Коллоквиум

«Алгоритмы и структуры данных»

Если найдете опечатки или неправильность пишите

Автор

18 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 Веществennозначный 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 Дополнительные знания, не требуемые для сдачи экзамена

14.1 Мастер-теорема о рекурсии

Теорема 14.1: 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. $a > b^c$): $S(q) = \Theta(q^k) = \Theta\left(\left(\frac{a}{b^c}\right)^{\log n}\right) = \Theta(n^{\log_b a})$

14.2 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 $k = k_1$.