RANDOMIZED ALGORITHMS

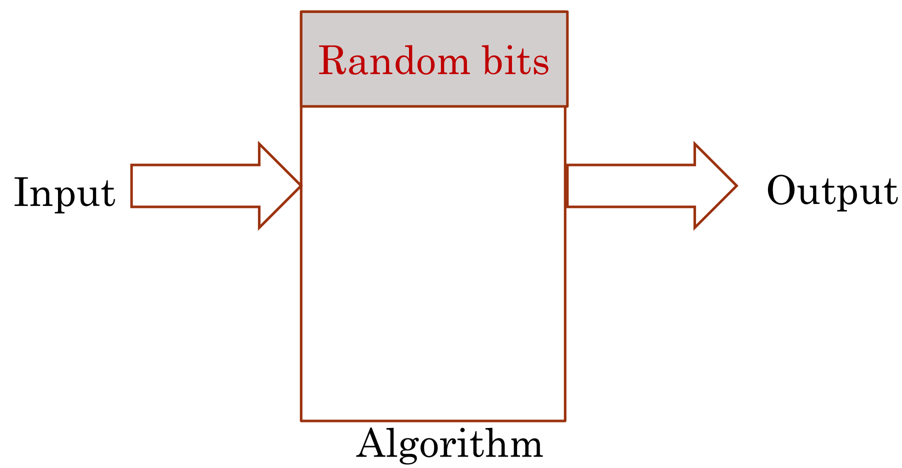
Module V

A short list of categories

- Algorithm types include:
  - Simple recursive algorithms
  - Backtracking algorithms
  - Divide and conquer algorithms
  - Dynamic programming algorithms
  - Greedy algorithms
  - Branch and bound algorithms
  - Brute force algorithms
  - Randomized algorithms

    - Also known as Monte Carlo algorithms or stochastic methods

Randomized Algorithm



- The **output** or the **running time** are **functions** of the **input** and **random bits chosen** .

Randomized algorithms

- A randomized algorithm is just one that depends on random numbers for its operation
- These are randomized algorithms:
  - Using random numbers to help to find a solution to a problem
  - Using random numbers to improve a solution to a problem
- These are related topics:
  - Getting or generating “random” numbers
  - Generating random data for testing (or other) purposes

Randomized Algorithms

- A randomized algorithm performs coin tosses (i.e., uses random bits) to control its execution
- It contains statements of the type

  $b \leftarrow random()$

  if $b = 0$

  do A …

  else { $b = 1$}

  do B …
- Its running time depends on the outcomes of the coin tosses

- We analyze the expected running time of a randomized algorithm under the following assumptions

  the coins are unbiased, and

  the coin tosses are independent

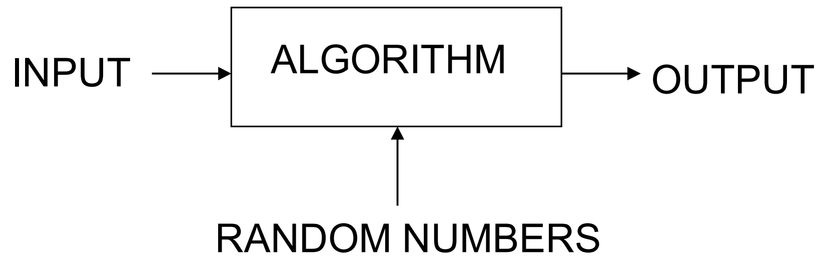
The worst-case running time of a randomized algorithm is often large but has very low probability (e.g., it occurs when all the coin tosses give "heads")

Pseudorandom numbers

- The computer is *not capable* of generating truly random numbers
  - The computer can only generate pseudorandom numbers--numbers that are generated by a formula
  - Pseudorandom numbers *look* random, but are perfectly predictable if you know the formula
    - Pseudorandom numbers are good enough for most purposes, but not all--for example, not for serious security applications
  - Devices for generating truly random numbers do exist
    - They are based on radioactive decay, or on lava lamps
- "Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin."

  —John von Neumann

Randomized Algorithms



- In addition to input, algorithm takes a source of random numbers and makes random choices during execution;

- Behavior can vary even on a fixed input;

Types of Randomized Algorithms

Randomized Las Vegas Algorithms:

- Output is always correct
- Running time is a **random variable**

Example: Randomized Quick Sort

Randomized Monte Carlo Algorithms:

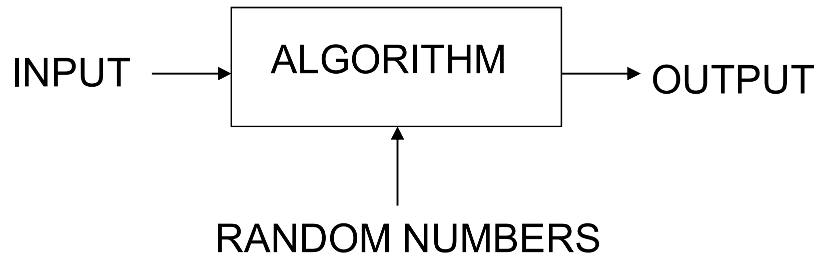
- Output may be incorrect with some probability
- Running time is deterministic.

Example: Randomized algorithm for approximate median

Motivation for Randomized Algorithms

- Simplicity;
- Performance;
- Reflects reality better (Online Algorithms);
- For many hard problems helps obtain better complexity bounds when compared to deterministic approaches;

Las Vegas Randomized Algorithms



Goal: Prove that for all input instances the algorithm solves the problem correctly and the expected number of steps is bounded by a polynomial in the input size.

Note: The expectation is over the random choices made by the algorithm.

Probabilistic Analysis of Algorithms

Input is assumed to be from a probability distribution.

Goal: Show that for all inputs the algorithm works correctly and for most inputs the number of steps is bounded by a polynomial in the size of the input.

Example : randomized Quick Sort

QuickSort(S)

QuickSort(S)

{ If (|S|>1)

Pick and remove an element x from S;

($S_{<x}$, $S_{>x}$)← Partition(S,x);

return(Concatenate(QuickSort($S_{<x}$), x, QuickSort($S_{>x}$))

}

QuickSort(S)

When the input S is stored in an array A

QuickSort(A,l, r)

{ If ($l < r$)

$x \leftarrow A[l]$;

$i \leftarrow$ **Partition**(A,l,r,x);

QuickSort(A,l, $i-1$);

QuickSort(A,$i+1$, r)

}

- **Average** case running time: O($n \log n$)
- **Worst** case running time: O(n^2)
- **Distribution sensitive**: Time taken depends upon the initial permutation of A.

Randomized QuickSort(S)
When the input S is stored in an array A

QuickSort(A,l, r)

{ If ($l < r$)

$x \leftarrow A[l]$;

$i \leftarrow$ Partition(A,l,r,x);

QuickSort(A,l, $i-1$);

QuickSort(A,$i+1$, r)

}

an element selected randomly uniformly from $A[l..r]$;

- **Distribution** insensitive: Time taken does **not** depend on initial permutation of A.
- Time taken **depends** upon the **random** choices of pivot elements.
  1. For a given input, Expected(**average)** running time: **$O(n \log n)$**
  2. **Worst** case running time: **$O(n^2)$**

Randomized Quick Sort

<u>Randomized-Partition(*A*, *p*, *r*)</u>

1. $i \leftarrow Random(p, r)$
2. exchange $A[r] \leftrightarrow A[i]$
3. **return Partition**(A, p, r)

<u>Randomized-Quicksort(*A*, *p*, *r*)</u>

1. **if** $p < r$
2. **then** $q \leftarrow$ **Randomized-Partition**(A, p, r)
3. **Randomized-Quicksort**$(A, p, q\text{-}1)$
4. **Randomized-Quicksort**$(A, q+1, r)$

Randomized Quick Sort

- Exchange $A[r]$ with an element chosen at random from $A[p \ldots r]$ in **Partition**.

- The pivot element is equally likely to be any of input elements.

- *For any given input, the behavior of Randomized Quick Sort* is determined not only by the input but also by the *random choices of the pivot.*

- We add randomization to Quick Sort to obtain for any input the expected performance of the algorithm to be good.