Randomized Algorithms

CS31005: Algorithms-II

Autumn 2022

IIT Kharagpur

Randomized Algorithms

- Algorithms that you have seen so far are all deterministic
  - Always follows the same execution path for the same input
- Randomized algorithms are non-deterministic
  - Makes some random choices in some steps of the algorithms
  - Output and/or running time of the algorithm may depend on the random choices made
  - If you run the algorithm more than once on the same input data, the results may differ depending on the random choice

A Simple Example: Choosing a Large Number

- Given n numbers, find a number that is ≥ median
- Simple deterministic algorithm takes O(n) time
- A simple randomized algorithm:
  - Choose k numbers randomly, $k < n$, and output the maximum
- Runs faster
- But this may not give the correct result always
  - But the probability that an incorrect result is output is less than $\frac{1}{2^k}$
  - Somewhat trivial example, but highlights the essence of a large class of randomized algorithms

Types of Randomized Algorithms

- Monte Carlo Algorithms
  - Randomized algorithms that always has the same time complexity, but may sometimes produce incorrect outputs depending on random choices made
    - Time complexity is deterministic, correctness is probabilistic
- Las Vegas Algorithm
  - Randomized algorithms that never produce incorrect output, but may have different time complexity depending on random choices made (including sometimes not terminating at all)
  - Time complexity is probabilistic, correctness is deterministic

Monte Carlo Algorithms

Monte Carlo Algorithms

- May sometimes produce incorrect output
- But will always terminate with same time complexity (usually polynomial to be of interest)
- Let p = probability that it will produce incorrect output
- Idea:
    - Design the algorithm such that $p < 1$ (typically $\leq \frac{1}{2}$)
    - Run the algorithm k times on the same input
    - Probability that it will produce incorrect output on all k trials $\leq \frac{1}{p^k}$
    - Can make the probability of error arbitrarily small by making k sufficiently large

Example: Primality Testing

- A decision problem: decide whether a given positive integer n is prime or not
  - Without loss of generality, we will assume n is odd
- Very important problem in practice
- Known polynomial time algorithm has high complexity, takes too much time in practice
- Many good randomized algorithms designed

- Basic Idea of randomized algorithms for primality testing
  - Try to find "witness of compositeness" randomly
    - A number such that know number-theoretic tests applied on it can prove that ***n*** is composite
  - If no witness can be found in large enough number of tries, declare ***n*** as prime
- Problem: A composite number may be declared as prime if the random choices fail to choose a witness in every try
  - So the Yes answer may be wrong
- But a prime is never declared as composite
  - So the No answer is always correct

- Monte Carlo algorithms like this for decision problems are said to have one-sided error
  - One of the Yes/No answers may be wrong, but not both
  - Yes-biased algorithm if the Yes answer is always correct, but the No answer may be wrong
  - No-biased algorithm if the No answer is always correct, but the Yes answer may be wrong
- A Monte Carlo Algorithm for decision problem can also exhibit two-sided error
  - Both Yes and No answers can be wrong with some bounded probability

- Back to primality testing: How can we make the probability of declaring a composite as prime (incorrect output) small?
  - Choose the witness type that you want to use such that there is an "abundance of witnesses" in the space from which the witness is chosen
  - If you choose from a space of size N, and you know a lower bound $\boldsymbol{m}$ on the number of witnesses of compositeness in that space, then the probability of incorrect output in each try is upper bounded by $\left(1 - \frac{m}{N}\right)$
  - Choose witness type so that $\boldsymbol{m}$ is proven to be large

- Simplest witness of compositeness: factor of n
  - Algorithm:
    - Pick an integer x randomly from 3 to $\sqrt{n}$
    - Check if x is a factor of n
    - If no, output Yes, else output No
  - Problem: factors are not abundant
    - 33 has only 2 factors, 3 and 11
    - 81 has only 3 factors, 3, 9, and 37
    - In general, m is very small compared to N
  - So this is not a good type of witness to use

- Witness: Fermat Witness
  - Fermat's Theorem: If ***n*** is prime and ***p*** is an integer co-prime to ***n***, then $a^{n-1} \equiv 1\ (mod\ n)$
    - But the inverse is not true, so this is a necessary condition for ***n*** to be prime, but not sufficient
    - There exists composite numbers that satisfy this for some ***a***
      - Example: ***n*** = 15 and ***a*** = 4
      - ***n*** is said to be a pseudoprime to base ***a***
  - Fermat witness of compositeness: An integer ***a*** $\in \mathbb{Z}_n^+$ such that
    - $gcd(a, n) \neq 1$ or $a^{n-1} \not\equiv 1\ (mod\ n)$
  - Algorithm:
    - Pick an integer ***a*** at random from $\mathbb{Z}_n^+$
    - Check if $gcd(a, n) = 1$ and $a^{n-1} \equiv 1\ (mod\ n)$
    - If yes, output Yes, else output No
  - How many such ***a***'s can be there in $\mathbb{Z}_n^+$?

Theorem: *If there exists a Fermat witness for a composite integer* $n > 1$ *which is relatively prime to* n*, then more than half of the integers from* 2 *to* $n-1$ *are Fermat witnesses for* n

- So the probability of incorrect output $< \frac{1}{2}$
- And we can make it arbitrarily low by repeating the algorithm sufficient number of times
- So looks like a good algorithm, right?

- Problem: Carmichael numbers
  - A composite integer $\boldsymbol{n}$ that does not have any Fermat Witness
  - The algorithm will always classify them as primes irrespective of any random choices made
- So the algorithm works for all integers except Carmichael numbers
  - Carmichael numbers are not abundant but still infinite
    - only 7 below 10000, and increasingly rare as the numbers grow large)
      - For example, only 585,355 below 10^{17}
  - No efficient algorithm known to detect if an integer is a Carmichael number

Miller-Rabin Algorithm

- Uses the following result in addition to Fermat's test:

  If $x^2 \equiv 1 \pmod{n}$ *has any solution other than the trivial solutions* $x \equiv \pm 1 \pmod{n}$, *then n must be composite*
- Write $n - 1 = 2^k m$, with odd m
- Choose a coprime to n
- Repeatedly square a^m and check if a non-trivial square root exists or not
  - Compute $(a^m)^{2^1}, (a^m)^{2^2}, (a^m)^{2^3}, \ldots, (a^m)^{2^k} = a^{n-1}$
  - Check if there exists j such that $(a^m)^{2^{j-1}} \not\equiv 1 \bmod n$ but $(a^m)^{2^j} \equiv 1 \bmod n$

Miller-Rabin Algorithm

```
Write $n - 1 = 2^k m$, where $k \geq 1$ and $m$ is odd
Randomly choose $a$, $1 < a < n$
If gcd($a$, $n$) ≠ 1 declare $n$ as composite
$b_0 = a^m$ mod $n$
If $b_0 = \pm 1$, stop and declare $n$ as prime
For $j = 1$ to $k - 1$
        Compute $b_j = (b_{j-1})^2$
        If $b_j = 1$ mod $n$   /* non-trivial root found */
              stop and declare $n$ as composite
        If $b_j = -1$ mod $n$, stop and declare $n$ as prime
Declare n as composite
```

- Can be shown that if $\boldsymbol{n}$ is composite, then at least $\frac{3}{4}^{th}$ of the numbers in $1 < \boldsymbol{a} < \boldsymbol{n}$ are witnesses of compositeness
- So probability of incorrect output $\leq \frac{1}{4}$
- Can repeat $\boldsymbol{t}$ times to reduce probability of incorrect output to $\leq \left(\frac{1}{4}\right)^{t}$
- Works for all numbers, including Carmichael numbers

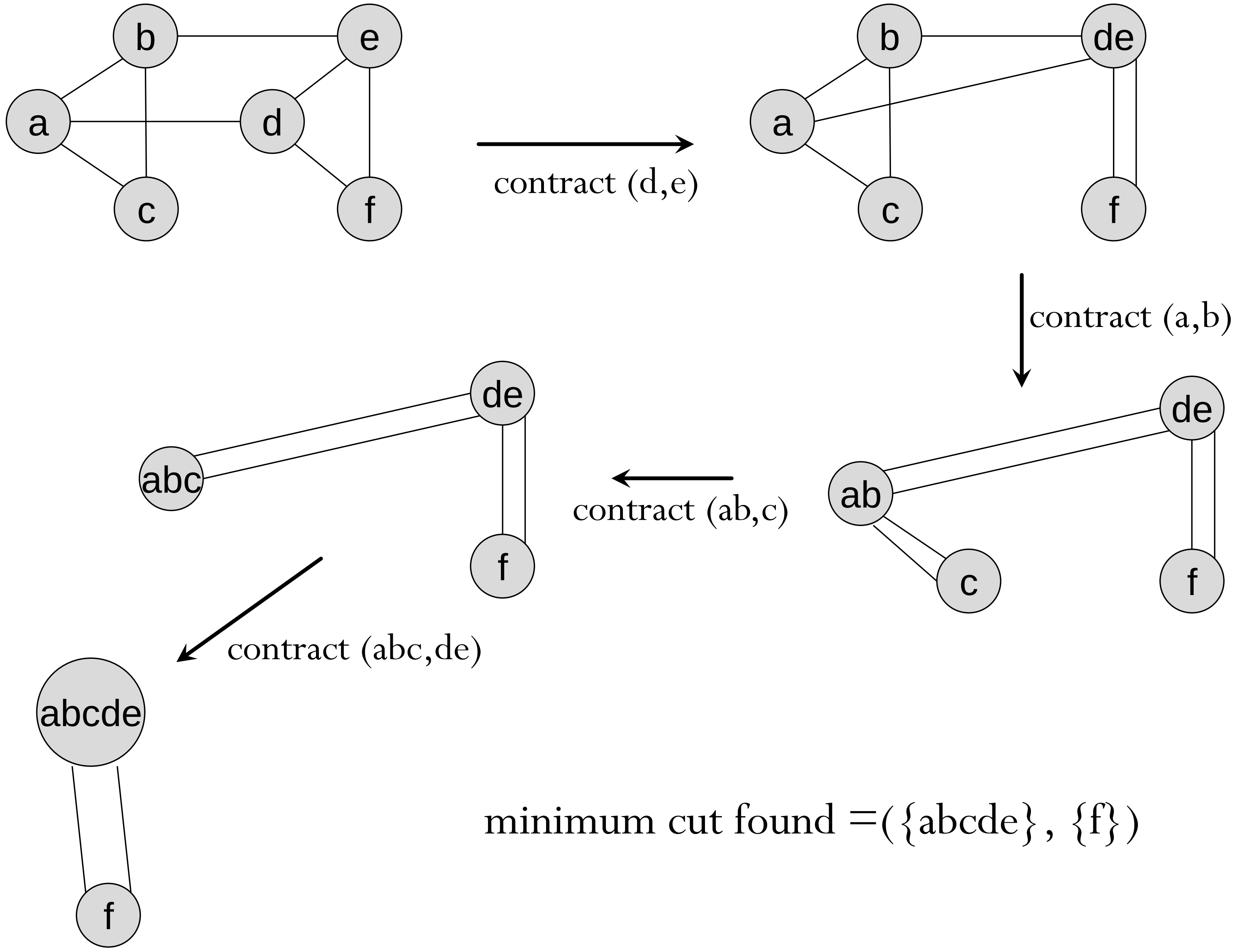
Example: Finding a Minimum Cut

- Input: An undirected connected graph $G = (V, E)$
- A cut in a graph is a partition of V into two sets S and (V – S)
  - Denoted as (S, V–S)
- Size of a cut is the number of edges with one endpoint in S and the other endpoint in V – S
- Minimum cut: The cut with the minimum size
  - Also defined as the least cardinality subset E_1 in E that disconnects G
- Can be found by repeated max-flow as we have seen
  - Best known complexity $O(|V|^3)$
- Karger's algorithm – a simple and elegant Monte Carlo randomized algorithm that works faster

Karger’s Algorithm

```
While $|V| > 2$
    Randomly pick an edge $(u,v)$ in E
    Contract $u$ and $v$ into a single vertex $uv$
Return the set of edges in E between the two
final vertices left as the minimum cut
```

- What does contracting (u, v) mean?
  - Add a node uv to V and remove nodes u and v from V
    - So V = V – {u, v} + uv
  - For each edge (u,w) in E for any $w \neq v$, remove (u,w) from E and add (uv, w) to E. Same for each edge (v, x) for any $x \neq u$. Remove all edges between u and v from E
- Note: This can result in parallel edges (more than one edge between same two nodes). Keep the parallel edges.



minimum cut found =({abcde}, {f})

- But this may not always give the minimum cut
  - Note that an edge disappears when you contract its endpoints
  - So if the algorithm picks an edge from the actual minimum cut, nodes from the two sides of the minimum cut are merged, and at least one edge of the minimum cut disappears

- So then why do this?
  - The chance of picking an edge in the minimum cut is low
    - Minimum cut has the smallest number of edges among all cuts
  - So more likely to get minimum cut than a non-minimum cut
  - But need to quantify "more likely"
    - What is the probability of getting a minimum cut?

Analysis

- Let $|V| = n$ and $|E| = m$
- We know that sum of degrees of all nodes $= 2m$
- So average degree of a node $= \frac{2m}{n}$
- Then, the size of the minimum cut can be at most $\frac{2m}{n}$
  - Consider the cut that separates the minimum degree node from the other nodes
  - Since average degree $= \frac{2m}{n}$, minimum degree of a node $\leq \frac{2m}{n}$
  - So need to remove only at most $\frac{2m}{n}$ edges to separate this node from the rest of the graph
  - Minimum cut size cannot be greater than this, as this is also a cut

- So the chance that an edge picked belongs to the minimum cut is at most $\frac{2}{n}$
  - Total *m* edges to choose from, at most $\frac{2m}{n}$ of which can belong to a minimum cut
- Probability of finding a minimum cut = Probability that none of the random choices choose an edge from the minimum cut

$$\geq \left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right)\left(1 - \frac{2}{n-2}\right) \ldots (1 - \frac{2}{3})$$

$$= \frac{n-2}{n} \times \frac{n-3}{n-1} \times \frac{n-4}{n-2} \times \cdots \frac{2}{4} . \frac{1}{3}$$

$$= \frac{2}{n(n-1)} > \frac{2}{n^2}$$

- So probability of not finding a minimum cut in one run is $< (1 - \frac{2}{n^2})$
  - This is not very good if ***n*** is large
- So improve the probability by running the algorithm ***k*** times
  - Take the best cut (minimum size) from the ***k*** runs
  - Probability of not getting a minimum cut after ***k*** runs is less than $(1 - \frac{2}{n^2})^k$

- We know that for any x ≥ 1,

$$\left(1 - \frac{1}{x}\right)^{x} \leq \frac{1}{e}$$

- So if we run the algorithm n^2 number of times and take the minimum among the minimum cuts reported in all these runs, the probability that it will not be a minimum cut is $\leq (1 - \frac{2}{n^2})^{n^2} \leq \frac{1}{e}$ (independent of n)
    - So success probability > 1/2
- If we run it $n^2 ln(n)$ times, the probability of not getting a minimum cut is $\leq \left(\frac{1}{e}\right)^{ln(n)} \leq \frac{1}{n}$
    - Finds the minimum cut with "high probability"

- But what about running time?
  - Each run can be implemented in $O(|V|^2)$ time with adjacency matrix
  - So total $O(|V|^4)$ time for n^2 runs
    - $O(|V|^4 lg|V|)$ if we want the success probability to be higher
  - But we already have a deterministic algorithm with lower complexity
    - So need to improve running time of Karger's algorithm if it is to be useful

Karger-Stein Algorithm

- Improvement over Karger's algorithm to improve running time
- Key Idea:
  - Chance of choosing a minimum cut edge increases as more and more edges are picked, as the number of edges remains the same but the number of nodes decreases
  - So do not run the algorithm till 2 vertices directly, as that increases the probability of failure at the later stages too much
  - Rather, run till the probability of the minimum cut being still there is $\geq \frac{1}{2}$ (Step 1). Then run two instances of the algorithm recursively on the remaining graph and take the minimum (Step 2)
  - So later stages that are more likely to contract a minimum cut edge are repeated more often, reducing chance of error

- The probability that the minimum cut stays after the 1st i steps (no min cut edges chosen)

  $$= \frac{n-2}{n} \times \frac{n-3}{n-1} \times \frac{n-4}{n-2} \times \cdots \frac{n-i-1}{n-i+1} \geq \frac{(n-i)^2}{n^2}$$

- Till how long does this probability stay $\geq \frac{1}{2}$?
  - Approximately till $i = \frac{n}{\sqrt{2}/(\sqrt{2}-1)}$
  - This corresponds to having $(n - i) = \frac{n}{\sqrt{2}}$ nodes left
- So in the first step, run till $\frac{n}{\sqrt{2}}$ nodes left

```
Karger-Stein(G, $n$)
  /* $n_0$ is a small constant */
  if ($n < n_0$) find the minimum cut by brute force
  else
        G$_1$ = G contracted down to $\frac{n}{\sqrt{2}}$ vertices
        (S$_1$, V – S$_1$) = Karger-Stein(G$_1$, $\frac{n}{\sqrt{2}}$)
        G$_2$ = G contracted down to $\frac{n}{\sqrt{2}}$ vertices
        (S$_2$, V – S$_2$) = Karger-Stein(G$_2$, $\frac{n}{\sqrt{2}}$)
        Choose the cut with minimum size
            between (S$_1$, V – S$_1$) and (S$_2$, V – S$_2$)
```

- Can show that the probability of finding a minimum cut is $\Omega\left({}^{1}/_{lgn}\right)$
- Time complexity for one run:

  $$T(n) = c \text{ if } n \leq n_0$$

  $$= 2T\left(\frac{n}{\sqrt{2}}\right) + O(n^2) \text{ otherwise}$$

  Solving by Master theorem gives

  $$T(n) = O(n^2 lgn)$$

- If we run ln^2n times, can reduce the probability of failure to $\leq \frac{1}{n}$
  - Time complexity $O(n^2lg^3n)$

Las Vegas Algorithm

Las Vegas Algorithm

- Randomized algorithms that always produces correct output, but whose time complexity can vary based on random choices made
- So correctness is deterministic, time complexity is probabilistic
    - Expected running time is important (should be polynomial to be of interest)
- Typical use
    - Improve performance
        - Ex.: Randomized quicksort, randomized selection
    - Searching in solution space

Example: Randomized Quicksort

- X = set of $\boldsymbol{n}$ numbers to sort (assume that all numbers are distinct for simplicity)
- Plain quicksort as you know it, with the difference that the pivot p is chosen randomly at each step
- Expected time complexity analysis:
  - Let T($\boldsymbol{n}$) denote running time of randomized quicksort with $\boldsymbol{n}$ elements
  - Then, T($\boldsymbol{n}$) = O($\boldsymbol{n} + \boldsymbol{C}$), where $\boldsymbol{C}$ is the total number of comparisons made in the partition (over all calls to quicksort, initial and recursive)
    - The $\boldsymbol{n}$ term comes because the number of recursive calls is O($\boldsymbol{n}$)
    - Swaps don't matter as it happens only as a result of comparison, so its (constant) time can be counted as part of the comparison cost

- So E[(T(n)] = n + E(C)
- So what is E(C)?
  - Let the elements of X in sorted order be $x_1, x_2, \ldots, x_n$
  - Let $X_{ij} = \{x_i, x_{i+1}, \ldots . x_j\}$ $1 \leq i < j \leq n$
  - Note that each pair of elements in X can be compared at most once during the algorithm
    - Any two numbers x_i and x_j will be compared only if either x_i or x_j is chosen as a pivot. If anything in between is chosen as a pivot, they will never be compared as they will go into different partitions. And once a number is chosen as a pivot, it is eliminated from all subsequent recursive calls so never compared with anything after that

Let $C_{ij} = 1$ if x_i is compared with x_j, 0 otherwise

Then $C = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} C_{ij}$

$\mathrm{E}[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[C_{ij}]$ /* by linearity of expectation */

$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[C_{ij} = 1]$

$\leq \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$ /* either x_i or x_j chosen as pivot */

$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k}$

$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$

$= \sum_{i=1}^{n-1} O(lgn)$

$= O(nlgn)$

- So $\mathrm{E}[\mathrm{T}(n)] = n + O(nlgn) = O(nlgn)$

Example: A Coloring Problem

- Given:
  - A set S of n items
  - k sets $S_1, S_2, \ldots S_k \subseteq S$ (not necessarily disjoint) such that
    - $S_i \neq S_j$ for all $i \neq j$
    - $|S_i| = r$ for all i, where $k \leq 2^{r-2}$
- Goal: color each item with one of two colors, red and blue, such that each set S_i contains at least one red and one blue item

- Algorithm

```
Repeat until a valid coloring is found
    Color each item either red of blue with probability $\frac{1}{2}$
    Check if the coloring obtained is a valid coloring
```

- Each iteration can be done in O(n + kr) time
- What is the expected number of iterations?
  - Need to find the probability of obtaining a valid coloring

- Let E_i = event that all items of S_i have color red
- Then $\Pr(E_i) = \left(\frac{1}{2}\right)^r$
- Then $\Pr(\bigcup_1^k E_i) \le k\left(\frac{1}{2}\right)^r \le 2^{r-2}\left(\frac{1}{2}\right)^r = \frac{1}{4}$
- Same for the event that all items of S_i are blue
- Thus, probability of all items being red or blue in any one set is $\le \frac{1}{2}$
- Hence, probability of getting a valid coloring in an iteration is $\ge \frac{1}{2}$
- Hence the expected number of iterations before getting a valid coloring is 2
- Expected time complexity = $O(\boldsymbol{n} + \boldsymbol{kr})$

Example: Bit Vector Search

- Input: A bit vector X[1..n]
- Goal: Output the index of a 1-bit
- Algorithm

```
Repeat
        Choose $i$ randomly between 1 and $n$
        If A[$i$] = 1 output $i$ and terminate
```

- May not terminate even if a 1-bit is there!
- But never gives a wrong index, so Las Vegas
- Typically in such cases, can stop the algorithm after a polynomial number of iterations and output a messge "*solution cannot be found*"

Relation between Monte Carlo and Las Vegas Algorithms

Claim: *Any Las Vegas algorithm with expected running time T can be converted into a Monte Carlo algorithm with running time cT and probability of error bounded by $\frac{1}{c}$ for any constant $c > 1$*

How:

- Run the Las Vegas algorithm for cT steps
- If no result is found, output an arbitrary result
- So deterministic time bound, but answer may be incorrect

- Note that for decision problems, outputting Yes/No arbitrarily may give two-sided error. Can you get it down to an algorithm with one-sided error?

- How does this bound the error probability of the Monte Carlo algorithm?
  - Probability of getting an incorrect output in the Monte Carlo algorithm = Probability of the Las Vegas algorithm not terminating in $\boldsymbol{c}T$ steps
  - Let X = random variable denoting time before termination of the Las Vegas algorithm
  - The error bound then follows directly from Markov Bound

[Markov Bound] *Let X be a nonnegative random variable with positive expected value. Then, for any constant $c > 0$,*

$$\Pr[X \geq \boldsymbol{c}\mathrm{E}[X]] \leq \frac{1}{c}$$

Can be proved easily from definition of expectation

Randomized Data Structure: Bloom Filter

Bloom Filters

- A randomized data structure for fast searching
- Keys represented in compressed storage as bit array
- Two operations supported – Insert and Search
- Search returns YES (present) or NO (not present)
  - NO is always correct
  - YES is correct with a probability
  - Similar to Monte Carlo with one-sided error
- Many practical applications in networks, content search etc.

Bloom Filter Operation

- A bit array A[0..m-1] of size m
- Initially all bits are set to 0
- A set of k random and independent hash functions h_0, h_1, …,h_{k-1} producing a hash value between 0 and $m - 1$
- Insert key x
  - Compute $y_i = h_i(x)$ for $i = 0, 1, \ldots, k - 1$
  - Set A[y_i] = 1 for $i = 0, 1, \ldots, k - 1$
  - ($y_0, y_1, y_2, \ldots, y_{k-1}$) is called the signature of x
- Search for a key x
  - Compute $y_i = h_i(x)$ for $i = 0, 1, \ldots, k - 1$
  - Answer YES if A[y_i] =1 for all i, NO otherwise

- NO answer is always correct
  - If x was inserted, corresponding y_i's must have been set to 1 for all i
- YES answers may be correct
  - y_i's may have been set to 1 due to insert of other keys
  - Note that all of them must have been set to 1 by insert of other keys
    - Could be insert of more than one key, each setting some bits

Example

- $m = 13$, $h_0 = x \bmod m$, $h_1 = 3x \bmod m$

0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0	0

- Insert 23, $y_0 = 23 \bmod 13 = 10$, $y_1 = 69 \bmod 13 = 4$

0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	1	0	0	0	0	0	1	0	0

Example

- Insert 44, $y_0 = 5, y_1 = 2$

0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	0	1	1	0	0	0	0	1	0	0

- Search for 23 or 44 obviously returns YES
- Search for 9, $y_0 = 9, y_1 = 1$, returns NO (both bits 0)
- Search for 11, $y_0 = 11, y_1 = 5$, returns NO (one bit 0)
- Search for 5, $y_0 = 5, y_1 = 2$, returns YES (both bits 1)
  - But 5 was not inserted, so false positive

Probability of False Positive

- Note that each hash function sets each bit in A with the same probability $\frac{1}{m}$
- The probability of setting a particular bit at least once during ***n*** insertions $= 1 - \left(1 - \frac{1}{m}\right)^{kn}$
- If a element ***u*** not inserted in the filter is searched, the ***k*** hash functions will produce ***k*** indexes uniformly at random, each of which is set with the above probability
- The probability of false positive = probability that all the ***k*** locations are set to 1 $= \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1 - e^{-\frac{kn}{m}}\right)^{k}$

- Probability of false positive depends on
  - Size *m* of bit vector
    - Larger *m* is, smaller the probability of false positive
  - Number of hash functions
    - Too few hash functions increases the chance of two different keys having the same signature, so higher chance of false positive
    - Too many hash functions set too many 1's for each key, filling up the bit vector fast, again increasing the chance of false positive

  Type of hash functions
    - Should be random and independent
  - Number of inserts *n*
    - For many applications, this is known a-priori
      - Example: content search in a memory cache that is first loaded from disk and then searched on every query

- Reducing the probability of false positive
  - Choosing number of hash functions ***k***
    - The probability is minimized with respect to ***k*** for $k \approx \frac{m}{n} ln2$
    - Use this to choose the number of hash functions ***k***
  - Choosing bit vector size ***m***
    - Minimized error probability $= 2^{-k} = \left(2^{-ln2}\right)^{\frac{m}{n}}$
    - If the desired error probability is ***p***, we can solve for ***p*** to get $m \approx -\frac{nlnp}{ln^2 2}$
    - As we mentioned, ***n*** is usually known, so ***m*** can be chosen suitably

- Bloom Filter summary
  - Space efficient way of answering the basic set membership question: Is an element in the set or not?
  - Yes answer may be wrong, No is always correct
  - Tradeoff between space used, no. of hash functions used, and the error probability bound desired
  - Application inputs *n* (estimated no. of insertions) and *p* (desired maximum error probability)
    - First compute *m* using the formula shown
    - Then compute *k*
    - Choose the *k* hash functions to be random and independent
  - Used in many practical applications