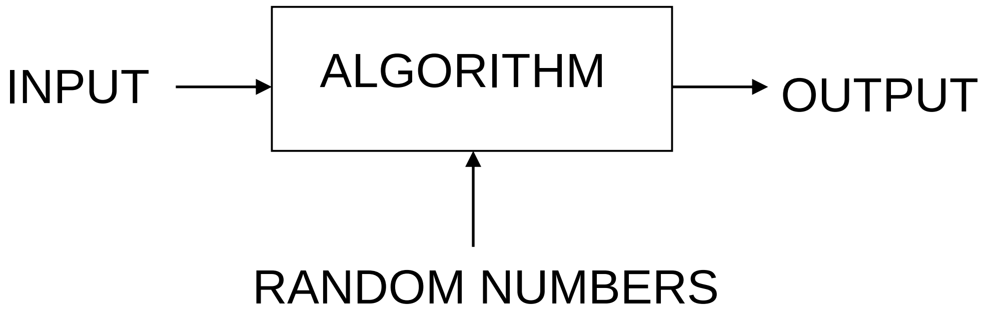
Introduction to Randomized Algorithms

Deterministic Algorithms

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

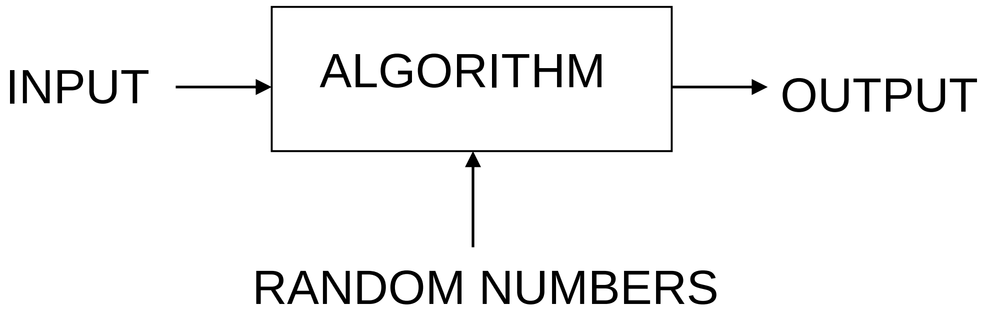
Randomized Algorithms



- 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

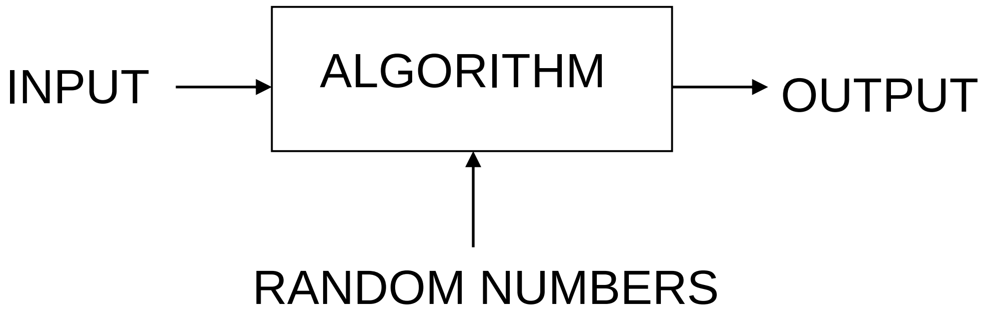
Las Vegas Randomized Algorithms



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.

Monte Carlo Randomized Algorithms



Goal: Prove that the algorithm

- with high probability solves the problem correctly;
- for every input 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.

Monte Carlo versus Las Vegas

Thus, Randomized algorithms are classified in two categories.

- **Las Vegas:**
  - ✓These algorithms always produce correct or optimum result.
  - ✓Time complexity of these algorithms is based on a random value and time complexity is evaluated as expected value.
  - ✓For example, Randomized QuickSort always sorts an input array and expected worst case time complexity of QuickSort is O(nLogn).
- **Monte Carlo:**
  - ✓Produce correct or optimum result with some probability.
  - ✓These algorithms have deterministic running time and it is generally easier to find out worst case time complexity.
  - ✓Examples include Karger's Algorithm and Fermet Method for Primality Testing.

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;

How to generate random numbers in Computers?

- True Random numbers can't be generated using computers as Computers use some algorithms to generate random numbers.
- Therefore, Computers can't generate pure random numbers, but it rather generates Pseudo Random Numbers.
- Computer generated random numbers appears random statistically.
- **How to Generate a random number in between low and high in C?**

```
srand(time(NULL));
int random = low + rand() % (high – low + 1); //You can use this as random number
```

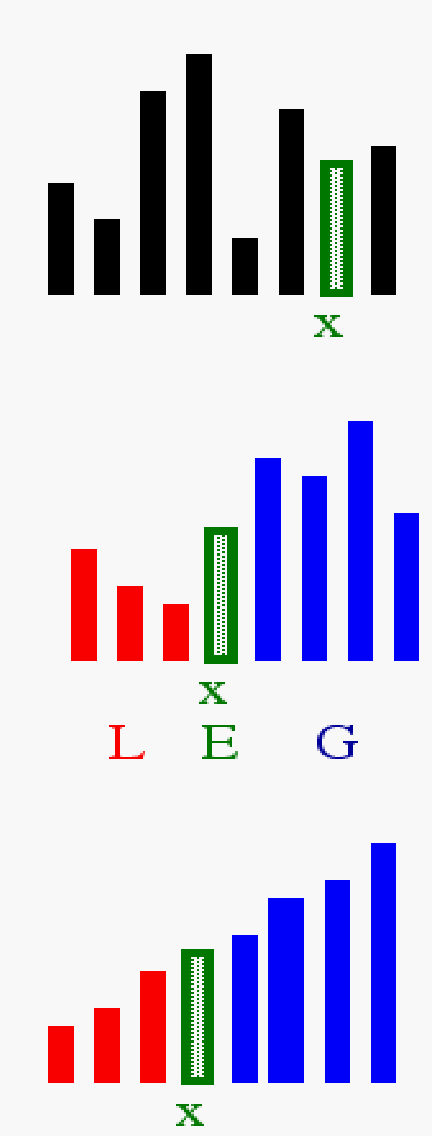
- **srand**() sets the seed which is used by **rand()** to generate "random" numbers.
- If you don't call **srand** before your first call to **rand**, it's as if you had called **srand**(1) to set the seed to one.

Quick Sort

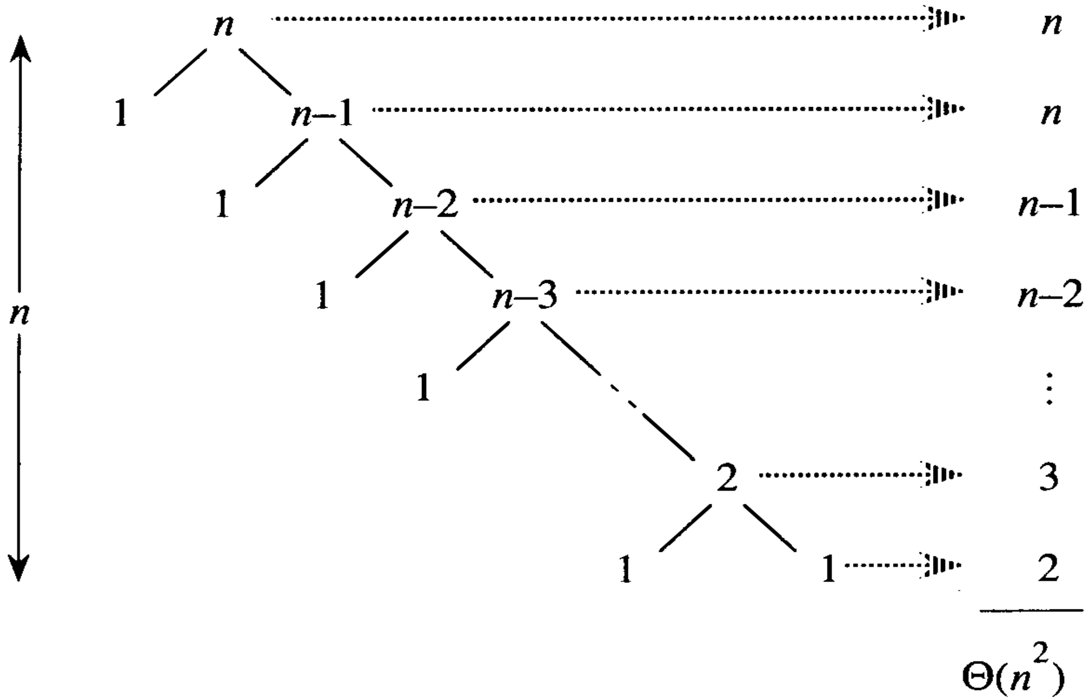
Select: pick an arbitrary element x in S to be the pivot.

Partition: rearrange elements so that elements with value less than x go to List L to the left of x and elements with value greater than x go to the List R to the right of x.

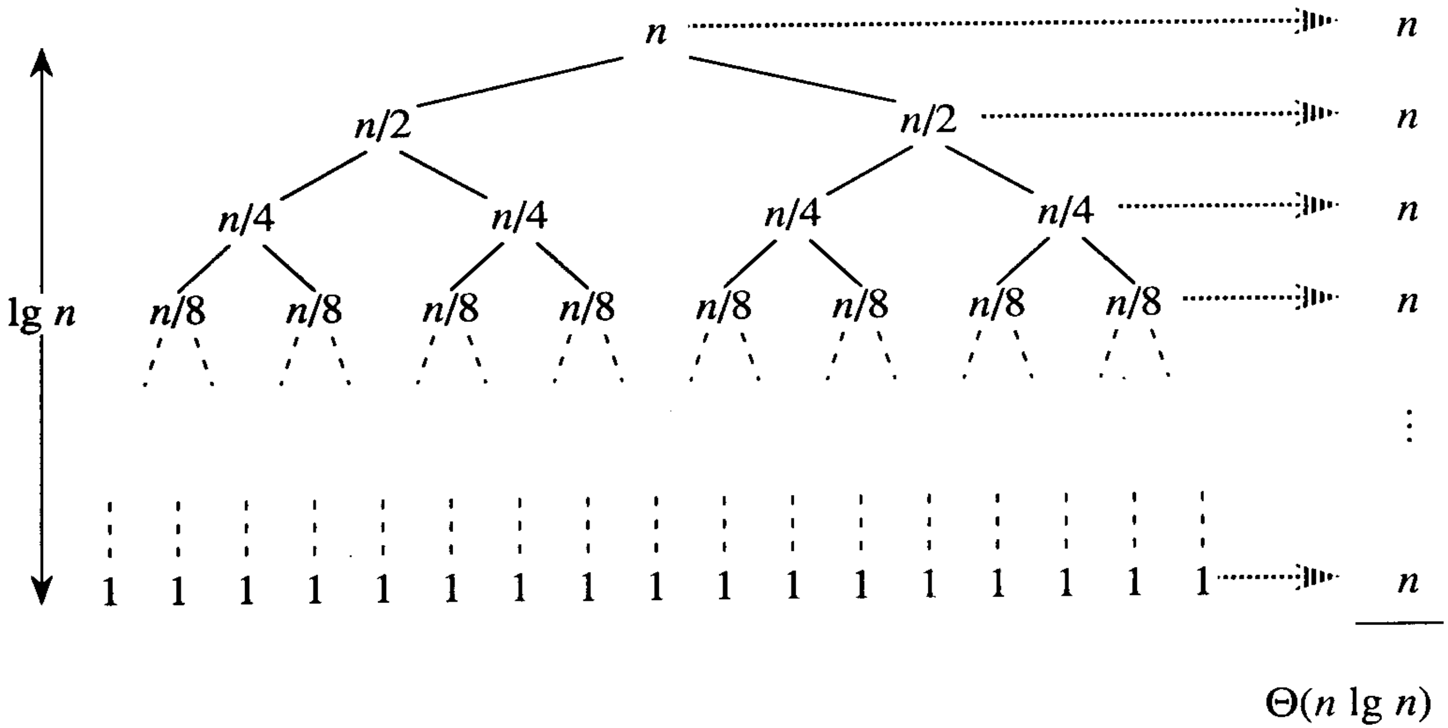
Recursion: recursively sort the lists L and R.



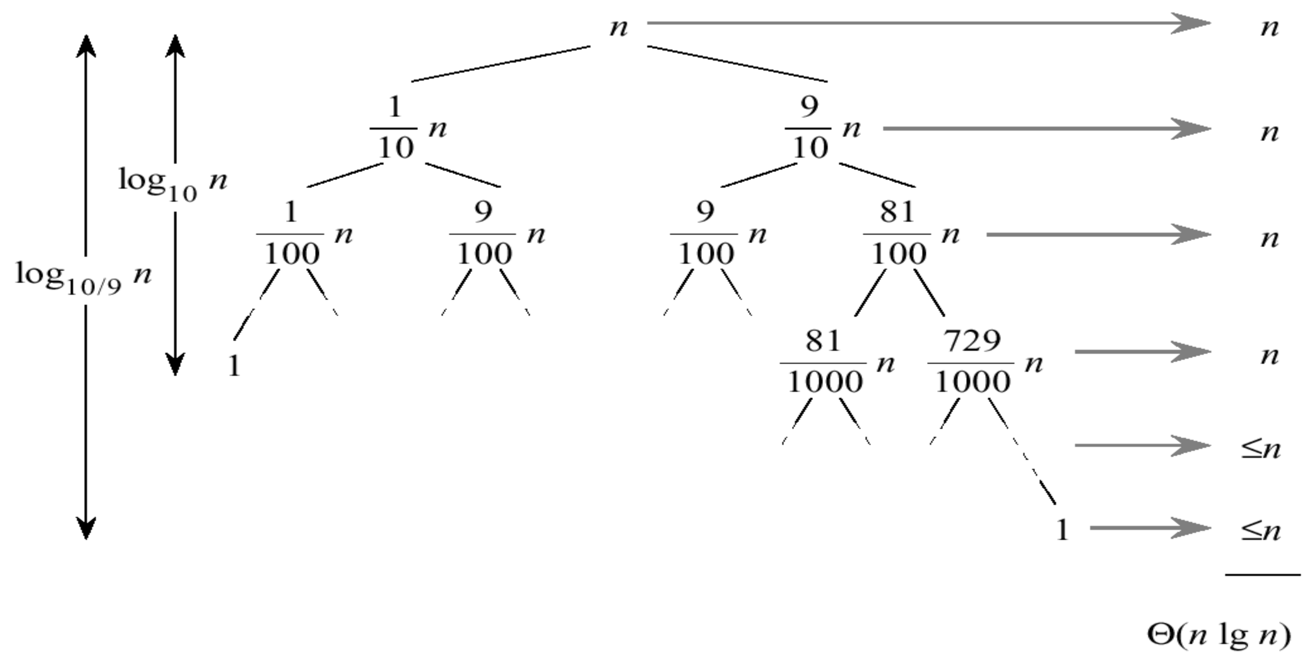
Worst Case Partitioning of Quick Sort



Best Case Partitioning of Quick Sort



Average Case of Quick Sort



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 <u>*random choices of the pivot.*</u>
- We add randomization to Quick Sort to obtain for any input the expected performance of the algorithm to be good.

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** **<u>Partition</u>**(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**$(A, q+1, r)$

```
int Partition(int arr[], int low, int high)
{       int pivot = arr[high]; // pivot
        int i = (low - 1); // Index of smaller element
        for (int j = low; j <= high - 1; j++)
         {      // If current element is smaller than or equal to pivot
                if (arr[j] <= pivot)
                 {      i++; // increment index of smaller element
                        swap(arr[i], arr[j]);
                }
        }
        swap(arr[i + 1], arr[high]);
        return (i + 1);
}
```

Analysis of Randomized Quick Sort

Linearity of Expectation

If X_1, X_2, …, X_n are random variables, then

$$E\left[\sum_{i=1}^{n} Xi\right] = \sum_{i=1}^{n} E[Xi]$$

Notation

z_2	z_9	z_8	z_3	z_5	z_4	z_1	z_6	z_{10}	z_7
2	9	8	3	5	4	1	6	10	7

- Rename the elements of A as $z_1, z_2, \ldots, z_n$, with z_i being the i^{th} smallest element (Rank "i").
- Define the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ be the set of elements between z_i and z_j, inclusive.

Expected Number of Total Comparisons in PARTITION

Let X_{ij} = I {z_i is compared to z_j } ⟵ indicator random variable

Let X be the total number of comparisons performed by the algorithm. Then

$$\left[X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \right]$$

The expected number of comparisons performed by the algorithm is

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E\left[X_{ij} \right]$$

by linearity of expectation

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \textit{ is compared to } z_j\}$$

Comparisons in PARTITION

<u>Observation 1</u>: Each pair of elements is compared at most once during the entire execution of the algorithm

- Elements are compared only to the pivot point!
- Pivot point is excluded from future calls to PARTITION

<u>Observation 2</u>: Only the pivot is compared with elements in both partitions

z_2	z_9	z_8	z_3	z_5	z_4	z_1	z_6	z_{10}	z_7
2	9	8	3	5	4	1	6	10	7

$Z_{1,6} = \{1, 2, 3, 4, 5, 6\}$

$\{7\}$
pivot

$Z_{8,9} = \{8, 9, 10\}$

Elements between different partitions are <u>never</u> compared

Comparisons in PARTITION

z_2	z_9	z_8	z_3	z_5	z_4	z_1	z_6	z_{10}	z_7
2	9	8	3	5	4	1	6	**10**	7

$Z_{1,6}$= {1, 2, 3, 4, 5, 6} {7} $Z_{8,9}$ = {8, 9, 10}

$$\Pr\{z_i \textit{ is compared to } z_j\}?$$

<u>Case 1</u>: pivot chosen such as: $z_i < x < z_j$

- z_i and z_j will never be compared

<u>Case 2</u>: z_i or z_j is the pivot

- z_i and z_j will be compared
- only if one of them is chosen as pivot before any other element in range z_i to z_j

Expected Number of Comparisons in PARTITION

Pr $\{Z_i$ is compared with $Z_j\}$

= Pr$\{Z_i$ or Z_j is chosen as pivot before other elements in $Z_{i,j}\}$ = 2 / (j-i+1)

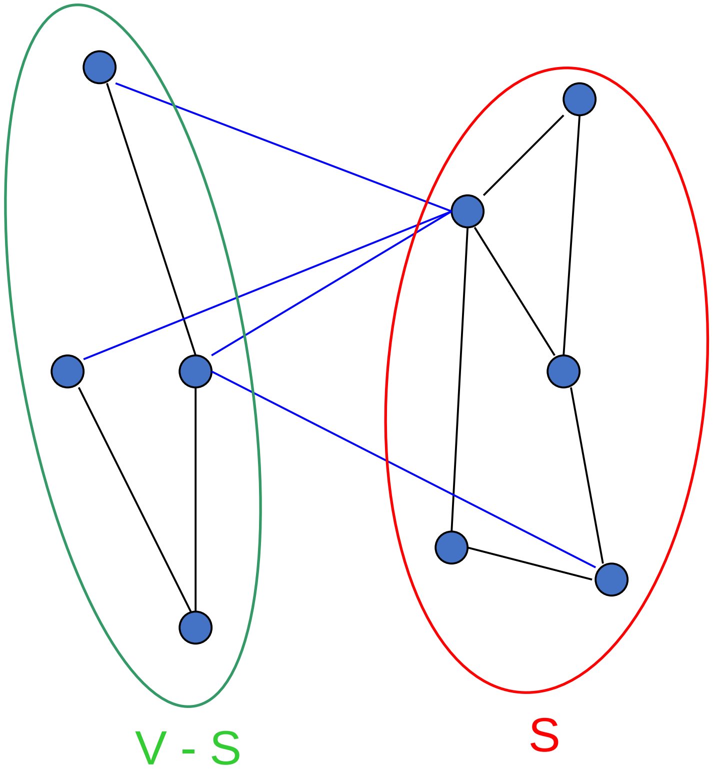
$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ } is\ compared\ to\ z_j\}$$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\lg n)$$

$= O(nlgn)$

Min-cut for Undirected Graphs

Given an undirected graph, a global *min-cut* is a cut (S,V-S) minimizing the number of *crossing edges*, where a crossing edge is an edge (u,v) s.t. u∈S and v∈ V-S.

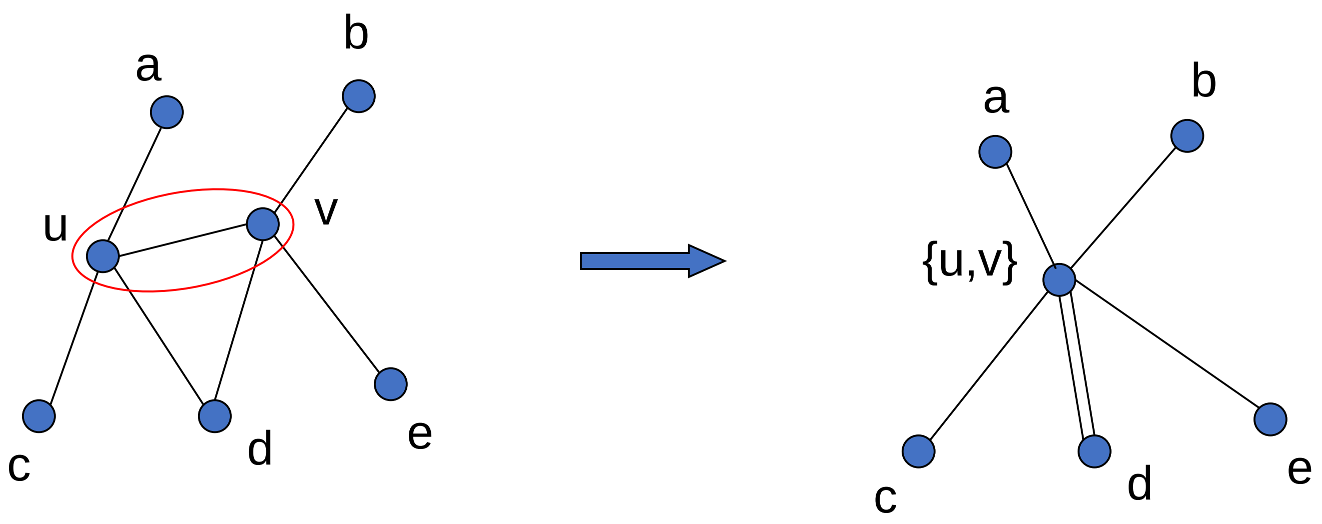


Graph Contraction

For an undirected graph G, we can construct a new graph G' by *contracting* two vertices u, v in G as follows:

- u and v become one vertex {u,v} and the edge (u,v) is removed;
- the other edges incident to u or v in G are now incident on the new vertex {u,v} in G';

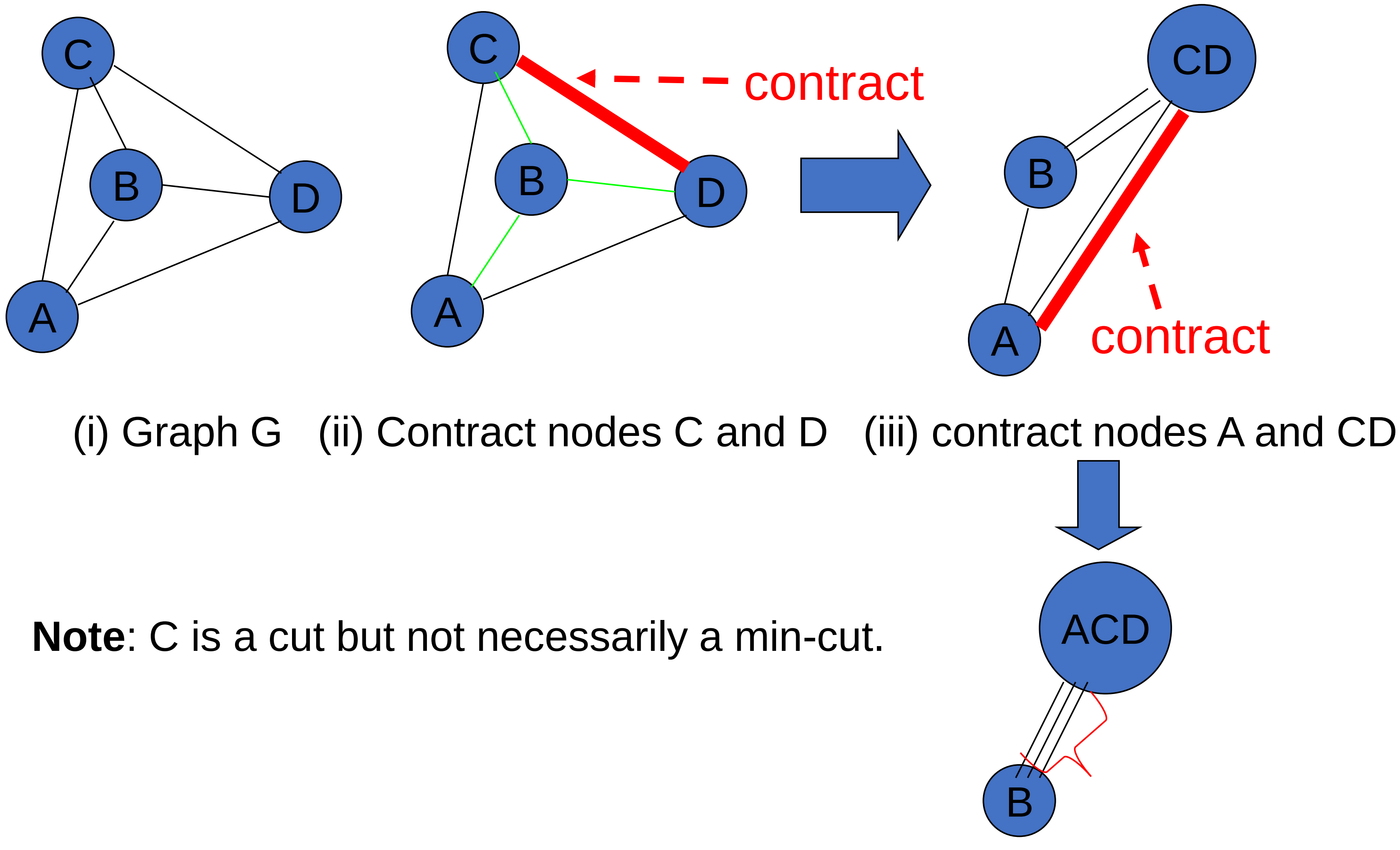
Note: There may be multi-edges between two vertices. We just keep them.



Graph G

Graph G'

Karger's Min-cut Algorithm



Note: C is a cut but not necessarily a min-cut.

(Iv) Cut C=*{(A,B), (B,C), (B,D)}*

Karger's Min-cut Algorithm

For i = 1 to $100n^2$

repeat

randomly pick an edge (u,v)

contract u and v

until two vertices are left

$c_i \leftarrow$ the number of edges between them

Output **min$_i$ c_i**

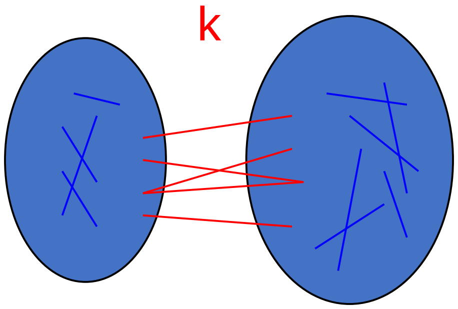
Key Idea

- Let *$C^* = \{c_1^*, c_2^*, \ldots, c_k^*\}$* be a min-cut in G and C^i be a cut determined by Karger's algorithm during some iteration i.
- C^i will be a min-cut for G if during iteration "i" none of the edges in C^* are contracted.
- If we can show that with *prob. $\Omega(1/n^2)$, where $n = |V|$, C^i will be a min-cut*, then by *repeatedly obtaining min-cuts $O(n^2)$ times* and taking minimum gives the min-cut with high prob.

Analysis of Karger's Min-Cut Algorithm

Analysis of Karger's Algorithm

Let k be the number of edges of min cut (S, V-S).

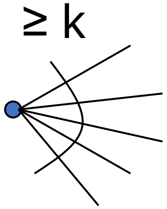


If we never picked a crossing edge in the

algorithm, then the number of edges between two last vertices is the correct answer.

The probability that in step 1 of an iteration a

crossing edge is not picked = (|E|-k)/|E|.



By def of min cut, we know that each vertex v has degree at least k, Otherwise the cut ({v}, V-{v}) is lighter.

Thus |E| ≥ nk/2 and (|E|-k)/|E| = 1 - k/|E| ≥ 1-2/n.

Analysis of Karger's Algorithm

- In step 1, Pr [no crossing edge picked] >= $1 - 2/n$
- Similarly, in step 2, Pr [no crossing edge picked] $\geq$ $1-2/(n-1)$
- In general, in step j, Pr [no crossing edge picked] $\geq$ $1-2/(n-j+1)$
- Pr {the n-2 contractions never contract a crossing edge}
  - = Pr [first step good]

    * Pr [second step good after surviving first step]

    * Pr [third step good after surviving first two steps]

    * …

    * Pr [(n-2)-th step good after surviving first n-3 steps]

    $\geq (1-2/n)\,(1-2/(n-1)) \ldots (1-2/3)$

    $= [(n-2)/n]\,[(n-3)(n-1)] \ldots [1/3] = 2/[n(n-1)] = \Omega(1/n^2)$