Коллоквиум

«Алгоритмы и структуры данных»

Если найдете опечатки или неправильность пишите

Автор

19 декабря 2025 г.

Содержание

1 Понятие временной и пространственной (по памяти) сложности алгоритма. Определение асимптотических сравнений O, Θ, Ω. Примеры

1.1 Временная и пространственная сложности алгоритма

Определение 1.1: Временная сложность алгоритма

Временной сложностью называется количество операций, выполняемых алгоритмом, в зависимости от входных данных.

Определение 1.2: Пространственная (по памяти) сложность алгоритма

Пространственной сложностью называется количество памяти, требуемой алгоритмом, в зависимости от входных данных.

1.2 Асимптотические сравнения

Определение 1.3: Асимптотическая положительность

Функция $f(n)$ является асимптотически положительной, если

$$\exists N_0 : \forall n > N_0, f(n) > 0.$$

Определение 1.4: Асимптотическая неотрицательность

Функция $f(n)$ является асимптотически неотрицательной, если

$$\exists N_0 : \forall n > N_0, f(n) \geq 0.$$

Определение 1.5: O-большое

Рассмотрим множество функций:

$$f(n) \in O(g(n)) \iff \exists N_0 > 0, \exists C > 0 : \forall n \geq N_0 \quad 0 \leq f(n) \leq C \cdot g(n)$$

Говорят что: f асимптотически не больше, чем g, или f не больше, чем g с точностью до константы. Также можно сказать что f это O-большое от g.

Определение 1.6: Ω (Омега)

Рассмотрим множество функций:

$$f(n) \in \Omega(g(n)) \iff \exists N_0 > 0, \exists C > 0 : \forall n \geq N_0 \quad f(n) \geq C \cdot g(n) \geq 0$$

Говорят что: f не меньше, чем g, или g — нижняя оценка для f. Также можно сказать что f это Ω от g.

Определение 1.7: θ (тета)

Рассмотрим множество функций:

$$f(n) \in \Theta(g(n)) \iff \exists N_0 > 0, \exists C_1 > 0, \exists C_2 > 0 : \forall n \geq N_0 \quad C_1 g(n) \leq f(n) \leq C_2 g(n)$$

Говорят что: $g(n)$ — асимптотически точная оценка $f(n)$. Также можно сказать что f это Θ от g.

Аналогии с неравенствами:

Соотношение	Аналогия
$f(n) = O(g(n))$	$a \leq b$
$f(n) = \Theta(g(n))$	$a = b$
$f(n) = \Omega(g(n))$	$a \geq b$

1.3 Примеры

Пример 1.1: Временная и пространственная сложность бинарного поиска

Временная сложность: $O(logn)$
Пространственная сложность: $O(1)$ (так как дополнительная память никак не зависит от входных данных)

Пример 1.2: Временная и пространственная сложность префикстых сумм

Временная сложность: $O(n)$
Пространственная сложность: $O(n)$ (если хотим сохранить исходный массив, создаём новый для префикстых сумм)

Пример 1.3: Временная и пространственная сложность дерева отрезков

Временная сложность построения: $O(nlogn)$
Временная сложность обработки запроса: $O(logn)$
Пространственная сложность: $O(n)$

2 Бинарный поиск. Бинарный поиск по ответу. Временная сложность

2.1 Бинарный поиск

Определение 2.1: Бинарный поиск

Бинарный поиск — алгоритм поиска элемента в отсортированном массиве, который делит поисковый интервал пополам, и продолжает поиск в нужной половине.

Алгоритм 2.1: Бинарный поиск

Вход: отсортированный массив arr, элемент $target$.
Выход: индекс элемента или -1, если не найден.

Реализация 2.1: Бинарный поиск

```
int binary_search(vector<int>& arr, int target) {
  int left = 0, right = arr.size() - 1;
  while (left <= right) {
    int mid = left + (right - left) / 2;
    if (arr[mid] == target) return mid;
    if (arr[mid] < target) left = mid + 1;
    else right = mid - 1;
  }
  return -1;
}
```

2.2 Бинарный поиск по ответу

Определение 2.2: Бинарный поиск по ответу

Бинарный поиск по ответу — алгоритм поиска ответа в диапазоне, который делит поисковый интервал пополам, и продолжает поиск в нужной половине.

Алгоритм 2.2: Бинарный поиск по ответу

Вход: отсортированный массив arr, требование $target$.
Выход: максимальная дельта между элементами, чтобы arr делился ровно на $target$ отрезков.

Реализация 2.2: Бинарный поиск по ответу

```
int check(const vector<int>& arr, int max_delta) {
  int count = 1;
  int begin_index = 0;
  for (size_t index = 1; index < arr.size(); ++index) {
    if (arr[index] - arr[begin_index] > max_delta) {
      ++count;
      begin_delta = index;
    }
  }
  return count;
}

int binary_search(const vector<int>& arr, int target) {
  int left = 0, right = INT_MAX;
  while (left <= right) {
    int mid = left + ((right - left) >> 1);
    if (target <= check(arr, mid)) {
      left = mid;
    } else {
```

```
      right = mid;
    }
  }
  return left;
}
```

2.3 Временная сложность

Рекуррентное соотношение: $T(n) = T(n/2) + \Theta(1)$.

По мастер-теореме ($a = 1, b = 2, c = 0$, случай 2): $T(n) = \Theta(\log n)$.

3 Вещественнозначный бинарный поиск (по ответу). Временная сложность

3.1 Вещественный бинарный поиск

Применяется для поиска корней уравнений или решения задач с вещественными числами, где требуется найти значение с заданной точностью.

Алгоритм 3.1: Вещественный бинарный поиск (поиск корня)

Задача: найти $\sqrt{n}$ с точностью ε.

Реализация 3.1: Вещественный бинарный поиск (поиск корня)

```
double sqrt_binary_search(double n, double epsilon = 1e-6) {
  double left = 0, right = n + 1;
  while (right - left > epsilon) {
    double mid = (left + right) / 2;
    if (mid * mid < n) left = mid;
    else right = mid;
  }
  return (left + right) / 2;
}
```

3.2 Вещественный бинарный поск по ответу

Просто совместить идеи бинарного поиска по ответу и вещественного бинарного поиска

3.3 Сложность вещественного бинарного поиска

Начальный интервал: $[0, n]$ (длина n). На каждой итерации интервал уменьшается в 2 раза. Требуемая точность: ε.

Количество итераций k: $\frac{n}{2^k} \leq \varepsilon \Rightarrow k \geq \log_2\left(\frac{n}{\varepsilon}\right)$.

Следовательно, $T(n) = O\left(\log\left(\frac{n}{\varepsilon}\right)\right)$.

4 Префиксные суммы. Обобщение на произвольную обратимую операцию

4.1 Префиксные суммы на массиве

Определение 4.1: Префиксные суммы

Префиксная сумма (prefix sum) массива $a[0..n-1]$ — это массив $p[0..n]$, где:

$$p[i] = \sum_{j=0}^{i-1} a[j] = a[0] + a[1] + \ldots + a[i-1]$$

Реализация 4.1: Построение префиксных сумм

```
vector<int> build_prefix_sum(vector<int>& arr) {
  vector<int> prefix(arr.size() + 1, 0);
  for (int i = 1; i <= arr.size(); i++) {
    prefix[i] = prefix[i-1] + arr[i-1];
  }
  return prefix;
}
```

Применение: быстрое вычисление суммы на отрезке $[l, r]$:

$$sum(l, r) = p[r+1] - p[l]$$

Реализация 4.2: Запрос суммы на отрезке

```
int range_sum(vector<int>& prefix, int l, int r) {
  return prefix[r+1] - prefix[l];
}
```

Сложность:

- Построение: $\Theta(n)$
- Запрос: $\Theta(1)$
- Память: $\Theta(n)$

4.2 Двумерные префиксные суммы

Для матрицы $a[m][n]$ строится матрица $p[m+1][n+1]$, где:

$$p[i][j] = \sum_{x=0}^{i-1} \sum_{y=0}^{j-1} a[x][y]$$

Реализация 4.3: Построение двумерных префиксных сумм

```
vector<vector<int>> build_prefix_sums_2d(const vector<vector<int>>& a) {
    int n = a.size();
    int m = a[0].size();
    vector<vector<int>> prefix_sum(n + 1, vector<int>(m + 1, 0));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            prefix_sum[i + 1][j + 1] = prefix_sum[i][j + 1] + prefix_sum[i + 1][j]
    - prefix_sum[i][j] + a[i][j];
        }
    }
    return prefix_sum;
}
```

Сумма в прямоугольнике $[x_1, y_1]$ до $[x_2, y_2]$:

$$sum = p[x_2 + 1][y_2 + 1] - p[x_1][y_2 + 1] - p[x_2 + 1][y_1] + p[x_1][y_1]$$

Реализация 4.4: Запрос суммы в прямоугольнике

```
int64_t get_sum(std::vector<std::vector<int>>& pref, int x1, int y1,
                int x2, int y2) {
return prefix[x2+1][y2+1] - prefix[x1][y2+1] - prefix[x2+1][y1]
       + prefix[x1][y1]
}
```

4.3 Обобщение на многомерный случай

Для обобщения на многомерный случай потребуется формула включения/исклчения. Применять по аналогии с одномерным и двуменым случаями в построении и поиске ответа.

4.4 Обобщение на произвольную обратимую операцию

Определение 4.2: Обратимая операция

Обратимой операцией называется операция, которую можно отменить, применив обратную операцию. Например, для * отменяющая операция — /, а для xor — сам xor.

По аналогии с суммой можно использовать любую другую обратимую операцию. Например, разность, умножение, деление, xor и т.д.

5 Факторизация числа. Функция Эйлера. Сводимость вычисления функции Эйлера к факторизации

5.1 Факторизация числа

Определение 5.1: Факторизцаия числа

Разбиение числа на его простые делители ($8 = 2 * 2 * 2$)

Алгоритм 5.1: Факторизация числа

Задача: Дано число n. Вывести его в виде произведения простых чисел.
Идея: Пройдёмся числом i по числам от 2 до $\sqrt{n}$. Если встретили число, тогда есть 2 варианта:

1. i - простое число. Если число n не делится на i, просто пропускаем это число. Иначе записываем в ответ это число i столько раз, сколько число n делится на i, а потом поделим n на i это количество раз. После этого n не будет делиться на i. То есть получится, что n больше не делится ни на одно простое число $<= i$.

2. i - составное число. Так как это число состоит из произведения простых чисел до i, а число n в данный момент не делится ни на одно простое число i, значит n гарантированно не делится на число i. То есть число n не делится ни на одно число $<= i$, будь то простое или составное.

В итоге получили, что число n или равно 1, или простое число (так как не делится ни на одно число $<= \sqrt{n}$), которое мы записываем в ответ.
Время работы: если число простое, то i просто пройдётся по числам от 2 до $\sqrt{n}$, что в итоге составит сложность $O(\sqrt{n})$. Если не простое, то получаем сложность не хуже $i + T(\frac{n}{2^i})$, что $T(n)$, значит меньше худшего случая, где n - простое число. (Лучшем случаем будет степень двойки, тогда сложность будет равна $O(logn)$)

Реализация 5.1: Факторицазия числа

```
std::vector<int> factorization(int number) {
  std::vector<int> answer;
  int num = 2;
  while (num * num <= number) {
    while (!(number % num)) {
      answer.push_back(num);
      number /= num;
    }
    ++num;
  }
  return answer;
}
```

5.2 Функция Эйлера

Определение 5.2: Функция Эйлера

Функция Эйлера $\varphi(a)$ определяется для всех целых положительных a и представляет собою количество чисел ряда $0, 1, \ldots, a-1$, взаимно простых с a.

5.3 Сводимость вычисления функции Эйлера к факторизации

Теорема 5.1: Формула для функции Эйлера

$$\varphi(a) = a\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\dots\left(1 - \frac{1}{p_k}\right)$$

где $p_1, p_2, \dots, p_k$ — различные простые делители числа a.

Пример: $\varphi(60) = 60 \cdot (1 - 1/2) \cdot (1 - 1/3) \cdot (1 - 1/5) = 60 \cdot 1/2 \cdot 2/3 \cdot 4/5 = 16$.

Задача 5.1: Выведение формулы Эйлера

1. Рассмотрим простой случай: n - простое число (то есть n = p). Очевидно, для любого простого числа количество чисел, взаимно простых с ним и меньших него равно p - 1.
2. Теперь пусть $n = p^\alpha$. Тогда числа, не взаимно простые с n и меньшие его - все числа, делящиеся на p. Таких ровно $\frac{p^\alpha}{p}$. Тогда $\varphi(n) = p^\alpha - \frac{p^\alpha}{p} = p^\alpha(1 - \frac{1}{p})$
3. Так как функция Эйлера мультипликативна (то есть $\varphi(a \cdot b) = \varphi(a) \cdot \varphi(b)$), если $n = p_1^{\alpha_1} \cdot \ldots \cdot p_k^{\alpha_k}$. Тогда $\varphi(n) = \varphi(p_1^{\alpha_1} \cdot \ldots \cdot p_k^{\alpha_k}) = \varphi(p_1^{\alpha_1}) \cdot \ldots \cdot \varphi(p_k^{\alpha_k}) = n(1 - \frac{1}{p_1})\ldots(1 - \frac{1}{p_k})$

6 Классическое решето Эратосфена за O(n log log n) (время работы б/д). Линейное решето Эратосфена

6.1 Решето Эратосфена, время работы

Алгоритм 6.1: Решето Эратосфена

Для составления таблицы простых чисел, не превосходящих данного целого N:

1. Выписываем числа $1, 2, \ldots, N$.
2. Первое, большее 1 число этого ряда есть 2. Оно делится только на 1 и само на себя и, следовательно, оно простое.
3. Вычеркиваем все кратные двум до N.
4. Теперь первое не вычеркнутое число есть 3. Оно не делится на 2. Оно простое, вычеркиваем все кратные 3.
5. И т.д.

Реализация 6.1: Решето Эратосфена

```
std::vector<int> Erat(int num) {
  std::vector<int> erat(num + 1, 0);
  for (int index = 2; index * index <= num; ++index) {
    if (!erat[index]) {
      for (int index1=index * index; index1 <= num; index1 += index) {
        erat[index1] = 1;
      }
    }
  }
  return erat;
}
```

Корректность:

Когда указанным способом будут вычеркнуты все числа кратные простым, меньших простого p, то все невычеркнутые, меньшие p^2, будут простыми. Действительно, всякое составное a, меньшее p^2, нами уже вычеркнуто как кратное его наименьшего простого делителя, который $\leq \sqrt{a} < p$.

Полезное примечание:

- При вычеркивании кратных простого p, это вычеркивание следует начинать с p^2.
- Составление таблицы простых чисел, не превосходящих N, будет закончено, когда вычеркнуты все составные кратные простым, не превосходящим $\sqrt{N}$.

Время работы решета Эратосфена: $O(n \log \log n)$ (без доказательства).

6.2 Линейное решето Эратосфена

Реализация 6.2: Решето Эратосфена

```
std::vector<int> Erat(int num) {
    std::vector<int> erat(num + 1, 0);
    std::vector<int> primes;
    for (size_t index = 2; index <= num; ++index) {
    if (erat[index] == 0) {
        primes.push_back(index);
        erat[index] = index;
    }
    for (size_t jndex = 0; jndex <       primes.size() && primes[jndex] <
        erat[jndex] && index * primes[jndex] <= num; ++jndex) {
        erat[index * primes[jndex]] = primes[jndex];
    }
    }
```

```
    return erat;
}
```

Идея:

Каждое составное число должно быть вычеркнуто 1 раз. Для этого пройдёмся по массиву и каждому числу запишем его наименьший делитель (см. реализацию)

Корректность:

Это довольно сложная тема, в которой я до конца не разобрался. Буду рад, если разъясните в личке, но на экзамене скорее всего такого не спросят. Если спросят, скорее всего вам попался Костя, поэтому соболезную)

Время работы:

$O(n)$ (б/д)

7 Арифметика в $\mathbb{Z}_m$. Малая теорема Ферма и теорема Эйлера (б/д). Критерий существования обратного по модулю. Поиск обратного по модулю.

7.1 Модульная арифметика

Определение 7.1: Сравнимость по модулю

Два целых a и b называются равноостаточными по модулю m или сравнимыми по модулю m, если они дают одинаковый остаток при делении на m. Сравнимость чисел a и b по модулю m записывается так:

$$a \equiv b \pmod{m}$$

Реализация 7.1: Модульная арифметика

```
const int64_t MOD = 1e9 + 7;

int64_t BinPow(int64_t a, int64_t n) {
  if (n == 1) {
    return a % MOD;
  }
  if (n % 2) {
    return (a * BinPow(a, n - 1)) % MOD;
  }
  int64_t ans = BinPow(a, n / 2);
  return ans * ans % MOD;
}

int64_t InverseModulo(int64_t a) {
    return BinPow(a, MOD - 2);
}

int64_t PlusMod(int64_t a, int64_t b) {
    return (a + b) % MOD;
}

int64_t MinusMod(int64_t a, int64_t b) {
    return (a - b + MOD) % MOD;
}

int64_t MultiplyMod(int64_t a, int64_t b) {
    return (a * b) % MOD;
}

int64_t DivideMod(int64_t a, int64_t b) {
    return (a * InverseModulo(b)) % MOD;
}
```

Алгоритм 7.1: Быстрое возведение в степень

Задача: Нужно возвести число a в степень n по простому модулю m
Реализация: см. выше
Идея: если у нас n чётная, то a^n можно записать как $(a^{n/2})^2$, вычислить рекурсивно $a^{n/2}$ и полученный ответ возвести в квадрат за $O(1)$. В итоге мы каждый раз (в худшем случае через раз) делим степень на 2 и рекурсивно получаем ответ, поэтому
Временная сложность: $O(\log n)$

7.2 Малая теорема Ферма и теорема Эйлера (б/д)

Теорема 7.1: Теорема Эйлера

Если $\gcd(a, m) = 1$, то

$$a^{\varphi(m)} \equiv 1 \pmod{m}$$

Теорема 7.2: Малая теорема Ферма

Частный случай теоремы Эйлера для простого p: если $\gcd(a, p) = 1$, то

$$a^{p-1} \equiv 1 \pmod{p}$$

7.3 Критерий существования обратного по модулю

Теорема 7.3: Критерий существования обратного по модулю

Обратное число x числу a по модулю m существует тогда и только тогда, тогда a и m взаимо просты.

Необходимость: Пусть НОД(a, m) = d. Тогда $ax \equiv 1(modm) \Leftrightarrow ax - mk = 1$. Т.к. и a, и m делится на d, всё выражение $ax - mk$ делится на d, значит и 1 делится на d, но единственный делитель 1 — само число 1. Значит $d = 1$

Достаточность: НОД(a, m) = $1 \Rightarrow \exists u, v : au + mv = 1$. Рассмотрим это уравнение по модулю m. Так как $mv \equiv 0(modm)$, $au \equiv 1(modm)$. Ч.Т.Д.

7.4 Поиск обратного по модулю

Идея: Рассмотрим частный случай простого модуля p. По малой теореме Ферма: $a^{p-1} \equiv 1(modp)$, следовательно, $a^{p-2} \equiv a^{-1}(modm)$. Тогда, используя быстрое возведение в степень, найдём a^{p-2}, что и будет являться обратным по простому модулю p для числа a. Сложность алгоритма для простого модуля: $O(logn)$

Теперь рассмотрим общий случай не простого модуля m. По теореме Эйлера: $a^{\varphi}(m) \equiv 1(modm)$, следовательно, $a^{\varphi(m)-1} \equiv a^{-1}(modm)$. Тогда, найдя $\varphi(m)$ за $\sqrt{m}$ через факторизацию числа, и используя быстрое возведение в степень, получим обратное по модулю m. Сложность такого алгоритма: $O(\sqrt{m} + \log n)$

8 Критерий существования обратного по модулю. Поиск обратного по модулю. Алгоритм Евклида (классический, б/д)

8.1 Критерий существования обратного по модулю

См. выше

8.2 Поиск обратного по модулю

См. выше

8.3 Алгоритм Евклида (классический, б/д)

Теорема 8.1: Основная теорема алгоритма Евклида

Если $a = bq + r$, то $(a, b) = (b, r)$.

Алгоритм 8.1: Алгоритм Евклида

Пусть a и b — положительные целые, и $a > b$, тогда по теореме о делении с остатком получим систему равенств:

$$\begin{aligned} a &= bq_1 + r_2, \quad 0 < r_2 < b \\ b &= r_2q_2 + r_3, \quad 0 < r_3 < r_2 \\ r_2 &= r_3q_3 + r_4, \quad 0 < r_4 < r_3 \\ &\vdots \end{aligned}$$

Этот ряд можно продолжать пока не получим 0. Заметим, что:

$$(a, b) = (b, r_2) = (r_2, r_3) = \ldots = (r_{n-1}, r_n) = r_n$$

Алгоритм 8.2: Реализация алгоритма Евклида

Реализация 8.1: Алгоритм Евклида

```
int euclid(int a, int b) {
    while (a && b) (a > b ? a %= b : b %= a);
    return a + b;
}
```

Сложность алгоритма Евклида: $O(\log(\min(a, b)))$.

9 Амортизированный анализ. Метод монеток. Метод потенциалов. Применение на примере push_back в динамическом массиве

9.1 Амортизационный анализ

Определение 9.1: Амортизационный анализ

Пусть наша программа состоит из элементарных кусков (операций), i-й из которых работает t_i.

- Реальное время – t_i
- Среднее время – $t_{\text{avg}} = \frac{\sum_i t_i}{n}$
- Амортизированное время одной операции – $a_i = t_i + \Delta\varphi_i$

9.2 Метод монет

9.2.1 Принцип

Определение 9.2

В методе в ходе группового анализа, разные операции оцениваются по-разному, в зависимости от их фактической стоимости. Величина, которая начисляется на операцию, называется амортизированной стоимостью (amortized cost). Если амортизированная стоимость операции превышает ее фактическую стоимость, то соответствующая разность присваивается определенным объектам структуры данных как кредит (credit). Кредит можно использовать впоследствии для компенсирующих выплат на операции, амортизированная стоимость которых меньше их фактической стоимости.

Важно: К выбору стоимости стоит подходить осторожно. Должно выполняться соотношение:

$$\sum_i a_i \geq \sum_i t_i$$

То есть полный кредит в любой момент времени должен быть неотрицательным.

9.2.2 Пример

Задача 9.1: push_back

Задача: доказать, что push_back в динамическом массиве работает за $O^*(1)$
Решение: применим метод монет:
Если в данный момент есть место для вставки элемента, сделаем это и добавим в банк 2 монеты (за выделение место под 2 элемента в будущем). Когда у нас закончится место, заберём деньги из банка и пустем их на выделение места под новые элемента. Это нам ничего не зудет стоить. Поэтому амортизированно у нас push_back будет работать за $O^*(1)$.

9.3 Метод потенциалов

9.3.1 Принцип

Определение 9.3: Метод потенциалов

Функция потенциала φ — отображает структуру данных D_i на действительное число $\varphi(D_i)$, которое является потенциалом, связанным со структурой данных. Амортизированная стоимость i-ой операции:

$$a_i = t_i + \varphi(D_i) - \varphi(D_{i-1})$$

Соотношение:

$$\sum_i a_i = \sum_i (t_i + \varphi(D_i) - \varphi(D_{i-1})) = \sum_i t_i + \varphi(D_n) - \varphi(D_0)$$

Если потенциал можно определить так, что $\varphi(D_n) - \varphi(D_0) \geq 0$, то суммарная амортизированная стоимость даст верхнюю границу полной фактической стоимости.

Соотношение: Если потребовать $\varphi(D_i) \geq \varphi(D_0), \forall i$, то мы как и в методе бухучета получим предоплату операций.

9.3.2 Пример

Задача 9.2: push_back

Задача: доказать, что push_back в динамическом массиве работает за $O^*(1)$
Решение: применим метод потенциалов:
Пусть $\varphi(D) = 2 \cdot size - capacity$
Тогда:

1. Без переаллоцирования памяти: $a_i = 1 + 2 \cdot (size + 1) - capacity - 2 \cdot size + capacity = 3$
2. С переаллоцированием памяти: $a_i = 1 + 2 \cdot (size + 1) - capacity \cdot 2 - 2 \cdot size + capacity \cdot 2 = 3$

10 Линейные контейнеры. Односвязный и двусвязный списки. Стек, очередь, дек (через списки и через кольцевой буффер)

10.1 Односвязный и двусвязный списки

Определение 10.1: Односвязный список

Односвязный список — последовательный набор из узлов с данными, где каждый узел знает, где лежит следующий за ним.

Реализация 10.1: Односвязный список

```
struct Node {
    int val;
    Node* next;

    Node(int _val) : val(_val), next(nullptr) {}
};
struct list {
    Node* first;
    Node* last;

    list() : first(nullptr), last(nullptr) {}

    bool is_empty() {
        return first == nullptr;
    }

    void push_back(string _val) {
        Node* p = new Node(_val);
        if (is_empty()) {
            first = p;
            last = p;
            return;
        }
        last->next = p;
        last = p;
    }

    void print() {
        if (is_empty()) return;
        Node* p = first;
        while (p) {
            cout << p->val << " ";
            p = p->next;
        }
        cout << endl;
    }
}
```

Определение 10.2: Двусвязный список

Двусвязный список — последовательный набор из узлов с данными, где каждый узел знает, где лежат следующий и предыдущий узлы.

Реализация 10.2: Двусвязный список

```
struct Node {
    int val;
    Node* next;
    Node* before*

    Node(int _val) : val(_val), next(nullptr) {}
};
```

```
struct list {
    Node* first;
    Node* last;

    list() : first(nullptr), last(nullptr) {}

    bool is_empty() {
        return first == nullptr;
    }

    void push_back(string _val) {
        Node* p = new Node(_val);
        if (is_empty()) {
            first = p;
            last = p;
            return;
        }
        last->next = p;
        p->before = last;
        last = p;
    }

    void print() {
        if (is_empty()) return;
        Node* p = first;
        while (p) {
            cout << p->val << " ";
            p = p->next;
        }
        cout << endl;
    }
}
```

10.2 Стек, очередь, дек (через списки и через кольцевой буффер)

Определение 10.3: Стек (stack)

Стек — АТД, который хранит элементы и предоставляет к ним доступ в рамках парадигмы LIFO (Last in, First Out).

Определение 10.4: Очередь (queue)

Очередь — АТД, который хранит элементы и предоставляет к ним доступ в рамках парадигмы FIFO (First in, First Out).

Определение 10.5: Дек (deque)

Дек — АТД, который представляет из себя двустороннюю очередь, то есть можно вставлять-/удалять в начало/конец.

Задача 10.1: Дек через списки

Идея: Дек можно реализовать с помощью двусвязного списка с операциями добавления в начало/конец и удаления из начала/конца. Однако при такой реализации обращение к элементу дека будет работать за $O(n)$, что подходит под требования, написанные в определении, однако это нас не устраивает, поэтому дек можно реализовать с помощью циклического буффера.

Задача 10.2: Дек через циклический буффер

Идея: Будем заполнять циклический буффер, пока есть место, если место закончилось и мы хотим добавить элемент, расширим capacity в два рада (по аналогии с динамическим массовом), и продолжим заполнение. Для лучшего понимания см. реализоцию

Реализация 10.3: Дек на циклическом буффере

```
  struct RingBuffer{
  void reserve(int new_cap) {
    if (new_cap <= capacity) {
      return;
    }
    int* tmp = new int[new_cap];
    for (int index = 0; index < size; ++index) {
      tmp[index] = data[(begin + index) % capacity];
    }
    delete[] data;
    begin = 0;
    capacity = new_cap;
    data = tmp;
  }

  void push_back(int elem) {
    if (size == capacity) {
      reserve(capacity * 2);
    }
    data[(begin + size) % capacity] = elem;
    ++size;
  }

  void push_front(int elem) {
    if (size == capacity) {
      reserve(capacity * 2);
    }
    begin = (begin + capacity - 1) % capacity;
    data[begin] = elem;
    ++size;
  }

  void pop_back() {
    if (size == 0) {
      return;
    }
    --size;
  }

  void pop_front() {
    if (size == 0) {
      return;
    }
    --size;
    begin = (begin + 1) % capacity;
  }

  int size = 0;
  int begin = 0;
  int capacity = 1;
  int* data = new int[1];
};
```

11 Линейные контейнеры. Очередь с поддержкой произвольной ассоциативной операции

11.1 Стек с поддержкой минимума

Давайте добавим к интерфейсу стека еще возможность возвращать минимум в нем. Будем хранить в узле стека не только сам элемент, но еще и минимум в стеке. При добавлении нового элемента в стек S для получения стека S' будем в голову S' добавлять минимум как минимум из значения элемента и значения минимума в голове S'.

Алгоритм 11.1: Идея реализации

Будем просто поддерживать стек, в котором первым элементом хранить сам элемент, а вторым - текущий минимум. При добавлении элемента новым минимумом будет или он или предыдущий элемент. Так как у нас есть доступ только к верхнему элементу, остальные минимумы пересчитывать не надо

Реализация 11.1: Минимум на стеке

```
#include <stack>

struct MinStack {
public:
    void Push(int element) {
        int min_element = (data_.empty() ? element :
        std::min(data_.top().second, element));
        data_.push({element, oper_element});
    }
    void Pop() {
        if (!data_.empty()) {
            data_.pop();
        }
    }
    int Top() {
        if (data_.empty()) {
            return -1;
        }
        return data_.top().first;
    }
    int Get() {
        if (data_.empty()) {
            return -1;
    }
        return data_.top().second;
    }
    bool Empty() {
        return data_.empty();
    }
private:
    std::stack<std::pair<int,int>> data_;
}
```

11.2 Очередь на двух стеках:

Задача. Дан стек в виде черного ящика. Надо соорудить очередь. Одного стека не хватит, воспользуемся двумя. Сделаем один стек на вход stack_in и второй на выход stack_out. В первый будем добавлять, а из второго — извлекать. Если при извлечении stack_out пуст, то перекидываем все элементы из stack_in в stack_out.

11.3 Очередь с минимумом:

Задача. Сделать очередь с поддержкой операции получения минимума в ней. Заведем очередь на двух стеках, поддерживающих минимум. Запрос минимума сводится к минимуму из минимумов в двух стеках.

Алгоритм 11.2: Идея реализации

Сделаем стек на минимуме, но когда удаляем элемент, будем удалять тот, что лежит в обратном стеке. Если он пустой, по "перельём"все элементы из основного стека в обратный. В итоге они будут иметь обратный порядок и ударяя верхний элемент в обратном стеке мы фактически будем удалять нижний элемент в обычном стеке.

Реализация 11.2: Минимум на очереди

```
#include<stack>
struct MinQueue {
public:
    void Push(int element) {
        in_data.Push(element);
    }
    void Pop() {
        if (out_data_.Empty()) {
            while (!in_data_.Empty()) {
                out_data_.Push(in_data_.Top());
            in_data_.Pop();
            }
        }
        if (out_data_.Empty()) {
            return;
        }
        out_data_.Pop();
    }
    int GetMin() {
        if (out_data_.Empty() && in_data_.Empty()) {
            return 0;
        }
        if (out_data.Empty()) {
            return in_data_.GetMin();
        }
        if (in_data_.Empty()) {
            return out_data_.GetMin();
        }
        return std::min(in_data_.GetMin(), out_data_.GetMin());
    }
    bool Empty() {
        return in_data.Empty() && out_data_.Empty();
    }
private:
    MinStack out_data_;
    MinStack in_data_;

}
```

11.4 Обобщение до произвольной ассоциативной операции

По аналогии с минимумом можно использовать любую ассоциативную операцию, будут небольшие различия в реализации (например, под такую реализацию можно легко подставить плюс в качестве ассоциативной операции, однако для минуса реализация будет немного отличаться)

12 Сортировки. Теорема о нижней границе времени сортировки. Стабильные сортировки

12.1 Сортировки

Постановка задачи:

Имеется последовательность из n элементов $x_1, \ldots, x_n$. Необходимо упорядочить элементы по неубыванию (невозрастанию), для удобства далее будем везде сортировать по неубыванию.

Значит нужно найти перестановку элементов: $p_1, p_2, p_3, \ldots, p_n$, т. ч. $x_{p_1} \leq x_{p_2} \leq \ldots \leq x_{p_n}$.

12.2 Теорема о нижней границе времени сортировки

Определение 12.1: Сортировка, основанная на сравнениях

Сортировку называют основанной на сравнениях, если она работает в следующем предположении: объекты можно только сравнивать.

Теорема 12.1: Нижняя оценка сортировки сравнениями

Сортировка, основанная на сравнениях, работает за $\Omega(N \log N)$.

Доказательство:

Необходимо найти единственную корректную перестановку среди их множества — всего есть $O(n!)$.

Всё что мы можем сделать это сравнить два элемента и понять, нужно ли их переставить. Число перестановок, которые остаются к исследованию: $\frac{n!}{2}$.

Тогда $T(n) \geq \log(n!) = \Omega(n \log n)$.

Теорема 12.2: lemm о факториале

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

Доказательство:

По формуле Стирлинга:

$$\log N! \sim \log \sqrt{2\pi n} \left(\frac{n}{e}\right)^n = \frac{1}{2}\log(2\pi n) + n \log n - n \log e = \Theta(N \log N)$$

12.3 Стабильные сортировки

Определение 12.2: Устойчивость (стабильность) сортировки

Устойчивость (стабильность) сортировки говорит о том, что элементы, одинаковые для сравнения, не поменяют своего расположения относительно друг друга после сортировки.

13 Сортировка слиянием. Посчёт количества инверсий

13.1 Сортировка слиянием

Идея «Разделяй и властвуй»:

1. Разбить массив на две примерно равные половины.
2. Решить все подзадачи, применив рекурсивно шаги 1 и 3, пока не дойдем до базы, в которой все тривиально (разбиваем массивы и дальше пополам, пока не дойдем до массива из одного элемента).
3. Объединить решения подзадач (слить два отсортированных массива в один большой).

Слияние двух отсортированных массивов:
Если у нас есть два отсортированных массива — можно их слить в третий, который будет тоже отсортирован за $O(n+m)$, где n, m — размеры массивов. Достигается это методом двух указателей.
Анализ сортировки слиянием:
Время работы составит $n\log(n)$, согласно мастер-теореме о рекурсии: $T(n) = 2T(n/2) + n \Rightarrow T(n) = \Theta(n \log n)$.
Потребляемая память составит $O(n)$, поскольку необходимо завести массив для слияний. Также алгоритму нужно $O(\log n)$ стековой памяти для хранения параметров рекурсивных вызовов.
К счастью, алгоритм можно реализовать итеративно и без рекурсии, начав решать задачу снизу-вверх: сначала сливаем соседние массивы размеров 1, затем размеров 2, затем 4, и так далее.

Реализация 13.1: Сортировка Слиянием

```
vector<int> Merge(std::vector<int> lhs, std::vector<int> rhs) {
    std::vector<int> answer(lhs.size() + rhs.size());
    size_t lhs_index = 0;
    size_t rhs_index = 0;
    while (lhs_index != lhs.size() || rhs_index != rhs.size()) {
        if (lhs_index == lhs.size()) {
            answer[lhs_index + rhs_index] = rhs[rhs_index++];
        } else if (rhs_index == rhs.size()) {
            answer[lhs_index + rhs_index] = lhs[lhs_index++];
        } else {
            if (lhs[lhs_index] <= rhs[rhs_index]) {
                answer[lhs_index + rhs_index] = lhs[lhs_index++];
            } else {
                answer[lhs_index + rhs_index] = rhs[rhs_index++];
            }
        }
    }
    return answer;
}
// vec - [0, n)
std::vector<int> MergeSort(const std::vector<int>& vec, size_t left_index,
size_t right_index) {
    if (right_index - left_index == 1) {
        return {vec[left_index]};
    }
    size_t min_index = (right_index + left_index) / 2;
    auto sorted_segment_1 = MergeSort(vec, left_index, mid_index);
    auto sorted_segment_2 = MergeSort(vec, mid_index, right_index);
    return Merge(sorted_segment_1, sorted_segment_2);
}
```

13.2 Подсчёт количества инверсий

Идея: при сортировке слиянием будем проверять на новые инверсии: если элемент с индексом j из правого массива оказался меньше элемента из левого с индексом i (в ходе их сравнения), то все элементы, начиная с элемента i образуют инверсии с элементом j из правого массива. Значит к ответу нам надо прибавить lhs.size() - i (количество элементов после i, включая сам i).

Реализация 13.2: Сортировка слиянием с подсчётом инверсий

```
int Merge(std::vector<int>& vec, int left_st, int left_end, int right_st, int
right_end) {
  int answer = 0;
  int l_index = left_st;
  int r_index = right_st;
  std::vector<int> ans;
  ans.reserve(left_end - left_st + right_end - right_st);
  while (l_index != left_end || r_index != right_end) {
    if (l_index == left_end) {
      ans.push_back(vec[r_index]);
      ++r_index;
    } else if (r_index == right_end) {
      ans.push_back(vec[l_index]);
      ++l_index;
    } else {
      if (vec[l_index] <= vec[r_index]) {
        ans.push_back(vec[l_index]);
        ++l_index;
      } else {
        answer += (left_end - l_index);
        ans.push_back(vec[r_index]);
        ++r_index;
      }
    }
  }
  for (int index = left_st; index < left_end; ++index) {
    vec[index] = ans[index - left_st];
  }
  for (int index = right_st; index < right_end; ++index) {
    vec[index] = ans[index + left_end - left_st - right_st];
  }
  return answer;
}

int MergeSort(std::vector<int>& vec, int left, int right) {
  if (right - left == 1) {
    return 0;
  }
  int mid = left + (right - left) / 2;
  int ans1 = MergeSort(vec, left, mid);
  int ans2 = MergeSort(vec, mid, right);
  return ans1 + ans2 + Merge(vec, left, mid, mid, right);
}
```

P.S. Здесь используется не стандартное слияние двух отсортированных массивов, а слияние двух отсортированных отрезков одного массива.

14 Бинарная пирамида. Определение. Поддержка свойства пирамиды. Основные операции. Пирамидальная сортировка

14.1 Определение

Определение 14.1: Двоичная куча (пирамида)

Двоичная куча (пирамида) — подвешенное бинарное дерево, обладающее следующими свойствами:

- Значение элемента в каждой вершине не меньше, чем значения в потомках.
- На i-ом (нумерация ведется с нуля) ярусе дерева, кроме быть может последнего — 2^i вершин
- Последний слой заполнен слева направо

Замечание: При желании можно вместо бинарной пирамиды использовать k-ичную пирамиду (например тернарную). Это уменьшает число ярусов дерева, необходимых для хранения элементов. На основе бинарной пирамиды можно реализовать очередь с приоритетом.

Лемма 14.1: Высота бинарной кучи

Число ярусов (высота) бинарной кучи из n элементов можно оценить, как $O(\log n)$.

Доказательство: Рассмотрим все ярусы кучи, без последнего. Тогда у нас имеется полная бинарная куча в которой $k < n$ элементов и ее высота на 1 меньше. У такой кучи каждый следующий ярус содержит в два раза больше элементов. Пусть высота этой кучи равна h. Тогда:

$$k = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

Из этого следует, что $h \leq \log_2 k$. Поскольку исходная куча содержит на 1 ярус больше, ее высоту можно оценить как $O(\log n)$.

Хранение в памяти:

Хранить кучу удобно в массиве (используем 0-индексацию). Пусть мы рассматриваем элемент с индексом i:

- Индекс левого ребенка: $2i + 1$
- Индекс правого ребенка: $2i + 2$
- Индекс родителя: $\lfloor \frac{i-1}{2} \rfloor$
- Элементы, которые являются листами, имеют индексы $\lceil \frac{n}{2} \rceil \dots n - 1$

14.2 Основные операции

Операция Sift Up (Просеивание вверх):

Сравним элемент с родительским элементом, если он меньше — поменяем их местами. Продолжим выполнение операции до тех пор, пока родитель не окажется больше или равным значению нашего элемента, либо пока элемент не станет корнем.

Операция Sift Down (Просеивание вниз):

Сравним элемент с дочерними элементами, если один из них больше, чем этот элемент, поменяем местами с большим из дочерних элементов. Продолжим выполнение операции до тех пор, пока элемент не окажется больше всех детей или не окажется листом.

Основные операции:

1. **Вставка:** Имеется куча (возможно, пустая), мы хотим добавить в нее элемент. Свойства кучи позволяют только добавить его на последний ярус. Такое добавление может нарушить свойства кучи: добавленный элемент может оказаться больше родителя. Исправить проблему возможно, если выполнить операцию SiftUp. Операция будет работать за $O(\log n)$.

2. **Извлечение максимума:** Имеется непустая куча, хотим извлечь максимальный элемент. Исходя из свойств кучи максимум всегда находится в корне. Также согласно свойствам кучи (все ярусы кроме последнего должны быть заполнены), на место извлекаемого элемента необходимо что-то поставить. Для этого подойдет последний элемент последнего яруса, но он может оказаться меньше какого-либо из детей корня. Исправить проблему возможно, если выполнить операцию sift down. Операция будет работать за $O(\log n)$.

3. **Increase/Decrease key:** Имея доступ к произвольному ключу можно поменять его значение. Операция будет работать за $O(\log n)$.

Построение кучи:

Можно построить за $O(n \log n)$, просто сделав n операций вставки. Можно сделать умнее (за $O(n)$). Давайте с поочередно от элемента с индексом $\frac{n}{2}$ до начала массива вызывать операцию SiftDown. Это будет работать, так как кучи из 1 вершины корректны, а далее мы восстанавливаем кучи из $k+1$ элемента и т. д.

Теорема 14.1: Построение кучи за O(n)

Построение бинарной кучи из n элементов (методом, описанным выше) работает за $O(n)$.

Реализация 14.1: Реализация бинарной кучи

```
struct Heap {
    public:
    Heap (const std::vector<int>& arr) : data_(arr) {
        for (size_t index = arr.size(); index + 1 > 0; ++index) {
            SiftDown(index);
        }
    }
    void Insert(const int& number) {
        data_.push_back(number);
        SiftUp(number);
    }
    int EraseMax() {
        std::swap(data_[0], data_.back());
        int answer = data_.back();
        data_.pop_back();
        SiftDown(0);
        return answer;
    }
    private:
    void SiftDown(size_t index) {
        if (((index + 1) << 1) >= data_.size()) {
            return;
        }
        size_t lchild = (index << 1) + 1;
        size_t rchild = (index << 1) + 2;
        if (data_[index] < data_[lchild]) {
            if (data_[index] < data_[rchild]) {
                if (data_[lchild] < data_[rchild]) {
                    std::swap(data_[index], data_[rchild]);
                    SiftDown(rchild);
                } else {
                    std::swap(data_[index], data_[lchild]);
                    SiftDown(lchild);
                }
            } else {
            std::swap(data_[index], data_[lchild]);
            SiftDown(lchild);
            }
        } else if (data_[index] < data_[rchild]) {
            std::swap(data_[index], data_[rchild]);
            SiftDown(rchild);
        }
    }
    void SiftUp(size_t index) {
        if (index == 0) {
            return;
        }
        if (data_[index] > data_[((index - 1) >> 1)]) {
```

```
            std::swap(data_[index], data_[((index - 1) >> 1)]);
            SiftUp(((index - 1) >> 1));
        }
    }
    std::vector<int> data_;
};
```

14.3 Пирамидальная сортировка

Алгоритм:

1. Хотим отсортировать массив A по неубыванию.
2. Построим бинарную кучу (на максимум) на основе массива A (в самом массиве, т. е. inplace).
3. Будем извлекать по очереди элементы пирамиды и ставить их в конец массива (это возможно, поскольку после извлечения из пирамиды в конце массива образуется неиспользованная ячейка).
4. В итоге получим отсортированный массив.

Анализ:

- Время работы пирамидальной сортировки составляет $O(n \log n)$
- Потребление доп. памяти пирамидальной сортировки составляет $O(1)$ (если не использовать рекурсию для реализации кучи)
- Сортировка не является устойчивой.

Первое верно, так как извлечение из кучи составляет $O(\log n)$. Второе — поскольку все операции можно без создания дополнительных массивов. Третье, поскольку в пирамиде равные элементы могут уйти в разные поддеревья пирамиды.

Примечание: Пирамида основана только на сравнениях $\Rightarrow$ сортировка тоже основана на сравнениях. Заметим, что она работает за $O(n \log n)$ даже в худшем случае.

Реализация 14.2: Heapsort

```
void heapify(int arr[], int n, int i){
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < n && arr[l] > arr[largest])
        largest = l;
    if (r < n && arr[r] > arr[largest])
        largest = r;
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    for (int i = n - 1; i >= 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
```

15 Быстрая сортировка. Процедура Patrition $O(1)$ доппамяти. Оценка времени со случайным pivot (б/д)

15.1 Быстрая сортировка

Алгоритм быстрой сортировки:

1. Как-нибудь выберем опорный элемент pivot (важно: значение должно присутствовать в массиве, например можно брать самый правый/самый левый/центральный).
2. Разобьем массив на две части, в одной все элементы больше или равны pivot, в другой меньше.
3. Поставим pivot на границе раздела (во избежание бесконечной рекурсии).
4. Рекурсивно решим две образованные подзадачи (шаги 1-3).

15.2 Разбиения

Разбиение Ломуто:

1. Заведем два указателя l, r, изначально оба показывают на начало массива.
2. Пойдем слева направо, l – будет показывать границу, до которой все элементы меньше опорного.
3. Таким образом для разбиения хватит одного цикла.

Реализация 15.1: Разбиение Ломуто

```
int lomuto_partition(int* a, int l, int r) {
  int pivot = a[l + (r - l) / 2];
  swap(&a[r], &a[l + (r - l) / 2]);
  int i = l;
  for (int j = l; j <= r; ++j) {
    if (a[j] < pivot) {
      swap(&a[i++], &a[j]);
    }
  }
  swap(&a[i], &a[r]);
  return i;
}
```

Разбиение Хоара:

1. Заведем два указателя l, r, левый начинает с начала, правый с конца.
2. Устремим их на встречу друг другу.
3. r пропускает все элементы, которые $\geq$ pivot, l – все, которые меньше pivot. Когда оба находят элементы, стоящие не так, меняем местами a_l и a_r.
4. Когда l, r встретятся – мы получим необходимое разбиение.

Реализация 15.2: Разбиение Хоара

```
int hoar_partition(int* a, int l, int r) {
  int piv_idx = l + (r - l) / 2;
  int pivot = a[piv_idx];
  int i = l;
  int j = r;
  while (i <= j) {
    while (a[i] < pivot) { i++; }
    while (a[j] > pivot) { j--; }
    if (i >= j) {
      return j;
    }
    swap(&a[i++], &a[j--]);
```

```
    }
    return j;
}
```

Хоар vs Ломуто:

На практике разбиение Хоара обычно работает заметно быстрее, чем разбиение Ломуто. Интуитивно это можно объяснить тем, что при разбиении Хоара сначала меняются местами как можно более удаленные друг от друга элементы, благодаря чему в массиве быстрее уменьшается число инверсий. На просторах интернета я нашел оценку числа обменов, которая для Хоара в 3 раза меньше чем для Ломуто.

Толстое разбиение (задача флага Нидерландов):

Поскольку числа обладают свойством трихотомии, решение задачи флага Нидерландов можно применить к разбиению элементов на три группы: $<$ pivot, $=$ pivot, $>$ pivot.

1. Заведем два указателя l, mid, r.
2. $l = 0, r = n - 1, \text{mid} = 0$
3. Будем поддерживать инвариант: $\text{arr}[r \dots n-1] > \text{pivot}$, $\text{arr}[0 \dots l] < \text{pivot}$, $\text{arr}[l \dots \text{mid}] == \text{pivot}$.

15.3 Оценка времени с случайным pivot (б/д)

Лемма 15.1: Среднее время работы

При случайном выборе pivot: $\mathbb{E}[X] = O(n \log n)$.

16 Поиск порядковой статистики. Оценка времени с случайным pivot

Определение 16.1: k-порядковая статистика

k-порядковая статистика — это k-ый по величине элемент.

Алгоритм поиска:

1. Выбрать опорный элемент.
2. Провести разбиение как в быстрой сортировке (любым методом, который нравится).
3. Искать ответ только с той стороны, где будет наша статистика.

Лемма 16.1: Время работы поиска k-ой статистики

Алгоритм в среднем работает при выборе случайного pivot за $O(n)$ (б/д).

Реализация 16.1: Поиск k-той порядковой статистики

```
int find_kth(std::vector<int> vec, size_t k) {
    return find_kth(vec, 0, vec.size(), k);
}
int find_kth(std::vector<int>& vec, size_t left_index,
             size_t right_index, size_t k) {
    if (left_index + 1 == right_index) {
        return vec[left_index];
    }
    int pivot = vec[left_index + (right_index - left_index) / 2];
    size_t index = left_index;
    size_t jndex = right_index - 1;
    while (index < jndex) {
        while (index < jndex && vec[index] < pivot) {
            ++index;
        }
        while (jndex > index && vec[jndex] >= pivot) {
            --jndex;
        }
        if (index == jndex) {
            break;
        }
        std::swap(vec[index], vec[jndex]);
    }
    if (index < k) {
        return find_kth(vec, index, right_index, k);
    }
    return find_kth(vec, left_index, index + 1, k);
}
```

17 Медиана медиан. Время поиска медианы медиан. Работа алгоритмов быстрой сортировки и порядковой статистики с использованием медианы медиан

17.1 Медиана медиан как pivot. Время работы

Алгоритм (Блума-Флойда-Прата-Ривеста-Тарьяна):

1. Разобьем массив на пятерки элементов.
2. В каждой пятерке выберем медиану. Получили массив медиан.
3. Для массива медиан ищем медиану (с помощью алгоритма поиска k-порядковой статистики), и используем ее как опорный элемент.

Обратите внимание, что в алгоритме используется взаимная рекурсия — поиск медианы в массиве медиан, тоже будет использовать медиану медиан.

Лемма 17.1: Гарантия разбиения

Медиана медиан гарантированно делит массив в соотношении не хуже чем $3:7$.

Доказательство: Сначала определим нижнюю границу для количества элементов, превышающих по величине опорный элемент x. В общем случае как минимум половина медиан, найденных на втором шаге, больше или равны медианы медиан x. Таким образом, как минимум $\frac{N}{10}$ групп содержат по 3 элемента, превышающих величину x, за исключением группы, в которой меньше 5 элементов и ещё одной группы, содержащей сам элемент x. Таким образом получаем, что количество элементов больших x не менее $\frac{3n}{10}$.

Лемма 17.2: Линейное время работы

Алгоритм нахождения k-й порядковой статистики с использованием медианы медиан работает за линейное время.

Доказательство: У нас есть три составляющих работы алгоритма на каждом шаге:

1. Время на разделение массива на пятерки и их сортировка: aN
2. Время на поиск медианы медиан $T(\frac{N}{5})$
3. Время на поиск k-й порядковой не превзойдет времени его поиска в большей доле, то есть $T(\frac{7N}{10})$

Таким образом, $T(N) \leq T(\frac{N}{5}) + T(\frac{7N}{10}) + aN$.
Пусть $T(N) \leq cN$ для некоторой c и для любого n до 140. Подставим это соотношение:

$$T(N) \leq T(\frac{N}{5}) + T(\frac{7N}{10}) + cN \leq \frac{cN}{5} + \frac{7 \cdot cN}{10} + aN = cN \cdot \frac{9}{10} + aN = cN + \left(-cN \cdot \frac{1}{10} + aN\right)$$

Тогда неравенство $T(N) \leq cN$ верно, если взять $c \geq 10a$. Поскольку такое c действительно можно выбрать, медиана медиан работает за линейное время.
Применение: Если использовать медиану медиан в качестве опорного элемента в быстрой сортировке, время работы сортировки будет $O(n \log N)$ в худшем случае.

Реализация 17.1: Поиск медианы за линию

```
int MedianOfMedians(std::vector<int> vec) {
    return MedianOfMedians(vec, vec.size() / 2);
}
int MedianOfMedians(std::vector<int>& vec, size_t k) {
    if (vec.size() <= 5) {
        std::sort(vec.begin(), vec.end());
        return vec[k];
    }
    vector<int> medians;
```

```
    for (size_t index = 0; index < vec.size(); index += 5) {
        size_t last_index = std::min(vec.size(), index + 5);
        sort(vec.begin() + index, vec.begin() + last_index);
        medians.push_back(vec[(last_index + index) / 2]);
    }
    int median = MedianOfMedians(medians, medians.size() / 2);
    vector<int> less_median, greater_median;
    for (size_t index = 0; index < vec.size(); ++index) {
        if (vec[index] <= median) {
            less_median.push_back(vec[index]);
        } else {
            greater_median.push_back(vec[index]);
        }
    }
    if (less_median.size() < k) {
        return MedianOfMedians(greater_median, k - less_median.size());
    }
    return MedianOfMedians(less_median, k);
}
```

18 Сортировка подсчётом. Стабильный вариант. Алгоритм LSD

18.1 Сортировка подсчётом (Counting Sort):

Пусть есть массив положительных чисел, элементы которого ограничены неким числом K. Почему бы просто не сосчитать какое значение вошло в него сколько раз? Затем выписать элементы по возрастанию нужное число раз!

Подводным камнем такой сортировки является отсутствие стабильности, но отчаиваться рано, ведь можно сделать эту сортировку стабильной.

18.2 Стабильный вариант

Заведём массив, в который будем записывать отсортированный массив (изначально заполнен нулями). Теперь на массиве, в котором хранили количество элементов, построем префикстые суммы. Пройдёмся по изначальному массивы с конца в начало и будем ставить элемент на последнее свободное место, выделенное в итоговом массиве. Для лучшего понимания см. реализацию.

Реализация 18.1: Стабильная сортировка подсчётом

```
void StableCountintSort(std::vector<char>& vec) { \/\/ english language signs
  std::vector<int> count(26, 0);
  std::vector<char> answer(vec.size());
  for (auto &elem: vec) {
    ++count[elem - 'a'];
  }
  for (int index = 1; index < 26; ++index) {
    count[index] += count[index - 1];
  }
  for (int index = vec.size() - 1; index > -1; --index) {
    answer[--count[vec[index] - 'a']] = vec[index];
  }
  vec = answer;
}
```

18.3 LSD (Least Significant Digit Sort):

Поймем, что целое число это набор байт. Что будет, если сортировать от последнего разряда к первому? Тут важно заметить, что нам КРАЙНЕ важна стабильность сортировки — без нее перемешаются старые результаты для менее значащих разрядов.

Время сортировки составляет $O(n + K)$, где K — макс. значение цифры в основании системы счисления, по которой число сортируется. Потребляемая доп. память составляет $O(n+K)$, поскольку нужен доп массив для совершения перестановки, и для хранения счетчиков.

Реализация 18.2: LSD

```
void LSPSort(std::vector<int>& vec) {
    auto vec_copy = vec;
    int max_element = 0;
    for (size_t index = 0; index < vec.size(); ++index) {
        max_element = std::max(max_element, vec[index]);
    }
    int digit_number = 1;
    size_t max_length = to_string(max_element).size();
    for (size_t length = 0; length < max_length; ++length) {
        std::vector<int> count(10);
        for (size_t index = 0; index < vec.size(); ++index) {
            int digit = (vec[index] / digit_number) % 10;
            ++count[digit];
        }
        for (size_t index_count = 1; index_count < count.size();
        ++index_count) {
            count[index_count] += count[index_count - 1];
        }
        for (size_t index = count.size() - 1; index + 1 > 0; ++index) {
            int digit = (vec[index] / digit_number) % 10;
```

```
            vec_copy[count[digit] - 1] = vec[index];
            --count[digit];
        }
        vec = vec_copy;
        digit_number *= 10;
    }
}
```

19 Бинарное дерево поиска. Определение. Операции Find, Insert, Erase. Поиск порядковой статистики, LowerBound

19.1 Бинарное дерево поиска

Определение 19.1: бинарное дерево поиска

BST (Бинарное Дерево Поиска) — двоичное дерево, правое и левое поддеревья которого тоже являются BST, все значения, лежащие в левом поддереве меньше или равны значения, лежащего в вершине, а все значения правого поддерева больше значения, лежащего в вершине.

19.2 Операции Find, Insert, Erase

Операция Find: поиск элемента, если существует, нужно вернуть ссылку на узел
Идея: давайте спускаться по дереву, начиная с корня, и сравнивать значение текущей вершины с тем, что мы ищем.

1. Если она равны: мы нашли что искали, вернём ссылку на текущую вершину
2. Если искомое значение меньше значения в данной вершине, по свойствам BST оно должно лежать в левом поддереве данной вершине, поэтому рекурсивно продолжим поиск в левом поддереве.
3. Если искомое значение больше значения в данной вершине, по аналогии с предыдущим пунктом продолжим поиск в правом поддереве

В итоге мы пройдёмся по единственному верному маршруту и вернём ссылку на нужную вершину, если таковую нашли.
Операция Insert: вставить элемент в подходящее место
Идея: так как это обычное BST, мы не хотим играться с переподвешиванием в операции Insert, поэтому пройдёмся по дереву и вставим вершину на свободное место, не нарушив порядка:

1. Если нашли пустое место, вставим туда новый элемент
2. Если текущее значение больше или равно вставляемого, пойдём искать свободное место в левое поддерево
3. Если текущее значение меньше вставляемого, пойдём искать свободное место в правое поддерево

Операция Erase: удалить элемент из дерева
Идея: по аналогии с предыдущими 2 операциями пройдёмся по дереву, однако теперь, тогда мы встретили подходящую вершину, стоит рассмотреть 3 случая:

1. У вершины нет детей. Тогда просто говорим, что у родителя нет этого ребёнка.
2. У вершины один ребёнок. Тогда говорим, что у родителя новым ребёнком становится единственный ребёной удаляемой вершины.
3. У вершины два ребёнка. Пусть мы находимся в вершине n, а вершина m - самый левый ребёнок правого поддерева n. Тогда:
   (a) Если m не существует (то есть у правого поддерева n нет ребёнка), значит на это место можно подвесить левое поддерево n, а в качестве нового ребёнка у родителя n объявить правое поддерево n.
   (b) Иначе у нас выполняется следующее: все значения левого поддерева n <= m (так как m лежит в правом поддереве), а также все значения правого поддерева n, не считая m, больше, чем m (так как m - самое левое значение правого поддерева). Тогда мы можем значение в n заменить на значение в m и рекурсивно удалить вершину m.

19.3 Поиск порядковой статистики, LowerBound

Поиск порядковой статистики: хотим получить ссылку на k-ый элемент в отсортированном массиве

Идея: будем в вершине дополнительно хранить размер её поддерева (включая саму вершину). Тогда для того, чтобы найти k-ый элемент в дереве, нужно просто рассмотреть 3 случая:

1. Размер левого поддерева = текущий k - 1. Тогда k-ым элементом будет сама вершина и мы вернём ссылку на неё.

2. Размер левого поддерева + 1 < текущий k. Это значит, что искомый элемент лежит в правом поддереве. Тогда новый k = k - 1 - размер левого поддерева, так как остальные элементы нас больше не интересую и лежат раньше искомого. Рекурсивно запустимся от правого поддерева с новым k.

3. Размер левого поддерева >= k. Тогда k останется прежде, так как остальные элементы лежат после k текущего поддерева и не влияют на k. Рекурсивно запустимся от левого поддерева со старым k.

LowerBound: найти первый элемент, не меньший x.

Идея: сначала будем идти постоянно вправо, пока значение в вершине не будет >= x. А затем будем идти максимально влево, пока значение в левой вершине не меньше x. В итоге у нас получится, что все значения в, лежащие левее, будут меньше, чем x, а все, лежащие правее, будут больше или равны. Вернём ссылку на эту вершину.

20 AVl - дерево, определение, балансировка,операции, теорема о высоте(б/д)

Определение 20.1: АВЛ дерево

АВЛ дерево — это сбалансированное двоичное дерево поиска, в котором для каждой вершины высота левого и правого поддерева отличается не более чем на 1. Название «АВЛ» образовано от фамилий авторов структуры: Адельсон-Вельский и Ландис.

20.1 Балансировка

Балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев |h(L)-h(R)| = 2 , изменяет связи предок-потомок в поддереве данной вершины так, чтобы восстановилось свойство дерева |h(L)-h(R)|≤ 1 , иначе ничего не меняет. Для балансировки будем хранить для каждой вершины разницу между высотой её левого и правого поддерева diff[i]=h(L)−h(R) Для балансировки вершины используются один из 4 типов вращений:

- Правое вращение (Right Rotation) — применяется при добавлении элемента в левое поддерево левого ребенка (LL случай).
- Левое вращение (Left Rotation) — применяется при добавлении элемента в правое поддерево правого ребенка (RR случай).
- Левое-правое вращение (Left-Right Rotation) — применяется при добавлении элемента в правое поддерево левого ребенка (LR случай).
- Правое-левое вращение (Right-Left Rotation) — применяется при добавлении элемента в левое поддерево правого ребенка (RL случай).

20.2 Операции

Основные операции в АВЛ дереве:

- Вставка: Как в бинарном дереве, потом по вершинам вверх переуказать высоту, и если надо вызвать балансировки приколы. Время O(log n)
- Удаление : Как в бинарном дереве, потом по вершинам вверх переуказать высоту, и если надо вызвать балансировки приколы. Время O(log n)
- Поиск : как в бинарном дереве поиска. Время O(log n
- Слияние двух AVL-деревьев. (на лекциях не было)
- Разделение AVL-дерева на два дерева. (на лекциях не было)

20.3 Теорема о высоте

Теорема 20.1: Теорема о высоте АВЛ дерева

Высота АВЛ дерева с n узлами составляет O(log n). Если более точно, то $h < 1.4404 * \log_2 n + 0.328$

Идея доказательства: Пусть $N(h)$ — минимальное количество узлов в АВЛ дереве высоты h. Тогда:

$$N(h) = 1 + N(h-1) + N(h-2)$$

Пояснение: чтобы минимизировать количество узлов при заданной высоте, нужно, чтобы одно из поддеревьев имело высоту $h-1$, а другое — $h-2$. Решая это рекуррентное соотношение, получаем, что $N(h)$ растет экспоненциально с h. Следовательно, высота h растет логарифмически с количеством узлов n. А дальше самостоятельно индукцией по h можно показать, что $N(h) \geq F_{h+2} - 1$, где F_k — k-е число Фибоначчи. А числа Фибоначчи растут экспоненциально с золотым сечением $\phi = \frac{1+\sqrt{5}}{2}$.

21 Декартово дерево, операции, время работы

Определение 21.1: Декартово дерево

Декартово дерево — это структура данных, которая сочетает свойства двоичного дерева поиска и кучи. Каждый узел содержит ключ и приоритет, и дерево упорядочено по ключам (как в двоичном дереве поиска) и по приоритетам (как в куче).

21.1 Операции

Основные операции в декартовом дереве:

- Split: Разделяет дерево на два дерева по значению x, где одно содержит все ключи $<$ x, а другое $>=$ x. Время работы: O(log n)
- Merge: Объединяет два дерева, где все ключи левого дерева меньше ключей правого дерева. Время работы: O(log n)
- Вставка: Вставляет новый узел, используя операции Split и Merge. Время работы: O(log n)
- Удаление: Удаляет узел с заданным ключом, используя операции Split и Merge. Время работы: O(log n)
- Поиск: Находит узел с заданным ключом. Время работы: O(log n)

21.2 Время работы

Амортизированное время выполнения основных операций (вставка, удаление, поиск) составляет O(log n), где n — количество узлов в дереве. Это достигается благодаря случайному распределению приоритетов, что обеспечивает сбалансированность дерева в среднем случае.

21.3 Выразимость операций

Используя Split и Merge, можно выразить все основные операции над декартовым деревом:

- Вставка: Разделить дерево на две части по ключу нового узла, затем объединить левую часть, новый узел и правую часть.
- Удаление: Разделить дерево на три части: узлы с ключами меньше удаляемого, сам узел и узлы с ключами больше удаляемого, затем объединить левую и правую части.
- Поиск: Стандартный поиск в двоичном дереве поиска.

очевидно что все эти операции работают за O(log n) в среднем случае.

21.4 Теорема о высоте декартового дерева

Теорема 21.1: Теорема о высоте декартового дерева

В декартовом дереве с n узлами, где приоритеты — независимые случайные величины с равномерным распределением, высота дерева составляет O(log n) с высокой вероятностью.

Идея доказательства: Тут много слов мат ожидание и вероятность, но руко махательная суть в том что приоритеты рандомные и независимые, поэтому дерево получается сбалансированным в среднем случае. Можно показать что вероятность того что высота дерева превысит c*log n, но это не спросят.

22 Splay дерево, операция splay, время работы, остальные операции

Определение 22.1: Splay дерево

Splay дерево — это самобалансирующееся двоичное дерево поиска, которое выполняет операцию splay для перемещения узла к корню дерева с помощью серии вращений. Эта операция помогает поддерживать часто используемые элементы ближе к корню, что улучшает среднее время доступа.

22.1 Операция splay

Операция splay перемещает узел с заданным ключом к корню дерева с помощью серии вращений. Существует три основных случая, которые определяют, какие вращения будут выполнены:

- Zig: Если узел является прямым потомком корня, выполняется одно правое или левое вращение.
- Zig-Zig: Если узел и его родитель являются левыми или правыми потомками своих родителей, выполняется два последовательных правых или левых вращения.
- Zig-Zag: Если узел является левым потомком правого родителя или правым потомком левого родителя, выполняется сначала одно вращение в одну сторону, а затем в другую.

22.2 Время работы

Амортизированное время выполнения операции splay составляет O(log n), где n — количество узлов в дереве. Это означает, что хотя отдельные операции могут занимать больше времени, среднее время на операцию остается логарифмическим при последовательном выполнении множества операций.

22.3 Остальные операции

Основные операции в Splay дереве:

- **Вставка**: Вставляет новый узел, а затем выполняет splay для этого узла или сплитим по ключу и потом подвесим слева и справа.
- **Удаление**: Выполняет splay для узла, который нужно удалить, а затем удаляет его.
- **Поиск**: Выполняет splay для узла с заданным ключом.
- **Сплит**: по значению x делает splay на минимальном элементе >= x, отсоединяя левое поддерево и как бы возвращая его и остаток того дерева
- **Слияние**: сливает два дерева, где все ключи левого меньше правого, делая splay на максимуме левого. Мы получаем у левого дерева вершину только с левым поддеревом, к которой можно прицепить правое дерево, как поддерево.

23 Декартово дерево по неявному ключу. Задача о развороте подотрезка

24 Дерево Фенвика

Определение 24.1: Дерево Фенвика

это массив той же длины, что и изначальный, в котором

$$t_i = \sum_{k=F(i)}^{i} a_k$$

Подходящие операции: в отличие от дерева отрезков дерево фенвика требует обратимости операции для вычисления её на отрезке.

24.1 Одномерный случай

Возможности:

1. Изменение элемента за O(logn)
2. Вычисление значения на префиксе за O(logn)

В дереве Фенвика используются 2 основные функции: убрать из числа все последние подряд идущие единицы, поменять первый 0 на 1. Делать мы это хотим за $O(1)$. Существует множество разлиных функций, подходящих для этой задачи, но мы рассмотрим 2 самые популярные:

В качестве функции, убирающей последние единицы, будем использовать $i\&(i+1)$, где & - это побитовое И. Если к числу прибавить 1, все последние подряд идущие единицы станут равны 0, а первый 0 превратится в единицу. Поэтому после применения побитового И, мы последние единицы занулим, а 0, который стал единицей обратно станет 0. В итоге получилось число без последних единиц.

В качестве функции, обращающей последний 0 в 1, будем использовать $i|(i+1)$, где | - это побитовое ИЛИ. Если в предыдущем пункте использовать вместо побитого И побитовое ИЛИ, получим, что последние единицы как были единицами, так и останутся, а 0, превратившийся в 1, станет единицей.

Теперь, используя эти 2 функции можно за $O(logn)$ быстро менять значение элемента и считать функцию на префиксе.

Почему работает за $O(logn)$? В числе $logn$ битов, поэтому, проходясь этими функциями, мы каждый раз работаем с новым битом. Значит мы рассмотрим не больше $O(logn)$ элементов.

Рассмотрим дерево Фенвика на примере операций суммы на отрезке и изменения в точке

Реализация 24.1: Дерево Фенвика

```
struct Fenwik {
  std::vector<int> data_;
  int size;

  void add(int index, int delta) {
    for (int ind = index; ind < size; ind = (ind | (ind + 1))) {
      data_[ind] += delta;
    }
  }

  int sum(int left, int right) {
    return sum(right) - sum(left - 1);
  }

  int sum(int index) {
    if (index < 0) {
      return 0;
    }
    int summ = 0;
    for (int ind = index; ind >= 0; ind = (ind & (ind + 1)) - 1) {
      summ += data_[ind];
    }
    return summ;
  }

  void build(const std::vector<int>& vec) {
```

```
      size = vec.size();
      data_.resize(vec.size());
      for (int index = 0; index < size; ++index) {
        add(index, vec[index]);
      }
    }
  };
```

24.2 Обобщение на n-мерный случай

В отличие от ДО Дерево Фенвика очень легко обобщается на n-мерный случай. Достаточно просто сделать n вложенных циклов.

25 хеш-функция. Коллизия. хеш-таблица на цепочках и основные операции. Гипотеза простого равномерного хеширования. Математическое ожидание длины цепочки в предположении гипотезы простого равномерного хеширования

Определение 25.1: хеш-функци

Это функция, преобразующая некий объект в число.

Определение 25.2: Коллизия

Коллизией называется существование двух разных захешированных объекта, хеши которых совпали. Т.е. $\exists\ x \neq y : f(x) = f(y)$

25.1 хеш-таблица на цепочках и основные операции

Проблема: в хэщ-таблице могут происходить коллизии и в таких случаях не ясно что нужно делать.

Решение: самое простое и интуитивно понятное решение - это вместо одного элемента хранить список (цепочку) элементов с одинаковым хешом. Обычно для таких целей используют двусвязный список, ведь зачастую длина цепочки не превышает 8-10 элементов, однако при большой длине цепочки возникают проблемы, которые мы рассмотрим далее

Основные операции:

1. **Вставка:** если нам не требуется уникальность элементов, то мы можем просто добавить новый элемент в двусвязный список за O(1), если же нам требуется уникальность, придётся за O(длины цепочки) пройтись по всему списку и проверить наличие. В итоге операция может работать за O(длины цепочки) в худшем случае

2. **Удаление:** как в обычном двусвязном списке: проходимся, встречаем, соединяем предыдущего и следующего. Работает в худшем случае за O(длины цепочки)

3. **Поиск:** так как у нас двусвязный список, нам ничего не остаётся, кроме как пройтись по всему списку и проверить наличие элемента. Работает за O(длины цепочки) в худшем случае

Мы видим, что при длиных цепочка все операции работают очень долго. Довольно очевидным решением будет использовать какое-нибудь сбалансированное дерево поиска (например, красно-чёрное). Однако при маленькой длине цепочки это не целесообразно, т.к. будет работать дольше и память есть больше, поэтому будем действовать так: если в какой-то момент длина цепочки стала больше 8 элементов, запустим построение сбалансированного дерева поиска и будем дальше работать с ним. После этого все операции будут работать за O(log n).

25.2 Гипотеза простого равномерного хеширования

Это гипотеза для упрощения анализа хеш-таблиц. Звучит так:

Любой элемент может равновероятностно попасть в любую из ячеек хеш-таблицы. Т.е. вероятность равна $P(f(k) = i) = \frac{1}{m}$

Иначе говоря, коллизия между двумя элементами не превышает вероятности совпадения 2 чисел от 1 до m, что составляет $\frac{1}{m}$

Важное уточнение: не существует идеальной хуш-функции, удовлетворяющей данной гипотизе.

25.3 Мат. ожидание длины цепочки при использовании гипотезы простого равномерного хеширования

Пусть X_i - индикатор, равный 1, если i-ый элемент попал в эту цепочку, и 0, если не попал. Тогда количество элементов, попавших в эту цепочку равно

$$X = \sum_{i=1}^{n} X_i$$

Мат.ожидание, что элемент i попал в цепочку равно $1 \cdot \frac{1}{m} + 0 \cdot (1 - \frac{1}{m}) = \frac{1}{m}$
Тогда мат. ожидани длины цепочки равно

$$E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i] = \frac{n}{m}$$

26 Динамическое программирование

26.1 Что это такое и с чем это едят

Определение 26.1: Динамическое программирование

Динамическое программирование — это когда у нас есть задача, которую непонятно как решать, и мы разбиваем ее на меньшие задачи, которые тоже непонятно как решать. (c) А.Кумок
А если серьёзно, то это подход к задачам, в котором мы говорим: мы не знаем как решать текущую задачу, попробуем сначала решить более лёгкие задачи, и на основе полученных ответов попробуем получить ответ на текущую задачу.

Рассмотрим несколько базовых примеров применения ДП:

26.2 Задача о наибольшей возрастающей подпоследовательности

26.2.1 Решение за $O(n^2)$

Идея: пройдёмся по массиву и будем пытаться улучшить уже посчитанный ответ: возьмём максимальный из тех ответов, что посчитаны ранее для элементов, меньших текущего, и запишем в как ответ для данного, добавив единицу (учтя себя). Для получения ответа пройдёмся по массиву ответов (после того, как уже все посчитали) и возьмём максимальный.

Реализация 26.1: НВП за $O(n^2)$

```
int Solve(const std::vector<int>& vec) {
  std::vector<int> dp(vec.size(), 0);
  for (size_t index = 0; index < vec.size(); ++index) {
    int max_ans = 0;
    for (int index1 = 0; index1 < index; ++index1) {
      if (vec[index1] < vec[index]) {
        max_ans = std::max(max_ans, dp[index1]);
      }
    }
    dp[index] = max_ans;
  }
  int answer = 0;
  for (auto &elem: dp) {
    answer = std::max(answer, elem);
  }
  return answer;
}
```

26.2.2 Решение за $O(nlogn)$

Здесь рассмотрим именно решение с помощью ДП. Если интересно решение без использование ДП за $O(nlogn)$, загляни в секцию дополнительных материалов (если там нет, значит я забыл добавить, напиши в личку - добавлю)

Идея: теперь будем хранить в ДП не ответы для элемента с индексом i, а элемент, на который заканчивается последовательность длины i (если таких несколько, будем хранить минимальный). Теперь мы можем бинарным поиском найти первый элемент, больший или равный данного и обновить ответ для него.

Важные наблюдения: все dp[i-1] $\leq$ dp[1], каждый элемент a[i] обновляет не более 1 элемента d[j]

Реализация 26.2: НВП за $O(nlogn)$

```
int Solve(const std::vector<int>& vec) {
  std::vector<int> dp(vec.size() + 1, INT_MAX);
  dp[0] = INT_MIN;
  int answer = 0;
```

```
    for (int i = 0; index < vec.size(); ++index) {
      auto index1 = std::lower_bound(dp.begin(), dp.end(), vec[index]) - dp.begin
();
      if (dp[index1] >= vec[index] && dp[index1 - 1] < vec[index]) {
        dp[index1] = vec[index];
        answer = std::max(answer, index1);
      }
    }
    return answer;
}
```

26.3 Наибольшая общая подпоследовательность

Идея: в dp[i][j] будем хранить ответ для последовательностей a[0...i], b[0...j]. Есть 2 варианта:

1. a[i] = b[j]. Тогда ответом будет ответ для последовательности a[0...i-1], b[0...j-1] + 1
2. a[i] != b[j]. Тогда ответом будет максимум из подпоследовательностей a[0...i], b[0...j-1] и a[0...i-1], b[0...j]

Ответ будет лежать в ячейке dp[n-1][m-1]

Реализация 26.3: НОП за $O(n^2)$

```
int Solve(const std::vector<int>& lhs, const std::vector<int>& rhs) {
  std::vector<std::vector<int>> dp(lhs.size(), std::vector<int> (rhs.size(), 0)
);
  for (size_t index = 0; index < lhs.size(); ++index) {
    for (size_t index1 = 0; index1 < rhs.size(); ++index1) {
      if (lhs[index] == rhs[index1]) {
        dp[index][index1] = dp[index - 1][index1 - 1] + 1;
      } else {
        dp[index][index1] = std::max(dp[index - 1][index1], dp[index][index1 -
1]);
      }
    }
  }
  return dp[lhs.size() - 1][rhs.size() - 1];
}
```

26.4 Задача о рюкзаке за O(nW)

Задача 26.1: Рюкзак

Даны предметы весом $w_1, w_2, ..., w_n$, каждый из которых стоит $c_1, c_2, ..., c_n$, а также есть рюкзак вместимостью W. Нужно в него засунуть вещи на максимальную сумму.

Идея: пусть dp[i][j] - ответ для первых i предметов с рюкзаком вместимостью j. Тогда dp[i][j] - это максимум из dp[i-1][j] и dp[i-1][j - w[i]] + cost[i]. Тогда ответ - max(dp[n][0], dp[n][1], ..., dp[n][W]).

Реализация 26.4: Рюкзак за $O(nW)$

```
int Solve(const std::vector<int>& weight, const std::vector<int>& cost, int
capacity) {
  std::vector<std::vector<int>> dp(cost.size() + 1, std::vector<int> (capacity
+ 1, 0));
  int answer = 0;
  for (size_t index = 1; index <= cost.size(); ++index) {
    for (size_t index1 = 1; index1 <= capacity; ++index1) {
      dp[index][index1] = dp[index - 1][index1];
      if (index1 >= weight[index - 1]) {
        dp[index][index1] = std::max(dp[index][index1], dp[index - 1][index1 -
weight[index - 1]] + cost[index - 1]);
      }
```

```
            answer = std::max(answer, dp[index][index1]);
        }
    }
    return answer;
}
```

27 Дополнительные знания, не требуемые для сдачи экзамена

27.1 Мастер-теорема о рекурсии

Теорема 27.1: Мастер-теорема о рекурсии

Имеется рекуррентное соотношение:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + n^c, \quad \text{где } a \geq 1, b > 1$$

Тогда:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & \text{если } a > b^c \\ \Theta(n^c \log n), & \text{если } a = b^c \\ \Theta(n^c), & \text{если } a < b^c \end{cases}$$

Доказательство (схема): Раскроем рекурренту:

$$\begin{aligned} T(n) &= n^c + a \cdot T(n/b) \\ &= n^c + a \cdot (n/b)^c + a^2 \cdot (n/b^2)^c + \dots \\ &= n^c \left(1 + \frac{a}{b^c} + \left(\frac{a}{b^c}\right)^2 + \dots\right) \\ &= n^c(1 + q + q^2 + \dots), \quad \text{где } q = \frac{a}{b^c} \end{aligned}$$

Количество членов: $k = \Theta(\log n)$.

- Если $q = 1$ (т.е. $a = b^c$): $S(q) = k + 1 = \Theta(\log n) \Rightarrow T(n) = \Theta(n^c \log n)$
- Если $q < 1$ (т.е. $a < b^c$): бесконечно убывающая геометрическая прогрессия, $T(n) = \Theta(n^c)$
- Если $q > 1$ (т.е. $a > b^c$): $S(q) = \Theta(q^k) = \Theta\left(\left(\frac{a}{b^c}\right)^{\log n}\right) = \Theta(n^{\log_b a})$

27.2 Доказательство основной теоремы алгоритма Евклида

Доказательство:

Пусть $(a, b) = k$, тогда и a и bq делятся на k, следовательно, $r = a - bq$ тоже делится на k.

С другой стороны, пусть $(b, r) = k_1$, тогда $a = bq + r$ делится на k_1.

Предположим, что $k_1 > k$ (или $k_1 < k$), но в силу того что k (и k_1) — НОД, получим противоречие, тогда $k = k_1$.