1. Теория вероятностей

Схемой Бернулли называется последовательность независимых испытаний, в каждом из которых возможны лишь два исхода — «успех» и «неудача», при этом успех в каждом испытании происходит с одной и той же вероятностью $p \in (0, 1)$, а неудача — с вероятностью $q = 1 - p$.

Формула Бернулли вероятность того, что событие наступит ровно k раз при n испытаниях

$$P_{k,n} = C_n^k \cdot p^k \cdot q^{n-k}$$

1.1. Локальная теорема Муавра — Лапласа

Если в схеме Бернулли n стремится к бесконечности, то

$$P(a \leq \frac{\mu - np}{\sqrt{npq}} \leq b) \approx \int_a^b \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx = \Phi(b) - \Phi(a)$$

где μ – количество успехов, $\int_a^b$ -интеграл Лапласа. Рекомендуется использовать при $n > 100$ и $\mu > 20$.

На заметку (для схемы Б.):

- $M[\mu] = np$ – математическое ожидание
- $D[\mu] = npq$ – дисперсия
- $\sigma = \sqrt{D[\mu]} = \sqrt{npq}$ – среднеквадратичное отклонение

И факты про интеграл Лапласа :

1. $\Phi(-x) = -\Phi(x)$ *важно!*
2. $\Phi(x) \approx 0.5$, если $x > 5$

1.2. Центральная предельная теорема

Утверждает о том, что сумма одинаково распределённых случайных и независимых случайных велечин, имеет распределение, близкое к нормальному.

Пусть $\xi_1 \dots \xi_n$ - последовательность случайных величин, $S_n = \sum_{i=1}^{n} \xi_i$, тогда:

$$\frac{S_n - nM[\xi_k]}{\sqrt{nD[\xi_k]}} \to N(0.1)$$

Следствие:

$$P(a \leq \frac{S_n - nM[\xi_k]}{\sqrt{nD[\xi_k]}} \leq b) \approx \Phi(b) - \Phi(a)$$

1.3. Неравенства Чебышева

1. $P(|\xi| \geq \varepsilon) \leq \dfrac{M[\xi]}{\varepsilon}$
2. $P(|\xi - M[\xi]| \geq \varepsilon) \leq \dfrac{D[\xi]}{\varepsilon^2}$
3. $P(|\xi - M[\xi]| < \varepsilon) \geq 1 - \dfrac{D[\xi]}{\varepsilon^2}$

1.4. Локальные предельные теоремы

- Пуассона: $P(\mu = k) \approx \dfrac{\lambda^k}{k!} e^{-\lambda}$, где $\lambda = np$. Использовать, когда $np \leq 10$
- Муавра-Лапласа: $P(\mu = k) \approx \dfrac{\Phi(x)}{\sigma}$, где $x = \dfrac{k - np}{\sigma}$

2. Дискретная математика

2.1. Частично упорядоченные множества

$(M, \leq)$ - Чум, его войства:

- рефлексивность $\forall x \in M : x \leq x$
- антисимметричность $\forall x, y \in M : x \leq y \wedge y \leq x \to x = y$
- транзитивность $\forall x, y, z \in M : x \leq y, y \leq z \to x \leq z$

Элементы **несравнимы**, когда $x \not\leq y, y \not\leq x$

2.1.1. Диаграмма Хасса

$a \prec b$

- $a \leq b, a \neq b$
- $a \leq c \leq b \to a = c \vee c = b$

Число отношений порядка на $\{1 \dots n\}$ равно $n!$

2.2. Графы

2.3. Голосарий

- Изоморфизм. Два графа называются изоморфными, если существует перестановка вершин, при которой они совпадают. Иначе говоря, два графа называются изоморфными, если существует взаимно-однозначное соответствие между их вершинами и рёбрами, которое сохраняет смежность и инцидентность (графы отличаются только названиями своих вершин).
- Маршрут в графе — это чередующаяся инцидентных последовательность вершин и рёбер

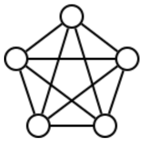
- Простая цепь в графе — маршрут, все рёбра которого различны.
- Цикл — замкнутая цепь.
- Простой цикл — цикл, не проходящий дважды через одну вершину.
- Гамильтонов путь — простой путь в графе, содержащий все вершины графа ровно по одному разу.
- Гамильтонов цикл — простой цикл в графе, содержащий все вершины графа ровно по одному разу.
- Двудольный граф — это граф (V, E), такой, что множество вершин V разбито на два непересекающихся подмножества V_1 V_2, причём всякое ребро E инцидентно вершине из V_1 и вершине из V_2

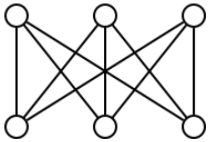
  Теорема Кенига Граф G является двудольным тогда и только тогда, когда все циклы в графе G имеют чётную длину.

  Следствие - алгоритм проверки на двудольность: за один проход в глубину. На каждом шаге обхода в глубину помечаем вершину. Допустим, мы пошли в первую вершину — помечаем её как 1. Затем просматриваем все смежные вершины, и если не помечена вершина, то на ней ставим пометку 2 и рекурсивно переходим в нее. Если же она помечена и на ней стоит та же пометка, что и у той, из которой шли (в нашем случае 1), значит граф не двудольный.
- Лемма о рукопожатиях — положение теории графов, согласно которому любой конечный неориентированный граф имеет чётное число вершин нечётных степеней.

$$\sum_{v \in V} \deg(v) = 2|E|$$

- Планарный граф — граф, который может быть изображён (уложен) на плоскости без пересечения рёбер.
  Теорема Понтрягина — Куратовского. Граф планарен тогда и только тогда, когда не содержит подграфов, стягивающихся в K_5 или $K_{3,3}$.





2.4. Булева алгерба

2.4.1. Правило резолюций

$$\frac{A \vee B, \neg A \vee C}{B \vee C}$$

3. Комбинаторные алгоритмы

3.1. Минимальный остов

Пусть $G = (V, E, c)$ — связный взвешенный неориентированный граф. Под весом $c(H)$ произвольного ненулевого подграфа H будем понимать сумму весов всех ребер подграфа H.

Остовом называется ацикличный подграф данного графа, содержащий все его вершины.

Ацикличный остовный подграф (содержащий все вершины графа G) будем называть остовным лесом графа G.

Остов T называется минимальным, если для любого остова T' выполняется неравенство $c(T) \leq c(T')$.

3.1.1. Алгоритм Борувки–Краскала

1. Сортируем ребра графа по возрастанию весов.
2. Тривиальный лес объявить растущим.
3. Проходим ребра в «отсортированном» порядке.
4. Для каждого ребра выполняем:
   - если вершины, соединяемые данным ребром лежат в разных деревьях растущего леса, то объединяем эти деревья в одно, а ребро добавляем к строящемуся остову.
   - если вершины, соединяемые данным ребром лежат в одном дереве растущего леса, то исключаем ребро из рассмотрения
5. Если есть еще нерассмотренные ребра и не все деревья объединены в одно, то переходим к шагу 3, иначе выход.

3.1.2. Ярника–Прима–Дейкстры

1. Выбирается произвольная вершина — она будет корнем остовного дерева.
2. Измеряется расстояние от нее до всех внешних по отношению к растущему дереву вершин (расстояние от внешней вершины до ближайшей к ней вершины дерева).
3. До тех пор пока в дерево не добавлены все вершины делать:
   - Найти вершину, расстояние от дерева до которой минимально.
   - Добавить ее к дереву.
   - Пересчитать расстояния от вершин до дерева следующим образом: если расстояние до какой–либо вершины из новой вершины меньше текущего расстояния от дерева, то старое расстояние от дерева заменить новым.

3.2. Пути в сетях

Взвешенный орграф $G = (V, E, c)$ называется сетью.
Пусть P — некоторый (v, w)–путь:

$$v = v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} \dots \xrightarrow{e_k} v_k = w$$

Величину $c(P)$ назовем длиной пути P:

$$c(P) = c(e_1) + c(e_2) + \dots + c(e_k)$$

Наименьшую из длин (v, w)–путей назовем расстоянием от v до w, . . . а тот (v, w) –путь, длина которого равна расстоянию от v до w , будем называть кратчайшим (v, w)–путем.

3.2.1. Алгоритм Форда–Беллмана

Поиска кратчайшего (s, t) пути, работает на ребрах с **отрицательными весами**

$D[v]$ — массив, в котором будет «накапливаться» расстояние от выделенной вершины s до всех остальных вершин графа;

ПРЕДШ$[v]$ — массив, в котором на месте v будет храниться номер вершины–предшественницы вершины v в кратчайшем (s, v)– маршруте

1. Для всех вершин $v \in V$ положить:
   - $D[v] := c(s, v)$;
   - если $c(s, v) < \infty$ ПРЕДШ$[v] := s$;
   - иначе ПРЕДШ$[v] := 0$;
2. Следующую операцию повторить $n{-}2$ раза:
   - для всех вершин $v \in V \setminus \{s\}$:
     $D[v] := min\{D[v], D[w] + c[w, v] | w \in V\}$
   - ПРЕДШ$[v] := w*$(та вершина, которая обеспечила минимум $D[v]$)

3.2.2. Алгоритм Дейкстры

Жадный, не будет работать если есть ребра с отрицательным весом.

F — множество еще не просмотренных вершин.

1. $D[s] := 0$;ПРЕДШ$[s] := 0$; $F := V \setminus \{s\}$
2. для всех $v \in F$ положим:
   - $D[v] := c[s, v]$;
   - ПРЕДШ$[v] := s$;
3. $n{-}1$ раз делать:

- выбрать $w \in F : D[w] = min\{D[u] | u \in F\}; F := F \setminus w$;
- для всех $v \in F$ делать:
  если $(D[w] + c[w, v] < D[v])$,
  то $D[v] := D[w] + c[w, v]$; ПРЕДШ$[v] := w$.

3.3. Потоки в сетях

Потоком в сети G называется функция $f : E \to R$, удовлетворяющая условиям:

- $0 \leq f(e) \leq c(e)$ для всех $e \in E$;
- $f(v-) = f(v+)$ для всех $v \in V \setminus \{s, t\}$

Здесь $f(v-) = \Sigma_{w \in v-} f(w, v)$, а$f(v+) = \Sigma_{w \in v+} f(v, w)$.

Пусть P — цепь из v в w. Для каждой дуги e цепи P положим

$$h(e) = \begin{cases} c(e) - f(e), & \text{если } e \text{ — прямая дуга,} \\ f(e), & \text{если } e \text{ — обратная дуга.} \end{cases}$$

Пусть $h(P) = min\{h(e) | e \in P\}$.

Цепь P из v в w называется f –дополняющей, если $h(P) > 0$.

3.3.1. Алгоритм Форда–Фалкерсона

1. Положить $f(e) = 0$ для всех дуг $e \in E$;
2. для текущего потока f искать f–дополняющую (s, t)–цепь;
3. если такая цепь P построена, то для всех прямых дуг цепи P положить

$$f(e) = \begin{cases} f(e) + h(P), & \text{для прямых дуг,} \\ f(e) - h(P), & \text{для обратных дуг,} \end{cases}$$

и вернулся на шаг 2;

4. иначе СТОП.

3.4. Парасочетания в двудольных графах

Паросочетанием в графе называется произвольное множество его ребер такое, что каждая вершина графа инцидентна не более, чем одному ребру из этого множества.

Граф $G = (V, E)$ называется двудольным, если множество его вершин V можно разбить на непересекающиеся множества X и Y такие, что каждое ребро $e \in E$ имеет вид $e = xy$, где $x \in X$, $y \in Y$.

Вершины, не принадлежащие ни одному ребру из паросочетания, называют **свободными относительно** M или просто **свободными**, а все прочие — **насыщенными**.

Удобно также все ребра, входящие в паросочетание M называть M–**темными** или просто **темными**, а все прочие — M–**светлыми** или просто **светлыми**.

Паросочетание, содержащее наибольшее число ребер, называется наибольшим.

Паросочетание, насыщающее все вершины двудольного графа, называется полным.

Паросочетание, не содержащееся ни в каком другом паросочетании, называется **максимальным** (по включению).

Пусть M — паросочетание в графе G. M–**чередующейся цепью** называется такая последовательность вершин и ребер вида

$$x_0, \quad x_0y_1, \quad y_1, \quad y_1x_2, \quad x_2, \quad \ldots, \quad x_k, \quad x_ky_{k+1}, \quad y_{k+1},$$

где $k > 0$, что все вершины этой цепи различны, x_0 и y_{k+1} — свободные, а все остальные вершины насыщенные в паросочетании M, причем каждое второе ребро принадлежит M (т.е. ребра вида $y_i x_{i+1}, i = 1, \ldots, k-1$ входят в M), а остальные ребра в M не входят.

Операция $M \oplus P$ определяется следующим образом: *все ребра из P, входившие в M из паросочетания исключаются, а все ребра из P, не входившие в M, в паросочетание добавляются.* Другими словами,

$$M \oplus P = (M \backslash P) \cup (P \backslash M).$$

В цепи P происходит «переключение цветов»: светлые ребра становятся темными и, наоборот, темные — светлыми.

3.4.1. Алгоритм Куна

Поиск максимального (и полного) парасочетания.

1. пустое паросочетание объявить текущим паросочетанием M;
2. если все вершины из X насыщены в M, то СТОП (M — полное паросочетание);
3. иначе выбрать произвольную свободную вершину $x \in X$ и искать M–чередующуюся цепь, начинающуюся в x;
4. если такая цепь P найдена, то положить $M := M \oplus P$ и вернуться на шаг 2;
5. иначе СТОП (полного паросочетания в заданном графе не существует).

3.5. Задача о назначениях

Пусть $G = (X, Y, E, c)$ — взвешенный двудольный граф, $|X| = |Y| = n$.

Весом паросочетания M называется сумма весов входящих в него

Задача о назначениях: в заданном взвешенном двудольном графе найти полное паросочетание минимального веса (**оптимальное паросочетание**).

Если веса всех ребер графа, инцидентных какой-либо вершине, увеличить (уменьшить) на одно и то же число, то всякое оптимальное паросочетание в графе с новыми весами является оптимальным и в графе с исходными весами

Пусть $X' \subseteq X$, $Y' \subseteq Y$ и $d \in R$. Будем говорить, что к графу $G = (X, Y, E, c)$ применена операция $(\mathbf{X'}, \mathbf{d}, \mathbf{Y'})$, если сначала из веса каждого ребра, инцидентного вершине из X', вычтено число d, а затем к весу каждого ребра, инцидентного вершине из Y', прибавлено число d.

Схема применения операции:

	Y'	$Y \backslash Y'$
X'	I	$II - d$
$X \backslash X'$	$IV + d$	III

3.5.1. Венгерский алгоритм

1. Преобразовать веса ребер так, чтобы веса всех ребер стали неотрицательными и каждой вершине стало инцидентно хотя бы одно ребро нулевого веса;
2. пустое паросочетание объявить текущим паросочетанием M;
3. если в графе все вершины насыщены относительно текущего паросочетания, то СТОП (текущее паросочетание оптимально);

4. иначе выбрать свободную вершину $x \in X$ и искать $M-$чередующуюся цепь, начинающуюся в X, из ребер нулевого веса;
5. если такая цепь построена, то положить $M := M \oplus P$ и вернуться на шаг 3;
6. иначе для множества вершин $X' \subseteq X$, $Y' \subseteq Y$, помеченных в ходе поиска (это вершины венгерского дерева), положить величину d равной $\min\{c(x, y) | x \in X', y \in Y \backslash Y'\}$ и применить к графу операцию (X', d, Y');
7. из тех вершин $x \in X'$, которым стало инцидентно хотя бы одно ребро нулевого веса, возобновить поиск $M-$чередующейся цепи по ребрам нулевого веса; если такая цепь P будет найдена, то положить $M := M \oplus P$ и вернуться на шаг 3; иначе вернуться на шаг 6 (множества X' и Y' при этом увеличатся).

4. Алгебра и геометрия

5. Математический анализ

6. C++

6.1. Максимальный уровень бинарного дерева

```
// A queue based C++ program to find maximum sum
#include <iostream>
#include <queue>

using namespace std ;

/* A binary tree node has data, pointer to left child
 and a pointer to right child */
struct Node
{
    int data ;
    struct Node * left, * right ;
};

// Function to find the maximum sum of a level in tree
// using level order traversal
int maxLevelSum(struct Node * root)
{
    // Base case
    if (root == NULL)
        return 0;

    // Initialize result
    int result = root->data;

    // Do Level order traversal keeping track of number
    // of nodes at every level.
    queue<Node*> q;
    q.push(root);
    while (!q.empty())
    {
        // Get the size of queue when the level order
        // traversal for one level finishes
        int count = q.size() ;

        // Iterate for all the nodes in the queue currently
        int sum = 0;
        while (count--)
        {
            // Dequeue an node from queue
            Node *temp = q.front();
            q.pop();
```

```
            // Add this node's value to current sum.
            sum = sum + temp->data;

            // Enqueue left and right children of
            // dequeued node
            if (temp->left != NULL)
                q.push(temp->left);
            if (temp->right != NULL)
                q.push(temp->right);
        }

        // Update the maximum node count value
        result = max(sum, result);
    }

    return result;
}

/* Helper function that allocates a new node with the
 given data and NULL left and right pointers. */
struct Node * newNode(int data)
{
    struct Node * node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

int main()
{
    struct Node *root = newNode(1);
    root->left        = newNode(2);
    root->right       = newNode(3);
    root->left->left  = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(8);
    root->right->right->left  = newNode(6);
    root->right->right->right  = newNode(7);


    cout << "Maximum level sum is "
    << maxLevelSum(root) << endl;
    return 0;
}
```

6.2. Все перестановки массива

```
#include <string.h>
#include <iostream>
```

```
using namespace std;

/* Function to swap values at two pointers */
void swap(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

void prnt(int a[], int n=3){
    for (int i=0;i<n;i++){
        cout<<a[i];
    }
     cout<<'\n';
}
int* permute(int a[], int r)
{
    int i;
    if (r==0){

        return a;
    }
    else
    {
        for (i = 0; i < r; i++)
        {
            swap(a[i], a[r-1]);
            prnt(permute(a, r-1));
            swap(a[i], a[r-1]);
        }
    }
    return a;
}

int main()
{
    int a[3] = {1,2,3};
    int r=3;
    permute(a, r);

}
```

6.3. Поиск в ширину

```
vector < vector<int> > g; // graph
int n;
```

```
int s;

queue<int> q;
q.push (s);
vector<bool> used (n);
vector<int> d (n), p (n);
used[s] = true;
p[s] = -1;
while (!q.empty()) {
  int v = q.front();
  q.pop();
  for (size_t i=0; i<g[v].size(); ++i) {
    int to = g[v][i];
    if (!used[to]) {
      used[to] = true;
      q.push (to);
      d[to] = d[v] + 1;
      p[to] = v;
    }
  }
}

if (!used[to])
  cout << "No␣path!";
else {
  vector<int> path;
  for (int v=to; v!=-1; v=p[v])
    path.push_back (v);
  reverse (path.begin(), path.end());
  cout << "Path:␣";
  for (size_t i=0; i<path.size(); ++i)
    cout << path[i] + 1 << "␣";
}
```

6.4. Число простых делителей

```
  int count_or_prime_dividers(int x){
    int res = 0;
    for (int i=2;i<=x;i++){
        if (x % i ==0){
            cout<<i<<'\n';
            res+=1;
        }
        while (x%i ==0){
            x = x/ i;
        }
    }
```

```
    return res;
}
```