**COMPSCI 330: Design and Analysis of Algorithms** 8/28/2014

Lecture #2

*Lecturer: Debmalya Panigrahi* *Scribe: Yilun Zhou*

# 1 Overview

This lecture presents two sorting algorithms, quicksort and mergesort, that are widely used and implemented. Their performances are compared. Using the two sorting algorithms, the concepts of worst-case analysis and average-case analysis are introduced. Lastly, the divide-and-conquer design paradigm is introduced along with two algorithms that demonstrated this technique.

# 2 Worst-Case Analysis vs. Average Case Analysis

Worst-case performance is the performance of a given algorithm on the worst input data, which will cause the algorithm to take the maximum amount of time to finish. Having a good worst-case performance is important during the design process. However, worst-case input instances can sometimes be rare. Thus, average case performance is often of interest for practical reasons. In some situations, an algorithm with a unideal worst-case performance bound may even be preferred over an algorithm with better worst-case performance due to the difference in average-case performance. Such a scenario is illustrated by quicksort and mergesort—quicksorts run in $O(n^2)$ time in the worse case but is much more widely used than mergesort with worst-case performance of $O(n \log n)$.

# 3 Divide-and-Conquer Design Paradigm

The divide-and-conquer design paradigm refers to the design method of breaking a problem down into one or more subproblems, solving each independently, and then combining the solution. A typical example is binary search, which finds an element in a sorted array. Because the array is sorted, each time only the median element is needed to be compared with the element to be searched for and then according to the result, either the smaller half or the greater half can be discarded. Thus, the problem size decreases by a half each time, leading to an $O(\log n)$-time algorithm.

# 4 Quicksort

## 4.1 Introduction

Quicksort is a sorting algorithm with worst-case running time of $O(n^2)$ and average-case time of $O(n \log n)$. It is used as many standard implementations of sorting function in many languages due to its excellent average case performance. The general procedure is shown below:
1. (Carefully or randomly) choose a pivot element.
2. Traverse the array to sort and partition the array into two groups, one group having elements smaller or equal to the pivot element and the other group having elements greater than the pivot.
3. Recursively sort the two groups.

## 4.2 Algorithm Pseudocode

The pseudocode for the algorithm is shown below:

```
1   // Qsort will sort the element A[i] through A[j]
2   Qsort(A,i,j)
3           s = PivotIndex(A,i,j)
4           // PivotIndex(A,i,j) will choose a pivot from A[i] through A[j]
5           // and return the index of the pivot
6           p = A[s]
7           for k = i to j
8                   if k==s
9                           skip
10                  else if A[k]<=p
11                          Add(A[k],B) // Add(e,A) will add e to array A
12                  else
13                          Add(A[k],C)
14          copy B into A[i] through A[j], followed by p and then C
15          Qsort(A,i,|B|+i-1)
16          Qsort(A,|B|+i+1,j)
```

The recursive step is specified on the last two lines of the procedure where quicksort recursively sorts the two smaller arrays. Note that there is no base case specified in the above pseduo-code it, but it would occur when the array contains only one element (at which point the algorithm should just return this singleton element).

## 4.3 Example

The following example illustrates the quicksort algorithm (everything other than pivots are in parentheses).
Original: (5 2 13 12 1 7 3 4) pivot:5
Iteration 1: (1 3 2 4) 5 (12 13 7) pivots: 1 and 12
Iteration 2: 1 (3 2 4) 5 (7) 12 (13) pivots: 3
*note that by choosing 12 as a pivot, the right array has been sorted
Iteration 3: 1 (2) 3 (4) 5 (7) 12 (13)
The sort is completed after three recursive calls.

## 4.4 Running Time

In the pseudocode there are several components.
1. The `for loop` in line 7-14 takes $O(n)$ time because it traverses the loop one time and copies the array once.
2. The recursive quicksort on line 15 and 16 takes variable amount of time because the length of the subarray is not constant over each input and over each pivot choice. Assume that there are $l$ elements to sort in line 15. Thus there are $n-l-1$ elements to sort in line 16. Therefore,

$$T(n) = T(l) + T(n-l-1) + O(n).$$

### 4.4.1 Worst-Case Analysis

The worst case happens if $l$ is equal to 0 or $n-1$. In this case,

$$\begin{aligned} T(n) &= T(0) + T(n-1) + O(n) \\ &= T(n-1) + O(n) \\ &= O(n^2). \end{aligned}$$

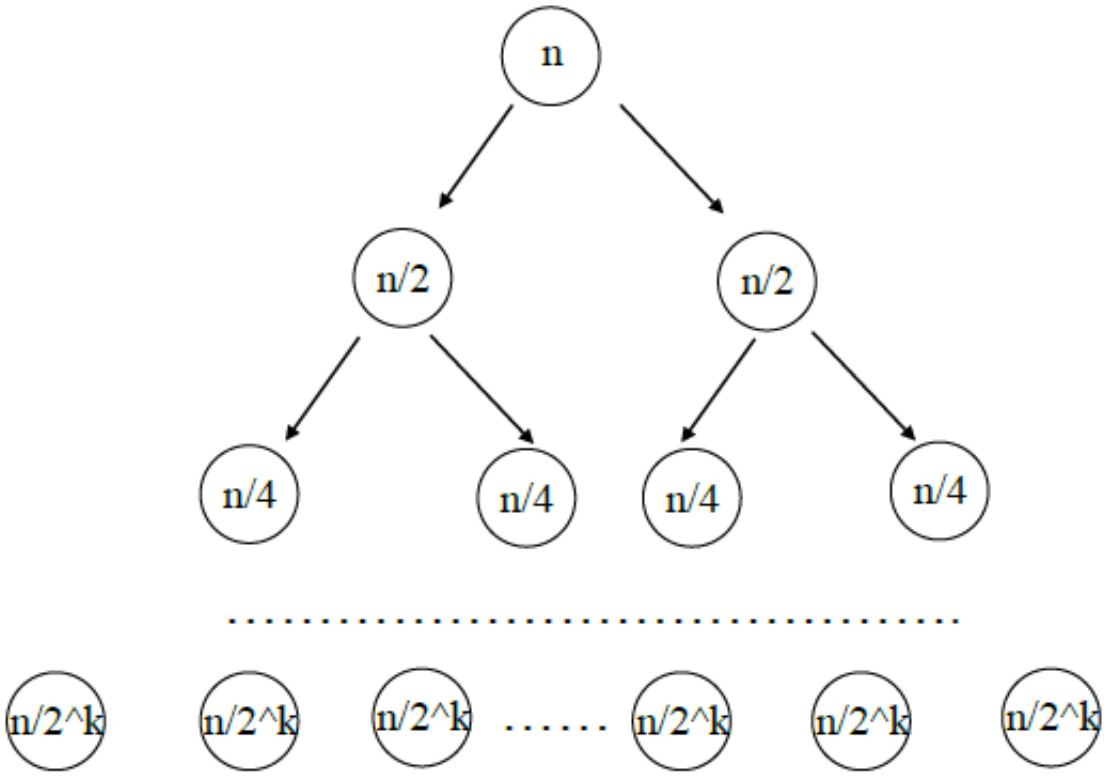
This case happens when the pivot element is the largest or the smallest in the array. In such a case, all other elements go to one side of the pivot, so the problem size only decreases by 1.

### 4.4.2 Average-Case Analysis

If the input data is random or the pivot is randomly chosen, over average we will select the median for our pivot at each step. Thus, $l = n/2$. Then the running time is:

$$\begin{aligned} T(n) &= T(n/2) + T(n/2-1) + O(n) \\ &\leq 2T(n/2) + O(n) \\ &= O(n \log n). \end{aligned}$$

The last step of the equation can be derived from the following diagram:



At $k$th level, there will be $2^k$ subproblems where each subproblem has size $n/2^k$. Because of the $O(n)$ term at the end of the recurrence relation, each of the $n/2^k$ problem will contribute $O(n/2^k)$ and $2^k$ problems will contribute $O(n)$ in total, which is the cost for one layer. Because the problem size goes down by a half at each layer, there are $\log n$ layers. Thus, the total size of the tree is $O(n \log n)$, which is the running time of the algorithm.

# 5 Mergesort

## 5.1 Introduction

Mergesort is a sorting algorithm with both worst-case and average-case performance of $O(n \log n)$. The algorithm is also a recursive divide-and-conquer algorithm, but instead of using a pivot to decide where to partition our subproblems, mergesort always divides the array equally. Mergesort then recursively sorts these two subarrays and then combines (or merges) them into one master sorted array.

## 5.2 Pseudocode

```
1   // Msort(A,i,j) will sort element A[i] through A[j] in A
2   Msort(A,i,j)
3           B=Msort(A,i,(i+j)/2)
4           C=Msort(A,(i+j)/2+1,j)
5           A=merge(B,C)
6
7   // Merge(B,C) assumes sorted arrays B and C
8   // and merges them into one big sorted array
9   Merge(B,C)
10          D=empty array
11          i=j=k=1
12          while(i<|B| and j<|C|)
13                  if(B[i]<C[j]) then D[k]=B[i], i++
14                  else then D[k]=C[j], j++
15                  k++
16          // At this point all elements of one array should be copied to D
17          Copy all elements of the other array into D
18          return D
```

## 5.3 Example

As an example, consider the following array:
7 1 5 2 0 -2
Mergesort will divide the array into two subarrays 7 1 5 and 2 0 -2. Then it will recursively sort these two arrays and give back 1 5 7 and -2 0 2. To merge these two arrays, we first compare 1 with -2. Because -2 is smaller, it is copied to the final sorted array. Then 1 is compared with 0 and 0 is copied to the final array. Then 1 and 2. This time 1 is smaller so 1 is copied to the final array. Next 5 is compared with 2 and 2 is copied to final array. Now the second array has been used up so 5 and 7 of the first array are copied to the

final array. The final array is
-2 0 1 2 5 7.

### 5.4 Performance

For mergesort, the worst-case and average-case performances are the same as the algorithm does not involve any random selection or subarrays of variable length. Line 3 and 4 of the pseudocode each takes $T(n/2)$ time and the `Merge`procedure takes $O(|B|+|C|)$ time, where $|B|$ and $|C|$ are the sizes of the array $B$ and $C$. The recurrence relation is thus:

$$\begin{aligned} T(n) &= 2T(n/2) + O(n) \\ &= O(n \log n). \end{aligned}$$

## 6 Reason for Choosing Quicksort over Mergesort

From an asymptotic (Big-Oh) viewpoint, mergesort is better than quicksort in the worst-case. Since they are equivalent in their average cases, there seems to be no reason for choosing quicksort. However, practically there are a few reasons for favoring quicksort over mergesort:

First, the worst-case for quicksort happens very infrequently. In fact one can argue that if we pick the pivot randomly, quicksort has an expected running time of $O(n \log n)$. It is also possible to pick the pivot in $O(n)$ time so that it is the exact median (an algorithm that finds the median in $O(n)$ time will be provided in the next lecture). Thus using this implementation, worst-case performance of quicksort can be improved to $O(n \log n)$.

Second, it is important to remember that asymptotic analysis hides constants. For example, two algorithms that run in $100n$ and $10n$ seconds for an input size of $n$ are asymptotically equivalent since both run in $O(n)$ time. However, in practice one algorithm will run 10 times faster than the other one. This difference exists in quicksort and mergesort, too. Although they are both $O(n \log n)$ algorithms, the constants hidden by mergesort tend to be larger. The exact coefficient depends on a lot of factors such as the exact implementation of the algorithm and the computer's processing speed; however, quicksort will typically outperform mergesort in pretty much any computing environment.

## 7 Summary

In this lecture, we used quicksort and mergesort to discuss worst-case performance, average-case performance, and divide-and-conquer design paradigm. The detailed implementation of the two algorithms were given and the running times for both worst-case and average-case were analyzed. The reasons behind quicksort's popularity was used to illustrate the importance of average-case performance, highlighting a situation when asymptotic analysis can obscure some important factors when translating formal analysis of algorithms into practical design.