Introduction to Randomized Algorithms III

Joaquim Madeira

Version 0.3 – November 2024

Overview

- Randomized Algorithms
- Monte Carlo Methods
- Random Sampling
- Fermat’s Primality Test
- Las Vegas Algorithms
- Monte Carlo Algorithms
- Randomized Algs. for Optimization Problems

RANDOMIZED ALGORITHMS

Randomized Algorithms

- Use a degree of randomness as part of an algorithm’s logic

- Algorithm behavior can be guided by random bits as an auxiliary input
  - Take decisions by tossing coins !

- Aiming at good performance on average !

Randomized Algorithms

- What is the effect of randomness?
- Algorithm running time and / or algorithm output are random variables
  - Determined by the random bits / by the coin tossing results

MONTE CARLO METHODS

Randomized Algorithms

- Monte Carlo methods
  - Rely on repeated random sampling to achieve numerical results
  - Often used to approximate numerical solutions for problems in Physics and Mathematics

Monte Carlo Methods

- Define the domain of possible values
- Generate random values inside the domain
  - Probability distribution ?

No, the short answer is you use whichever distributions best represent your variables.

The uniform is rarely appropriate.

- Process the generated values
  - Deterministic computation
- Compute the desired result
  - Approximation !!

What are we computing ?

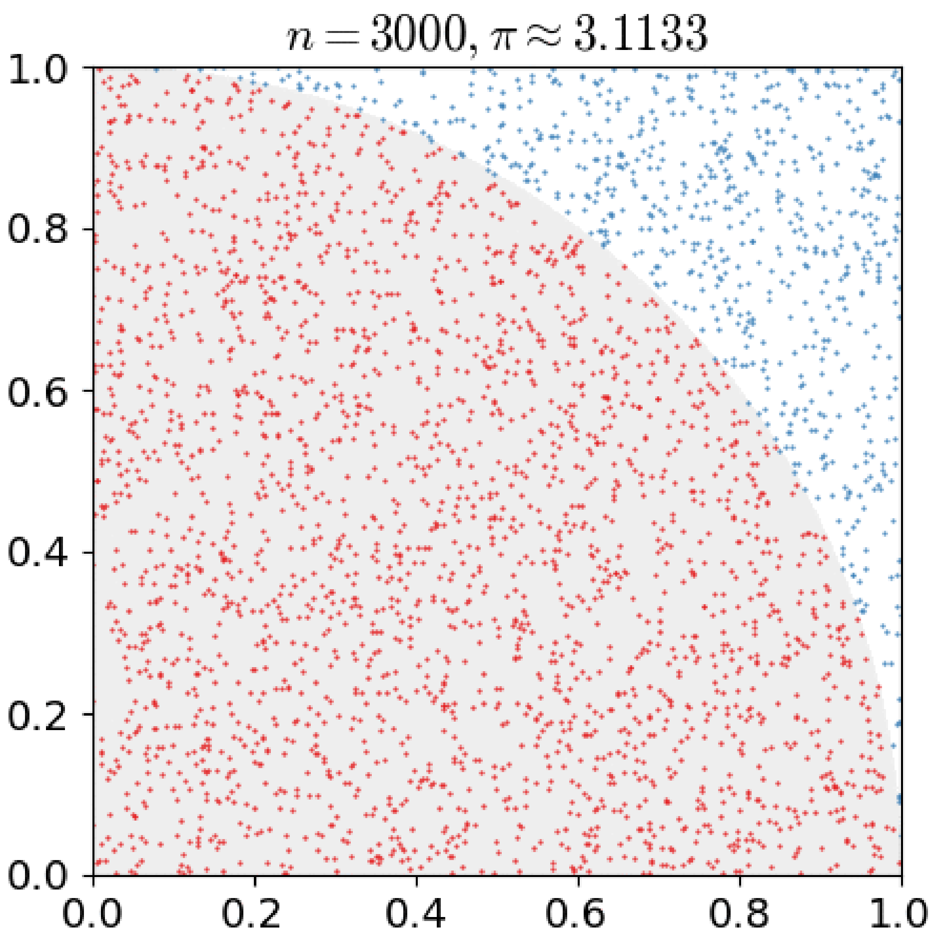
```
def monte_carlo_approximation( num_points ):
```

Approximatting π

```
    domain_area = 4

    inside_counter = 0

    for i in range( num_points ):

        x = random.uniform( -1, 1 )

        y = random.uniform( -1, 1 )

        if x * x + y * y <= 1.0:

            inside_counter += 1

    return ( inside_counter / num_points ) * domain_area
```

portion of samples inside circle

Another method



[Wikipedia]

TASK

Task – A (very) simple example

- The birth rate ratio of boys to girls is 51 to 49
- What is the probability of having two children who are both girls ?

- Use repeated random sampling
- And a large number of repetitions to approximate that probability value

```
boys_and_girls_domain = 51 * ['b'] + 49 * ['g']

def monte_carlo_approximation(num_points):
  inside_counter = 0
  for i in range(num_points):
    two_children = np.random.choice(boys_and_girls_domain, size=2)
    if (two_children == ['g', 'g']).all():
      inside_counter += 1

  return (inside_counter/num_points) * len(boys_and_girls_domain)
```

LAS VEGAS ALGORITHMS

Randomized Algorithms

- Las Vegas algorithms
  - Use the random bits to reduce expected running time or memory usage
  - BUT always terminate with a correct result, in a bounded amount of time

Example – Las Vegas Algorithm

- Array of n >= 2 elements, half are ‘a’, the other half ‘b’
- Find an ‘a’ in the array

```
repeat
      Randomly select an array element;
until ‘a’ is found;
```

- Algorithm succeeds with probability 1
- Expected running time over many calls is O(1)

MONTE CARLO ALGORITHMS

Probabilistic Algorithms

- Randomized algorithms that:
- Have a chance of producing an incorrect result
  - Monte Carlo algorithms
- Have a chance of failing to produce a result
  - Signaling a failure
  - Failing to terminate (!?!)

Example – Monte Carlo Algorithm

- Array of n >= 2 elements, half are ‘a’, the other half ‘b’
- Find an ‘a’ in the array

```
i = 0;
repeat
     Randomly select an array element;
     i = i + 1;
until i == k or ‘a’ is found;
```

Example – Monte Carlo Algorithm

- Run time is fixed !
- If an ‘a’ is found, the algorithm succeeds; else, it fails !
  - Compare with the Las Vegas version

- What is the probability of having found an ‘a’ after k iterations ?
  - Is it “large” or “small” ?

- Expected running time over many calls is O(1)

RANDOM SAMPLING

Choosing a Random Sample

- How to choose a random sample of size K from a set of size N ?
  - Random sampling a "population"

- Possible goal – Identify common features
- Why sampling ? – Population is too large !

- Every subset of size K < N must be given equal chance of being chosen !
  - How many such subsets ?

Choosing a Random Sample

```
in_sample[i] = false, for i=1 to N;
while( count < K )
     r = rand_int( 1, N );
     if( not in_sample[r] )
          in_sample[r] = true;
          count++;
Read the selected samples from file;
```

Choosing a Random Sample

- Do we have a valid random sample ?
  - Higher probability for some subsets ?
- Yes, the selection process is unbiased !
- What is the probability of each subset of K elements being chosen ?

Choosing a Random Sample

- Problem : we might need a VERY LARGE boolean array !

- K << N , usually

- Use an integer array of size K to keep the sorted set of selected indices !

- How to modify the previous algorithm ?

Choosing a Random Sample

- ISSUE : we are assuming that the entire set of size N is known in advance

- What if that is not the case ?
  - Need a on-line algorithm !

- Reservoir Sampling algorithms
  - N is unknown
  - N is TOO LARGE

Reservoir Sampling – Algorithm R

```
// Jeffrey Vitter, 1985

// 1 - Fill the reservoir array

Read K objects into array reservoir[1..K];

num_objs_read = K;
```

Reservoir Sampling – Algorithm R

```
// 2 - Sampling and Replacement

while( not end of input )

      obj = Get next object;

      r = rand_int( 1, num_objs_read );

      if( r <= K )

            reservoir[r] = obj;

      num_objs_read++;
```

TASKS

Random Sampling – Tasks

- Implement the three previous random sampling algorithms
- Analyze their behaviour for different test cases
  - How many random numbers are generated by the first algorithms ?
  - How many reservoir replacements are done in the last algorithm ?

HOW TO ADDRESS DECISION PROBLEMS ?

Monte Carlo Algorithms – Recap

- Guaranteed to be fast !
- Might not find the correct solution !
- BUT, the probability of finding a correct answer can be computed and controlled
  - Incorrect answer with negligible probability !
  - That is what makes them useful !

Decision Problems

- Decision problems
  - Answer is yes / no – true / false

- Some decision problems are “difficult”
  - No known polynomial algorithm, at the moment
  - Alternatives to “brute-force” for large instances ?

- Monte Carlo algorithms are useful here !
  - Fast execution
  - Negligible error probability

Monte Carlo Algorithms

- For decision problems, Monte Carlo algorithms can be:

- Yes-biased
  - A yes answer is always correct !
  - A no answer might be correct, with some probab.
- No-biased
  - A no answer is always correct !
  - A yes answer might be correct, with some probab.

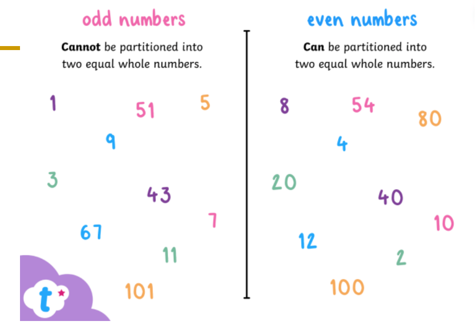
PRIMALITY TESTING

Primality Testing

- Given a positive integer p
- Is it a prime ?
  - Yes / No
- Important decision problem ?
  - Applications ?
- Naïve deterministic algorithm ?

```python
def is_prime(n):
    """
    Naive primality test.

    Parameters:
        n (int): The number to test for primality.

    Returns:
        bool: True if n is prime, False otherwise.
    """
    if n <= 1:
        return False  # 0 and 1 are not prime numbers
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

# Example usage
number = 29
if is_prime(number):
    print(f"{number} is a prime number.")
else:
    print(f"{number} is not a prime number.")
```

Naïve Primality Testing - Task



- Implement a first, naïve brute-force primality testing algorithm

- Improve your previous algorithm in order to avoid using unnecessary divisors !!
  - Even divisors ? for even number only verifying for 2 is necessary
  - Stopping criterion ? sqrt of n

- What is the largest prime you can find in a few seconds ?

Fermat's Primality Test

```
boolean fermat_test( P ) // P > 3
    a = rand_int( 2, P - 2 );
    if( power( a, P - 1 ) % P != 1 )
        return false; // Composite !!
    return true; // Meaning ?
```

(Certainly not prime)

(prime with some probability)

No-Biased problem !!!!!!!!!!

Does it always work ?

15 and 341 are not prime numbers

- Result for p = 15 and a = 2 ? 2^14 % 15 = 4 != 1
  - Composite !
  - 2 is a Fermat-witness for 15
- Result for p = 341 and a = 3 ? 3^340 % 341 = 56 != 1
  - Composite ! $p = 11 \times 31$
  - 3 is a Fermat-witness for 341
- Result for p = 341 and a = 2 ? 2^340 % 341 = 1
  - 2 is a Fermat-liar for 341 !! Lying witness

Iterated Fermat's Primality Test

```
boolean is_prime( P, K ) // P > 3
    for( i = 0; i < K; i++ )
        // Repeating Fermat's test
        a = rand_int( 2, P - 2 );
        if( power( a, P - 1 ) % P != 1 )
            return false; // Composite !
    return true; // PROBABLY prime !
```

Replication to increase certainty

Fermat's Primality Testing

- Fermat's Little Theorem (1640)
  - If the integer number p is prime, then for every integer a, 1 <= a < p

$$a^{p-1} \bmod p = 1$$

- How accurate is a true answer ?
- How much confidence in a true answer ?
  - Fermat liars vs Fermat witnesses
  - What is the proportion of Fermat witnesses ?

Fermat's Primality Testing

- Theorem
  - If the integer number p is not a prime, then at most half of the integers a, 1 <= a < p, satisfy the equation in Fermat's Little Theorem.
- Consequence ?
  - 1 test – error probability at most 50%
  - 2 tests – error probability at most 25%
  - …
  - 10 tests – error probability is negligible (!?!)

Remarks

- 1 and (p – 1) are trivial Fermat-liars
  - Do not use them !
- Exponentiation and integer division are "expensive" operations
- There are some "stubborn" composite numbers
  - Carmichael numbers : 561, 1105, 1729, 2465, …

    will be identified as primes with Fermat Primality Testing, even though they are not

Carmichael Numbers

- 561, 1105, 1729, 2465, 2821, 6601, …
- Odd composite number n which satisfies

$$b^{n-1} \equiv 1 \ (\mathrm{mod}\ n)$$

  - For all integers b which are relatively prime to n
- What happens for 561 for a few values of b ?
- Consequence for Fermat's primality test ?

The operator $\equiv$ is a symbol used in **modular arithmetic** and is read as "is congruent to." It indicates that two numbers have the same remainder when divided by a given modulus.

In mathematical terms:

$$a \equiv b \pmod{n}$$

Primality Testing – Better alternatives

- Solovay-Strassen, 1977
  - First Monte Carlo algorithm for primality testing
- Miller-Rabin, 1980
- Baillie-PSW, 1980
- Also, PRIMES is in P – 2002
  - But, Monte Carlo primality testing mostly used !

TASKS

Primality Testing

- Implement Fermat's primality test

- Generate some random positive integers and check if any of them is a prime
  - Check your results using the OEIS

- Use the Fermat's primality test to list the first Mersenne primes
  - It is not the fastest way !! But, it is OK for us…
  - Check your results using the OEIS

Primo de Mersenne é um número de Mersenne (número da forma $M_n = 2^n - 1$, com "n" número natural) que também é um número primo. Nem todo número de Mersenne é primo: *entre os números de Mersenne, com efeito, há aqueles que são primos*; porém, além do número um, que é número de Mersenne ($M_1 = 1$), porém *não*-primo, pois singular, há também números de Mersenne compostos.

RANDOMIZED SEARCHING & SORTING

Las Vegas Algorithms – Recap

- Guaranteed to give the correct answer !

- BUT their execution time is probabilistic !
  - Although expected to be fast in general

- The probability of an efficient execution time can be computed and controlled
  - Very long execution times with low probability !
  - That is what makes them useful !

Randomized Search

```
boolean las_vegas_search( a[], N, Target )

    for( i = 0; i < N; i++ )

        // Ensure no repeated indices
        test = rand_int( 0, N - 1 );

        if( a[test] == Target )

            return true;

    return false;
```

Randomized Search

- On average, faster than linear search
  - Whenever the array contains multiple occurrences of the target

- Compare the performance of linear search and randomized search
  - Generate random arrays
  - Select random targets
  - Count the number of array comparisons !

Randomized Quicksort

QuickSort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

Complexity Analysis of Quick Sort

Time Complexity:

- **Best Case:** ($\Omega(n \log n)$), Occurs when the pivot element divides the array into two equal halves.
- **Average Case** ($\theta(n \log n)$), On average, the pivot divides the array into two parts, but not necessarily equal.
- **Worst Case:** ($O(n^2)$), Occurs when the smallest or largest element is always chosen as the pivot (e.g., sorted arrays).

- How can we transform Quicksort into a randomized algorithm ?

- The probability of Worst Case behaviour will be much smaller !

- Standard vs randomized Quicksort
  - Compare their performance
  - Generate random arrays of large sizes
  - Number of array comparisons and exchanges ?

HOW TO ADDRESS OPTIMIZATION PROBLEMS ?

Randomized Algs. for Opt. Problems

- Compute an approximate solution for optimization problems

- Execute k runs of a randomized algorithm

- Final result ?
- The best of the k solutions computed
  - Regarding the optimization goal
  - I.e., the objective function

Probability of (in)success

- Probability of computing no optimal solution, in a given run of the algorithm

$$\left(1 - \frac{1}{n}\right)$$

- After n runs we have

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

- The probability of having obtained an optimal solution is at least

$$1 - \frac{1}{e}$$

Nearly-optimal solutions

- We are usually happy with a feasible solution that does not differ much from an optimal solution
- It is an approximation strategy !!
- Move from exponential or factorial time complexity to polynomial time complexity !

Approximation Accuracy – Min Prob

- Minimize function f()
- Approximate solution : s_a
- Exact solution : s^*

- Relative error : $re(s_a) = (f(s_a) - f(s^*)) / f(s^*)$

- Accuracy ratio : $r(s_a) = f(s_a) / f(s^*)$

- Performance ratio : R_A
  - The lowest upper bound of possible $r(s_a)$ values
  - Should be as close to 1 as possible
  - Indicates the quality of the approximation algorithm

Approximation Accuracy – Max Prob

- Maximize function f()
- Approximate solution : s_a
- Exact solution : s^*

- Relative error : $re(s_a) = (f(s^*) – f(s_a)) / f(s^*)$

- Accuracy ratio : $r(s_a) = f(s^*) / f(s_a)$

- Performance ratio : R_A
  - The largest lower bound of possible $r(s_a)$ values
  - Should be as close to 1 as possible

Main Goals

- Improve the accuracy ratio
  - I.e., the “quality” of the approximate solution
- Produce feasible solutions whose cost / value is not very far from the optimal cost / value
- With high probability !
- And without taking too much time !!

TASK
– THE TSP

Tasks – TSP

- Develop a randomized algorithm for the TSP

- For some test instances
- Compute optimal solutions using exhaustive search
- Compute approximate solutions after 100, 1000, 10000, … iterations
- Evaluate the accuracy of the obtained approximate solutions

REFERENCES

References

- D. Vrajitoru and W. Knight, *Practical Analysis of Algorithms*, Springer, 2014
  - Chapter 6
- J. Hromkovic, *Design and Analysis of Randomized Algorithms*, Springer, 2005
  - Chapter 2
- M. Dietzfelbinger, *Primality Testing in Polynomial Time*, Springer, 2004
  - Chapter 5