**Lecture Notes CS:5350** Randomized Algorithms
**Lecture 15**: Oct 15, 2019
**Scribe**: Cory Kromer-Edwards

## 1 Randomized Algorithms

A randomized algorithm is one that has access to a sequence of random bits, which it uses in making decisions.

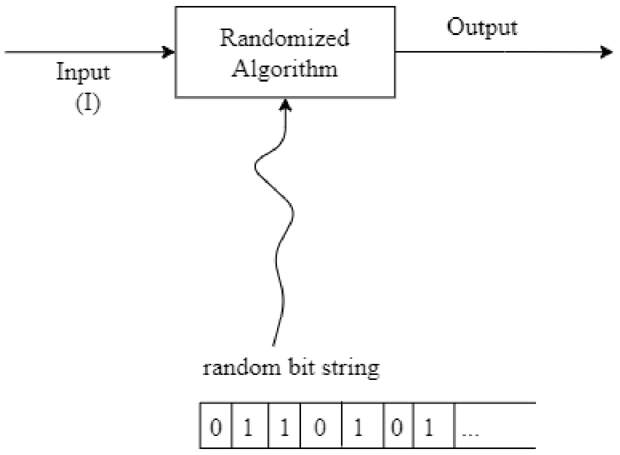

Figure 1: An example of a black box randomized algorithm

**Definition: ($T_A(I)$)** A randomized algorithm $A$ can behave differently on different runs of the same input $I$. Therefore, running time of algorithm $A$ on input $I$ is a random variable given as $T_A(I)$.

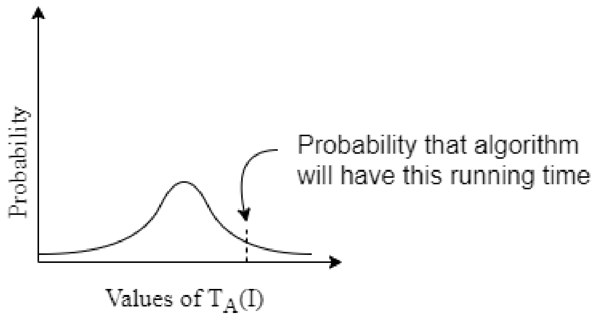

Figure 2: Visual of the probability distribution $T_A(I)$

So the running time of algorithm $A$ on an input of size n can be defined in two ways:

(i)
$$T(n) = \max_{I:|I|=n} \mathbb{E}[T_A(I)] \tag{1}$$

where $\mathbb{E}$ is the expected value over the algorithm's randomness. Note that this is similar to the usual *worst case* definition of the running time of a deterministic algorithm, except that for randomized algorithms we take the worst case over the expectation of the running time on input $I$.

(ii) Let $t_A(I)$ be a value such that

$$Prob[T_A(I) \geq t_A(I)] \leq \frac{1}{n}\cdot \tag{2}$$

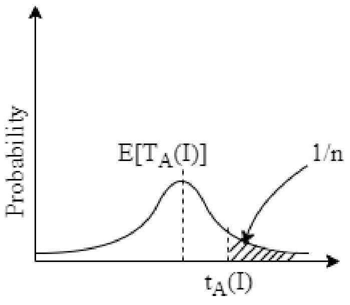

Figure 3: The probability distribution of $T_A(I)$, visually showing Inequality 2

The probability that the random variable $T_A(I)$ exceeds $t_A(I)$ is at most $\frac{1}{n}$. The value $t_A(I)$ from Inequality 2 can then be used in the following alternate definition of the running time of a randomized algorithm.

$$T(N) = \max_{I:|I|=n} t_A(I) \tag{3}$$

Just like the running time is a random variable, the output of $A$ on input $I$ is also a random variable, denoted $O_A(I)$. We allow randomized algorithms to make errors provided we can control the error probability. See Figure 4. We try to design algorithms for which there is an upper bound on this error probability. In particular, for some $\epsilon > 0$, we might be interested in designing algorithms such that Equation 4 holds.

$$Prob[O_A(I) \text{ is wrong}] \leq \epsilon\cdot \tag{4}$$

**Definition: Las Vegas** algorithms are randomized algorithms that have no error ($\epsilon = 0$). These algorithms run until a correct answer is produced.
**Definition: Monte Carlo** algorithms are randomized algorithms that can have error ($\epsilon > 0$). Some examples of Monte Carlo algorithms are the Miller-Robin algorithm for Primality Testing, finger printing algorithm for Verifying Matrix Multiplication, and Karger's MinCut algorithm.

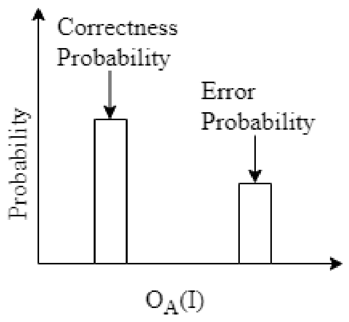

Figure 4: Showing correctness probability compared to error probability

## 2 Balanced Partition

Balanced Partition is a "toy" example problem for which we will show both a Monte Carlo algorithm and a Las Vegas algorithm.
**BalancedPartition (BP)**
**Input:** A list L of distinct integers
**Output:** A partition of L into sublists $L_1$ and $L_2$ such that the following two conditions apply:

(i) $\frac{|L|}{3} \leq |L_1| \leq \frac{2}{3}|L|$

(ii) for all $x \in L_1$ and $y \in L_2$, $x < y$

```
Result: Lists $L_1$ and $L_2$
Function BP(L: list) : (list, list) is
    Pick an index $i \in \{1, 2, \ldots, |L|\}$ uniformly at random
    $L_1 \longleftarrow \emptyset$, $L_2 \longleftarrow \emptyset$
    for each $l \in L$ do
        if $l \leq L[i]$ then
            append $l$ to $L_1$
        else
            append $l$ to $L_2$
        end
    end
    output ($L_1$, $L_2$)
end
```

**Algorithm 1:** Original Balanced Partition function

**Notes:**

- This algorithm has a deterministic running time of $O(n)$, assuming that a random index (in step 2) can be picked in $O(1)$ time.
- It is a Monte Carlo algorithm with $Prob[O_A(I) \text{ is wrong}] \leq \frac{2}{3}$. This can be seen better in Figure 5 where the two white parts represent choiced of indices that lead to wrong output.

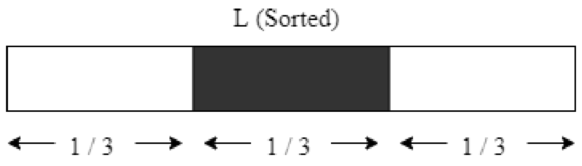

Figure 5: Showing input L sorted with correct answer (gray) and wrong answer (white)

There are two ways to extend BP to amplify the correctness probability. These are given in Algorithm 3 and Algorithm 2.

```
Result: Lists $L_1$ and $L_2$
Function BP-LasVegas(L: list) : (list, list) is
    repeat
        $(L_1, L_2) \longleftarrow BP(L)$
    until $\frac{|L|}{3} \leq |L_1| \leq \frac{2}{3}|L|$ ;
    output $(L_1, L_2)$
end
```

**Algorithm 2:** Las Vegas version of Balanced Partition

```
Result: Lists $L_1$ and $L_2$
Function BP-MonteCarlo(L: list) : (list, list) is
    for $i \longleftarrow 1$ to $k$ do
        $(L_1, L_2) \longleftarrow BP(L)$
        if $\frac{|L|}{3} \leq |L_1| \leq \frac{2}{3}|L|$ then
            output $(L_1, L_2)$
        end
    end
    output error
end
```

**Algorithm 3:** Monte Carlo version of Balanced Partition

## 2.1 BP-LasVegas

Note that there is no error in this algorithm ($\epsilon = 0$). Let $I_{BP-LV}(L)$ = number of iterations performed by BP-LV. The distribution of $I_{BP-LV}(L)$ can be seen in Figure 6.
It can be seen in Figure 6 that $I_{BP-LV}(L)$ is geometrically distributed with probability $\frac{1}{3}$.

**Corollary.** $\mathbb{E}[I_{BP-LV}(L)] = 3$

**Corollary.** $\mathbb{E}[I_{BP-LV}(L)] = O(|L|)$

**Corollary.** $\mathbb{E}[T_{BP-LV}(n)] = O(n)$

$$I_{BP\text{-}LV} \begin{cases} 1 \text{ with probability } 1/3 \\ 2 \text{ with probability } (2/3) * (1/3) \\ 3 \text{ with probability } (2/3)^2 * (1/3) \\ 4 \text{ with probability } (2/3)^3 * (1/3) \\ \ldots \\ i \text{ with probability } (2/3)^{i-1} * (1/3) \end{cases}$$

Figure 6: The distribution of $I_{BP-LV}(L)$

### 2.2 BP-MonteCarlo

The running time of the BP-MonteCarlo algorithm is not a random variable and is $O(kn)$. This can be easily seen as the running time of BP is $O(n)$ and $k$ times in worst case.
The error probability of the BP-MonteCarlo algorithm is $(\frac{2}{3})^k$. This is because the error probability per run of BP is $\frac{2}{3}$, and, if the algorithm is run k times, that means that it did not have a correct answer on run 1 and run 2 and run 3 and so on and using in depence this gives an error probability of $\frac{2}{3} \cdot \frac{2}{3} \cdot \ldots \cdot \frac{2}{3}$ ($k$ times).

For choosing an appropriate $k$ in the BP-MonteCarlo algorithm, suppose we want

$$Prob[O_{BP-MC}(L) = \text{ wrong}] \leq \epsilon$$

for a given, $\epsilon > 0$. To ensure this, we pick $k$ such that:

$$\left(\frac{2}{3}\right)^k \leq \epsilon \Rightarrow \left(\frac{3}{2}\right)^k \geq \frac{1}{\epsilon} \Rightarrow k \geq \log_{\frac{3}{2}}(\frac{1}{\epsilon})$$

Therefore, we want $k = \lceil \log_{\frac{3}{2}}(\frac{1}{\epsilon}) \rceil$.

## 3 The MinCut Problem

When we studied the MaxFlow problem, we also studied the dual $s - t$ MinCut problem. Here we consider the "global" MinCut problem. Here we are not given a specific $s$ or $t$; instead we are asked to find minimum size cut over all possible $(s, t)$ pairs.
**MinCut**
**Input:** A connected graph $G = (V, E)$
**Output:** A set $E' \subseteq E$ of smallest size such that $G \setminus E'$ is a disconnected graph

### 3.1 Karger's MinCut algorithm

Karger's MinCut algorithm performs operations called *contracts* many times. The graphs in this algorithm may have parallel edges.
**Definition:** A **contract** is an operation represented as given as *contract*$(G, \{u, v\})$ where $\{u, v\}$ is an edge in G. Each contract removes two nodes, $u$ and $v$, from $G$ along with the edge(s) that connect $u$ to $v$. The contract then adds in one single node $uv$ such that $uv$ keeps all outgoing edges from both $u$ and $v$ to other nodes in the graph.
**Example:** An example graph can be seen in Figure 8.

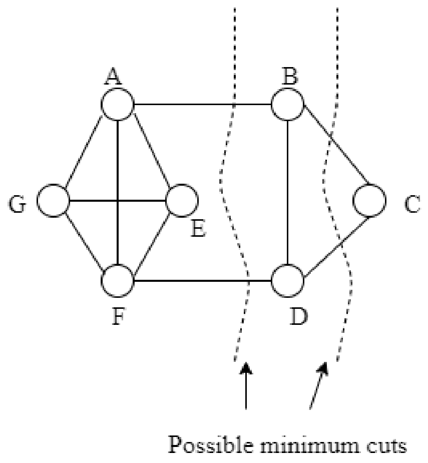

Figure 7: An example of possible minimum cuts for this graph

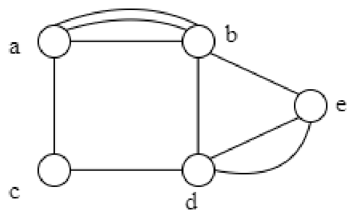

Figure 8: An example graph that could be input to the contract operation

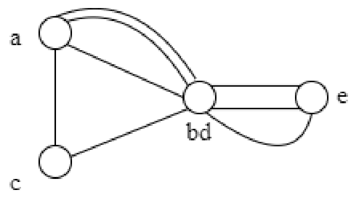

Figure 9: Graph after we contract nodes *b* and *d* into node *bd*

In Figure 8, if we contracted nodes *b* and *d* and the new node that would be created would be named *bd*, we would get Figure 9 as a result.

Karger's MinCut algorithm will be discussed further in the next lecture.