (Contd.)

How could we move the `pivot` to the proper place? (Contd.)



Efficiency of Quicksort

- We saw in the "Partitioning" section that a single partition runs in $O(N)$ time.
- How many number of levels of recursion occur if there are N items in the array?
- For example, if $N = 100$, If we keep dividing 100 by 2, and count how many times we do this, we get the series 100, 50, 25, 12, 6, 3, 1, which is about seven levels of recursion.
- This is approximately equal to $\log_2 N$.
- Thus, the quicksort algorithm operates in $O(N \log N)$ time.