Randomized Algorithms

Marasani Jayasurya AM.EN.U4AIE20048
M Mahadev AM.EN.U4AIE20045

Contents

Randomized Algorithms

- An algorithm that uses random numbers to decide what to do next anywhere in its logic.
- Randomness is used to reduce the time complexity, space complexity of other algorithms.
- There are three types of Randomized Algorithms:
    1. Monte Carlo Algorithms
    2. Las Vegas Algorithms
    3. Atlantic City Algorithms

Types of Randomized Algorithms

Monte Carlo

- It may produce incorrect answer
- we are able to bound its probability
- By running it many times on independent variables we can make the failure probability arbitrary small at the expense of running time
- Eg; Primality Testing

Las Vegas

- Always gives the true answer
- Running time is random
- Running time is bounded
- Eg; Quick Sort

Atlantic City

- Lies in between Monte Carlo and Las Vegas
- It is always almost fast
- It is always almost correct
- Designing these algorithms is extremely complex process
- Very few of them are in existence

Advantages and Disadvantages

Advantages:

- Speed of randomized algorithm may be faster than any deterministic algorithm.
- These algorithms are simpler even if not faster.
- Sometimes randomized algorithms are best for practical scenarios.
- Randomized ideas lead to deterministic algorithms.

Disadvantage:

- There isn't a guarantee that the problem will be solved at all or, in some cases, even an upper time limit to obtain the solution.
- Quality is dependent on quality of random number generator used as part of the algorithm.

Freivald's Algorithm for Matrix Product Checking

- It's a Monte Carlo Algorithm
- Probabilistic randomized algorithm.
- Checks Two Matrices Product is equal to Matrix C i.e A*B = C
- Time Complexity is $O(n^2)$
- Naive Approach has Time Complexity $O(n^3)$
- Algorithm:
  - Input n*n Matrices, i.e A, B and C.
  - To verify A×B=C, choose a n×1 column vector r with randomly choosing 0 or 1.
  - Compute A×(B×r) and C×r which takes $O(n^2)$ time.
  - if A×(B×r) ≠ C×r, Then Output False else if A×(B×r) = C×r Output True.
- All Matrices should be square matrices.

- **Error Analysis:**
  - **Probability of Error,**
    - **Case 1: A×B = C The Probability of Error is 0**
    - **Case 2: A×B ≠ C The Probability of Error is less than 1/2 and for k iterations probability of failure less than $1/2^k$.**

Input:

$$AB = \begin{bmatrix} 2 & 3 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix} \stackrel{?}{=} \begin{bmatrix} 6 & 5 \\ 8 & 7 \end{bmatrix} = C.$$

$$\vec{r} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\begin{aligned} A \times (B\vec{r}) - C\vec{r} &= \begin{bmatrix} 2 & 3 \\ 3 & 4 \end{bmatrix} \left(\begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) - \begin{bmatrix} 6 & 5 \\ 8 & 7 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 2 & 3 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \end{bmatrix} - \begin{bmatrix} 11 \\ 15 \end{bmatrix} \\ &= \begin{bmatrix} 11 \\ 15 \end{bmatrix} - \begin{bmatrix} 11 \\ 15 \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \end{aligned}$$

Output Screenshot:

```
Enter the number of iterations:
100

False
```

Approximate Median Algorithm

- It's a Monte Carlo Algorithm
- Non-deterministic algorithm
- Naive Approach:
  - Median of an unsorted array time complexity is O(n Log n)
  - We need to sort the array first.
  - After sorting the array we need to use median formula to find the Median.
  - Median of a sorted array of size n is defined as the middle element when n is odd and average of middle two elements when n is even.
- The algorithm produces incorrect result with probability less than or equal to $2/n^2$.

- Algorithm:
  - Randomly choose k elements from the array where k=c log n (c is some constant)
  - Insert then into a set.
  - Sort elements of the set.
  - Return median of the set i.e. (k/2)th element from the set
- Time Complexity is O((Log n) x (Log Log n))
- **Input Screenshot:**

```
Enter the size of array: 10
Enter the values of array:
1
2
3
4
5
6
7
8
9
0
```

Output Screenshot:

```
Enter the any contstant value1000
Approximate Median: 4
```

Primality Testing

- Given a positive integer, we need to check if the number is prime or not.
- A prime is a natural number greater than 1 that has no positive divisors other than 1 and itself.
- A simple solution is to iterate through all numbers from 2 to n-1 and for every number check if it divides n. If we find any number that divides, we return false.
- This method has time complexity O(n).
- Instead of Checking till n-1 we can check upto $\sqrt{n}$ because the larger factor of n must be a multiple of smaller factor of n which has been already checked.
- This method has time complexity O($\sqrt{n}$).
- The better approaches for finding the given number is prime or not are:
  - Fermat Method
  - Miller - Rabin
  - Solovay - Strassen

Fermat Method:

- **Fermat's Little Theorem:** If n is a prime number, then for every a, $1 < a < n-1$

$$a^{n-1} \equiv 1 \pmod{n}$$

OR

$$a^{n-1} \% n = 1$$

- If a given number is prime, then this method always returns true.
- If the given number is composite (or non-prime), then it may return true or false, but the probability of producing incorrect results for composite is low and can be reduced by doing more iterations.
- **Algorithm:**
  1) Repeat following k times
     a) Pick a randomly in the range [2, n - 2]
     b) If gcd(a, n) ≠ 1, then return false
     c) If $a^{n--1} \equiv 1 \pmod{n}$, then return true
  2) Return true [probably prime]
- Time complexity of this solution is O(k Log n)
- This method is used if a rapid method is needed for filtering, for example in the key generation phase of the RSA public key cryptographic algorithm.
- Carmichael numbers are some composite numbers with the property that for every $a < n$, $\gcd(a, n) = 1$ and $a^{n-1} \equiv 1 \pmod{n}$.

Solovay - Strassen Test:

- It is a probabilistic test to determine if a number is composite or probably prime.
- We divide Solovay Strassen Primality Test algorithm in following two parts,
  - (1) Find the value of Euler Criterion formula
  - (2) Find Jacobi Symbol for given value
- **Eulerian Criterion rule:**

$$a^{\frac{n-1}{2}} \equiv x \pmod{n}$$

Where, a is any random variable from 2 to (n-1),
n: given number for primality test.

- **Jacobi Symbol:**

```
Jacobi (a, n) {
    j = 1
    while (a not 0) do {
        while (a even) do {
            a = a/2
            if (n = 3 (mod 8) or n = 5 (mod 8)) then
                j = -j
            }
        interchange (a, n)
        if (a = 3 (mod 4) and n = 3 (mod 4)) then
            j = -j
            a = a mod n
    }
    Return j
}
```

When a is even positive number then, $\left(\frac{a}{n}\right) = (-1)^{\left(\frac{n^2-1}{8}\right)} = \begin{cases} 1, & if\ n \equiv 1,7 (mod\ 8) \\ -1, & if\ n \equiv 3,5 (mod\ 8) \end{cases}$

When a is odd positive number then $\left(\frac{a}{n}\right) = (-1)^{\left(\frac{n-1}{2}\right)} = \begin{cases} 1, & if\ n \equiv 1 (mod\ 4) \\ -1, & if\ n \equiv 3 (mod\ 4) \end{cases}$

We compare this Jacobi symbol with Euler criterion formula and if both are same then the number is Prime and if both are different then number is Composite.

- The time complexity of this test is O(k Log n).
- It is possible for the algorithm to return an incorrect answer.
- If the input n is indeed prime, then the output will always probably be correctly prime.
- However, if the input n is composite, then it is possible for the output to probably be incorrect prime.
- The number n is then called an Euler-Jacobi pseudoprime.

Primality Testing inputs and Outputs

Fermat's Method:

```
Enter the Number:34059687
Number of Iterations:400
The number is not Prime
```

Solovay -Strassen Test:

```
Enter the number: 3405987
Enter the number of iterations: 500
The number is not Prime
```

Approximation of pi

- Here, we are trying to approximate the value of pi using random values which are given as points inside the a square
- The random points can be anywhere inside either the square or inside the circle which is also common to the square.
- We then calculate the ratio of number points that
- lied inside the circle and total number of generated points.
- From there the value of pi is estimated using the equation given below

$$\pi = 4 * \frac{\text{no. of points generated inside the circle}}{\text{no. of points generated inside the square}}$$

Algorithm:

```
radius= 1
x,y randomly_generated , range: (-1 to 1)
The range  (-1 to 1)
if it lies inside the circle:
   increment the circle points and square points
if not:
   increment only the square points
pi = 4 * circle_points / square_points
print(pi)
```

Time Complexity:

The Time Complexity of this algorithm is O(n)

Output screenshot

```
Enter the number of iteration: 100
Final Estimation of Pi= 3.1444
```

Randomized Binary Search

- In randomized binary search, instead of picking the middle element, we pick a random element within the range as our pivot. .
- **Inputs:**
  - Input range which is sorted
  - A key (for searching element)
- **Output:**
  - Shows the index of the searched element(key).
- **Time Complexity:**
  - Randomized binary search has logarithmic time complexity i.e O(Log(n))

Algorithm:

Recursive Algorithm

```
Pivot= random index in between [left, right]
pivot=left+ rand()%(right-left+1)
If array[pivot]==key
        Key is found. Return
Else If array[pivot]<key
        Search only in the right half,
        thus set left= pivot +1, new range [pivot+1, right]
Else if array[pivot]>key
        Search only in the left half,
        thus set right= pivot -1, new range [left, pivot-1]
If left>right
        Break and element is not found
```

Output Screenshot:

```
Enter the elements of the array: 8 7 4 5 3 6
Enter the element you need to search: 4
The searched element is not present
```

Iterative Algorithm:

```
Defining a function for  generating the random value
Defining another function to search :
   while the left is less than or equal to right :
   If x greater, ignore left half
   if we reach here, then element was
   else not present return -1
```

Output Screenshot:

```
Enter the elements of the array1 2 3 4 5 6 7
Enter the element you need to search: 5
Element is present at index 4
```

Randomized Quick Sort

- Randomized Quick Sort, we use a random number to pick the next pivot (or we randomly shuffle the array).
- Here, using random pivoting we first partition the array in place such that all elements to the left of the pivot element are smaller, while all elements to the right of the pivot are greater than the pivot.
- The inputs are :
  - 1. The array
  - 2. The lowest number in the array
  - 3. The highest number in the array.
- The expected output is :The sorted array.
- **Time Complexity:** The worst-case time complexity of the algorithm is O(n Log n)

Algorithm:

Lumoto Algorithm

```
The function random generator generates random integer:
          between the starting index and the ending index
The function partition is to rearrange according to the pivot
The function partitioner is to  swap the starting index of the array and the
pivot
If Element < pivot :
 to the left of the  pivot
If Element > pivot :
 Placed to the right of the pivot
The function quicksort:
   if stat index < stop index:
      partitioner(arr,start, stop)
        call left quick sort    (left of the pivot)
        right quick sort        (right of the pivot)
```

Output Screenshot

```
Enter the elements of the array:  34 2 42  2 2
[2, 2, 2, 34, 42]
```

Algorithm:

Hoare Partition:

```
partition(arr[], lo, hi)
Initialize left and right index
   while(True):
          Find a value in left side greater than pivot
          while arr[i] < pivot
             Find a value in right side smaller than pivot
          while arr[j] > pivot
          if i >= j:
              return j
              else:
                        swap arr[i] with arr[j]
The function partitioner is to swap the starting index of the array and the pivot
quicksort(arr[], lo, hi)
   if lo < hi
       p = partition_r(arr, lo, hi)
       quicksort(arr, lo, p)
       quicksort(arr, p+1, hi)
```

Output Screenshot:

```
Enter the elements of the array: 5 4 3 2 1
[1, 2, 3, 4, 5]
```

Thank You