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:

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

```
Merge-Sort (A,p,r)
if $(p < r)$ then
    $q = (p + r)/2$
    Merge-Sort $(A, p, q)$
    Merge-Sort $(A, q + 1, r)$
    Merge $(A, p, q, r)$
end if
```

```
Merge$(A, p, q, r)$
copy $A$ to $B$
$p1 \Leftarrow p$, $p2 \Leftarrow q + 1$
while $(p1 \leq q$ or $p2 \leq r)$ do
    if $(B[p1] < B[p2])$ then
        $A[i] = b[p1]$
        $p1 \Leftarrow p1 + 1$
    else
        $A[i] = B[p2]$
        $p2 \Leftarrow p2 + 1$
    end if
    $i \Leftarrow i + 1$
end while
```

Algorithm 2.1: Merge-Sort

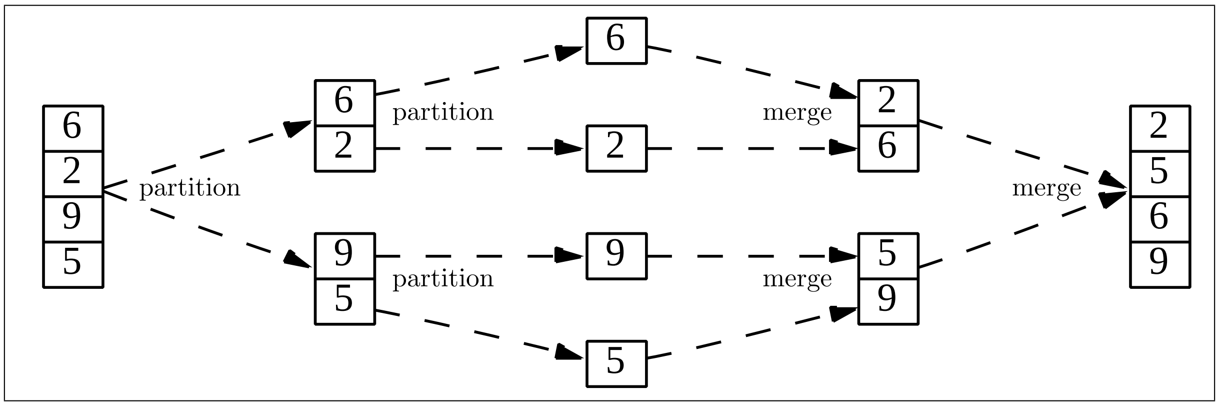

Figure 1: Merge Sort algorithm

One of the main disadvantages of this algorithm is that it requires additional storage, proportional to the size of the input array, as can be seen from Algorithm Merge-Sort.

2.2 Quick Sort

This is an in-place sort algorithm that uses the divide and conquer paradigm. It picks an element from the array (the pivot) and partitions the remaining elements about the pivot. The elements are partitioned into those greater than and those less than this pivot. The algorithm afterwards recursively sorts the partitions. Quick sort has a worst-case running time of $O(n^2)$, but typically it runs in time $O(n \log n)$. Figure 2 illustrates the quick sort algorithm.

If we are lucky enough to find an element that splits the array into two equal sets every time, the number of steps required to complete the sort is

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

where $T(n)$ represents the time taken by this method to sort n elements. The solution of this recurrence is:

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

However, the adversary can provide a reverse-sorted array as input, which is clearly the worst-case for Quick-Sort. In this case, the array will always be split into two *unbalanced* sets, a set of one element and a set consisting of the remainder

Input: A set of elements S
Output: The elements of S sorted in ascending order.

1. Pick the first element Y of S as a pivot
2. Partition the set S into two subsets, $S1$ and $S2$ such that every element of $S1$ is less than or equal to Y and every element in $S2$ is greater than Y.
3. Recursively sort $S1$ and $S2$ (using Quick-Sort)

Algorithm 2.2: Quick-Sort

{**26**	5	37	1	61	11	59	15	48	19}
{ **11**	5	19	1	15}	**26**	{59	61	48	37}
{1	5}	**11**	{19	15}	**26**	{59	61	48	37}
1	5	**11**	15	19	**26**	{48	37}	59	{61}
1	5	**11**	15	19	**26**	37	48	**59**	{61}
{1	5	**11**	15	19	**26**	48	37	**59**	61}

Figure 2: Quick sort algorithm

of the array. We find that the number of steps required to complete the sort is

$$T(n) - T(n-1) + n$$

Solving this recurrence we get:

$$T(n) - \frac{n(n-1)}{2}$$

Thus, the running time for the worst case is $O(n^2)$. In the next section we shall see how a randomized sort algorithm limits such an adversary.

2.3 Randomized Quick Sort

The randomized quick sort procedure is identical to the Quick-Sort procedure discussed above; the only difference is that the pivot is chosen randomly.

Input: A set of elements S
Output: The elements of S sorted in ascending order.

1. Choose an element Y uniformly at random from S: every element in S has an equal probability of being chosen.
2. By comparing each element of S with Y, determine the set $S1$ of elements smaller than Y and the set $S2$ of elements larger than Y.
3. Recursively sort $S1$ and $S2$. Output the sorted version of $S1$, followed by Y, and then the sorted version of $S2$.

Algorithm 2.3: Random Quick-Sort

As usual for sorting algorithms, we measure the running time of the random quick sort in terms of the number of comparisons it performs since this is the dominant cost in any reasonable implementation. For this randomized version, we are interested in the expected number of comparisons in an execution of this algorithm. Note that all the comparisons are performed in Step 2, in which we compare a randomly chosen partitioning element to the remaining elements. For $1 < i < n$, let $S(i)$ denote the element of rank i (the i^{th} smallest element) in the set S. Define the random variable X_{ij} to assume the value 1 if $S(i)$ and $S(j)$ are compared in an execution, and the value 0 otherwise. Thus, X_{ij} is a count of comparisons between $S(i)$ and $S(j)$. It should be noted that two different elements are only compared once, that means comparing $S(i)$ with $S(j)$ is the same event as comparing $S(j)$ with $S(i)$. Thus, $X_{ij} = X_{ji}$ and so the total number of comparisons is

$$\sum_{i=1}^{n} \sum_{j>1} X_{ij}$$

We are interested in the expected number of comparisons, which is clearly

$$\mathbf{E}\left[\sum_{i=1}^{n}\sum_{j>1} X_{ij}\right] = \sum_{i=1}^{n}\sum_{j>1}\mathbf{E}\left[X_{ij}\right]$$

Let p_{ij} denote the probability that $S(i)$ and $S(j)$ are compared in an execution. Since X_{ij} is a Bernoulli random variable that only assumes the values 0 and 1,

$$\mathbf{E}\left[X_{ij}\right] = p_{ij}\cdot 1 + (i - p_{ij})\cdot 0 = p_{ij}$$

Thus, the expected number of comparisons is

$$\sum_{i=1}^{n}\sum_{j>1} p_{ij}$$

To facilitate the determination of p_{ij}, one can view the execution of the algorithm as a binary tree T, each node of which is labeled with a distinct element of S. The root or the tree is labeled with the element Y chosen in Step 1, the left sub-tree of Y contains the elements in $S1$, and the right sub-tree of Y contains the elements in $S2$. The structures of the two sub-trees are determined recursively by the executions of the random quick sort algorithm on $S1$, and $S2$. The root Y is compared to the elements in the two sub-trees, but no comparison is performed between an element of the left sub-tree and an element of the right sub-tree. Thus, there is a comparison between $S(i)$ and $S(j)$ if and only if one of these elements is an ancestor of the other.

The output of the algorithm is nothing but the inorder traversal of this randomized binary search tree T. For the analysis let us consider the level-order traversal of T. the level-order is the permutation π of S obtained by visiting the nodes of T in increasing order of level numbers, and in left-to-right order within each level as illustrated in Figure 3.

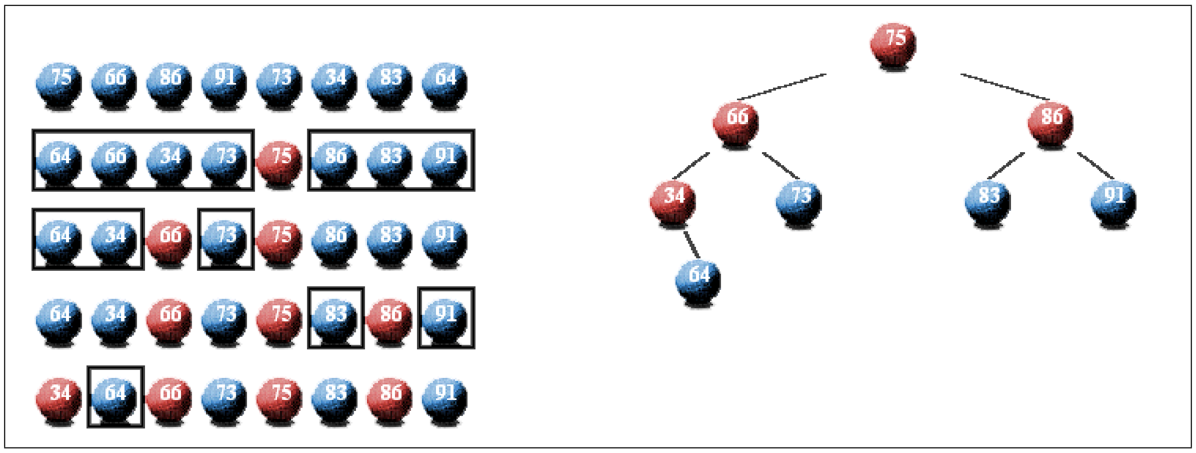

Figure 3: Random Quick Sort Algorithm

To compute p_{ij}, we make two observations.

1. $S(i)$ and $S(j)$ are compared if and only if one is an ancestor of the other. This happens only if $S(i)$ or $S(j)$ comes first in the level order among the elements that are larger than $S(i)$ and smaller than $S(j)$. $S(i)$ and $S(j)$ goes into different partitions and are never compared if and only if an element in between them is picked first by the random algorithm.

2. Since the partitioning element is chosen uniformly, the probability of picking $S(i)$ or $S(j)$ first in the set $\{S(i), S(i+1), ..., S(j)\}$ is $2/(j - i + 1)$.

Accordingly, The expected number of comparisons is

$$\sum_{i=1}^{n}\sum_{j>1}\frac{2}{j-i+1}$$

Let $k = j - i + 1$; thus the expected number of comparisons

$$\begin{aligned} &= \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k} \leqslant \sum_{i=1}^{n} \sum_{k}^{n} \frac{2}{k} \\ &\leqslant 2 \sum_{i=1}^{n} H_n \leqslant 2nH_n \\ &\leqslant n \ln n \end{aligned}$$

Note that H_n is the n^{th} Harmonic number, and is roughly $\ln n$.

Accordingly, the expected running time is $O(n \log n)$. The expected running time depends only on random choices made by the algorithm and not on any assumptions about the distribution of the input. Behavior can vary even on the same input.

In the next few sections we will continue to provide more examples on randomized algorithms.

3 Min-Cut Algorithm

Consider a graph-theoretic example. Let G be a connected, undirected multi-graph with n vertices. A multigraph may contain multiple edges between any pair of vertices. A cut in G is a set of edges whose removal results in G being broken into two or more components. A min-cut is a cut of minimum cardinality as seen in Figure 4.

Minimum Cut Problem
Input: Undirected multi-graph G with n vertices.
Output: The smallest set of edges in G, which partitions the vertices into two nonempty sets.

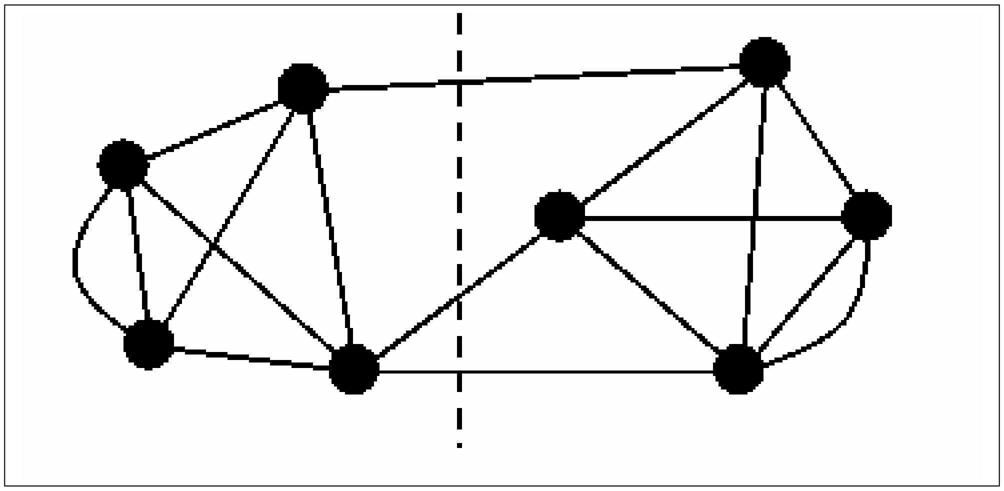

Figure 4: Min-Cut

Pick an edge uniformly at random and merge the two vertices at its end-points. If as a result there are several edges between some pairs of (newly formed) vertices, retain them all. Edges between vertices that are merged are removed, so that there are never any self-loops. We refer to this process of merging the two end-points of an edge into a single vertex as the contraction of that edge, as illustrated in figure 5.

With each contraction, the number of vertices of G decreases by one. The crucial observation is that an edge contraction does not reduce the min-cut size in G. This is because every cut in the graph at any intermediate stage is a cut in the original graph. The algorithm continues the contraction process until only two vertices remain, at this point, the set of edges between these two vertices is a cut in G and is output as a candidate min-cut. Now we proceed with the analysis.

Let k be the min-cut size. We fix our attention on a particular cut C with k-edges. Clearly, G has at least $\frac{kn}{2}$ edges; otherwise there would be a vertex of degree less than k, and its incident edges would be a min-cut of size less than k. we

Input: Undirected multi-graph G with n vertices.
Output: A cut in G of minimum cardinality.

1. Pick an edge at random and merge the two vertices at its end-points (contraction process).
2. Edges between vertices that are merged are removed, so that there are never any self-loops.
3. Continue to contract edges until only two vertices remain.
4. The set of edges between these two vertices is a cut in G.

Algorithm 3.1: Min Cut

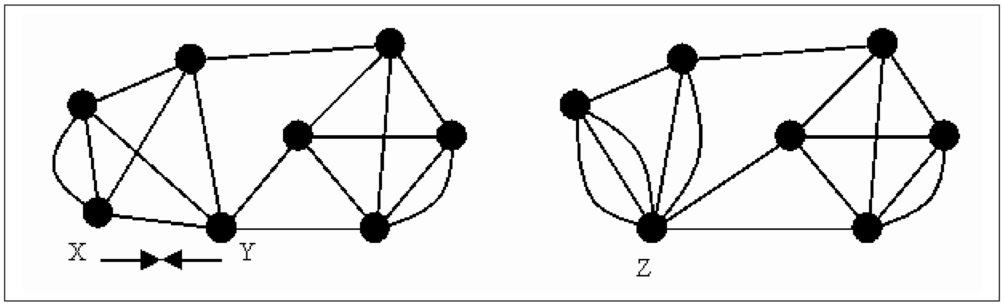

Figure 5: Edge contraction

will bound from below, the probability that no edge of C is ever contracted during the execution of the algorithm, so that the edges surviving till the end are exactly the edges in C.

Let E_i denote the event of not picking an edge of C at the i^{th} step, for $1 \leqslant i \leqslant n-2$. The probability that the edge is randomly chosen in the first step is in C is at most

$$\frac{k}{\frac{nk}{2}} = \frac{2}{n}$$

so that $\mathbf{Pr}[E_1] \geqslant 1 - \frac{2}{n}$. Assuming that E_1 occurs, during the second step there are at least $\frac{k(n-1)}{2}$ edges, so the probability of picking an edge in C is at most $\frac{2}{n-1}$, so that $\mathbf{Pr}[E_2|E_1] \geqslant 1 - \frac{2}{n-1}$. At the i^{th} step, the number or remaining vertices is $n-i+1$. The size of the min-cut is still at least k, so the graph has at least $\frac{k(n-i+1)}{2}$ edges remaining at this step. Thus, $\mathbf{Pr}\left[E_i | \cap_{j=1}^{i-2} E_j\right] \geqslant 1 - \frac{2}{n-i+1}$. Therefore, the probability that no edge of C is ever picked in the process is $\mathbf{Pr}\left[\cap_{i=1}^{n-2} E_i\right]$. The convenient method to compute the probability of their intersection is to use the following general formula (see appendix A).

$$\mathbf{Pr}\left[\bigcap_{i=1}^{k} E_i\right] = \prod_{i=1}^{k} \mathbf{Pr}\left[E_i | \bigcap_{j=1}^{i-1} E_j\right]$$

Accordingly,

$$\mathbf{Pr}\left[\bigcap_{i=1}^{n-2} E_i\right] \geqslant \prod_{i=1}^{n-2}\left(1 - \frac{2}{n-i+1}\right) = \frac{2}{n(n-1)}$$

Therefore, the probability of discovering a particular min-cut is larger than $\frac{2}{n^2}$. Thus, our algorithm may err in declaring the cut it outputs to be a min-cut. Suppose we repeat the above algorithm $\frac{n^2}{2}$ times, the probability that a min-cut is not found in any of the attempts is at most

$$(1 - \frac{2}{n^2})^{\frac{n^2}{2}} < \frac{1}{e}$$

By this process of repetition, we have managed to reduce the probability of failure from $(1 - \frac{2}{n^2})$ to a more respectable $\frac{1}{e}$. Further executions of the algorithm will make the failure probability arbitrarily small - the only consideration being that repetitions increase the running time. Note the extreme simplicity of the randomized algorithm. In contrast, most deterministic algorithms for this problem are based on network flow and are considerably more complicated.

4 Las Vegas and Monte Carlo

The randomized sorting algorithm and the min-cut algorithm illustrate two different types of randomized algorithms. The sorting algorithm always terminates with the correct solution, but its running time varies from an expected running time of $O(n \log n)$ to a worst-case running time of $O(n^2)$. We call such an algorithm a *Las Vegas algorithm*.

In contrast, the min-cut algorithm sometimes generates an incorrect solution. However, the failure probability can be made arbitrarily small by executing the algorithm several times with independent random choices each time. This is an example of a *Monte Carlo algorithm*. For decision problems (problems for which the answer to an instance is YES or NO), there are two kinds of Monte Carlo algorithms: those with *one-sided error* and those with *two-sided error*. A Monte Carlo algorithm has two-sided error if there is a non-zero probability of an error when it outputs either yes or no. If the probability of an error is zero for either a yes or a no output, then the Monte Carlo algorithm is said to have one-sided error.

A Las Vegas algorithm is a Monte Carlo algorithm with error probability 0. That is to say, when it terminates with a Yes or No result, that result is always correct. However, there is a small probability that the algorithm itself never terminates.

As for the efficiency, we say that a Las Vegas algorithm is efficient if on any input its expected running time is bounded by a polynomial function of the input size. Similarly, we say that a Monte Carlo algorithm is efficient if on any input its worst-case running time is bounded by a polynomial function of the input size.

Which one is better, Monte Carlo or Las Vegas? The answer depends on the problem itself. In some cases, a wrong answer is not a viable alternative. Because a Las Vegas algorithm is a Monte Carlo algorithm with error probability 0, we can say that, in most cases, a Las Vegas algorithm is better than a Monte Carlo algorithm. However, there is a cost/benefit relationship between these two alternatives and, in some cases, we do not want to pay for the extra time needed by the Las Vegas algorithm to generate an answer.

5 Binary Planar Partitions

Binary planar partitions have a number of applications in computer graphics; one of them is the problem of hidden line elimination. For further details about the hidden line elimination, refer to [1]. One approach to this problem is to use a binary partition tree as shown in Figure 6.

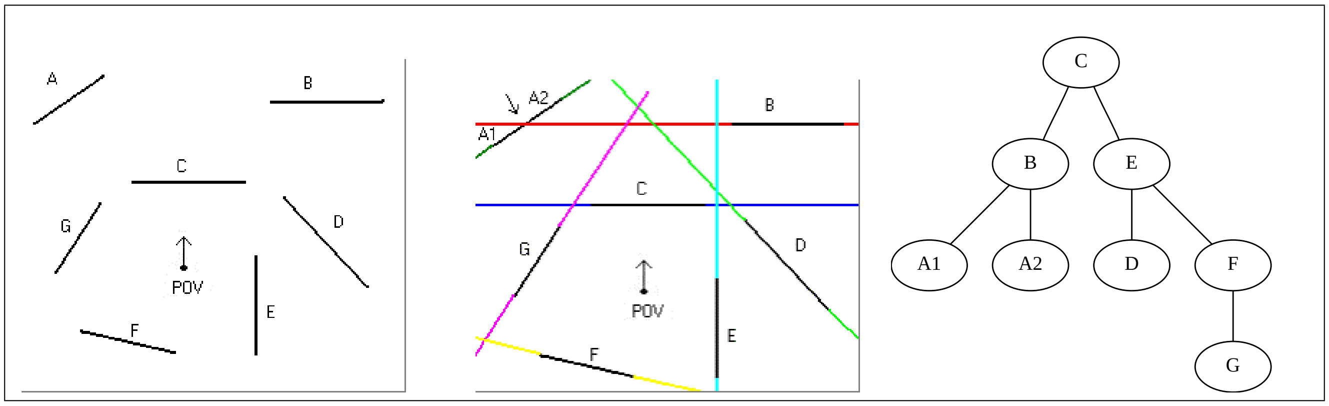

Figure 6: An example of a binary planar partition

A binary planar partition consists of a binary tree. Every internal node of the tree has two children. Associated with each node v of the tree is a region $r(v)$ of the plane. Associated with each internal node v of the tree is a line $l(v)$ that intersects $r(v)$. The region corresponding to the root is the entire plane. The region $r(v)$ is partitioned by $l(v)$ into two regions $r_1(v)$ and $r_2(v)$, which are the regions associated with the two children of v. Thus, any region r of the partition is bounded by the partition lines on the path from the root to the node corresponding to r in the tree. We now define the problem formally as follows.

Binary Planar Partitions Problem
Input: A set $S = \{s_1, s_2, ..., s_n\}$ of non-intersecting line segments in the plane.
Output: A binary planar partition such that every region in the partition contains at most one line segment (or a portion of one line segment).

The storage requirement of a particular binary planar partition is the number of nodes in the associated tree. Clearly we want this tree to have as few nodes as possible. It is not hard to see that the best case is the one in which we are able to partition the n line segments without generating other new line segments (which are created when the original line segments are broken). In this case, we will need $(n-1)$ lines to partition the n line segments. Therefore, the total number of nodes in the binary planar partition tree (of n leaves and $(n-1)$ interior nodes) is $(2n-1)$. However, there is no guarantee that there is always a partition of size $O(n)$. We will prove using a randomized algorithm that there always exists a binary planar partition of size $O(n \log n)$.

For a line segment s, let $l(s)$ denote the line obtained by extending (if necessary) s on both sides to infinity. For the set $S = \{s_1, s_2, ..., s_n\}$ of line segments, a simple and natural class of partitions is the set of autopartitions, which are formed by only using lines from the set $\{l(s_1), l(s_2), ..., l(s_n)\}$ in constructing the partition. The algorithm is defined as follows.

Input: A set $S = \{s_1, s_2, ..., s_n\}$ of non-intersecting line segments.
Output: A binary autopartition P_π of S.

1. Pick a permutation of π of $\{1, 2, ..., n\}$ uniformly at random from $n!$ possible permutations.
2. While a region contains more than one segment, cut it with $l(s_i)$ where i is the first in the ordering π such that s_i cuts that region.

Algorithm 5.1: RandAuto

Theorem: 5.1 *The expected size of the autopartition produced by RandAuto is $O(n \log n)$.*

Proof: *For line segments u and v, define $index(u,v)$ to be i if $l(u)$ intersects $i-1$ other segments before hitting v, and $index(u,v) = \infty$ if $l(u)$ does not hit v. since a segment u can be extended in two directions, it is possible that $index(u,v) = index(u,w)$ for two different lines u and w.*

Let us denote by $u \dashv v$ the event that $l(u)$ cuts v in the constructed partition. Let $index(u,v) = i$, and let $u_1, u_2, ..., u_{i-1}$ be the segments that $l(u)$ intersects before hitting v. the event $u \dashv v$ happened only if u occurs before any of $\{u_1, u_2, ..., u_{i-1}, v\}$ in the randomly chosen permutation π. The probability that this happens is $\frac{1}{i+1}$.

Let $C_{u,v}$ be an indicator variable which is 1 if u cuts v and 0 otherwise; clearly,

$$\mathbf{E}[C_{u,v}] = \mathbf{Pr}[u \dashv v] = \frac{1}{index(u,v)+1}$$

The size of P_π equals n plus the number of intersections due to cuts. Thus, its expectation is $n + \mathbf{E}\left[\sum_u \sum_v C_{u,v}\right]$ and by linearity of expectation this equals

$$n + \sum_u \sum_{v \neq u} \mathbf{Pr}[u \dashv v] = n + \sum_u \sum_{v \neq u} \frac{1}{index(u,v)+1}$$

For any line segment u and any finite positive integer i, there are at most two line segments v and w such that $index(u,v)$ and $index(u,w)$ equals i. This is because the extension of the segment u along either of the two possible directions will meet any other line segment at most once. Thus, in each of the two directions, there is a total ordering on the points of intersection with other segments and the index values increase monotonically. This implies that

$$\sum_u \sum_{v \neq u} \frac{1}{index(u,v)+1} \leqslant \sum_{i=1}^{n-1} \frac{2}{i+1}$$

Thus, the expected size of P_π is bounded by

$$n + 2\sum_u \sum_{v \neq u} \frac{1}{i+1} \leqslant n + 2nH_n$$

which is $O(n \log n)$. □

Since the expected size of the binary planar partition constructed by the algorithm is $O(n \log n)$ on any input, there must exist a binary autopartition of size $O(n \log n)$ for every input. This follows from the simple fact that any random variable assumes at least one value that is no greater than its expectation (and indeed, one that is no less then its expectation). Thus we have used a probabilistic argument to assert that the binary autopartition of size $O(n \log n)$ exists with absolute certainty rather than with some probability.

6 Computation Model and Complexity Classes

In this section we discuss Turing machine as a model of computation, and follow this with a review of complexity classes.

6.1 Turing Machines

We will use the Turing machine model to discuss complexity-theory issues.

Definition 6.1 **Deterministic Turing Machine:** *A deterministic Turing machine is a quadruple $M = (S, \Sigma, \delta, s)$. Here S is a finite set of states, of which $s \in S$ is the machine's initial state. The machine uses a finite set of symbols, denoted Σ; this set includes special symbols $BLANK$ and $FIRST$. (Σ^* represents all finite sequences of strings from Σ.) The function δ is the transition function of the Turing machine, mapping $S \times \Sigma$ to $(S \cup \{HALT, YES, NO\}) \times \Sigma \times \{\leftarrow, \rightarrow, STAY\}$. The machine has three halting states $HALT$ (the halting state), YES (the accepting state), and NO (the rejecting state) (these are states, but formally not in S).*

The Turing machine may be described as consisting of a finite-state control, a read-write head, and an unbounded sequential tape. Depending on the current state and symbol read on the tape, the machine can change its state and move the head to the left or right. A detailed elaboration on Turing Machines can be found in [1].

Definition 6.2 **Probabilistic Turing Machine:** *A probabilistic Turing machine is a Turing machine augmented with the ability to generate an unbiased coin flip in one step. At each step the machine makes a move by flipping a coin. On any input x, a probabilistic Turing machine accepts x with some probability, and we study this probability.*

6.2 Complexity Classes

We now define some basic complexity. For these definitions, the underlying model of computation is assumed to be the Turing machine.

A complexity class is a collection of languages all of whose recognition problems can be solved under prescribed bounds on the computational resources. We are primarily interested in various forms of efficient algorithms, where efficient is defined as being polynomial time. The following definitions list some interesting complexity classes.

Definition 6.3 **P Class:** *The class P consists of all languages L that have a polynomial time algorithm A such that for any input $x \in \Sigma^*$*

- $x \in L \Rightarrow A(x)$ *accepts.*
- $x \notin L \Rightarrow A(x)$ *rejects.*

Definition 6.4 **NP Class:** *The class NP consists of all languages L that have a polynomial time algorithm A such that for any input $x \in \Sigma^*$*

- $x \in L \Rightarrow \exists y \in \Sigma^*$, $A(x, y)$ *accepts, where* $|y|$ *is bounded by a polynomial in* $|x|$.
- $x \notin L \Rightarrow \forall y \in \Sigma^*$, $A(x, y)$ *rejects.*

A useful view of P and NP is the following. The class P consists of all languages L such that for any $x \in L$, a proof of membership $x \in L$ can be found and verified efficiently. On the other hand, NP consists of all languages L such that for any $x \in L$, a proof of membership $x \in L$ can be verified efficiently.

For any complexity class C, we define the complementary class co-C as the set of languages whose complement is in the class C. That is,

$$\text{co-C} = \{L | \bar{L} \in \text{C}\}$$

It is obvious that P=co-P and P $\subseteq$ NP $\cap$ co-NP. We generalize these classes to allow for randomized algorithms. The basic idea is to replace the existential and universal quantifiers in the definition of NP by probabilistic requirements.

Definition 6.5 **RP Class:** *The class RP (for Randomized Polynomial Time) consists of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input $x \in \Sigma^*$*

- $x \in L \Rightarrow \mathbf{Pr}[A(x) accepts] \geqslant \frac{1}{2}$.
- $x \notin L \Rightarrow \mathbf{Pr}[A(x) accepts] = 0$.

The choice of the bound error probability is arbitrary. Observe that an RP algorithm is a Monte Carlo algorithm that can err only when $x \in L$. This is referred to as one-sided error. The class co-RP consists of languages that have polynomial-time randomized algorithms erring only in case when $x \notin L$. A problem belonging to both RP and co-RP can be solved by a randomized algorithm with zero-sided error, i.e., a Las Vegas algorithm.

Definition 6.6 **ZPP Class:** *The class ZPP (for Zero-error Probabilistic Polynomial Time) is a class of languages that have Las Vegas algorithms running in expected polynomial time.*

Definition 6.7 **PP Class:** *The class PP (for Probabilistic Polynomial Time) consists of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input* $x \in \Sigma^*$

- $x \in L \Rightarrow \mathbf{Pr}\left[A(x) accepts\right] > \frac{1}{2}$.
- $x \notin L \Rightarrow \mathbf{Pr}\left[A(x) accepts\right] < \frac{1}{2}$.

Unfortunately, the definition of class PP is rather weak; because we have no bound on how far from $\frac{1}{2}$ the probabilities are. A more useful class of two-sided error randomized algorithms corresponds to the following complexity class.

Definition 6.8 **BPP Class:** *The class BPP (for Bounded-error Probabilistic Polynomial time) consists of all languages L that have a randomized algorithm A running in worst-case polynomial time such that for any input* $x \in \Sigma^*$

- $x \in L \Rightarrow \mathbf{Pr}\left[A(x) accepts\right] \geqslant \frac{3}{4}$.
- $x \notin L \Rightarrow \mathbf{Pr}\left[A(x) accepts\right] \leqslant \frac{1}{4}$.

The relationship between complexity classes is illustrated in Figure 7.

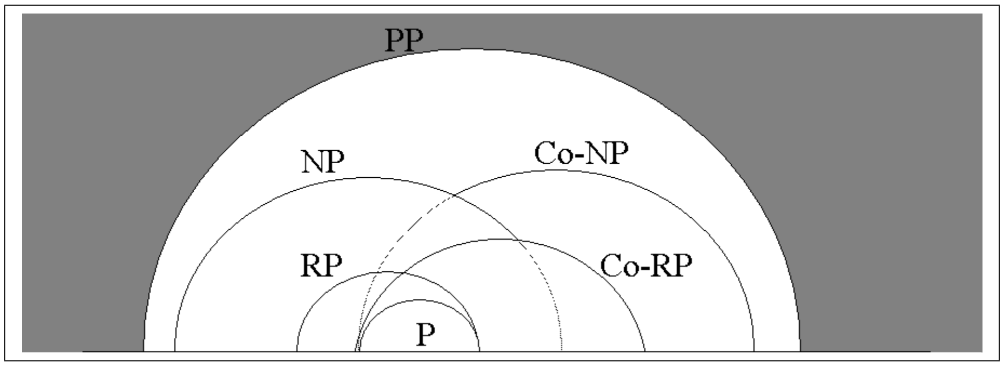

Figure 7: Complexity Classes Relationship

Exercise
Prove the following:

1. RP $\subseteq$ BPP $\subseteq$ PP
2. PP = co-PP
3. BPP = co-BPP
4. NP $\subseteq$ PP
5. ZPP = RP $\cap$ co-RP

Appendix A: Show that

$$\mathbf{Pr}\left[\bigcap_{i=1}^{k} E_i\right] = \prod_{i=1}^{k} \mathbf{Pr}\left[E_i | \bigcap_{j=1}^{i-1} E_j\right]$$

<u>Proof</u>: *Given two events E_1 and E_2, the probability of E_1 given than E_2 has occured is given by the following:*

$$\mathbf{Pr}\left[E_2|E_1\right] = \frac{\mathbf{Pr}\left[E_2E_1\right]}{\mathbf{Pr}\left[E_1\right]}$$

Acordingly,

$$\mathbf{Pr}\left[E_2E_1\right] = \mathbf{Pr}\left[E_2|E_1\right] \times \mathbf{Pr}\left[E_1\right]$$

Using this equation we will decompose $\mathbf{Pr}\left[\cap_{i=1}^{k} E_i\right]$ as follows:

$$\mathbf{Pr}\left[\bigcap_{i=1}^{k} E_i\right] = \mathbf{Pr}\left[E_kE_{k-1}...E_3E_2E_1\right] = \mathbf{Pr}\left[E_k|E_{k-1}...E_1\right] \times \mathbf{Pr}\left[E_{k-1}...E_1\right]$$

Similarly,

$$\mathbf{Pr}\left[E_{k-1}...E_3E_2E_1\right] = \mathbf{Pr}\left[E_{k-1}|E_{k-2}...E_1\right] \times \mathbf{Pr}\left[E_{k-2}...E_1\right]$$

Therefore,

$$\mathbf{Pr}\left[E_k...E_3E_2E_1\right] = \mathbf{Pr}\left[E_k|E_{k-1}...E_1\right] \times ... \times \mathbf{Pr}\left[E_3|E_2E_1\right] \times \mathbf{Pr}\left[E_2|E_1\right] \times \mathbf{Pr}\left[E_1\right]$$

In other words,

$$\mathbf{Pr}\left[\bigcap_{i=1}^{k} E_i\right] = \prod_{i=1}^{k} \mathbf{Pr}\left[E_i | \bigcap_{j=1}^{i-1} E_j\right]$$

□

References

[1] Rajeev Motwami and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, Cambridge, England, 1995.

[2] C.E. Leiserson T.H. Cormen and R.L. Rivest. *Introduction to Algorithms.* MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

Mathematics and Computer Science Division,
http://www-fp.mcs.anl.gov/division/welcome/default.asp

Dictionary of Algorithms, Data Structures, and Problems,
http://www.nist.gov/dads/terms.html

HPCC WWW Server,
http://hpcc.engin.umich.edu/CFD/users/charlton/Thesis/html/node28.html

Random binary search trees and Quicksort,
http://www.cs.mcgill.ca/ cs251/OldCourses/1997/topic10/

Symbolcraft : Graphics Notes,
http://symbolcraft.com/pjl/graphics/bsp/