Алгоритмы

1. Анализ алгоритмов. Понятие о сложности по времени и по памяти. Асимптотика, O- символика. Доказательство корректности алгоритмов.

Определение: (O-нотация)

Пусть $f, g : \mathbb{N} \to \mathbb{N}$. Тогда $f = \mathcal{O}(g)$, если $\exists C > 0$ и $\exists N : \forall n > N$:

$$f(x) \leq Cg(x)$$

Утв

$f = \mathcal{O}(g) \Leftrightarrow \exists C > 0\ \forall n \in \mathbb{N} : f(n) \leq Cg(n)$.

Определение

$f, g : \mathbb{N} \to \mathbb{N}$. Тогда $f = \Omega(g)$ или $g = \mathcal{O}(f)$ или $\exists C > 0\ f(n) \geq Cg(n)$.

Определение

$f = \Theta(g)$, если $f = \mathcal{O}(g)$ и $g = \mathcal{O}(f)$ или $\exists C_1, C_2 > 0\ \forall n \in \mathbb{N} C_1 g(n) \leq f(n) \leq C_2 g(n)$.

Примеры:

- $n = \mathcal{O}(n^2)$, $n^2 = \Omega(n)$;
- $3n + n^2 = \Theta(n^2)$;
- $n \log n = \mathcal{O}(n^2)$;
- $\log^{10} n = \mathcal{O}(n^{1/5})$.

Теорема: (Мастер Теорема)

Пусть $T(n)$ – время работы какого-то алгоритма на входе длины n, причем $T(n) = aT(\frac{n}{b}) + f(n)$. Тогда

1. Если $\exists \varepsilon > 0 : f(n) = \mathcal{O}(n^{\log_b a - \varepsilon})$, то
$$T(n) = \Theta(n^{\log_b a})$$
2. Если $f(n) = \Theta(n^{\log_b a})$, то
$$T(n) = \Theta(n^{\log_b a} \cdot \log n)$$
3. Если $\exists \varepsilon > 0 : f(n) = \Omega(n^{\log_b a + \varepsilon})$, причем $\exists c < 1 : a \cdot f^{n/b} \leq c \cdot f(n)$, ($\forall n$, начиная с некоторого), то $T(n) = \Theta(f(n))$.

Следствие: Пусть $T(n) = 2T(n/2) + \Theta(n)$. Тогда $T(n) = \Theta(n \log n)$.

Задание Префиксные суммы. Пусть $a_1, \ldots, a_n$ – статический массив. l, r – индексы. Сообщить $a_l + a_{l+1} + \ldots + a_r$

Наивное решение: $\mathcal{O}(n \cdot q)$, где q – количество запросов;

Префиксные суммы:

1: pref[0] = 0
2: pref[i] = $a_1 + \ldots + a_i$
3: pref[$i+1$] = $a_1 + \ldots + a_i + a_{i+1}$

Это проходит за $\mathcal{O}(n)$. Итого

$$a_l + \ldots a_r = pref[r] - pref[l-1]$$

Решение $\mathcal{O}(n+q)$

Задание Бинарный поиск Отсортированный массив $a_1 \leq a_2 \leq \ldots \leq a_n$. Вопрос: есть ли x в этом массиве.

Наивное: $\mathcal{O}(n \cdot q)$

$\mathcal{O}(q \cdot \log n + n)$ Пусть $l = 1$, $r = n$. x лежит в отрезке $[l, r]$. Посмотрим на середину массива с индексом $m = \frac{l+r}{2}$. Тогда:

1. Если $a[m] = x$ – выдать да;
2. Если $a[m] < x$, то ответ справа $l = m$, повторяем;
3. Иначе $r = m$.

Повторять
  int $m = \frac{(l+r)}{2}$
  Если тогда $a[m] == x$
    Выдать *yes*
  иначе если $a[m] < x$
    $l = m$
  иначе
    $r = m$
  Конец условия
Пока выполняется $(r - l > 1)$
Если (**тогда** $a[l] == x || a[r] == x$)
  Выдать *yes*
иначе
  Выдать *no*
Конец условия

2. Строки и операции над ними. Представление строк. Вычисление длины, конкатенация. Алгоритмы поиска подстроки в строке.

3. Сортировки. Нижняя теоретико-информационная оценка сложности задачи сортировки. Алгоритмы сортировки вставками, пузырьком, быстрая сортировка, сортировка слиянием. Оценка сложности.

Сортировки

Постановка задачи: Дан массив объектов $a_1, \dots, a_n$. На объектах задано отношение порядка:

$$\forall x, y, \quad x < y \text{ или } x = y \text{ или } x > y$$

Цель: переставить элементы, чтобы они шли в порядке неубывания (невозрастания). Или также найти перестановку: $\sigma : [n] \to [n]$:

$$a_{\sigma(1)} \leq a_{\sigma(2)} \leq \dots \leq a_{\sigma(n)}$$

Теорема

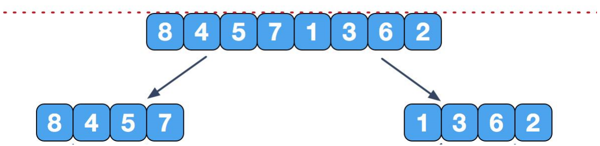
Любой алгоритм сортировки, основанный на сравнениях, требует $\Omega(n \log n)$ сравнений в худшем случае на массиве длины n.

Лемма

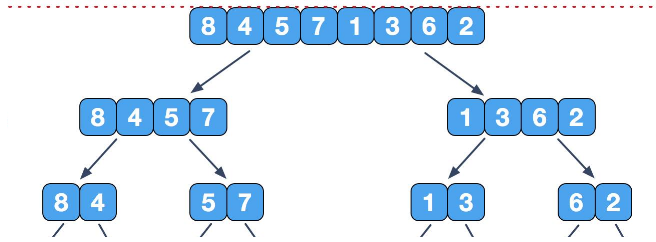
$\log(n!) = \Theta(n \log n)$

Сортировка слиянием

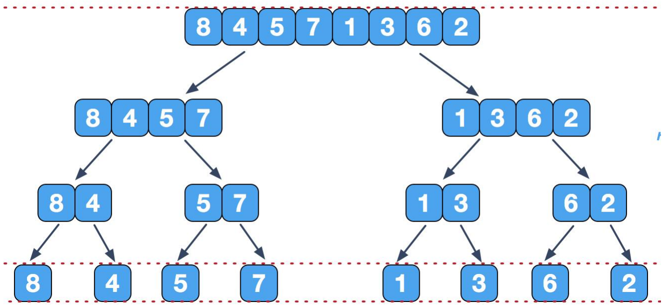
Сортировка за $\mathcal{O}(n \log n)$, основанная на сравнениях. Разобьем массив на два равных куска:



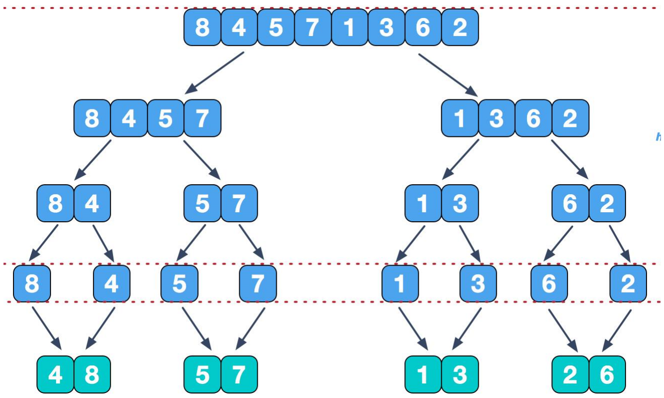
Рекурсивно продолжаем:



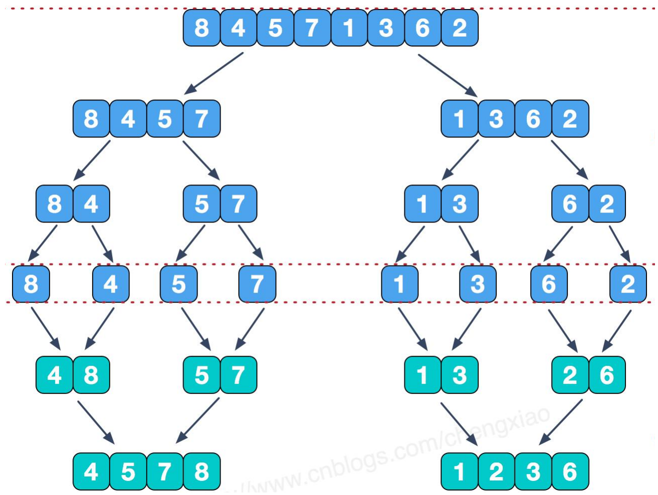
Следующий шаг:



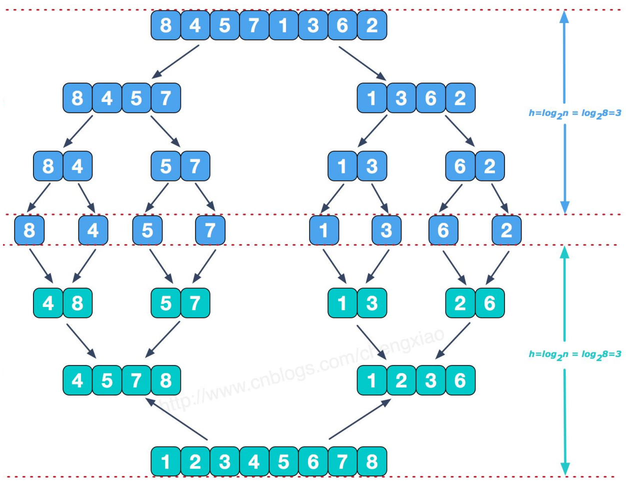
Сортируем полученное:



Склеиваем:



Заводим два указателя на отсортированные части. Записываем минимальное из элементов указателя и сдвигаем тот, который записали, повторяем:



Псевдокод:

```
Функция MergeSort(A)
    Если l тогдаen(A) == 1
        Возвратить
    Конец условия
    B, C – две половины A
    MergeSort(B)
    MergeSort(C)
    Merge(B,C,A)
Конец функции
Функция Merge(B,C,A)
    i=0, j=0, p=0
    Повторять
        Если ( тогдаB[i] < C[j])
            A[p++] = B[i]
            ++i
        иначе
            A[p++] = C[j]
            ++j
        Конец условия
    Пока выполняется (i < len(B) and j < len(C))
    Повторять
        A[p++] = B[i]
        ++i
    Пока выполняется (i < len(B))
    Повторять
        A[p++] = C[j]
        ++j
    Пока выполняется (j < len(C))
Конец функции
```

Количество инверсий

Неотсортированный массив $a_1, \ldots, a_n$. Инверсия $(i,j)\ :\ i < j,\ a_i > a_j$. Пусть $f(A)$ – число инверсий в массиве A. $f(A) = f(B) + f(C)+$ количество инверсий, пересекающих разделитель. В момент, когда выбирается $B[i]$ все числа, стоящие слева от $B[i]$ его меньше, причем там ровно j элементов из C. К ответу добавить j.

Нерекурсивная реализация MergeSort

Будем считать, что n – степень двойки. (Если нет, то добавим слева или справа меньшие или большие, соответственно элементы, чем остальные в массиве. После сортировки их выкинем)

```
Функция MergeSort(a)
    queue<vector<int» q
    Цикл (int i = 0; i < n; ++i) do
        q.push(a[i])
    Конец цикла
    Пока (q.size() > 1) do
        vector<int> a = q.front()
        q.pop()
        vector<int> b = q.front()
        q.pop()
        q.push(Merge(a,b))
    Конец цикла
Конец функции
```

Ответ: $q.front()$. Время: $\mathcal{O}(n \log n)$, Память: $\mathcal{O}(n)$

Быстрая сортировка (Quick sort)

Массив чисел: $a_1, \ldots, a_n$. Пусть x – случайный элемент массива. $Partition(A, x)$ – перемешивание массива в виде:

$$\underbrace{[\ < x\ ]}_{Quick\ Sort}[\ = x\ ]\underbrace{[\ > x\ ]}_{Quick\ Sort}$$

Псевдокод

```
Функция QuickSort(A)
    Если ( тогдаlen(A) == 1)
        Возвратить
    Конец условия
    x – случайный элемент A
    Partition(A,x)
    B – числа <= x, C – числа > x
    QuickSort(B)
    QuickSort(C)
Конец функции
```

В худшем случае: $\Omega(n^2)$.

Теорема

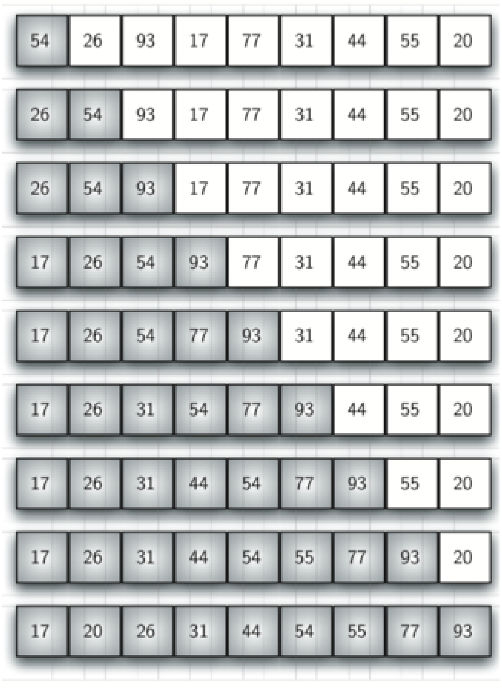
Математическим ожиданием времени работы на массиве длины n есть $\Theta(n \log n)$.

Сортировка вставками (InsertSort)

Псевдокод

```
Функция InsertSort(a)
    Если ( тогдаlen(A) == 1)
        Возвратить
    Конец условия
    Цикл (i=1; i < n; ++i) do
        j = i-1
        Пока (j >= 0 and a[j] > a[j+1]) do
            swap(a[j], a[j+1])
            –j
        Конец цикла
    Конец цикла
Конец функции
```

Худший случай – массив отсортирован в обратном порядке. Сложность $\mathcal{O}(n^2)$



Сортировка пузырьком (BubbleSort)

Псевдокод

```
Функция BubbleSort(a)
    Цикл (int i = 0; i < n-2; ++i) do
        Цикл (int j = 0; j < n-2; ++j) do
            Если a тогда[j] > a[j+1]
                swap(a[j], a[j+1])
            Конец условия
        Конец цикла
    Конец цикла
Конец функции
```

Сложность: $\mathcal{O}(n^2)$

Оптимизация

1. После i итераций, в конце i чисел отсортировано;

```
Функция BubbleSort(a)
    Цикл (int i = 0; i < n-2; ++i) do
        Цикл (int j = 0; j < n-i-2; ++j) do
            Если a тогда[j] > a[j+1]
                swap(a[j], a[j+1])
            Конец условия
        Конец цикла
    Конец цикла
Конец функции
```

2. Если не было *swap*, то массив уже отсортирован.

```
Функция BubbleSort(a)
    i=0
    isSwapped=True
    Пока isSwapped do
        isSwapped=False
        Цикл (int j = 0; j < n-i-2; ++j) do
            Если a[j] > a[j+1] тогда
                swap(a[j], a[j+1])
                isSwapped = True
            Конец условия
        Конец цикла
        ++i
    Конец цикла
Конец функции
```

4. Представление матриц и векторов. Алгоритмы умножения матриц и эффективные способы их реализации. Численные методы решения систем линейных уравнений.

Векторы – массивы двух типов. Статические массивы – структуры фиксированного размера. Динамические – размер можно менять в процессе выполнения программы. В $c++ - std :: vector$. Позволяет эффективно добавлять элементы в конце и удалять последние элементы.

```
std::vector<int> v = {2,4,3};
v.push_back(3); // [2,4,3,3]
v.push_back(5); // [2,4,3,5]
std::cout << v.back() << "\n"; // 5
v.pop_back(); // [2,4,3]
```

Вектор – частный случай матрицы размера $n \times 1$, где n – длина вектора. Итого: матрица представляет собой двумерный массив (вектор векторов). Операции над матрицами в линейной алгебре. Умножение двух двух матриц за время $O(n^3)$:

```
for (int i = 1; i <= n; ++i) {
    for (int j = 1; j <= n; ++j) {
        for (int k = 1; k <= n; ++k) {
            ans[i][j] += a[i][k]*b[k][j];
        }
    }
}
```

Возведение в степень матрицы считается также, как и для возведения числа:

$$A^n = \begin{cases} A \cdot A^{n-1}, & \text{если } n - \text{четное} \\ (A^{1/2})^2, & \text{если } n \text{ четное.} \end{cases}$$

Время: $\mathcal{O}(k^3 \log n)$. k – размерность квадратной матрицы, n – степень

Методы решения СЛАУ

Метод Гаусса

Система уравнений имеет хорошую плотность и среднюю обусловленность:

$$Ax = b.$$

За счет элементарных преобразования над строками (см. Линейная алгебра) матрица СЛАУ преобразуется в верхнюю (или нижнюю) треугольную – прямой ход метода. Обратный ход метода – определяются неизвестные.

Прямой ход. На первом шаге алгоритма выберем диагональный элемент в качестве главного (pivot), причем такой, чтобы он не был равен нулю (если равен, то меняем строки местами), строку и столбец, на пересечении которых стоит pivot, объявим ведущими. Обнулим элементы ведущего столбца. В результате повторения для каждого столбца (2-й шаг – pivot = a^1_{22}). В результате прямого хода получим верхнетреугольную матрицу вида:

$$\left[\begin{array}{ccccc|c} a_{11} & a_{12} & a_{13} & \dots & a_{1n} & b_1 \\ 0 & a^1_{22} & a^1_{23} & \dots & a^1_{2n} & b_2 \\ 0 & 0 & a^2_{33} & \dots & a^2_{3n} & b_3 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & a^{n-1}_{nn} & b^{n-1}_n \end{array}\right]$$

В обратном ходе алгоритма выражаем итеративно в обратном порядке значения решения:

$$\begin{cases} a^{n-1}_{nn} x_n = b^{n-1}_n & \Longrightarrow x_n \\ a^{n-2}_{n-1} x_{n-1} + a^{n-2}_{n-1} x_n = b^{n-2}_{n-1} & \Longrightarrow x_{n-1} \\ --- & --- \\ a_{11} x_1 + \dots + a_{1n} x_n = b_1 & \Longrightarrow x_1. \end{cases}$$

Замечание

Условия совместности системы (теорема Кронекера-Капелли) никто не отменял

Замечание

Так же при таком методе возможно получения определителя, но надо хранить информацию о перестановке строк (определитель меняет знак):

$$\det A = a_{11} a^1_{22} \dots a^{n-1}_{nn}.$$

Сложность алгоритма: $\mathcal{O}(n^3)$.

Метод прогонки

Метод прогонки эффективен, но работает только для матриц с трехдиагональной структурой. Является частным случаем метода Гаусса. Пусть СЛУ имеет вид:

$$\begin{cases} b_1 x_1 + c_1 x_2 = d_1, \\ a_2 x_1 + b_2 x_2 + c_2 x_3 = d_2, \\ \qquad a_3 x_2 + b_3 x_3 + c_3 x + 4 = d_3, \\ --- \\ \qquad a_{n-1} x_{n-2} + b_{n-1} x_{n-1} + c_{n-1} x_n = d_{n-1} \\ \qquad\qquad a_n x_{n-1} + b_n x_n = d_n, \; c_n = 0 \end{cases}$$

Решение ищем в виде:

$$x_i = A_i x_{i+1} + B_i,$$

где A_i, B_i – прогоночные коэффициенты, подлежащие определению. Чтобы их выразить:

$$x_1 = \frac{-c_1}{b_1} x_2 + \frac{d_1}{b_1} = A_1 x_2 + B_1$$

Тогда

$$A_1 = \frac{-c_1}{b_1},\ B_1 = \frac{d_1}{b_1}.$$

Из второго уравнению аналогично выразим x_2 через x_3:

$$x_2 = \frac{-c_2}{b_2 + a_2 A_1} x_3 + \frac{d_2 - a_2 B_1}{b_2 + a_2 A_1} = A_2 x_3 + B_2,$$

откуда

$$A_2 = \frac{-c_2}{b_2 + a_2 A_1},\ B_2 = \frac{d_2 - a_2 B_1}{b_2 + a_2 A_1}.$$

Тогда

$$x_i = \frac{-c_i}{b_i + a_i A_{i-1}} x_{i+1} + \frac{d_i - a_i B_{i-1}}{b_i + a_i A_{i-1}},$$

следовательно:

$$A_i = \frac{-c_i}{b_i + a_i A_{i-1}},\ B_i = \frac{d_i - a_i B_{i-1}}{b_i + a_i A_{i-1}}.$$

Из последнего уравнения СЛУ:

$$x_n = \frac{-c_n}{b_n + a_n A_{n-1}} x_{n+1} + \frac{d_n - a_n B_{n-1}}{b_n + a_n A_{n-1}} = 0 \cdot x_{n+1} + B_n,$$

то есть

$$A_n = 0\ (c_n = 0), \qquad B_n = \frac{d_n - a_n B_{n-1}}{b_n + a_n A_{n-1}} = x_n$$

В результате прогоночные коэффициенты вычисляются следующим образом:

$$A_i = \frac{-c_i}{b_i + a_i A_{i-1}},\ B_i = \frac{d_i - a_i B_{i-1}}{b_i + a_i A_{i-1}},\quad i = \overline{2, n-1}$$

Причем (из-за того, что $a_1 = 0$)

$$A_1 = \frac{-c_1}{b_1},\ B_1 = \frac{d_1}{b_1}$$

И так как $c_n = 0$:

$$A_n = 0, \qquad B_n = \frac{d_n - a_n B_{n-1}}{b_n + a_n A_{n-1}}.$$

Обратный ход метода прогонки:

$$\begin{cases} x_n = A_n x_{n+1} + B_n = B_n, \\ x_{n-1} = A_{n-1} x_n + B_{n-1}, \\ x_{n-2} = A_{n-2} x_{n-1} + B_{n-2}, \\ --- \\ x_1 = A_1 x_2 + B_1. \end{cases}$$

Алгоритмическая сложность: $\mathcal{O}(8n + 1) = \mathcal{O}(n)$.

Итерационные методы решения СЛУ

Используются при большом количестве уравнений обычные методы труднореализуемы. Методы последовательных приближений, в которых в последующих вычислениях используются предыдущие называются итерационными.

Метод простых итераций

Рассмотрим СЛУ:

$$\begin{cases} a_{11}\underline{x_1} + a_{12}x_2 + \ldots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}\underline{x_2} + \ldots + a_{2n}x_n = b_2, \\ --- \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}\underline{x_n} = b_n \end{cases}$$

с невырожденной матрицей A. Представим в виде:

$$x = \beta + \alpha x,$$

где

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_n \end{bmatrix}, \quad \alpha = \begin{bmatrix} \alpha_{11} & \ldots & \alpha_{1n} \\ \vdots & \ddots & \vdots \\ \alpha_{n1} & \ldots & \alpha_{nn} \end{bmatrix}.$$

Разрешим первую систему относительно подчеркнутых неизвестных при ненулевых диагональных элементах (если равен – меняем местами). Получим выражения

$$\beta_i = \frac{b_i}{a_{ii}}; \qquad \alpha_{ij} = \begin{cases} -\dfrac{a_{ij}}{a_{ii}},\ i, j = \overline{1, n},\ i \neq j; \\ 0, \quad i = j,\ i = \overline{1, n}. \end{cases}$$

В качестве нулевого приближения выберем $x^{(0)} = \beta$, или $\left(x_1^{(0)}\ x_2^{(0)}\ \ldots\ x_n^{(0)}\right)^\top = (\beta_1\ \beta_2\ \ldots\ \beta_n)$. Тогда метод простых итераций – последовательные вычисления вида:

$$\begin{cases} x^{(0)} = \beta, \\ x^{(1)} = \beta + \alpha x^{(0)}, \\ x^{(2)} = \beta + \alpha x^{(1)}, \\ --- \\ x^{(k)} = \beta + \alpha x^{(k-1)}. \end{cases}$$

Теорема: (О сходимости МПИ)

Метод простых итераций сходится к единственному решению СЛУ при любом начальном приближении $x^{(0)}$, если какая-либо норма матрицы α эквивалентной системы меньше 1:

$$||\alpha|| < 1\ \forall ||\alpha|| \text{ и } \forall x^{(0)}.$$

Метод Зейделя

Метод простых итераций медленно сходится, для ускорения существует метод Зейделя. Идея заключается в том, что при вычислении компонента $x_i^{(k+1)}$ вектора неизвестных на $(k+1)$-й итерации используются значения $x_1^{(k+1)}, \dots, x_{i-1}^{(k+1)}$, уже вычисленные. Значения остальных $x_i^{(k)}, \dots, x_n^{(k)}$ из предыдущей итерации. Как и в МПИ $x^{(0)} = (\beta_1\ \beta_2\ \dots\ \beta_n)^\top$. Тогда метод Зейделя для известного вектора $\left(x_1^{(k)}\ x_2^{(k)}\ \dots\ x_n^{(k)}\right)^\top$ на k-ой итерации будет иметь вид:

$$\begin{cases} x_1^{(k+1)} = \beta_1 + \alpha_{11}x_1^{(k)} + \alpha_{12}x_2^{(k)} + \dots + \alpha_{1n}x_n^{(k)}, \\ x_2^{(k+1)} = \beta_2 + \alpha_{21}x_1^{(k+1)} + \alpha_{22}x_2^{(k)} + \dots + \alpha_{2n}x_n^{(k)}, \\ x_3^{(k+1)} = \beta_3 + \alpha_{31}x_1^{(k+1)} + \alpha_{32}x_2^{(k+1)} + \dots + \alpha_{3n}x_n^{(k)}, \\ --- \\ x_n^{(k+1)} = \beta_n + \alpha_{n1}x_1^{(k+1)} + \alpha_{n2}x_2^{(k+1)} + \dots + \alpha_{nn}x_n^{(k)} \end{cases}$$

Можно заметить $x^{(k+1)} = \beta + Bx^{(k+1)} + Cx^{(k)}$, где B – нижняя треугольная матрица с диагональными элементами, равными 0, а – верхнетреугольная с диагональными элементами, не равными нулю. $\alpha = B + C$. Итого:

$$x^{(k+1)} = (E - B)^{-1}Cx^{(k)} + (E - B)^{-1}\beta.$$

Таким образом, метод Зейделя – МПИ с матрицей правых частей $\alpha = (E - B)^{-1}C$ и вектором правых частей $\beta = (E - B)^{-1}\beta$.

5. Численное дифференцирование и интегрирование. Численные методы для решения систем дифференциальных уравнений.

6. Граф. Ориентированный граф. Представления графа. Обход графа в глубину и в ширину. Топологическая сортировка. Подсчет числа путей в орграфе.

Дадим определения графа и ориентированного графа.

Определение: (Неориентированный граф)

Пара $G = (V, E)$, где V – множество, $E \subset C_V^2$ (множество упорядоченных пар)

Определение: (Ориентированный граф)

Ориентированный граф $G = (V, E)$, V – множество, $E \subset V \times V$

Определение: (Путь)

Путь $v_1, v_2, \dots v_k$ – последовательность вершин таких, что $(v_1, v_2) \in E,\ (v_2, v_3) \in E, \dots, (v_{k-1}, v_k) \in E$.

Путь называется реберно простым, если ребра в нем не повторяются. Путь называется вершинно простым, если вершины в нем не повторяются.

Замечание

Из вершинной простоты следует реберная.

Определение: (Цикл)

Путь $v_1, \dots, v_k$ называется циклом, если $v_1 = v_k$.

Замечание

Определения реберной и вершинной простоты наследуются.

Определение: (Достижимость)

Из вершины u достижима вершина v, если существует путь $v_1, \dots, v_k\ :\ v_1 = u,\ v_k = v$.

Определение: (Связность)

Если G – неориентированный граф, то отношение связности:

$$u \sim v,$$

если из u есть путь в v.

Замечание

Отношение связности в неориентированном графе – отношение эквивалентности.

Определение

Классы эквивалентности по отношению связности – компоненты связности.

Определение

Если G – ориентированный граф, то введем отношение сильной связности:

$$u \sim v,$$

если $\exists$ путь из u в v и $\exists$ путь из v в u.

Замечание

σ – отношение эквивалентности.

Определение: (Комп-нты сильной связности)

Компоненты сильной связности – классы эквивалентности по отношению сильной связности.

Хранение графов

Матрица смежности

$$M_{ij} = \begin{cases} 1, & \text{если } (i,j) \in E \\ 0, & \text{иначе}. \end{cases}$$

+ за $\mathcal{O}(1)$ проверка наличия ребра;

- требует $\Omega(n^2)$ памяти.

Список ребер

$$\begin{array}{l} u_1, v_1 \\ u_2, v_2 \\ \vdots \\ u_m, v_m \end{array}$$

+ естественность;

- неудобство обработки.

Список смежности

$$\begin{array}{l} l_1 \\ \vdots \\ l_n \end{array} \qquad l_u = \{v : (u,v) \in E\}$$

+ требует $\mathcal{O}(n+m)$ памяти;

+ для каждой вершины знаем множество соседей;

- за $\mathcal{O}(1)$ нельзя проверить наличие ядра в графе.

DFS

```
std::vector<std::vector<int>> g;
std::vector<int> tin, tout;
int timer = 0;
std::vector<string> color; // WHITE
vector<int> parent;
void dfs(int v, int p = -1) {
    tin[v] = timer++;
    parent[v] = p;
    color[v] = GRAY; // GRAY -- IN WORK
    for (int to: g[v]) {
        if (color[to] != WHITE)
            continue;
        dfs(to, v);
    }
    tout[v] = timer++;
    color[v] = BLACK; // the end
}
```

Лемма: (о белых путях)

Если в момент $tin[v]$ есть некий путь из v по белым вершинам, то к моменту $tout[v]$ все вершины этого пути будут черными.

7. Алгоритмы поиска кратчайших путей в графе. Алгоритм Дейкстры. Алгоритм Форда-Беллмана. Алгоритм Флойда. Алгоритм A*.

8. Недетерминированные конечные автоматы, различные варианты определения. Детерминированные конечные автоматы. Их эквивалентность. Машина Тьюринга.