Lecture 17: Quick Sort

Rui Wang
Summary of mergesort

• Merging two sorted array is a key step.
  – A real-world example: merging two piles of sorted papers.
• Divide and conquer: split an input array to two halves, sort each half recursively, and merge.
• Can be converted to a non-recursive version.
  – How?
• $O(N \cdot \log N)$ cost
• Requires additional memory space.
Quick Sort

• The most popular sorting algorithm.
• Divide and conquer.
• Uses recursion.
• Fast, and sort ‘in-place’ (i.e. does not require additional memory space)
Partition (Split)

• A **key step** in quicksort

• Given an input array, and a **pivot value**
  – Think of the pivot value as a threshold.

• Partition the array to two groups: all elements smaller than the pivot are on the left, and those larger than the pivot are on the right.

• Example: 42  89  63  12  94  27  78  3  50  36
  pivot: 40
Partition (Split)

• **Some observations:**
  • The order of elements in each group does not matter (each group is unsorted).
  • What if the pivot value is the value of an actual element in the array?

• Workshop Applet Demo.
Partition (Split)

• How to write code to accomplish partitioning?
• Think about it for a while.

1. Assume you are allowed additional memory space.
2. Assume you must perform in-place partition (i.e. no additional memory space allowed).
Partition (Split)

- If additional memory space is allowed, you can do this in two passes:
  1. Prepare a workspace array.
  2. Loop over the input array the first time, copy elements smaller than the pivot value one-by-one to the workspace array.
  3. Loop over the input array a second time, copy elements larger than the pivot value one-by-one to the workspace array, starting from where you left off in step 2.
Partition (Split)

• If **additional memory space is allowed**, you can do this in two passes.
• This seems quite simple.
• Now think about how do to eliminate the workspace array?
  – This will also eliminate the need of two passes.
Partition (Split)

• **In-space partition** (no additional memory space.)
• Let’s look at two approaches:
• Approach 1:
  Keep a `storeIndex` variable. This tracks where the next element, if smaller than pivot, should be stored.
Partition (Split)

• **In-space partition** (no additional memory space.)
• Let’s look at two approaches:
• Approach 1:

  Keep a `storeIndex` variable. This tracks where the next element, if smaller than pivot, should be stored.

  Scan the input array, if a smaller-than-pivot element is encountered, swap it with `array[storeIndex]`; otherwise keep going.
Partition (Split)

• Approach 1:

```java
int partition(double[] a, double pivot) {
    int storeIndex = 0;
    for(int i=0; i<a.length; i++) {
        if (a[i] <= pivot) {
            swap a[i] and a[storeIndex];
            storeIndex ++;
        }
    }
    return storeIndex;
}
```
Partition (Split)

• Approach 1:

  Example.

  Note the return value: it’s the index to the first element in the right sub-array.
  In other words, it’s the index of the first element that’s larger than the pivot value.
Partition (Split)

• Approach 2 (textbook):

Keep a `leftPtr` and a `RightPtr`. These track the current element being examined that should be stored on the left or on the right.

These two pointers move in alternating steps: `leftPtr` increments, and `RightPtr` decrements.

Left scan stops if it sees a larger element; right scan stops if it sees a smaller element; then swap the two.

This continues until left crosses right.
public int partitionIt(int left, int right, long pivot)
{
    int leftPtr = left - 1;
    int rightPtr = right + 1;
    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 partitionIt()
public int partitionIt(int left, int right, long pivot)
{
    int leftPtr = left - 1;
    int rightPtr = right + 1;
    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 partitionIt()
public int partitionIt(int left, int right, long pivot)
{
    int leftPtr = left - 1;
    int rightPtr = right + 1;
    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, // partition done
            break;
        else // not crossed, so
            swap(leftPtr, rightPtr); // swap elements
    } // end while(true)
    return leftPtr; // return partition
} // end partitionIt()
Partition (Split)

• Approach 2 (textbook):
  The return value is the same with before.

Comparing the two approaches:
• I personally like the first approach better, because it’s easier to understand and write.
• Therefore I won’t spent more time explaining the textbook approach. If you are interested, read pp. 330-331.
Partition (Split)

Some observations:
• The array is not necessarily partitioned in half.
  – This depends on the pivot value.
  – In some cases the split can be highly unbalanced.
• The array is by no means sorted.
  – But we are getting closer to that goal.
• What’s the cost of partition?
Partition (Split)

Some observations:

• The array is not necessarily partitioned in half.
  – This depends on the pivot value.
  – In some cases the split can be highly unbalanced.

• The array is by no means sorted.
  – But we are getting closer to that goal.

• What’s the cost of partition?

O(N)
Quick Sort

• Partition is the key step in quicksort.
• Once we have it, quicksort is pretty simple:
  – Partition (this splits the array into two: left and right)
  – Sort the left part, and sort the right part (how?)
Quick Sort

• Let’s take a look at the code first:

```java
public void recQuickSort(int left, int right)
{
    if(right-left <= 0)  // if size is 1,
        return;  // it's already sorted
    else  // size is 2 or larger
    {
        // partition range
        int partition = partitionIt(left, right);
        recQuickSort(left, partition-1);  // sort left side
        recQuickSort(partition+1, right);  // sort right side
    }
}
```
Quick Sort

• Let’s take a look at the code first:

```java
public void recQuickSort(int left, int right)
{
    if(right-left <= 0) // if size is 1,
        return; // it's already sorted
    else // size is 2 or larger
    {
        // partition range
        int partition = partitionIt(left, right);
        recQuickSort(left, partition-1); // sort left side
        recQuickSort(partition+1, right); // sort right side
    }
}
```
Quick Sort

Why does this work?

• After one partition, all elements in the left subarray are smaller than those on the right.
• The return value of partition marks the boundary.
Quick Sort

Why does this work?

• If we then sort the left, and sort the right (using recursion), then everything is sorted.
• The base case is when there is only one element to be sorted.
• What about the element at the partition boundary?
Quick Sort

Choosing a pivot value

• This influences the **efficiency** of quicksort a lot!
• Some ideas:
  1. The pivot should be **the value of an actual element**;
  2. You can pick an **arbitrary** element; ideally it should partition the array into two balanced subarrays.

Now let’s look at where the pivot element would appear after the partition?
Partition (Split)

• Approach 1:

```java
int partition(double[] a, int pivotIdx) {
    int storeIndex = 0;
    double pivot = a[pivotIdx];
    for(int i=0; i<a.length; i++) {
        if (a[i] <= pivot) {
            swap a[i] and a[storeIndex];
            storeIndex ++;
        }
    }
    return storeIndex;
}
```
Quick Sort

Choosing a pivot value

• It can appear anywhere in the left subarray.
• However, we would like it to appear right at the boundary of the left and right subarrays, so that it will be at its final sorted position.
• How do you achieve this?
Quick Sort

Choosing a pivot value

• It can appear anywhere in the left subarray.
• However, we would like it to appear right at the boundary of the left and right subarrays, so that it will be at its final sorted position.
• How do you achieve this?
  – Before partition, swap it to the right end of the array; afterwards, swap it back.
Quick Sort

```c
int partition(int left, int right, int pivotIdx) {
    double pivot = a[pivotIdx];
    swap a[pivotIdx] and a[right];  // swap pivot to end
    int storeIndex = left;
    for(int i=left; i<right; i++) {  // scan elements,
        if (a[i] <= pivot) {  // excluding the right end
            swap a[i] and a[storeIndex];
            storeIndex ++;
        }
    }
    swap a[right] and a[storeIndex];  // swap pivot to its final position
    return storeIndex;
}
```
Quick Sort

In Short:

• After each partition, the pivot element appears at its final sorted position; in addition, all elements smaller than it are on the left, and those larger on the right.
• The pivot element will **not be touched again** (since it’s already at its final resting place), and hence it’s not included in the recursion.

```c
int partition = partitionIt(left, right);
recQuickSort(left, partition-1);  // sort left side
recQuickSort(partition+1, right);  // sort right side
```
Quick Sort

Choosing a pivot value
• So which element do we pick as pivot element?
• Could be a random element, or could be the element at the right end.

\[
\text{pivotIdx} = \text{random(left, right)}; \\
\text{pivotIdx} = \text{right}; \quad \text{Problem?}
\]
Quick Sort

Choosing a pivot value

• So which element do we pick as pivot element?
• Could be a random element, or could be the element at the right end.

\[
pivotIdx = \text{random}(\text{left, right});
\]

\[
pivotIdx = \text{right};
\]

• Ideally it should result in equally sized left and right subarrays. But how to achieve this?
Quick Sort

Choosing a pivot value

- If we know the median, this will be easy.
- However, finding median itself is non-trivial.
- Can we perhaps approximate the median?
Median-of-Three Partitioning

• We don’t know the median, but let’s approximate it by the median of three elements in the array: the first, last, and the center.
Median-of-Three Partitioning

- This is **fast**, and has a good chance of giving us something **close to** the real median.

![Diagram](image-url)
Median-of-Three Partitioning

• The textbook discusses some additional benefits of median-of-three for their 
  partition method (approach 2).

• This doesn’t matter much if you use our partition method (approach 1).
Quick Sort

Some other considerations:

• Handling small subarrays.
• Use insertion sort for small subarrays.
• Quicksort followed by insertion sort.
• Removing recursion.
Quick Sort

Cost Analysis:

• Assume that on average, each partition splits the array in half.
• How many levels of subarray until we reach the base case?
• What’s the cost per level?
• Total cost:
Quick Sort

Cost Analysis:

• Assume that on average, each partition splits the array in half.
• How many levels of subarray until we reach the base case? \(\Theta(\log N)\)
• What’s the cost per level? \(\Theta(N)\)
• Total cost: \(\Theta(N \times \log N)\)
Quick Sort

Cost Analysis:

• Assume that on average, each partition splits the array in half.

• How many levels of subarray until we reach the base case? $\Rightarrow O(\log N)$

• What’s the cost per level? $\Rightarrow O(N)$

• Total cost: $O(N \times \log N)$

• How does this compare with mergesort?
Summary

• Partition is a key step in quick sort.
• Quicksort partition the input array to two sub-arrays, then sort each subarray recursively.
• It sorts in-place.
• It has $O(N \times \log N)$ cost.
• These features make it the most popular sorting algorithm.