# Random Probability

Calculate random probability is another common pattern in math problem. The following are the examples:

## 382. Linked List Random Node

Medium

Given a singly linked list, return a random node's value from the linked list. Each node must have the **same probability** of being chosen.

**Follow up:**  
What if the linked list is extremely large and its length is unknown to you? Could you solve this efficiently without using extra space?

**Example:**

// Init a singly linked list [1,2,3].

ListNode head = new ListNode(1);

head.next = new ListNode(2);

head.next.next = new ListNode(3);

Solution solution = new Solution(head);

// getRandom() should return either 1, 2, or 3 randomly. Each element should have equal probability of returning.

solution.getRandom();

### Analysis:

The key point of this problem is that you do not know the size of the nodes in the very beginning, so you can assume the first node probability is 1/1, and the second one is 1/2 and the third one is 1/3, if the later one is chosen it will override the early ones.

/// <summary>

/// Leet code #382. Linked List Random Node

///

/// Given a singly linked list, return a random node's value from the linked

/// list.

/// Each node must have the same probability of being chosen.

/// Follow up:

/// What if the linked list is extremely large and its length is unknown to

/// you? Could you solve this efficiently without using extra space?

///

/// Example:

/// Init a singly linked list [1,2,3].

/// ListNode head = new ListNode(1);

/// head.next = new ListNode(2);

/// head.next.next = new ListNode(3);

/// Solution solution = new Solution(head);

/// </summary>

int LeetCodeLinkedList::getRandom(ListNode \*head)

{

int count = 0;

int value = 0;

while (head != nullptr)

{

count++;

if (count == 1)

{

value = head->val;

}

else

{

if (rand() % count == 0)

{

value = head->val;

}

}

head = head->next;

}

return value;

}

## 384. Shuffle an Array

Medium

Shuffle a set of numbers without duplicates.

**Example:**

// Init an array with set 1, 2, and 3.

int[] nums = {1,2,3};

Solution solution = new Solution(nums);

// Shuffle the array [1,2,3] and return its result. Any permutation of [1,2,3] must equally likely to be returned.

solution.shuffle();

// Resets the array back to its original configuration [1,2,3].

solution.reset();

// Returns the random shuffling of array [1,2,3].

solution.shuffle();

### Analysis:

For each card you randomly select which card to swap.

/// <summary>

/// Leet code #384. Shuffle an Array

/// Shuffle a set of numbers without duplicates.

/// Example:

/// Init an array with set 1, 2, and 3.

/// int[] nums = {1,2,3};

/// Solution solution = new Solution(nums);

///

/// Shuffle the array [1,2,3] and return its result. Any permutation of

/// [1,2,3] must equally likely to be returned solution.shuffle();

///

/// Resets the array back to its original configuration [1,2,3].

/// solution.reset();

/// Returns the random shuffling of array [1,2,3].

/// solution.shuffle();

/// </summary>

vector<int> LeetCode::shuffle(vector<int> nums)

{

for (size\_t i = 0; i < nums.size(); i++) {

int pos = rand() % (nums.size());

swap(nums[pos], nums[i]);

}

return nums;

}

## 478. Generate Random Point in a Circle

Medium

Given the radius and x-y positions of the center of a circle, write a function randPoint which generates a uniform random point in the circle.

Note:

1. input and output values are in [floating-point](https://www.webopedia.com/TERM/F/floating_point_number.html).
2. radius and x-y position of the center of the circle is passed into the class constructor.
3. a point on the circumference of the circle is considered to be in the circle.
4. randPoint returns a size 2 array containing x-position and y-position of the random point, in that order.

**Example 1:**

**Input:**

["Solution","randPoint","randPoint","randPoint"]

[[1,0,0],[],[],[]]

**Output:** [null,[-0.72939,-0.65505],[-0.78502,-0.28626],[-0.83119,-0.19803]]

**Example 2:**

**Input:**

["Solution","randPoint","randPoint","randPoint"]

[[10,5,-7.5],[],[],[]]

**Output:** [null,[11.52438,-8.33273],[2.46992,-16.21705],[11.13430,-12.42337]]

**Explanation of Input Syntax:**

The input is two lists: the subroutines called and their arguments. Solution's constructor has three arguments, the radius, x-position of the center, and y-position of the center of the circle. randPoint has no arguments. Arguments are always wrapped with a list, even if there aren't any.

### Analysis:

First you pick the random degree from 0 to 360, then pick the distance from center, remember the area is the square of radius, the far from center the more probability.

/// <summary>

/// Leet code #478. Generate Random Point in a Circle

///

/// Given the radius and x-y positions of the center of a circle, write

/// a function randPoint which generates a uniform random point in the

/// circle.

///

/// Note:

///

/// 1. input and output values are in floating-point.

/// 2. radius and x-y position of the center of the circle is passed into

/// the class constructor.

/// 3. a point on the circumference of the circle is considered to be in

/// the circle.

/// 4. randPoint returns a size 2 array containing x-position and

/// y-position of the random point, in that order.

///

/// Example 1:

/// Input:

/// ["Solution","randPoint","randPoint","randPoint"]

/// [[1,0,0],[],[],[]]

///

/// Output: [null,[-0.72939,-0.65505],[-0.78502,-0.28626],

/// [-0.83119,-0.19803]]

///

/// Example 2:

/// Input:

/// ["Solution","randPoint","randPoint","randPoint"]

/// [[10,5,-7.5],[],[],[]]

/// Output: [null,[11.52438,-8.33273],[2.46992,-16.21705],

/// [11.13430,-12.42337]]

/// Explanation of Input Syntax:

///

/// The input is two lists: the subroutines called and their arguments.

/// Solution's constructor has three arguments, the radius, x-position

/// of the center, and y-position of the center of the circle. randPoint

/// has no arguments. Arguments are always wrapped with a list, even if

/// there aren't any.

/// </summary>

class RandomCirclePoint

{

private:

double m\_radius;

double m\_x\_center;

double m\_y\_center;

const double pi = 3.1415926;

public:

RandomCirclePoint(double radius, double x\_center, double y\_center)

{

m\_radius = radius;

m\_x\_center = x\_center;

m\_y\_center = y\_center;

srand((unsigned int)time(0));

}

vector<double> randPoint()

{

double r = m\_radius \* sqrt(((double)rand() / (double)(RAND\_MAX)));

double angle = 2 \* pi \* ((double)rand() / (double)(RAND\_MAX));

double x = r \* cos(angle) + m\_x\_center;

double y = r \* sin(angle) + m\_y\_center;

return vector<double> {x, y};

}

};

## 528. Random Pick with Weight

Medium

Given an array w of positive integers, where w[i] describes the weight of index i, write a function pickIndex which randomly picks an index in proportion to its weight.

Note:

1. 1 <= w.length <= 10000
2. 1 <= w[i] <= 10^5
3. pickIndex will be called at most 10000 times.

**Example 1:**

**Input:**

["Solution","pickIndex"]

[[[1]],[]]

**Output:** [null,0]

**Example 2:**

**Input:**

["Solution","pickIndex","pickIndex","pickIndex","pickIndex","pickIndex"]

[[[1,3]],[],[],[],[],[]]

**Output:** [null,0,1,1,1,0]

**Explanation of Input Syntax:**

The input is two lists: the subroutines called and their arguments. Solution's constructor has one argument, the array w. pickIndex has no arguments. Arguments are always wrapped with a list, even if there aren't any.

### Analysis:

Calculate prefix accumulated sum for each number and do random within total count and do binary search.

/// <summary>

/// Leet code #528. Random Pick with Weight

///

/// Given an array w of positive integers, where w[i] describes the weight

/// of index i, write a function pickIndex which randomly picks an index

/// in proportion to its weight.

///

/// Note:

/// 1. 1 <= w.length <= 10000

/// 2. 1 <= w[i] <= 10^5

/// 3. pickIndex will be called at most 10000 times.

///

/// Example 1:

/// Input:

/// ["Solution","pickIndex"]

/// [[[1]],[]]

/// Output: [null,0]

///

/// Example 2:

///

/// Input:

/// ["Solution","pickIndex","pickIndex","pickIndex","pickIndex",

/// "pickIndex"]

/// [[[1,3]],[],[],[],[],[]]

/// Output: [null,0,1,1,1,0]

///

/// Explanation of Input Syntax:

///

/// The input is two lists: the subroutines called and their arguments.

/// Solution's constructor has one argument, the array w. pickIndex has no

/// arguments. Arguments are always wrapped with a list, even if there

/// aren't any.

/// </summary>

class RandomWithWeight

{

private:

vector<int> sum;

public:

RandomWithWeight(vector<int> w)

{

sum = vector<int>(w.size());

for (size\_t i = 0; i < w.size(); i++)

{

sum[i] = (i > 0) ? sum[i - 1] + w[i] : w[i];

}

srand((unsigned int)time(0));

}

int pickIndex()

{

int value = rand() % sum[sum.size() - 1] + 1;

auto itr = lower\_bound(sum.begin(), sum.end(), value);

int index = itr - sum.begin();

return index;

}

};

## 497. Random Point in Non-overlapping Rectangles

Medium

Given a list of **non-overlapping** axis-aligned rectangles rects, write a function pick which randomly and uniformily picks an **integer point** in the space covered by the rectangles.

Note:

1. An **integer point** is a point that has integer coordinates.
2. A point on the perimeter of a rectangle is **included** in the space covered by the rectangles.
3. ith rectangle = rects[i] = [x1,y1,x2,y2], where [x1, y1] are the integer coordinates of the bottom-left corner, and [x2, y2] are the integer coordinates of the top-right corner.
4. length and width of each rectangle does not exceed 2000.
5. 1 <= rects.length <= 100
6. pick return a point as an array of integer coordinates [p\_x, p\_y]
7. pick is called at most 10000 times.

**Example 1:**

**Input:**

["Solution","pick","pick","pick"]

[[[[1,1,5,5]]],[],[],[]]

**Output:**

[null,[4,1],[4,1],[3,3]]

**Example 2:**

**Input:**

["Solution","pick","pick","pick","pick","pick"]

[[[[-2,-2,-1,-1],[1,0,3,0]]],[],[],[],[],[]]

**Output:**

[null,[-1,-2],[2,0],[-2,-1],[3,0],[-2,-2]]

**Explanation of Input Syntax:**

The input is two lists: the subroutines called and their arguments. Solution's constructor has one argument, the array of rectangles rects. pick has no arguments. Arguments are always wrapped with a list, even if there aren't any.

### Analysis:

The probability for the point to locate in each rectangle depends on their area, and then divide by widths to get the y position and the remainder is the x position.

/// <summary>

/// Leet code #497. Random Point in Non-overlapping Rectangles

///

/// Given a list of non-overlapping axis-aligned rectangles rects, write

/// a function pick which randomly and uniformily picks an integer point

/// in the space covered by the rectangles.

///

/// Note:

///

/// An integer point is a point that has integer coordinates.

/// A point on the perimeter of a rectangle is included in the space

/// covered by the rectangles.

/// ith rectangle = rects[i] = [x1,y1,x2,y2], where [x1, y1] are the

/// integer coordinates of the bottom-left corner, and [x2, y2] are the

/// integer coordinates of the top-right corner.

/// length and width of each rectangle does not exceed 2000.

/// 1 <= rects.length <= 100

/// pick return a point as an array of integer coordinates [p\_x, p\_y]

/// pick is called at most 10000 times.

/// Example 1:

///

/// Input:

/// ["Solution","pick","pick","pick"]

/// [[[[1,1,5,5]]],[],[],[]]

/// Output:

/// [null,[4,1],[4,1],[3,3]]

/// Example 2:

///

/// Input:

/// ["Solution","pick","pick","pick","pick","pick"]

/// [[[[-2,-2,-1,-1],[1,0,3,0]]],[],[],[],[],[]]

/// Output:

/// [null,[-1,-2],[2,0],[-2,-1],[3,0],[-2,-2]]

/// Explanation of Input Syntax:

///

/// The input is two lists: the subroutines called and their arguments.

/// Solution's constructor has one argument, the array of rectangles

/// rects. pick has no arguments. Arguments are always wrapped with a

/// list, even if there aren't any.

/// </summary>

class RandomRectanglePoint

{

private:

vector<vector<int>> m\_rectangles;

vector<int> m\_sum;

public:

RandomRectanglePoint(vector<vector<int>> rects)

{

for (size\_t i = 0; i < rects.size(); i++)

{

m\_rectangles.push\_back(rects[i]);

int area = (rects[i][2] - rects[i][0] + 1) \* (rects[i][3] - rects[i][1] + 1);

if (m\_sum.empty())

{

// we start from 0, so it is a point

m\_sum.push\_back(area);

}

else

{

m\_sum.push\_back(m\_sum.back() + area);

}

}

srand((unsigned int)time(0));

}

vector<int> pick()

{

vector<int> result = { 0, 0 };

if (m\_sum.empty()) return result;

int rand\_num = rand() % m\_sum.back() + 1;

int index = lower\_bound(m\_sum.begin(), m\_sum.end(), rand\_num) - m\_sum.begin();

int remaining = (index == 0 ? rand\_num : rand\_num - m\_sum[index - 1]);

int x = m\_rectangles[index][0] + (remaining - 1) % (m\_rectangles[index][2] - m\_rectangles[index][0] + 1);

int y = m\_rectangles[index][1] + (remaining - 1) / (m\_rectangles[index][2] - m\_rectangles[index][0] + 1);

result = { x, y };

return result;

}

};

## 519. Random Flip Matrix

Medium

You are given the number of rows n\_rows and number of columns n\_cols of a 2D binary matrix where all values are initially 0. Write a function flip which chooses a 0 value [uniformly at random](https://en.wikipedia.org/wiki/Discrete_uniform_distribution), changes it to 1, and then returns the position [row.id, col.id] of that value. Also, write a function reset which sets all values back to 0. **Try to minimize the number of calls to system's Math.random()** and optimize the time and space complexity.

Note:

1. 1 <= n\_rows, n\_cols <= 10000
2. 0 <= row.id < n\_rows and 0 <= col.id < n\_cols
3. flip will not be called when the matrix has no 0 values left.
4. the total number of calls to flip and reset will not exceed 1000.

**Example 1:**

**Input:**

["Solution","flip","flip","flip","flip"]

[[2,3],[],[],[],[]]

**Output:** [null,[0,1],[1,2],[1,0],[1,1]]

**Example 2:**

**Input:**

["Solution","flip","flip","reset","flip"]

[[1,2],[],[],[],[]]

**Output:** [null,[0,0],[0,1],null,[0,0]]

**Explanation of Input Syntax:**

The input is two lists: the subroutines called and their arguments. Solution's constructor has two arguments, n\_rows and n\_cols. flip and reset have no arguments. Arguments are always wrapped with a list, even if there aren't any.

### Analysis:

We count total number unset cells, and count the unset cells in each row.

/// <summary>

/// Leet code #519. Random Flip Matrix

///

/// You are given the number of rows n\_rows and number of columns n\_cols

/// of a 2D binary matrix where all values are initially 0. Write a

/// function flip which chooses a 0 value uniformly at random, changes it

/// to 1, and then returns the position [row.id, col.id] of that value.

/// Also, write a function reset which sets all values back to 0. Try to

/// minimize the number of calls to system's Math.random() and optimize

/// the time and space complexity.

///

/// Note:

///

/// 1. 1 <= n\_rows, n\_cols <= 10000

/// 2. 0 <= row.id < n\_rows and 0 <= col.id < n\_cols

/// 3. flip will not be called when the matrix has no 0 values left.

/// 4. the total number of calls to flip and reset will not exceed 1000.

/// Example 1:

///

/// Input:

/// ["Solution","flip","flip","flip","flip"]

/// [[2,3],[],[],[],[]]

/// Output: [null,[0,1],[1,2],[1,0],[1,1]]

///

/// Example 2:

///

/// Input:

/// ["Solution","flip","flip","reset","flip"]

/// [[1,2],[],[],[],[]]

/// Output: [null,[0,0],[0,1],null,[0,0]]

///

/// Explanation of Input Syntax:

/// The input is two lists: the subroutines called and their arguments.

/// Solution's constructor has two arguments, n\_rows and n\_cols. flip and

/// reset have no arguments. Arguments are always wrapped with a list,

/// even if there aren't any.

/// </summary>

class RandomFlipMatrix

{

private:

int rows;

int cols;

int size;

vector<bitset<10000>> matrix;

vector<int> row\_slots;

public:

RandomFlipMatrix(int n\_rows, int n\_cols)

{

rows = n\_rows;

cols = n\_cols;

reset();

srand((unsigned int)time(0));

}

vector<int> flip()

{

vector<int> result;

int index = rand() % size;

int sum = 0;

for (int i = 0; i < rows; i++)

{

sum += row\_slots[i];

if (index >= sum) continue;

int remaining = sum - index;

for (int j = 0; j < cols; j++)

{

if (matrix[i].test(j)) continue;

remaining--;

if (remaining == 0)

{

result.push\_back(i);

result.push\_back(j);

matrix[i].set(j);

size--;

row\_slots[i]--;

return result;

}

}

}

return result;

}

void reset()

{

matrix = vector<bitset<10000>>(rows);

row\_slots = vector<int>(rows, cols);

size = rows \* cols;

}

};

## 710. Random Pick with Blacklist

Hard

Given a blacklist B containing unique integers from [0, N), write a function to return a uniform random integer from [0, N) which is **NOT** in B.

Optimize it such that it minimizes the call to system’s Math.random().

**Note:**

1. 1 <= N <= 1000000000
2. 0 <= B.length < min(100000, N)
3. [0, N) does NOT include N. See [interval notation](https://en.wikipedia.org/wiki/Interval_(mathematics)).

**Example 1:**

**Input:**

["Solution","pick","pick","pick"]

[[1,[]],[],[],[]]

**Output:** [null,0,0,0]

**Example 2:**

**Input:**

["Solution","pick","pick","pick"]

[[2,[]],[],[],[]]

**Output:** [null,1,1,1]

**Example 3:**

**Input:**

["Solution","pick","pick","pick"]

[[3,[1]],[],[],[]]

**Output:** [null,0,0,2]

**Example 4:**

**Input:**

["Solution","pick","pick","pick"]

[[4,[2]],[],[],[]]

**Output:** [null,1,3,1]

**Explanation of Input Syntax:**

The input is two lists: the subroutines called and their arguments. Solution's constructor has two arguments, N and the blacklist B. pick has no arguments. Arguments are always wrapped with a list, even if there aren't any.

### Analysis:

With the blacklist, you can divide the whole range into multiple segments and each with accumulated sum for the valid numbers. This become a weighted random pick.

There is another solution, you can swap the blacklist hole with the valid numbers, when you hit the blacklisted number, simply return the swapped number in it.

/// <summary>

/// Leet code #710. Random Pick with Blacklist

///

/// Given a blacklist B containing unique integers from [0, N), write a

/// function to return a uniform random integer from [0, N) which is NOT

/// in B.

///

/// Optimize it such that it minimizes the call to system’s Math.random().

///

/// Note:

/// 1. 1 <= N <= 1000000000

/// 2. 0 <= B.length < min(100000, N)

/// 3. [0, N) does NOT include N. See interval notation.

///

/// Example 1:

/// Input:

/// ["Solution","pick","pick","pick"]

/// [[1,[]],[],[],[]]

/// Output: [null,0,0,0]

///

/// Example 2:

/// Input:

/// ["Solution","pick","pick","pick"]

/// [[2,[]],[],[],[]]

/// Output: [null,1,1,1]

///

/// Example 3:

/// Input:

/// ["Solution","pick","pick","pick"]

/// [[3,[1]],[],[],[]]

/// Output: [null,0,0,2]

///

/// Example 4:

/// Input:

/// ["Solution","pick","pick","pick"]

/// [[4,[2]],[],[],[]]

/// Output: [null,1,3,1]

/// Explanation of Input Syntax:

/// The input is two lists: the subroutines called and their arguments.

/// Solution's constructor has two arguments, N and the blacklist B. pick has

/// no arguments. Arguments are always wrapped with a list, even if there

/// aren't any.

/// </summary>

class Solution

{

private:

vector<pair<int, int>> m\_Range;

vector<int> m\_Sum;

unsigned int m\_Size;

public:

Solution(int N, vector<int> blacklist)

{

sort(blacklist.begin(), blacklist.end());

int sum = 0;

int first = 0;

for (size\_t i = 0; i <= blacklist.size(); i++)

{

int last;

if (i == blacklist.size())

{

last = N;

}

else

{

last = blacklist[i];

}

if (first != last)

{

sum += last - first;

m\_Sum.push\_back(sum);

m\_Range.push\_back(make\_pair(first, last));

}

first = last + 1;

}

m\_Size = N - blacklist.size();

srand((unsigned int)time(NULL));

}

int pick()

{

int random = rand() % m\_Size;

auto itr = upper\_bound(m\_Sum.begin(), m\_Sum.end(), random);

if (itr == m\_Sum.end()) return -1;

int index = itr - m\_Sum.begin();

int result = m\_Range[index].second - (m\_Sum[index] - random);

return result;

}

};

class SolutionII

{

private:

unordered\_map<int, int> m\_BlackMap;

unsigned int m\_Size;

public:

SolutionII(int N, vector<int> blacklist)

{

sort(blacklist.begin(), blacklist.end());

for (size\_t i = 0; i < blacklist.size(); i++)

{

m\_BlackMap[blacklist[i]] = blacklist[i];

}

m\_Size = N - blacklist.size();

int last = N - 1;

for (size\_t i = 0; i < blacklist.size(); i++)

{

while (m\_BlackMap.count(last) > 0) last--;

m\_BlackMap[blacklist[i]] = last;

last--;

}

srand((unsigned int)time(NULL));

}

int pick()

{

int index = rand() % m\_Size;

if (m\_BlackMap.count(index) > 0)

{

return m\_BlackMap[index];

}

else

{

return index;

}

}

};