Advanced Sorting

Quicksort - Mergesort
Divide and Conquer

Strategy:
1. Divide instance of problem into smaller instances
2. Solve smaller instances (recursively)
3. Combine smaller solutions to solve original instance
Divide-and-conquer technique

A problem of size $n$

Subproblem 1 of size $n/2$

- A solution to subproblem 1

Subproblem 2 of size $n/2$

- A solution to subproblem 2

A solution to the original problem
Quicksort

- Uses divide-and-conquer strategy.
- Partitions a list into smaller and smaller sublists about a value called the *pivot*.
- The algorithm locates the pivot in its final position in such a way that every value to the left of the pivot is $\leq$ pivot and every value to the right of the pivot is $\geq$ pivot.
- Unlike mergesort, quicksort is an *in-place* sorting algorithm.
Quicksort Algorithm

Given an array of \( n \) elements (e.g., integers):

- If array only contains one element, return
- Else
  - pick one element to use as \textit{pivot}.
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results
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
        double pivot = theArray[right];
        int partition = partitionIt(left, right);
        recQuickSort(left, partition-1); // sort left side
        recQuickSort(partition+1, right); // sort right side
    }
}
public int partitionIt(int left, int right, double pivot)
{
    int leftPtr = left-1; // left (after ++)  
    int rightPtr = right; // right-1 (after --)  
    while(true)
    {
        // find bigger item
        while(theArray[++leftPtr] < pivot)
        {
            // (nop)
        } // end while(true)
        // find smaller item
        // find smaller item
        while(rightPtr > 0 && 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)
    swap(leftPtr, right); // restore pivot  
    return leftPtr; // return pivot location
} // end partitionIt()

//----------------------------------------------------------------------------
• 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.
Choosing pivot element

The pivot value should be the key value of an actual data item; this item is called the *pivot*.

- You can pick a data item to be the pivot more or less at random. For simplicity, let's say we always pick the item on the right end of the subarray being partitioned.

- After the partition, if the pivot is inserted at the boundary between the left and right subarrays, it will be in its final sorted position.
Quicksort Analysis

• Assume that keys are random, uniformly distributed.

• What is best case running time?
  – Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  – Depth of recursion tree? $O(\log_2 n)$
  – Number of accesses in partition? $O(n)$
Quicksort Analysis

• Assume that keys are random, uniformly distributed.
• Best case running time: $O(n \log_2 n)$
• Worst case running time?
  – Recursion:
    1. Partition splits array in two sub-arrays:
       • one sub-array of size 0
       • the other sub-array of size n-1
    2. Quicksort each sub-array
  – Depth of recursion tree? $O(n)$
  – Number of accesses per partition? $O(n)$
Quicksort Analysis

• Assume that keys are random, uniformly distributed.
• Best case running time: $O(n \log_2 n)$
• Worst case running time: $O(n^2)$!!!
Analysis of quicksort—best case

• Suppose each partition operation divides the array almost exactly in half

• Then the depth of the recursion in $\log_2 n$
  – Because that’s how many times we can halve $n$

• However, there are many recursions!
  – How can we figure this out?
  – We note that
    • Each partition is linear over its subarray
    • All the partitions at one level cover the array
Partitioning at various levels
Best case II

- We cut the array size in half each time
- So the depth of the recursion is $\log_2 n$
- At each level of the recursion, all the partitions at that level do work that is linear in $n$
- $O(\log_2 n) \times O(n) = O(n \log_2 n)$
- Hence in the average case, quicksort has time complexity $O(n \log_2 n)$
- What about the worst case?
Worst case

• In the worst case, partitioning always divides the size $n$ array into these three parts:
  – A length one part, containing the pivot itself
  – A length zero part, and
  – A length $n-1$ part, containing everything else
• We don’t recur on the zero-length part
• Recurring on the length $n-1$ part requires (in the worst case) recurring to depth $n-1$
Worst case partitioning
Worst case for quicksort

• In the worst case, recursion may be $n$ levels deep (for an array of size $n$)
• But the partitioning work done at each level is still $n$
• $O(n) \times O(n) = O(n^2)$
• So worst case for Quicksort is $O(n^2)$
• When does this happen?
  – When the array is sorted to begin with!
Merge Sort

• Apply divide-and-conquer to sorting problem
• Problem: Given $n$ elements, sort elements into non-decreasing order
• The idea in the mergesort is to divide an array in half, sort each half, and then use the merge() method to merge the two halves into a single sorted array
• Divide-and-Conquer:
  – If $n=1$ terminate (every one-element list is already sorted)
  – If $n>1$, partition elements into two or more sub-collections; sort each; combine into a single sorted list
Partitioning

• Let’s try to achieve balanced partitioning
• A gets \(\frac{n}{2}\) elements, B gets rest half
• Sort A and B recursively
• Combine sorted A and B using a process called \textit{merge}, which combines two sorted lists into one
Merge Sort Algorithm

1. If a list has 1 element or 0 elements it is sorted
2. If a list has more than 2 split into into 2 separate lists
3. Perform this algorithm on each of those smaller lists
4. Take the 2 sorted lists and merge them together
Merge Sort

When implementing one temporary array is used instead of multiple temporary arrays.

Why?
Merge Sort code

* perform a merge sort on the data in c
* @param c c != null, all elements of c
* are the same data type
 */

public static void mergeSort(Comparable[] c) {
    Comparable[] temp = new Comparable[ c.length ];
    sort(c, temp, 0, c.length - 1);
}

private static void sort(Comparable[] list, Comparable[] temp, int low, int high)
{
    if( low < high) {
        int center = (low + high) / 2;
        sort(list, temp, low, center);
        sort(list, temp, center + 1, high);
        merge(list, temp, low, center + 1, high);
    }
}

private static void merge( Comparable[] list, Comparable[] temp,
        int leftPos, int rightPos, int rightEnd)
{
    int leftEnd = rightPos - 1;
    int tempPos = leftPos;
    int numElements = rightEnd - leftPos + 1;
    //main loop
    while( leftPos <= leftEnd && rightPos <= rightEnd){
        if( list[ leftPos ].compareTo(list[rightPos]) <= 0){
            temp[ tempPos ] = list[ leftPos ];
            leftPos++;
        }
        else{
            temp[ tempPos ] = list[ rightPos ];
            rightPos++;
        }
        tempPos++;
    }
    //copy rest of left half
    while( leftPos <= leftEnd){
        temp[ tempPos ] = list[ leftPos ];
        tempPos++;
        leftPos++;
    }
    //copy rest of right half
    while( rightPos <= rightEnd){
        temp[ tempPos ] = list[ rightPos ];
        tempPos++;
        rightPos++;
    }
    //Copy temp back into list
    for(int i = 0; i < numElements; i++, rightEnd--)
        list[ rightEnd ] = temp[ rightEnd ];
}
Example

- Partition into lists of size n/2
Example Cont’d

- Merge

![Diagram of merging elements][1]

[1]: #Example-Contd.png

```
[2, 3, 4, 5, 6, 7, 8, 10]

[3, 4, 6, 10]
```

```
[2, 5, 7, 8]
```

```
[2, 10]
```

```
[3, 6]
```

```
[2, 8]
```

```
[5, 7]
```
Recurrence for merge sort

• We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small $n$, but only when it has no effect on the asymptotic solution to the recurrence.

• Several ways to find a good upper bound on $T(n)$.

$T(n) = \Theta(1)$ if $n = 1$; $2T(n/2) + \Theta(n)$ if $n > 1$. 
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$h = \lg n$

$\Theta(1)$ #leaves = $n$ $\Theta(n)$

Total = $\Theta(n \lg n)$
• Mergesort requires temporary array for merging = $O(N)$ extra space

• Can we do in place sorting without extra space?

• Want a divide and conquer strategy that does not use the $O(N)$ extra space
Conclusions

• $\Theta(n \log n)$ grows more slowly than $\Theta(n^2)$.

• Therefore, merge sort asymptotically beats insertion sort in the worst case.

• In practice, merge sort beats insertion sort for $n > 30$ or so.
Stability

Stable sorting algorithms maintain relative order of records with equal keys,

- *i.e.*, for 2 records $R$ and $S$ with equal keys,
- where $R$ appears before $S$ in the input,
- $R$ will also appear before $S$ in the output.