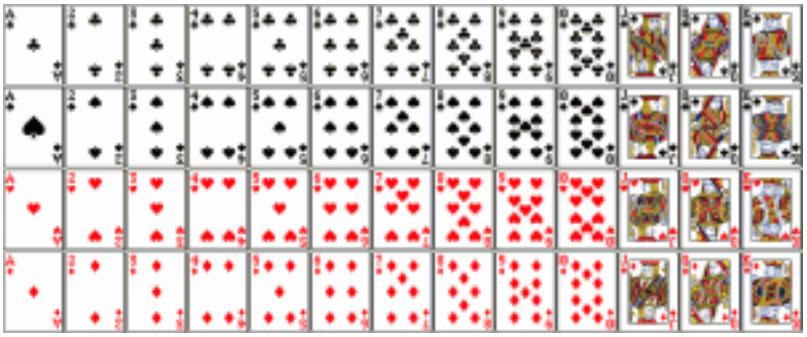
# Contents

# Chapter 1

# Shuffle a given array

Given an array, write a program to generate a random permutation of array elements. This question is also asked as "shuffle a deck of cards" or "randomize a given array".



Let the given array be *arr[]*. A simple solution is to create an auxiliary array *temp[]* which is initially a copy of *arr[]*. Randomly select an element from *temp[]*, copy the randomly selected element to *arr[0]* and remove the selected element from *temp[]*. Repeat the same process n times and keep copying elements to *arr[1], arr[2], … .* The time complexity of this solution will be O(n^2).

Fisher–Yates shuffle Algorithm works in O(n) time complexity. The assumption here is, we are given a function rand() that generates random number in O(1) time.
The idea is to start from the last element, swap it with a randomly selected element from the whole array (including last). Now consider the array from 0 to n-2 (size reduced by 1), and repeat the process till we hit the first element.

Following is the detailed algorithm

```
To shuffle an array a of n elements (indices 0..n-1):
  for i from n - 1 downto 1 do
       j = random integer with 0
Following is C++ implementation of this algorithm.

// C Program to shuffle a given array

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
// A utility function to swap to integers
void swap (int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// A utility function to print an array
void printArray (int arr[], int n)
{
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// A function to generate a random permutation of arr[]
void randomize ( int arr[], int n )
{
    // Use a different seed value so that we don't get same
    // result each time we run this program
    srand ( time(NULL) );

    // Start from the last element and swap one by one. We don't
    // need to run for the first element that's why i > 0
    for (int i = n-1; i > 0; i--)
    {
        // Pick a random index from 0 to i
        int j = rand() % (i+1);

        // Swap arr[i] with the element at random index
        swap(&arr[i], &arr[j]);
    }
}

// Driver program to test above function.
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
    int n = sizeof(arr)/ sizeof(arr[0]);
    randomize (arr, n);
    printArray(arr, n);

    return 0;
}
```

Output:

```
 7 8 4 6 3 1 2 5
```

The above function assumes that rand() generates a random number.

Time Complexity: O(n), assuming that the function rand() takes O(1) time.

**How does this work?**
The probability that ith element (including the last one) goes to last position is 1/n, because we randomly pick an element in first iteration.

The probability that ith element goes to second last position can be proved to be 1/n by dividing it in two cases.
*Case 1: i = n-1 (index of last element)*:
The probability of last element going to second last position is = (probability that last element doesn't stay at its original position) x (probability that the index picked in previous step is picked again so that the last element is swapped)
So the probability = ((n-1)/n) x (1/(n-1)) = 1/n
*Case 2: 0 :*
*The probability of ith element going to second position = (probability that ith element is not picked in previous iteration) x (probability that ith element is picked in this iteration)*
*So the probability = ((n-1)/n) x (1/(n-1)) = 1/n*

We can easily generalize above proof for any other position.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

http://www.geeksforgeeks.org/shuffle-a-given-array/

# Chapter 2

# Select a Random Node from a Singly Linked List

Given a singly linked list, select a random node from linked list (the probability of picking a node should be 1/N if there are N nodes in list). You are given a random number generator.

Below is a Simple Solution
1) Count number of nodes by traversing the list.
2) Traverse the list again and select every node with probability 1/N. The selection can be done by generating a random number from 0 to N-i for i'th node, and selecting the i'th node node only if generated number is equal to 0 (or any other fixed number from 0 to N-i).

We get uniform probabilities with above schemes.

```
i = 1, probability of selecting first node = 1/N
i = 2, probability of selecting second node =
                   [probability that first node is not selected] *
                   [probability that second node is selected]
                  = ((N-1)/N)* 1/(N-1)
                  = 1/N
```

Similarly, probabilities of other selecting other nodes is 1/N

The above solution requires two traversals of linked list.

**How to select a random node with only one traversal allowed?**
The idea is to use Reservoir Sampling. Following are the steps. This is a simpler version of Reservoir Sampling as we need to select only one key instead of k keys.

```
(1) Initialize result as first node
   result = head->key
(2) Initialize n = 2
(3) Now one by one consider all nodes from 2nd node onward.
    (3.a) Generate a random number from 0 to n-1.
         Let the generated random number is j.
    (3.b) If j is equal to 0 (we could choose other fixed number
          between 0 to n-1), then replace result with current node.
    (3.c) n = n+1
    (3.d) current = current->next
```

Below is C implementation of above algorithm.

```
/* C program to randomly select a node from a singly
   linked list */
#include<stdio.h>
#include<stdlib.h>
#include <time.h>

/* Link list node */
struct node
{
    int key;
    struct node* next;
};

// A reservoir sampling based function to print a
// random node from a linked list
void printRandom(struct node *head)
{
    // IF list is empty
    if (head == NULL)
       return;

    // Use a different seed value so that we don't get
    // same result each time we run this program
    srand(time(NULL));

    // Initialize result as first node
    int result = head->key;

    // Iterate from the (k+1)th element to nth element
    struct node *current = head;
    int n;
    for (n=2; current!=NULL; n++)
    {
        // change result with probability 1/n
        if (rand() % n == 0)
           result = current->key;

        // Move to next node
        current = current->next;
    }

    printf("Randomly selected key is %d\n", result);
}

/* BELOW FUNCTIONS ARE JUST UTILITY TO TEST  */

/* A utility function to create a new node */
struct node *newNode(int new_key)
{
    /* allocate node */
    struct node* new_node =
```

```
        (struct node*) malloc(sizeof(struct node));

    /* put in the key  */
    new_node->key  = new_key;
    new_node->next =  NULL;

    return new_node;
}


/* A utility function to insert a node at the beginning
  of linked list */
void push(struct node** head_ref, int new_key)
{
    /* allocate node */
    struct node* new_node = new node;

    /* put in the key  */
    new_node->key  = new_key;

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}


// Driver program to test above functions
int main()
{
    struct node *head = NULL;
    push(&head, 5);
    push(&head, 20);
    push(&head, 4);
    push(&head, 3);
    push(&head, 30);

    printRandom(head);

    return 0;
}


 Randomly selected key is 4
```

Note that the above program is based on outcome of a random function and may produce different output.

**How does this work?**
Let there be total N nodes in list. It is easier to understand from last node.

The probability that last node is result simply 1/N [For last or N'th node, we generate a random number between 0 to N-1 and make last node as result if the generated number is 0 (or any other fixed number]

The probability that second last node is result should also be 1/N.

```
The probability that the second last node is result
          = [Probability that the second last node replaces result] X
            [Probability that the last node doesn't replace the result]
          = [1 / (N-1)] * [(N-1)/N]
          = 1/N
```

Similarly we can show probability for 3rd last node and other nodes.

This article is contributed by **Rajeev**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

http://www.geeksforgeeks.org/select-a-random-node-from-a-singly-linked-list/

# Chapter 3

# Linearity of Expectation

This post is about a mathematical concept, but it covers one of the required topics to understand Randomized Algorithms.

Let us consider the following simple problem.

Given a fair dice with 6 faces, the dice is thrown n times, find expected value of sum of all results.

For example, if n = 2, there are total 36 possible outcomes.

```
(1, 1), (1, 2), .... (1, 6)
(2, 1), (2, 2), .... (2, 6)
.................
.................
(6, 1), (6, 2), ..... (6, 6)
```

**Expected value** of a discrete random variable is R defined as following. Suppose R can take value $r_1$ with probability $p_1$, value $r_2$ with probability $p_2$, and so on, up to value $r_k$ with probability $p_k$. Then the expectation of this random variable R is defined as

```
    E[R] = r1*p1 + r2*p2 + ... rk*pk
```

Let us calculate expected value for the above example.

```
Expected Value of sum = 2*1/36 + 3*1/6 + .... + 7*1/36 +
of two dice throws       3*1/36 + 4*1/6 + .... + 8*1/36 +
                         ........................
                         ........................
                         7*1/36 + 8*1/6 + .... + 12*1/36

                       =  7
```

The above way to solve the problem becomes difficult when there are more dice throws.

If we know about linearity of expectation, then we can quickly solve the above problem for any number of throws.

**Linearity of Expectation:** Let $R_1$ and $R_2$ be two discrete random variables on some probability space, then

```
E[R1 + R2] = E[R1] + E[R2]
```

Using the above formula, we can quickly solve the dice problem.

```
Expected Value of sum of 2 dice throws = 2*(Expected value of one dice throw)
                                       = 2*(1/6 + 2/6 + .... 6/6)
                                       = 2*7/2
                                       = 7

Expected value of sum for n dice throws is = n * 7/2 = 3.5 * n
```

Some interesting facts about Linearly of Expectation:
1) Linearity of expectation holds for both dependent and independent events. On the other hand the rule $E[R_1R_2] = E[R_1]*E[R_2]$ is true only for independent events.

2) Linearity of expectation holds for any number of random variables on some probability space. Let $R_1$, $R_2$, $R_3$, … $R_k$ be k random variables, then
$E[R_1 + R_2 + R_3 + … + R_k] = E[R_1] + E[R_2] + E[R_3] + … + E[R_k]$

**Another example that can be easily solved with linearity of expectation:**
Hat-Check Problem: Let there be group of n men where every man has one hat. The hats are redistributed and every man gets a random hat back. What is the expected number of men that get their original hat back.

Solution: Let $R_i$ be a random variable, the value of random variable is 1 if i'th man gets the same hat back, otherwise 0.

```
So the expected number of men to get the right hat back is
  = E[R1] + E[R2]  +  .. + E[Rn]
  = P(R1 = 1) + P(R2 = 1) + .... + P(Rn = 1)
  [Here P(Ri = 1)  indicates probability that Ri is 1]
  = 1/n + 1/n + ... + 1/n
  = 1
```

So on average 1 person gets the right hat back.

**Exercise:**
1) Given a fair coin, what is the expected number of heads when coin is tossed n times.

2) Balls and Bins: Suppose we have m balls, labeled i = 1, … , m and n bins, labeled j = 1, .. ,n. Each ball is thrown into one of the bin independently and uniformly at random.
a) What is the expected number of balls in every bin
b) What is the expected number of empty bins.

3) Coupon Collector: Suppose there are n types of coupons in a lottery and each lot contains one coupon (with probability 1 = n each). How many lots have to be bought (in expectation) until we have at least one coupon of each type.

See following for solution of Coupon Collector.

Expected Number of Trials until Success

Linearity of expectation is useful in algorithms. For example, expected time complexity of random algorithms like randomized quick sort is evaluated using linearity of expectation (See thisfor reference).

**References:**

http://www.cse.iitd.ac.in/~mohanty/col106/Resources/linearity_expectation.pdf

http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-042j-mathematics-for-computer-science-fall-2010
video-lectures/lecture-22-expectation-i/

This article is contributed by **Shubham Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

http://www.geeksforgeeks.org/linearity-of-expectation/

# Chapter 4

# Expected Number of Trials until Success

Consider the following famous puzzle.

*In a country, all families want a boy. They keep having babies till a boy is born. What is the expected ratio of boys and girls in the country?*

This puzzle can be easily solved if we know following interesting result in probability and expectation.

**If probability of success is p in every trial, then expected number of trials until success is 1/p**

**Proof:** Let R be a random variable that indicates number of trials until success.

```
The expected value of R is sum of following infinite series
E[R] = 1*p + 2*(1-p)*p + 3*(1-p)2*p + 4*(1-p)3*p + ........

Taking 'p' out
E[R] = p[1 + 2*(1-p) + 3*(1-p)2 + 4*(1-p)3 + .......] ---->(1)

Multiplying both sides with '(1-p)' and subtracting
(1-p)*E[R] = p[1*(1-p) + 2*(1-p)2 + 3*(1-p)3 + .......] --->(2)

Subtracting (2) from (1), we get

p*E[R] = p[1 + (1-p) + (1-p)2 + (1-p)3 + ........]

Canceling p from both sides
E[R] = [1 + (1-p) + (1-p)2 + (1-p)3 + ........]

Above is an  infinite geometric  progression with ratio (1-p).
Since (1-p) is less than, we can apply sum formula.
  E[R] = 1/[1 - (1-p)]
       = 1/p
```

***Solution of Boys/Girls ratio puzzle:***
Let us use the above result to solve the puzzle. In the given puzzle, probability of success in every trial is 1/2 (assuming that girls and boys are equally likely).

```
Let p be probability of having a baby boy.
Number of kids until a baby boy is born = 1/p
                                        = 1/(1/2)
                                        = 2
Since expected number of kids in a family is 2,
ratio of boys and girls is 50:50.
```

Let us discuss another problem that uses above result.

**Coupon Collector Problem:**
*Suppose there are n types of coupons in a lottery and each lot contains one coupon (with probability 1 = n each). How many lots have to be bought (in expectation) until we have at least one coupon of each type.*

The solution of this problem is also based on above result.

Let $X_i$ be the number of lots bought before i'th new coupon is collected.

Note that $X_1$ is 1 as the first coupon is always a new coupon (not collected before).

Let 'p' be probability that 2nd coupon is collected in next buy. The value of p is (n-1)/n. So the number of trials needed before 2nd new coupon is picked is 1/p which means n/(n-1). [This is where we use above result]

Similarly, the number of trials needed before 3rd new coupon is collected is n/(n-2)

```
Using Linearity of expectation,
we can say that the total number of expected trials =
                  1 + n/(n-1) + n/(n-2) + n/(n-3) + .... + n/2 + n/1
               = n[1/n + 1/(n-1) + 1/(n-2) + 1/(n-3) + ....+ 1/2 + 1/1]
               = n * Hn
Here Hn is n-th Harmonic number

Since Logn n
Exercise:

1) A 6 faced fair dice is thrown until a '5' is seen as result of dice throw. What is the expected number of th
2) What is the ratio of boys and girls in above puzzle if probability of a baby boy is 1/3?
Reference:

http://www.cse.iitd.ac.in/~mohanty/col106/Resources/linearity_expectation.pdf
http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-042j-mathematics-for-computer-s
Please write comments if you find anything incorrect, or you want to share more information about the topic di
```

## Source

http://www.geeksforgeeks.org/expected-number-of-trials-before-success/

# Chapter 5

# Reservoir Sampling

Reservoir sampling is a family of randomized algorithms for randomly choosing $k$ samples from a list of $n$ items, where $n$ is either a very large or unknown number. Typically $n$ is large enough that the list doesn't fit into main memory. For example, a list of search queries in Google and Facebook.

So we are given a big array (or stream) of numbers (to simplify), and we need to write an efficient function to randomly select $k$ numbers where *1 . Let the input array be* stream[].

A **simple solution** is to create an array *reservoir[]* of maximum size $k$. One by one randomly select an item from *stream[0..n-1]*. If the selected item is not previously selected, then put it in *reservoir[]*. To check if an item is previously selected or not, we need to search the item in *reservoir[]*. The time complexity of this algorithm will be *O(k^2)*. This can be costly if $k$ is big. Also, this is not efficient if the input is in the form of a stream.

It **can be solved in *O(n)* time**. The solution also suits well for input in the form of stream. The idea is similar to thispost. Following are the steps.

**1)** Create an array *reservoir[0..k-1]* and copy first $k$ items of *stream[]* to it.
**2)** Now one by one consider all items from *(k+1)*th item to $n$th item.
...**a)** Generate a random number from 0 to $i$ where $i$ is index of current item in *stream[]*. Let the generated random number is $j$.
...**b)** If $j$ is in range 0 to *k-1*, replace *reservoir[j]* with *arr[i]*

Following is C implementation of the above algorithm.

```
// An efficient program to randomly select k items from a stream of items

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// A utility function to print an array
void printArray(int stream[], int n)
{
    for (int i = 0; i < n; i++)
        printf("%d ", stream[i]);
    printf("\n");
}

// A function to randomly select k items from stream[0..n-1].
```

```
void selectKItems(int stream[], int n, int k)
{
    int i;  // index for elements in stream[]

    // reservoir[] is the output array. Initialize it with
    // first k elements from stream[]
    int reservoir[k];
    for (i = 0; i < k; i++)
        reservoir[i] = stream[i];

    // Use a different seed value so that we don't get
    // same result each time we run this program
    srand(time(NULL));

    // Iterate from the (k+1)th element to nth element
    for (; i < n; i++)
    {
        // Pick a random index from 0 to i.
        int j = rand() % (i+1);

        // If the randomly  picked index is smaller than k, then replace
        // the element present at the index with new element from stream
        if (j < k)
          reservoir[j] = stream[i];
    }

    printf("Following are k randomly selected items \n");
    printArray(reservoir, k);
}

// Driver program to test above function.
int main()
{
    int stream[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
    int n = sizeof(stream)/sizeof(stream[0]);
    int k = 5;
    selectKItems(stream, n, k);
    return 0;
}
```

Output:

```
  Following are k randomly selected items
6 2 11 8 12
```

Time Complexity: O(n)

**How does this work?**
To prove that this solution works perfectly, we must prove that the probability that any item *stream[i]* where *0 will be in final* reservoir[] *is* k/n. *Let us divide the proof in two cases as first* k *items are treated differently.*

**Case 1: For last *n-k* stream items, i.e., for *stream[i]* where *k***
For every such stream item *stream[i]*, we pick a random index from 0 to $i$ and if the picked index is one of the first $k$ indexes, we replace the element at picked index with *stream[i]*

To simplify the proof, let us first consider the *last item*. The probability that the last item is in final reservoir = The probability that one of the first $k$ indexes is picked for last item = $k/n$ (the probability of picking one of the $k$ items from a list of size $n$)

Let us now consider the *second last item*. The probability that the second last item is in final *reservoir[]* = [Probability that one of the first $k$ indexes is picked in iteration for *stream[n-2]*] X [Probability that the index picked in iteration for *stream[n-1]* is not same as index picked for *stream[n-2]* ] = *[k/(n-1)]*[(n-1)/n]* = *k/n*.

Similarly, we can consider other items for all stream items from *stream[n-1]* to *stream[k]* and generalize the proof.

**Case 2: For first *k* stream items, i.e., for *stream[i]* where *0***
The first $k$ items are initially copied to *reservoir[]* and may be removed later in iterations for *stream[k]* to *stream[n]*.
The probability that an item from *stream[0..k-1]* is in final array = Probability that the item is not picked when items *stream[k], stream[k+1], …. stream[n-1]* are considered = *[k/(k+1)] x [(k+1)/(k+2)] x [(k+2)/(k+3)] x … x [(n-1)/n] = k/n*

References:
http://en.wikipedia.org/wiki/Reservoir_sampling

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

http://www.geeksforgeeks.org/reservoir-sampling/

# Chapter 6

# K'th Smallest/Largest Element in Unsorted Array | Set 2 (Expected Linear Time)

We recommend to read following post as a prerequisite of this post.

K'th Smallest/Largest Element in Unsorted Array | Set 1

Given an array and a number k where k is smaller than size of array, we need to find the k'th smallest element in the given array. It is given that ll array elements are distinct.

Examples:

```
 Input: arr[] = {7, 10, 4, 3, 20, 15}
       k = 3
Output: 7

Input: arr[] = {7, 10, 4, 3, 20, 15}
       k = 4
Output: 10
```

We have discussed three different solutions here.

In this post method 4 is discussed which is mainly an extension of method 3 (QuickSelect) discussed in the previouspost. The idea is to randomly pick a pivot element. To implement randomized partition, we use a random function, rand() to generate index between l and r, swap the element at randomly generated index with the last element, and finally call the standard partition process which uses last element as pivot.

Following is C++ implementation of above Randomized QuickSelect.

```
// C++ implementation of randomized quickSelect
#include<iostream>
#include<climits>
#include<cstdlib>
using namespace std;

int randomPartition(int arr[], int l, int r);
```

```
// This function returns k'th smallest element in arr[l..r] using
// QuickSort based method.  ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int kthSmallest(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k > 0 && k <= r - l + 1)
    {
        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = randomPartition(arr, l, r);

        // If position is same as k
        if (pos-l == k-1)
            return arr[pos];
        if (pos-l > k-1)  // If position is more, recur for left subarray
            return kthSmallest(arr, l, pos-1, k);

        // Else recur for right subarray
        return kthSmallest(arr, pos+1, r, k-pos+l-1);
    }

    // If k is more than number of elements in array
    return INT_MAX;
}

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Standard partition process of QuickSort().  It considers the last
// element as pivot and moves all smaller element to left of it and
// greater elements to right. This function is used by randomPartition()
int partition(int arr[], int l, int r)
{
    int x = arr[r], i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[r]);
    return i;
}

// Picks a random pivot element between l and r and partitions
// arr[l..r] arount the randomly picked element using partition()
int randomPartition(int arr[], int l, int r)
```

```
{
    int n = r-l+1;
    int pivot = rand() % n;
    swap(&arr[l + pivot], &arr[r]);
    return partition(arr, l, r);
}

// Driver program to test above methods
int main()
{
    int arr[] = {12, 3, 5, 7, 4, 19, 26};
    int n = sizeof(arr)/sizeof(arr[0]), k = 3;
    cout << "K'th smallest element is " << kthSmallest(arr, 0, n-1, k);
    return 0;
}
```

Output:

```
K'th smallest element is 5
```

**Time Complexity:**
The worst case time complexity of the above solution is still $O(n^2)$. In worst case, the randomized function may always pick a corner element. The expected time complexity of above randomized QuickSelect is $\Theta(n)$, see CLRS book or MIT video lecture for proof. The assumption in the analysis is, random number generator is equally likely to generate any number in the input range.

**Sources:**

MIT Video Lecture on Order Statistics, Median

Introduction to Algorithms by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L.

This article is contributed by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

http://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-2-expected-linear-time/

Category: Arrays

Post navigation

← Amazon Interview Experience | Set 151 (For SDE) Amazon Interview Experience | Set 152 →

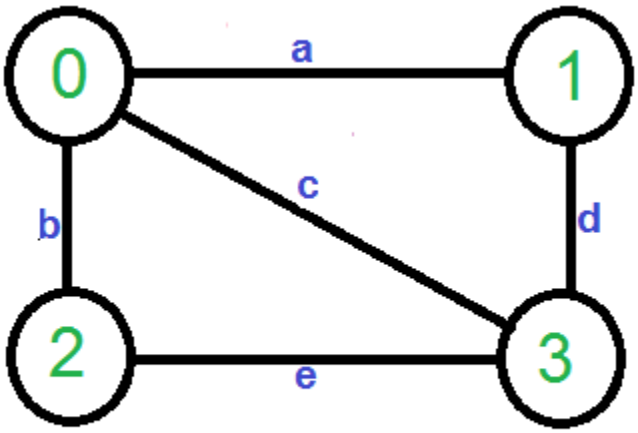
Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

# Chapter 7

# Karger's algorithm for Minimum Cut | Set 1 (Introduction and Implementation)

Given an undirected and unweighted graph, find the smallest cut (smallest number of edges that disconnects the graph into two components).
The input graph may have parallel edges.

For example consider the following example, the smallest cut has 2 edges.



Min-Cut for above graph is either **{a, d}** OR **{b, e}**

A Simple Solution use Max-Flow based s-t cut algorithm to find minimum cut. Consider every pair of vertices as source 's' and sink 't', and call minimum s-t cut algorithm to find the s-t cut. Return minimum of all s-t cuts. Best possible time complexity of this algorithm is $O(V^5)$ for a graph. [How? there are total possible $V^2$ pairs and s-t cut algorithm for one pair takes $O(V*E)$ time and $E = O(V^2)$].

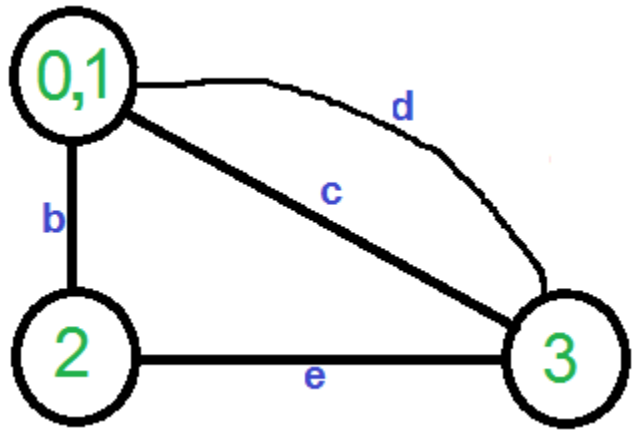
Below is simple Karger's Algorithm for this purpose. Below Karger's algorithm can be implemented in $O(E) = O(V^2)$ time.

```
1)  Initialize contracted graph CG as copy of original graph
```

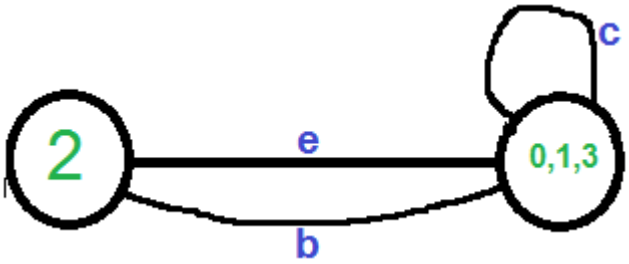
```
2)  While there are more than 2 vertices.
      a) Pick a random edge (u, v) in the contracted graph.
      b) Merge (or contract) u and v into a single vertex (update
         the contracted graph).
      c) Remove self-loops
3) Return cut represented by two vertices.
```

Let us understand above algorithm through the example given.

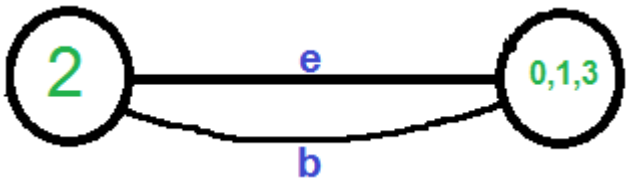
Let the first randomly picked vertex be '**a**' which connects vertices 0 and 1. We remove this edge and contract the graph (combine vertices 0 and 1). We get the following graph.




Let the next randomly picked edge be 'd'. We remove this edge and combine vertices (0,1) and 3.

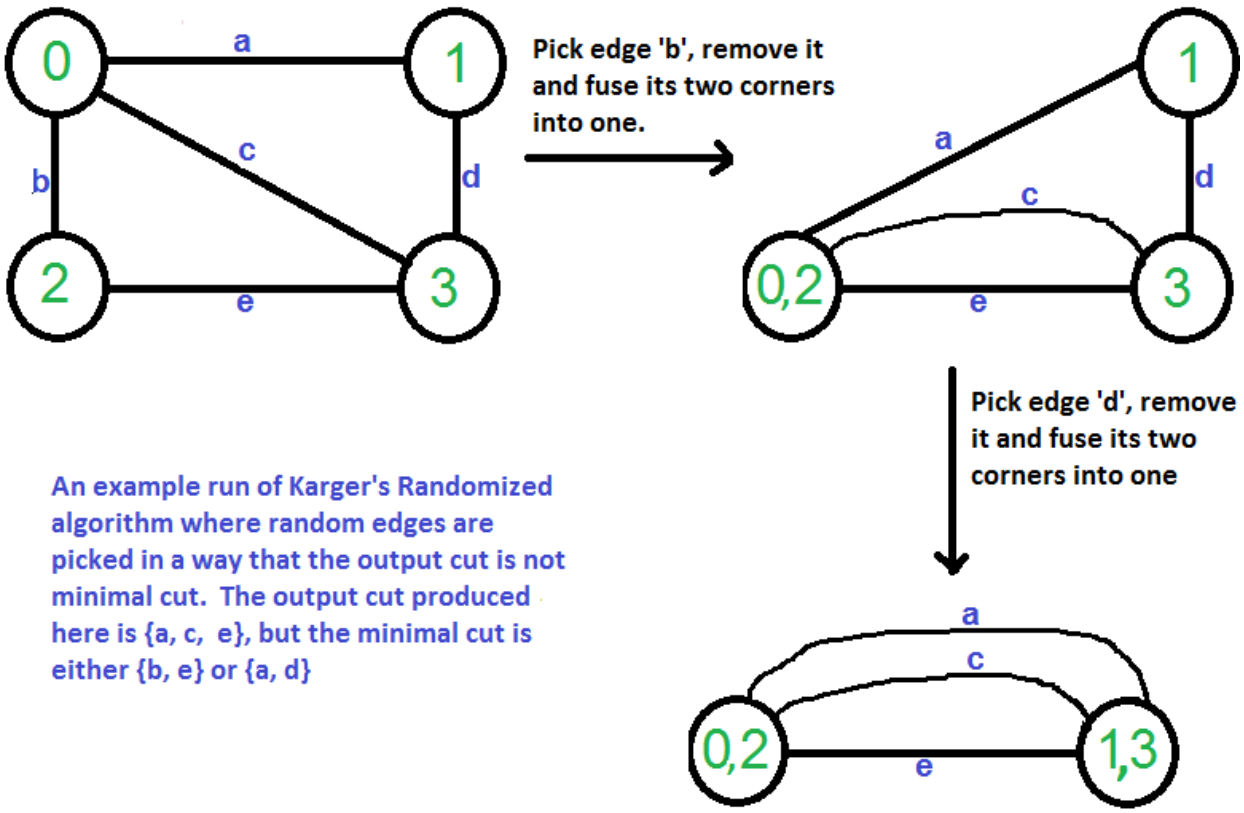



We need to remove self-loops in the graph. So we remove edge 'c'




Now graph has two vertices, so we stop. The number of edges in the resultant graph is the cut produced by Karger's algorithm.

***Karger's algorithm is a Monte Carlo algorithm and cut produced by it may not be minimum.*** For example, the following diagram shows that a different order of picking random edges produces a min-cut of size 3.




Below is C++ implementation of above algorithm. The input graph is represented as a collection of edges andunion-find data structure is used to keep track of components.

```
// Karger's algorithm to find Minimum Cut in an
// undirected, unweighted and connected graph.
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// a structure to represent a unweighted edge in graph
struct Edge
{
    int src, dest;
};

// a structure to represent a connected, undirected
// and unweighted graph as a collection of edges.
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    // Since the graph is undirected, the edge
    // from src to dest is also edge from dest
    // to src. Both are counted as 1 edge here.
    Edge* edge;
```

```
};

// A structure to represent a subset for union-find
struct subset
{
    int parent;
    int rank;
};

// Function prototypes for union-find (These functions are defined
// after kargerMinCut() )
int find(struct subset subsets[], int i);
void Union(struct subset subsets[], int x, int y);

// A very basic implementation of Karger's randomized
// algorithm for finding the minimum cut. Please note
// that Karger's algorithm is a Monte Carlo Randomized algo
// and the cut returned by the algorithm may not be
// minimum always
int kargerMinCut(struct Graph* graph)
{
    // Get data of given graph
    int V = graph->V, E = graph->E;
    Edge *edge = graph->edge;

    // Allocate memory for creating V subsets.
    struct subset *subsets = new subset[V];

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Initially there are V vertices in
    // contracted graph
    int vertices = V;

    // Keep contracting vertices until there are
    // 2 vertices.
    while (vertices > 2)
    {
       // Pick a random edge
       int i = rand() % E;

       // Find vertices (or sets) of two corners
       // of current edge
       int subset1 = find(subsets, edge[i].src);
       int subset2 = find(subsets, edge[i].dest);

       // If two corners belong to same subset,
       // then no point considering this edge
       if (subset1 == subset2)
```

```
          continue;

        // Else contract the edge (or combine the
        // corners of edge into one vertex)
        else
        {
           printf("Contracting edge %d-%d\n",
                  edge[i].src, edge[i].dest);
           vertices--;
           Union(subsets, subset1, subset2);
        }
    }

    // Now we have two vertices (or subsets) left in
    // the contracted graph, so count the edges between
    // two components and return the count.
    int cutedges = 0;
    for (int i=0; i<E; i++)
    {
        int subset1 = find(subsets, edge[i].src);
        int subset2 = find(subsets, edge[i].dest);
        if (subset1 != subset2)
          cutedges++;
    }

    return cutedges;
}

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i
    // (path compression)
    if (subsets[i].parent != i)
      subsets[i].parent =
             find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high
    // rank tree (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
```

```
    // If ranks are same, then make one as root and
    // increment its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

// Driver program to test above functions
int main()
{
    /* Let us create following unweighted graph
        0------1
        | \    |
        |   \  |
        |     \|
        2------3   */
    int V = 4;  // Number of vertices in graph
    int E = 5;  // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;

    // add edge 0-2
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;

    // add edge 0-3
    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;

    // add edge 1-3
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;

    // add edge 2-3
    graph->edge[4].src = 2;
    graph->edge[4].dest = 3;

    // Use a different seed value for every run.
```

```
    srand(time(NULL));

    printf("\nCut found by Karger's randomized algo is %d\n",
           kargerMinCut(graph));

    return 0;
}
```

Output:

```
 Contracting edge 0-2
Contracting edge 0-3

Cut found by Karger's randomized algo is 2
```

***Note that the above program is based on outcome of a random function and may produce different output.***

In this post, we have discussed simple Karger's algorithm and have seen that the algorithm doesn't always produce min-cut. The above algorithm produces min-cut with probability greater or equal to that $1/(n^2)$. In the next post, we will discuss proof of this and how repeated calls of Karger's algorithm can improve this probability. We will also discuss applications of Karger's algorithm.

**References:**
http://en.wikipedia.org/wiki/Karger%27s_algorithm

https://www.youtube.com/watch?v=P0l8jMDQTEQ

https://www.cs.princeton.edu/courses/archive/fall13/cos521/lecnotes/lec2final.pdf

http://web.stanford.edu/class/archive/cs/cs161/cs161.1138/lectures/11/Small11.pdf

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

http://www.geeksforgeeks.org/kargers-algorithm-for-minimum-cut-set-1-introduction-and-implementation/