Introduction to Randomized Algorithms

Ashraf M. Osman

Department of Computer Science and Electrical Engineering,
West Virginia University,
Morgantown, WV
osman@csee.wvu.edu

1 Introduction

There are two principal advantages to randomized algorithms. The first advantage is performance; randomized algorithms run faster than the best-known deterministic algorithms for many problems. The second advantage is that many randomized algorithms are simpler to describe and implement than deterministic algorithms of comparable performance.

There are general principles that lie at the heart of almost all randomized algorithms, despite the multitude of areas in which they find application. We focus in these notes on the classical adversary paradigm.

Foiling an adversary: *The classical adversary for a deterministic algorithm establishes a lower bound on the running time of the algorithm by constructing an input on which the algorithm fares poorly.*

A randomized algorithm can be viewed as a probability distribution on a set of deterministic algorithms. While the adversary may be able to construct an input that foils one (or a small fraction) of the deterministic algorithms in a set, it is difficult to devise a single input that is likely to defeat a randomly chosen algorithm. We will illustrate the adversary paradigm for sorting algorithms and show how randomized algorithms perform.

2 Sorting Algorithms

This problem is concerned with sorting an array of n elements into ascending order.

Sorting Problem
Input: An array S of n elements.
Output: a permutation of the input such that $S[i] \leq S[i+1], i = 1, ..., n-1$.

We will provide the analyses of sorting algorithms in the following sections.

2.1 Merge-Sort

The Merge-Sort sort algorithm splits the array to be sorted into two groups, recursively sorts each group, and merges them into a final, sorted sequence. Algorithm (2.1) illustrates the merge sort algorithm.

The procedure Merge-Sort(A, p, r) sorts the elements in the subarray $A[p..r]$. If $p \geq r$ the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index q that partitions $A[p..r]$ into two arrays: $A[p..q]$, and $A[q+1..r]$. The procedure Merge(A, p, q, r) uses two pointers to the subarrays $p1$ and $p2$ to merge the two subarrays into one sorted array. This procedure has to use another array $B[p..r]$ to be able to merge the subarrays as seen in the algorithm. Examining the merge sort algorithm, we conclude that the number of steps required to complete the sort is

$$T(n) = 2T(\frac{n}{2}) + n,$$

where $T(n)$ represents the time taken by this method to sort n elements. This recurrence can be solved to give:

$$T(n) = O(n \log n)$$