1. Оценка сложности алгоритма по времени (Time Complexity)

В алгоритмах нас волнует не *секунды* (они зависят от железа), а *количество операций* в зависимости от размера входных данных N.

Асимптотические нотации

Мы используем «О-большое» (O), чтобы оценить **худший случай** (верхняя граница).

Определение: $f(n) = O(g(n))$ iff $\exists C > 0, n_0 > 0 : \forall n \geq n_0, |f(n)| \leq C \cdot |g(n)|$

Простыми словами: начиная с какого-то n, наш алгоритм работает не медленнее, чем $g(n)$, умноженная на константу.

Важные правила:
1. **Константы отбрасываются:** $O(2n) \Rightarrow O(n)$, $O(500) \Rightarrow O(1)$.
2. **Младшие члены отбрасываются:** $O(n^2 + n + 100) \Rightarrow O(n^2)$.

Основные классы сложности (от лучшего к худшему)

Нотация	Название	Пример
$O(1)$	Константная	Взять элемент массива по индексу, `map[key]` (в среднем).
$O(\log n)$	Логарифмическая	Бинарный поиск (`std::lower_bound`), операции в `std::set`.
$O(n)$	Линейная	Проход по массиву циклом `for`, поиск минимума.
$O(n \log n)$	Линеарифическая	Быстрые сортировки (`MergeSort`, `HeapSort`, `std::sort`).
$O(n^2)$	Квадратичная	Вложенные циклы, `BubbleSort`, `InsertionSort`.
$O(2^n)$	Экспоненциальная	Перебор всех подмножеств (рюкзак брутфорсом).
$O(n!)$	Факториальная	Перебор перестановок (задача коммивояжера).

2. Оценка сложности алгоритма по памяти (Space Complexity)

Оцениваем, сколько **дополнительной** памяти (Auxiliary Space) требует алгоритм относительно входных данных.

Важные моменты:

1. **Переменные:** $O(1)$ (если их фиксированное число).
2. **Массивы/Векторы:** $O(n)$, если создаем копию данных.
3. **Рекурсия (Стек вызовов):** Это часто забывают! Глубина рекурсии жрет память.
   - Например, `quicksort` в худшем случае займет $O(n)$ доп. памяти
   - В сбалансированном дереве — $O(\log n)$.

Примеры оценки памяти

1. **Сортировка пузырьком:** Мы меняем элементы местами внутри массива (`swap`). Доп. память не нужна. $S(n) = O(1)$ (In-place).

2. **Сортировка слиянием (Merge Sort):** Нам нужен дополнительный буфер для слияния частей. $S(n) = O(n)$.

3. **Рекурсивный Фибоначчи (наивный):**

```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n-1) + fib(n-2); // максимальная глубина стека - n; S(n) = O(n)
}
```

3. Сортировка вставками (Insertion Sort)

Самый интуитивный алгоритм. Именно так люди сортируют карты в руке: берем новую карту и пихаем ее в нужное место среди уже отсортированных.

Принцип работы

Мы делим массив на две части:

1. **Отсортированная часть** (слева). Изначально там только один элемент $A[0]$.
2. **Неотсортированная часть** (справа).

На каждом шаге мы берем первый элемент из *неотсортированной* части и «вставляем» его на правильное место в *отсортированную*, сдвигая элементы, которые больше него, вправо.

Инвариант цикла

В начале итерации i подмассив $A[0..i-1]$ состоит из элементов, которые были в этом диапазоне изначально, но теперь они **отсортированы**.

Реализация (C++)

```
void insertionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; i++) {
        int key = arr[i]; // Элемент, который вставляем
        int j = i - 1;

        // Сдвигаем элементы arr[0..i-1], которые больше key,
        // на одну позицию вправо
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key; // Вставляем на освободившееся место
    }
}
```

Анализ сложности

Время (Time Complexity)

1. **Худший случай (Worst Case):** Массив отсортирован в обратном порядке. Для каждого элемента i нам придется делать i сравнений и сдвигов.

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \approx O(n^2)$$

2. **Лучший случай (Best Case):** Массив уже отсортирован. Внутренний цикл `while` ни разу не выполнится (условие `arr[j] > key` сразу ложно). Мы просто пройдемся по массиву.

$$O(n)$$

3. **Средний случай (Average Case):** В среднем приходится сдвигать половину элементов слева.

$$O(n^2)$$

Память (Space Complexity)

$O(1)$. Сортировка происходит «на месте» (in-place). Нам нужна только одна доп. переменная `key`.

Свойства алгоритма

1. **Устойчивость (Stable):** Да. Мы не меняем порядок равных элементов (условие `arr[j] > key` строгое, если элементы равны, сдвига не будет, и новый встанет *после* равного).
2. **Адаптивность (Adaptive):** Да. Если массив «почти отсортирован», алгоритм работает очень быстро (близко к $O(n)$).
3. **Online-алгоритм:** Да. Мы можем сортировать данные по мере их поступления (по одному).

Зачем это нужно?

Казалось бы, $O(n^2)$. Но Insertion Sort — хорош на **маленьких массивах**.

Почему:

1. Очень простой код (мало оверхеда).
2. Хорошо работает с кэшем процессора (последовательный доступ к памяти).

Реальный кейс: В `std::sort` (C++) или `pdqsort` (Go/Rust) используется гибридный подход. Когда рекурсия QuickSort делит массив на куски размером меньше 16-32 элементов, она переключается на Insertion Sort, потому что на таких объемах он быстрее асимптотически крутых алгоритмов.

4. Сортировка слиянием (Merge Sort)

Это алгоритм класса «Разделяй и Властвуй». Он гарантирует отличную скорость даже на самых плохих данных, но требует «жертв» по памяти.

Визуализация процесса

Мы дробим массив пополам, пока не получим кучу одиночных элементов (а один элемент всегда отсортирован). Потом мы начинаем склеивать (сливать) их обратно, но уже в правильном порядке.

[**38, 27, 43, 3**]

↓

38, 27   43, 3

↓

38   27   43   3

Merge

27, 38   3, 43

↓

[**3, 27, 38, 43**]

Алгоритм

1. **Divide (Разделение):** Находим середину массива и рекурсивно вызываем сортировку для левой и правой половин.
2. **Conquer (Властвование):** Если размер подмассива равен 1 — он уже отсортирован. Возвращаем его.
3. **Combine (Слияние):** Берем два отсортированных массива и сливаем их в один общий с помощью **двух указателей**.

Реализация (C++)

```
// Функция слияния двух отсортированных частей
void merge(vector<int>& arr, int left, int mid, int right) {
    // Временные векторы (жрут память!)
    vector<int> L(arr.begin() + left, arr.begin() + mid + 1);
    vector<int> R(arr.begin() + mid + 1, arr.begin() + right + 1);

    int i = 0, j = 0, k = left;

    // Сравниваем головы двух массивов и берем меньший
    while (i < L.size() && j < R.size()) {
        if (L[i] <= R[j]) { // <= важно для устойчивости (Stable)
            arr[k++] = L[i++];
        } else {
            arr[k++] = R[j++];
        }
    }

    // Докидываем остатки, если они есть
    while (i < L.size()) arr[k++] = L[i++];
    while (j < R.size()) arr[k++] = R[j++];
}

void mergeSort(vector<int>& arr, int left, int right) {
    if (left >= right) return; // Базовый случай

    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);      // Сортируем левую часть
    mergeSort(arr, mid + 1, right); // Сортируем правую часть
    merge(arr, left, mid, right);   // Сливаем
}
```

5. Быстрая сортировка (Quick Sort)

Один из самых эффективных алгоритмов сортировки общего назначения. Разработан Тони Хоаром в 1960 году. Относится к классу алгоритмов «Разделяй и властвуй» (Divide and Conquer).

В отличие от сортировки слиянием, где основной работой является объединение отсортированных частей, в быстрой сортировке основная вычислительная сложность приходится на этап **разделения** (partitioning).

Принцип работы

Алгоритм состоит из трех этапов:

1. **Выбор опорного элемента (Pivot):** Из массива выбирается один элемент. От его выбора зависит эффективность работы алгоритма.
2. **Разбиение (Partition):** Перераспределение элементов в массиве таким образом, что:
   - Все элементы *меньше* опорного помещаются слева от него.
   - Все элементы *больше* опорного помещаются справа.
   - Опорный элемент занимает свое окончательное (отсортированное) положение. (в Хоара необязательно)
3. **Рекурсия:** Рекурсивный запуск алгоритма для левого и правого подмассивов (исключая опорный элемент).

Визуализация этапа Partition

Рассмотрим массив, где в качестве опорного элемента (Pivot) выбран первый элемент (**4**).

1. Исходный массив (Pivot = 4):

4	7	1	3	2	6	5

↓

2. Процесс разделения:

Перемещаем элементы < 4 влево, > 4 вправо.

3	1	2	**4**	7	6	5

Теперь 4 находится на своем месте. Рекурсивно сортируем синюю и розовую части.

Реализация (C++)

В данной реализации используется схема разбиения Ломуто (Lomuto partition scheme), где опорным элементом выбирается последний элемент.

```
// Функция разделения массива
int partition(vector<int>& arr, int low, int high) {
    int pivot = arr[high]; // Опорный элемент
    int i = (low - 1);     // Индекс меньшего элемента

    for (int j = low; j <= high - 1; j++) {
        // Если текущий элемент меньше или равен опорному
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    // Ставим опорный элемент на правильную позицию
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}

void quickSort(vector<int>& arr, int low, int high) {
    if (low < high) {
        // pi - индекс разделения (partitioning index)
        int pi = partition(arr, low, high);

        // Рекурсивно сортируем элементы до и после разделения
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Реализация схемы Хоара (C++)

Данный вариант эффективнее схемы Ломуто по количеству операций записи в память (swap), так как обмен происходит только тогда, когда это действительно необходимо (инверсия относительно pivot).

```cpp
int partitionHoare(std::vector<int>& arr, int low, int high) {
    // В качестве опорного выбираем первый элемент (для простоты)
    // На практике лучше брать arr[(low + high) / 2]
    int pivot = arr[low];
    int i = low - 1;
    int j = high + 1;

    while (true) {
        // Сдвигаем левый указатель вправо, пока элементы меньше pivot
        do {
            i++;
        } while (arr[i] < pivot);

        // Сдвигаем правый указатель влево, пока элементы больше pivot
        do {
            j--;
        } while (arr[j] > pivot);

        // Если указатели пересеклись, разбиение завершено
        if (i >= j)
            return j;

        // Иначе меняем элементы местами
        std::swap(arr[i], arr[j]);
    }
}

void quickSortHoare(std::vector<int>& arr, int low, int high) {
    if (low < high) {
        // Получаем индекс разделения
        int p = partitionHoare(arr, low, high);

        // Важно: в схеме Хоара элемент p включается в ЛЕВУЮ часть
        quickSortHoare(arr, low, p);
        quickSortHoare(arr, p + 1, high);
    }
}
```

Анализ сложности

Время (Time Complexity)

Скорость работы критически зависит от выбора опорного элемента и структуры входных данных.

1. **Средний и лучший случай:** $O(N \log N)$. Это достигается, когда опорный элемент делит массив на две примерно равные части. Глубина рекурсии составляет $\log_2 N$.
2. **Худший случай:** $O(N^2)$. Происходит, если на каждом этапе массив делится на части размером 1 и $N - 1$. *Пример:* Массив уже отсортирован (в прямом или обратном порядке), а в качестве Pivot всегда берется первый или последний элемент.

Память (Space Complexity)

Алгоритм работает **in-place** (без выделения дополнительного массива), но расходует стековую память на рекурсивные вызовы.

Средний случай: $\boldsymbol{O(\log N)}$**. Худший случай:** $O(N)$ (при деградации до $O(N^2)$).

Стратегии выбора Pivot

Чтобы избежать худшего случая ($O(N^2)$), применяются следующие эвристики:
1. **Случайный выбор (Randomized QuickSort):** Pivot выбирается рандомно. Это делает вероятность худшего случая крайне низкой.
2. **Медиана трех (Median-of-three):** В качестве Pivot берется медиана между первым, средним и последним элементами.

Свойства алгоритма

1. **Неустойчивость (Unstable):** Да. В процессе разделения относительный порядок равных элементов может быть нарушен.
2. **In-place:** Да. Не требует $O(N)$ дополнительной памяти, как Merge Sort.
3. **Кэш-локальность:** Высокая. Quick Sort работает быстрее Merge Sort и Heap Sort на практике, так как последовательно обращается к памяти, эффективно используя кэш процессора.

Примечание: Схема Хоара vs Схема Ломуто

В коде выше использована **схема Ломуто** (однонаправленный проход). Она проще в реализации и гарантирует, что после этапа `partition` опорный элемент (Pivot) встает на свое **финальное отсортированное место**.

Однако оригинальная **схема Хоара** (два указателя, движущихся навстречу друг другу) работает иначе:
1. Она эффективнее (делает в среднем в 3 раза меньше обменов).
2. **Важно:** После разбиения Хоара опорный элемент **не обязательно** оказывается на своем финальном месте. Функция разбиения возвращает индекс, разделяющий массив на две части (элементы $\leq$ pivot и элементы $\geq$ pivot), но сам pivot может находиться где угодно внутри соответствующей части.

6. Сортировка подсчетом (Counting Sort)

Алгоритм сортировки, применимый, когда входные данные являются целыми числами в ограниченном диапазоне. Основная идея — подсчитать, сколько раз встречается каждый элемент, и на основе этого определить его позицию в отсортированном массиве.

Визуализация процесса

Пусть дан массив чисел в диапазоне от 0 до 5.

1. Входной массив (Input):

1	4	1	2	7	5	2

2. Массив подсчета (Count Array):

Индексы соответствуют значениям элементов. Значения — количеству их вхождений.

0	2	2	0	1	1	0	1
0	1	2	3	4	5	6	7

3. Восстановление (Output):

Раскладываем числа согласно их количеству.

1	1	2	2	4	5	7

Алгоритм (Стабильная версия)

Для сохранения свойства устойчивости (stability) алгоритм усложняется:

1. **Подсчет частот:** Создаем массив `count`, где `count[i]` — число вхождений элемента `i`.
2. **Префиксные суммы:** Модифицируем `count`, чтобы `count[i]` содержал количество элементов, *меньших либо равных* `i`. Это фактически позиция последнего вхождения числа `i` в выходном массиве.
3. **Расстановка:** Проходим по исходному массиву **с конца**. Для каждого числа `x` помещаем его в выходной массив на позицию `count[x] - 1` и уменьшаем `count[x]`.

Реализация (C++)

```cpp
void countingSort(std::vector<int>& arr) {
    if (arr.empty()) return;

    // 1. Поиск диапазона
    int max_val = *std::max_element(arr.begin(), arr.end());
    int min_val = *std::min_element(arr.begin(), arr.end());
    int range = max_val - min_val + 1;

    // 2. Массив частот
    std::vector<int> count(range, 0);
    std::vector<int> output(arr.size());

    // Подсчет вхождений
    for (int num : arr) {
        count[num - min_val]++;
    }

    // 3. Префиксные суммы (накопление)
    // count[i] теперь хранит позицию элемента
    for (int i = 1; i < range; i++) {
        count[i] += count[i - 1];
    }

    // 4. Построение выходного массива (идем с конца для устойчивости)
    for (int i = arr.size() - 1; i >= 0; i--) {
        int num = arr[i];
        int index_in_count = num - min_val;
        output[count[index_in_count] - 1] = num;
        count[index_in_count]--;
    }

    // Копируем обратно
    arr = output;
}
```

Зачем нужен `num - min_val`?

1. **Поддержка отрицательных чисел:** Индексы массива в C++ начинаются с 0. Если в данных есть число -5, мы не можем обратиться к `count[-5]`. Сдвиг переносит минимальный элемент в индекс 0.
2. **Оптимизация памяти:** Если сортируем числа в диапазоне $[1000, 1005]$, нам нужен массив размером всего 6, а не 1006. Без сдвига мы бы тратили память впустую на «пустоту» от 0 до 999.

Анализ сложности

Пусть N — количество элементов, K — диапазон значений $(\max - \min + 1)$.

Время (Time Complexity)

$$O(N + K)$$

Мы проходим по массиву длины N и по массиву подсчета длины K линейно. Если $K \approx N$, то алгоритм работает за линейное время $O(N)$.

Память (Space Complexity)

$$O(N + K)$$

Требуется память под массив частот (K) и выходной массив (N).

Ограничения и применимость

1. **Только целые числа:** Нельзя сортировать `float` или строки (напрямую), так как они не могут быть индексами массива.
2. **Зависимость от диапазона:** Если K (разброс значений) значительно больше N (например, сортируем массив `[1, 1000000]`), алгоритм становится крайне неэффективным по памяти и времени ($O(N^2)$ (при K большем N, K $\approx N^2$, будет $O(N + N^2)) \Rightarrow O(N^2)$ или хуже в контексте памяти).
3. **Устойчивость:** Реализация через префиксные суммы является **устойчивой** (Stable). Это критически важно, так как Counting Sort часто используется как подпрограмма в **Radix Sort**.

7. Цифровая сортировка (Radix Sort)

Алгоритм сортировки, работающий с элементами как с последовательностями цифр (или символов). Не использует сравнения элементов между собой. Является обобщением сортировки подсчетом.

Основная идея: сортировать числа не целиком, а по разрядам (цифрам), используя устойчивую сортировку (обычно Counting Sort) в качестве подпрограммы.

Параметры:
- N — количество элементов.
- b — основание системы счисления (base/radix). Например, 10, 2, или 256.
- d — количество разрядов (длина числа).

Варианты реализации

Существует два принципиально разных подхода к порядку обработки разрядов:

1. **LSD (Least Significant Digit):** Сортировка от младшего разряда к старшему (справа налево).
2. **MSD (Most Significant Digit):** Сортировка от старшего разряда к младшему (слева направо).

1. LSD Radix Sort (Младший разряд — первый)

Это классическая, итеративная версия алгоритма. Мы последовательно сортируем массив по последней цифре, затем по предпоследней, и так до первой.

> **Важное условие**
> Используемая на каждом этапе под-сортировка должна быть **устойчивой (stable)**. *Почему?* Когда мы сортируем по разряду 10^1 (десятки), мы не должны перемешать порядок, который мы выстроили на этапе 10^0 (единицы), если десятки у чисел равны.

Пример (LSD)

Массив: `[170, 045, 075, 090, 802, 024, 002, 066]`

1. **Сортируем по единицам (последняя цифра):** `[170, 090, 802, 002, 024, 045, 075, 066]` (Обратите внимание: 802 идет перед 002, так как в исходном массиве 802 было раньше. Устойчивость сохранена).

2. **Сортируем по десяткам:** `[802, 002, 024, 045, 066, 170, 075, 090]` (Сейчас массив отсортирован по последним двум цифрам).

3. **Сортируем по сотням:** `[002, 024, 045, 066, 075, 090, 170, 802]` **Массив полностью отсортирован.**

Реализация LSD (C++)

```cpp
// Вспомогательная функция (Counting Sort по конкретному разряду exp)
void countSortByDigit(vector<int>& arr, int exp) {
    int n = arr.size();
    vector<int> output(n);
    vector<int> count(10, 0); // Система счисления 10

    // 1. Считаем вхождения (digit = (arr[i] / exp) % 10)
    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    // 2. Префиксные суммы
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // 3. Формируем output (идем с конца для устойчивости!)
    for (int i = n - 1; i >= 0; i--) {
        int digit = (arr[i] / exp) % 10;
        output[count[digit] - 1] = arr[i];
        count[digit]--;
    }

    arr = output;
}

void radixSortLSD(vector<int>& arr) {
    if (arr.empty()) return;
    int max_val = *max_element(arr.begin(), arr.end());

    // Проходим по разрядам: 1, 10, 100... пока exp <= max_val
    for (int exp = 1; max_val / exp > 0; exp *= 10) {
        countSortByDigit(arr, exp);
    }
}
```

2. MSD Radix Sort (Старший разряд — первый)

Более естественный для человека способ (как поиск слова в словаре). Мы смотрим на первую букву/цифру и раскладываем элементы по «корзинам» (buckets). Внутри каждой корзины рекурсивно запускаем сортировку для следующего разряда.

Особенности:

- Рекурсивный алгоритм.
- Может работать с элементами переменной длины (например, строками).
- Не требует устойчивости под-сортировки (так как мы разбиваем массив на независимые подгруппы).

Алгоритм MSD

1. Разделить массив на группы (корзины) по значению текущего старшего разряда (i).
2. Рекурсивно применить алгоритм к каждой корзине для разряда $i + 1$.
3. Соединить корзины обратно.

Пример (MSD)

Массив: `[170, 045, 075, 090, 802, 024, 002, 066]`

1. **Смотрим на сотни (разряд 100):**
   - Корзина „0“: `[045, 075, 090, 024, 002, 066]` -> **Рекурсия**
   - Корзина „1“: `[170]` -> (Один элемент, готово)
   - …
   - Корзина „8“: `[802]` -> (Один элемент, готово)
2. **Рекурсия внутри корзины „0“ (смотрим на десятки):**
   - Корзина „0“: `[002]`
   - Корзина „2“: `[024]`
   - Корзина „4“: `[045]`
   - … и т.д.

Реализация MSD Radix Sort (C++)

В отличие от LSD, здесь мы используем рекурсию.

```cpp
#include <iostream>

int digit_at(int x, int r) { return (int)(x / pow(10, r - 1)) % 10; }

int get_max(int arr[], int len) {
  if (len <= 0) return -INFINITY;
  int max = arr[0];

  for (int i = 0; i < len; ++i) {
    max = std::max(arr[i], max);
  }

  return max;
}

void msd(int arr[], int left, int right, int r) {

  if (right <= left || r == 0) return;

  int cnt[10] = {0};
  for (int i = left; i <= right; ++i) {
    int c = digit_at(arr[i], r);
    cnt[c]++;
  }

  for (int i = 1; i < 10; ++i) {
    cnt[i] += cnt[i - 1];
  }

  int temp[right - left + 1];
  for (int i = right; i >= left; --i) {
    int c = digit_at(arr[i], r);
    temp[cnt[c] - 1] = arr[i];
    cnt[c]--;
  }

  for (int i = left; i <= right; ++i) {
    arr[i] = temp[i];
  }

  // r = 1, | 321 231 431 101 | 122 | 234 | 346
  // cnt = {0, 4, 1, 0, 1, 0, 1};
  // cnt = {0, 4, 5, 5, 6, 6, 7};

  for (int i = 0; i < 10; i++) {
    int gs = (i == 0) ? 0 : cnt[i - 1];
    int ge = cnt[i] - 1;
    if (ge >= gs) {
```

```
      msd(arr, left + gs, left + ge, r - 1);
    }
  }
}

void msd_sort(int arr[], int len) {

  if (len <= 1) return;
  int max = get_max(arr, len);
  int digits = (max == 0) ? 1 : (int)floor(log10(max)) + 1;
  // 2345 -> 3.12312412 -> 3 -> 4

  msd(arr, 0, len - 1, digits);

}

int main() {

  int arr[] = {2100, 1300, 3050, 4000};

  msd_sort(arr, 4);

  for (int i = 0; i < 4; ++i) std::cout << arr[i] << ' ';

}
```

Сравнение LSD и MSD

LSD	**MSD**
Проще в реализации (цикл).	Сложнее (рекурсия, управление памятью).
Всегда просматривает **все** разряды у **всех** чисел.	Может остановиться раньше, если префикс уникален (быстрее на строках).
Удобен для чисел фиксированной длины (`int32`, `int64`).	Идеален для сортировки строк (lexicographical sort).
Требует устойчивой сортировки на каждом шаге.	Не требует устойчивости (разбиение на группы).

Анализ сложности (Radix Sort в целом)

Пусть d — количество разрядов, b — основание системы счисления (размер массива подсчета), N — количество элементов.

Время (Time Complexity)

$$O(d \times (N + b))$$

На каждом из d этапов мы выполняем Counting Sort, который занимает $O(N + b)$.

Инсайт: Если мы сортируем `int32`, мы можем выбрать основание $b = 256$ (1 байт). Тогда количество проходов $d = 4$ (так как 4 байта в int). В этом случае сложность $O(4(N + 256)) \approx O(N)$. Это линейная сортировка!

Память (Space Complexity)

$$O(N + b)$$

Требуется буфер для выходного массива (N) и массив подсчета (b). В MSD версии добавляется память под стек рекурсии.

Применимость

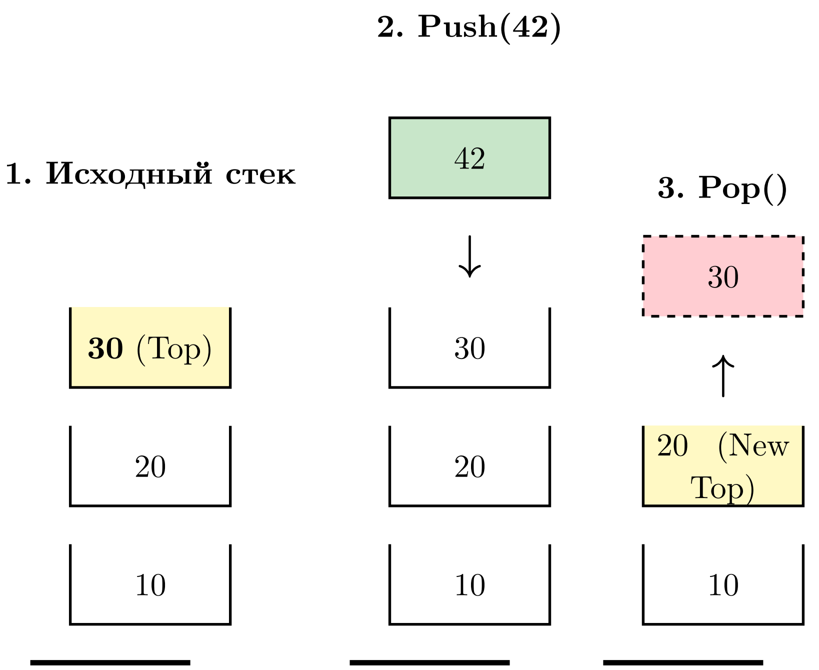
Radix Sort часто быстрее $O(N \log N)$ алгоритмов (QuickSort, MergeSort) на больших массивах чисел или строк, так как не выполняет дорогих операций сравнения. Однако, он проигрывает по кэш-локальности (random access при записи в `output`).

8. Стек (Stack)

Абстрактная структура данных, работающая по принципу **LIFO (Last In, First Out)** — «последним пришел, первым ушел».

Аналогия из жизни: стопка тарелок. Вы можете положить новую тарелку только наверх и взять тарелку только сверху. Чтобы добраться до нижней, нужно убрать все верхние.

Визуализация LIFO



Основные операции

Все операции в стеке выполняются за **константное время** $O(1)$, так как мы работаем только с одним концом структуры (вершиной).

Операция	Описание	Сложность
`push(x)`	Добавляет элемент `x` на вершину стека.	$O(1)$
`pop()`	Удаляет элемент с вершины. **Внимание:** в C++ `std::stack::pop` возвращает `void`, а не удаленный элемент.	$O(1)$
`top()` / `peek()`	Возвращает значение элемента на вершине, не удаляя его.	$O(1)$
`empty()`	Проверяет, пуст ли стек (true/false).	$O(1)$

`size()`	Возвращает количество элементов в стеке.	$O(1)$

Реализация (на базе динамического массива)

В C++ стандартный адаптер `std::stack` по умолчанию использует `std::deque` (или `std::vector`), закрывая доступ к произвольным индексам.

Пример реализации «вручную» на векторе:

```
template <typename T>
class Stack {
private:
    std::vector<T> data;

public:
    // Добавление элемента
    void push(T val) {
        data.push_back(val);
    }

    // Удаление элемента
    void pop() {
        if (!data.empty()) {
            data.pop_back();
        } else {
            throw std::out_of_range("Stack underflow");
        }
    }

    // Просмотр вершины
    T top() {
        if (!data.empty()) {
            return data.back();
        }
        throw std::out_of_range("Stack is empty");
    }

    bool empty() {
        return data.empty();
    }

    size_t size() {
        return data.size();
    }
};
```

Области применения

1. **Управление памятью (Call Stack):** Хранение локальных переменных и адресов возврата при вызове функций. Рекурсия работает именно благодаря системному стеку.
2. **Обратная польская запись (RPN):** Вычисление выражений вида `3 4 +` (где операторы идут после операндов).
3. **Скобочные последовательности:** Проверка правильности расстановки скобок `(([]))`.
4. **Обход графов (DFS):** Поиск в глубину использует стек (явно или через рекурсию).
5. **Отмена операций (Undo):** В текстовых редакторах (Ctrl+Z).

13. Стек на связных списках (Linked List Stack)

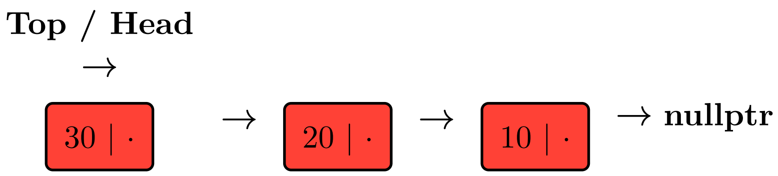
Реализация стека (LIFO), использующая односвязный список для хранения элементов.

Ключевая идея: Вершина стека (`Top`) соответствует **Голове** (`Head`) списка.

- `push` → Вставка в начало списка.
- `pop` → Удаление первого элемента списка.

Почему именно в начало? В односвязном списке вставка и удаление из начала выполняются за $O(1)$. Удаление с конца (хвоста) требовало бы $O(N)$, так как нам нужно найти предпоследний элемент, чтобы обнулить его `next`.

Визуализация



Доступ (Push/Pop) происходит только через Top.

Реализация (C++)

```
template <typename T>
class LinkedStack {
private:
    struct Node {
        T data;
        Node* next;
        Node(T val, Node* n = nullptr) : data(val), next(n) {}
    };

    Node* head; // Указатель на вершину стека
    size_t sz;

public:
    LinkedStack() : head(nullptr), sz(0) {}

    // Очистка памяти
    ~LinkedStack() {
        while (!empty()) pop();
    }

    // Push: O(1)
    void push(T val) {
        // Создаем узел и сразу направляем его на текущую голову
```

```cpp
        head = new Node(val, head);
        sz++;
    }

    // Pop: O(1)
    void pop() {
        if (empty()) throw std::underflow_error("Stack is empty");

        Node* oldHead = head;
        head = head->next; // Сдвигаем голову
        delete oldHead;    // Освобождаем память
        sz--;
    }

    // Top: O(1)
    T top() {
        if (empty()) throw std::underflow_error("Stack is empty");
        return head->data;
    }

    bool empty() { return head == nullptr; }
    size_t size() { return sz; }
};
```

Сравнение: Стек на Векторе vs Стек на Списке

Характеристика	Вектор (Array-based)	Список (Linked-based)
Выделение памяти	Блоками. Редко, но требует копирования всего массива при росте.	Для каждого элемента отдельно (`new`).
Стабильность времени	Амортизированное $O(1)$. Иногда случаются лаги при ресайзе.	Строгое $O(1)$. Операция всегда занимает одно и то же время.
Потребление памяти	Экономично (нет указателей), но может быть зарезервировано лишнее место (`capacity`).	Оверхед на указатели (8-16 байт) для каждого элемента + фрагментация кучи.
Кэш-локальность	Отличная (данные лежат рядом).	Плохая (узлы разбросаны по памяти).

Когда использовать Linked Stack?

1. Когда критически важна **гарантированная** скорость операции (Real-time системы), и мы не можем допустить задержку на реаллокацию вектора.

2. Когда размер стека непредсказуем и сильно варьируется.
3. Когда копирование объектов (при ресайзе вектора) слишком дорогое.

9. Очередь (Queue)

Абстрактная структура данных, работающая по принципу **FIFO (First In, First Out)** — «первым пришел — первым ушел».

Элементы добавляются строго в один конец (хвост/back), а извлекаются строго из другого конца (голова/front).

Визуализация FIFO

Выход (Pop) ← | 10 | 20 | 30 | **40** | ← **Вход** (Push)

Элемент 10 (голова) уйдет следующим. Элемент 40 (хвост) пришел последним.

Основные операции

Как и в стеке, операции должны выполняться за константное время $O(1)$.

Операция	**Описание**	**Сложность**
`push(x)` / `enqueue`	Добавляет элемент `x` в конец очереди.	$O(1)$
`pop()` / `dequeue`	Удаляет элемент из начала очереди.	$O(1)$
`front()` / `peek`	Возвращает первый элемент (голову), не удаляя его.	$O(1)$
`back()`	Возвращает последний элемент (хвост).	$O(1)$
`empty()`	Проверяет, пуста ли очередь.	$O(1)$
`size()`	Возвращает количество элементов.	$O(1)$

Проблема реализации на массиве

Если реализовывать очередь на обычном динамическом массиве (`std::vector`):

- `push` (добавление в конец) работает за $O(1)$.
- `pop` (удаление из начала) требует **сдвига всех остальных элементов** влево на одну позицию. Это стоит $O(N)$.

Решение: Использовать **Циклический буфер (Circular Buffer)** или Связный список.

Реализация: Циклический буфер (Кольцевая очередь)

Мы используем массив фиксированного размера и два указателя: `head` (начало) и `tail` (конец). Когда указатель доходит до конца массива, он перескакивает на индекс 0 (по модулю размера).

```
template <typename T>
class CircularQueue {
    std::vector<T> data;
    int head = 0;
    int tail = 0;
    int count = 0;
    int capacity;

public:
    CircularQueue(int size) : data(size), capacity(size) {}

    void push(T val) {
        if (count == capacity) throw std::overflow_error("Queue full");
        data[tail] = val;
        tail = (tail + 1) % capacity; // Зацикливание
        count++;
    }

    void pop() {
        if (count == 0) throw std::underflow_error("Queue empty");
        head = (head + 1) % capacity; // Зацикливание
        count--;
    }

    T front() {
        if (count == 0) throw std::underflow_error("Queue empty");
        return data[head];
    }
};
```

Примечание: В C++ стандартный контейнер `std::queue` по умолчанию использует `std::deque` (двусвязную очередь), которая состоит из кусочков (чанков) памяти, что позволяет делать `push` и `pop` с обоих концов за $O(1)$ без реаллокаций всего массива.

Области применения

1. **Буферизация данных:** Буфер клавиатуры, очередь печати принтера, буферизация видео (потоковая передача).
2. **Обход графов (BFS):** Поиск в ширину использует очередь для хранения вершин, которые нужно посетить.
3. **Планировщики задач:** Очередь процессов в операционной системе (Round Robin).
4. **Паттерны проектирования:** Очереди сообщений (Message Queues) в распределенных системах (Kafka, RabbitMQ) для асинхронной обработки.

14. Очередь на связных списках (Linked List Queue)

Реализация очереди с использованием односвязного списка. Этот подход обеспечивает динамическое управление памятью (очередь растет по мере необходимости) и гарантирует выполнение всех операций за $O(1)$.

В отличие от стека, где нам достаточно знать только верхушку (`head`), в очереди нам необходимо поддерживать два указателя:
1. `head` (Голова) — для удаления элементов (Pop).
2. `tail` (Хвост) — для добавления элементов (Push).

Визуализация структуры

Элементы добавляются в хвост и забираются с головы.

Head ↓ **Tail** ↓

10 → | 20 → | 30 → | 40 → | null

Поток данных: ← Pop (удаление) происходит здесь. Push (вставка) происходит здесь ←

Алгоритм операций

1. Push (Enqueue) - Вставка в конец

1. Создать новый узел `newNode`.
2. Если очередь пуста: `head = tail = newNode`.
3. Если не пуста:
   - `tail->next = newNode` (привязываем новый узел к текущему хвосту).
   - `tail = newNode` (перемещаем указатель хвоста на новый узел).

2. Pop (Dequeue) - Удаление из начала

1. Проверить на пустоту (Underflow).
2. Сохранить указатель на удаляемый узел: `temp = head`.
3. Сдвинуть голову: `head = head->next`.
4. **Важный граничный случай:** Если после удаления `head` стал `nullptr` (очередь опустела), необходимо обнулить и `tail` (`tail = nullptr`), иначе он останется указывать на удаленную память (dangling pointer).
5. Удалить `temp` из памяти.

Реализация (C++)

```
template <typename T>
struct Node {
    T val;
    Node* next;
    Node(T v) : val(v), next(nullptr) {}
};

template <typename T>
class ListQueue {
private:
    Node<T>* head;
    Node<T>* tail;
    size_t sz;

public:
    ListQueue() : head(nullptr), tail(nullptr), sz(0) {}

    // Деструктор для очистки памяти
    ~ListQueue() {
        while (!empty()) pop();
    }

    void push(T val) {
        Node<T>* newNode = new Node<T>(val);
        if (empty()) {
            head = tail = newNode;
        } else {
            tail->next = newNode; // Связываем текущий хвост с новым
            tail = newNode;       // Обновляем хвост
        }
        sz++;
    }

    void pop() {
        if (empty()) throw std::underflow_error("Queue is empty");

        Node<T>* temp = head;
        head = head->next;

        if (head == nullptr) {
            tail = nullptr; // Очередь опустела
        }

        delete temp;
        sz--;
    }

    T front() {
        if (empty()) throw std::underflow_error("Queue is empty");
        return head->val;
```

```
    }

    bool empty() {
        return head == nullptr;
    }
};
```

Сравнение реализаций очереди

Критерий	**На массиве (Circular Buffer)**	**На списке (Linked List)**
Память	Экономична (нет указателей), но фиксированный размер.	Оверхед на указатели (`next`) для каждого элемента.
Скорость	Высокая кэш-локальность (данные рядом).	Низкая кэш-локальность (скачки по памяти).
Масштабируемость	Требует переаллокации ($O(N)$) при переполнении.	Динамически растет без задержек.
Сложность операций	Всегда $O(1)$.	Всегда $O(1)$ (но с выделением памяти `new`).

Вывод

Реализация на списках предпочтительна, когда:

1. Заранее неизвестно количество элементов.
2. Количество элементов сильно варьируется (то пусто, то миллион).
3. Важна гарантия $O(1)$ на вставку (нет пауз на ресайз массива).

10. Односвязный список (Singly Linked List)

Линейная структура данных, состоящая из узлов (Nodes). Каждый узел содержит:
1. Данные (`data`).
2. Указатель на следующий узел (`next`).

Последний узел указывает на `nullptr`.

Главное отличие от массива: Элементы в памяти могут располагаться хаотично (не последовательно). Связь поддерживается только через указатели.

Визуализация

Head $\rightarrow$

$\boxed{12 \mid \cdot} \rightarrow \boxed{99 \mid \cdot} \rightarrow \boxed{37 \mid \cdot} \rightarrow$ **nullptr**

Структура узла (C++)

```
template <typename T>
struct Node {
    T data;
    Node* next;
    Node(T val) : data(val), next(nullptr) {}
};
```

Операции и сложность

Операция	**Сложность**	**Комментарий**
Доступ по индексу	$O(N)$	Нужно пройти N раз по `next` от головы.
Вставка в начало	$O(1)$	Создать узел, перекинуть `head`.
Вставка в конец	$O(N)$	Если нет указателя `tail`. Если есть `tail` — $O(1)$.
Удаление первого	$O(1)$	Сдвинуть `head` на `head->next`.
Удаление произвольного	$O(N)$	Нужно найти **предыдущий** элемент, чтобы перекинуть его ссылку `next`.

Плюсы и минусы

1. **Плюсы:** Динамический размер (нет реаллокаций), дешевая вставка/удаление, если известна позиция.

- **Минусы:** Нет произвольного доступа (random access), накладные расходы на хранение указателя (4 или 8 байт на узел), плохая кэш-локальность.

11. Двусвязный список (Doubly Linked List)

Улучшенная версия списка, где навигация возможна в обоих направлениях. Каждый узел содержит:
1. Данные (`data`).
2. Указатель на следующий узел (`next`).
3. Указатель на предыдущий узел (`prev`).

Визуализация

null $\leftrightarrow$ $\boxed{\cdot \mid 12 \mid \cdot}$ $\leftrightarrow$ $\boxed{\cdot \mid 99 \mid \cdot}$ $\leftrightarrow$ $\boxed{\cdot \mid 37 \mid \cdot}$ $\leftrightarrow$ **null**

Структура узла (C++)

```
template <typename T>
struct DNode {
    T data;
    DNode* next;
    DNode* prev;
    DNode(T val) : data(val), next(nullptr), prev(nullptr) {}
};
```

Ключевые отличия от односвязного

1. **Удаление узла за $O(1)$:**. В односвязном списке, чтобы удалить узел `X`, нам нужно иметь указатель на `prev`. В двусвязном списке у `X` уже есть `X->prev`. Мы просто делаем:

   ```
   X->prev->next = X->next;
   X->next->prev = X->prev;
   delete X;
   ```

2. **Двусторонний обход:** Можно итерироваться с конца в начало (`tail -> head`).

3. **Память:** Требует больше памяти (два указателя на каждый элемент). Оверхед на 64-битной системе — 16 байт на элемент.

Сравнение структур (Массив vs Списки)

Характеристика	Массив	Односвязный	Двусвязный
Доступ (Index)	$O(1)$	$O(N)$	$O(N)$
Вставка в начало	$O(N)$ (сдвиг)	$O(1)$	$O(1)$
Вставка в середину	$O(N)$	$O(1)$ (если есть ptr)	$O(1)$ (если есть ptr)
Память (Оверхед)	Минимальный	Средний (1 ptr)	Высокий (2 ptrs)
Кэш-локальность	Отличная	Плохая	Плохая

Применение в реальных системах

- **Односвязный:** Реализация хеш-таблиц (метод цепочек), простые стеки.
- **Двусвязный:** `std::list` в C++, реализация LRU-кэша, список окон в ОС (Alt+Tab), undo/redo операции.

12. Циклический список (Circular Linked List)

Разновидность связного списка, в котором последний узел указывает не на `nullptr`, а обратно на первый узел. Таким образом, список образует кольцо.

Может быть как **односвязным** (Singly Circular), так и **двусвязным** (Doubly Circular).

Визуализация

Head →

A | · → B | · → C | · → **to A**

Особенности реализации

Структура узла (`Node`) остается такой же, как в обычном списке. Изменяется логика операций и условие остановки при обходе.

Главное отличие: У нас нет `nullptr`. Если мы будем просто идти по `next`, мы попадем в бесконечный цикл.

Обход списка (Traversal)

Для обхода используется цикл `do-while`, чтобы гарантированно посетить первый элемент и остановиться, когда снова вернемся к нему.

```
void printList(Node* head) {
    if (!head) return;

    Node* current = head;
    do {
        std::cout << current->data << " ";
        current = current->next;
    } while (current != head);
}
```

Вставка элемента

Есть два нюанса вставки (например, в начало):

1. В **односвязном** циклическом списке, чтобы вставить новый элемент перед `head` (сделать его новым `head`), нам нужно найти **последний** элемент (`tail`), чтобы перенаправить его `next` на новый узел. Это занимает $O(N)$.
2. **Оптимизация:** Часто хранят указатель не на `head`, а на `tail`. Тогда:
   - Доступ к началу: `tail->next`.
   - Доступ к концу: `tail`.
   - Вставка в начало и конец становится $O(1)$.

Двусвязный циклический список

Самая мощная вариация.

- `head->prev` указывает на `tail`.
- `tail->next` указывает на `head`.

Это позволяет двигаться по кругу в любом направлении бесконечно.

Преимущества и Недостатки

Преимущества	**Недостатки**
Любой узел может быть стартовой точкой для полного обхода.	Риск бесконечного цикла, если не следить за условием выхода.
Удобно для реализации кольцевых структур (буферов).	Сложнее реализация вставки/удаления (нужно поддерживать замкнутость).
Быстрый доступ к концу и началу (если храним `tail`).	

Примеры использования

1. **Планировщик задач (Round Robin):** ОС выделяет квант времени каждому процессу по очереди. Когда список процессов заканчивается, планировщик переходит к первому.
2. **Задача Иосифа Флавия:** Классическая задача на выбывание людей из круга.
3. **Слайдеры изображений / Плейлисты:** Переключение «Next» на последнем элементе возвращает к первому.

15. Бинарный поиск (Binary Search)

Эффективный алгоритм поиска элемента в **отсортированном** массиве.

Основная идея: на каждом шаге мы сравниваем искомый элемент со **средним** элементом массива. Если они не равны, мы можем отбросить половину массива, так как знаем, что искомый элемент точно не там (благодаря сортировке).

Визуализация процесса

Ищем число **7** в массиве.

Шаг 1: Весь массив

Диапазон `[L, R]`. Средний элемент `mid` $= 5$.
$5 < 7$, значит, все слева нам не нужно.

1	3	5	7	9	11	13	15	19

Шаг 2: Правая половина

Новый диапазон `[L, R]`. Средний элемент `mid` $= 11$.
$11 > 7$, значит, все справа нам не нужно.

1	3	5	7	9	11	13	15	19

Шаг 3: Сужение

Новый диапазон `[L, R]`. Средний элемент `mid` $= 7$.
$7 == 7$. **Найдено!**

1	3	5	**7**	9	11	13	15	19

Реализация (C++)

Классический итеративный подход.

```
int binarySearch(const std::vector<int>& arr, int target) {
    int left = 0;
    int right = arr.size() - 1;

    while (left <= right) {
        // Защита от переполнения (integer overflow)
        // Вместо (left + right) / 2
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid; // Элемент найден
        }
        if (arr[mid] < target) {
            left = mid + 1; // Ищем в правой половине
        } else {
            right = mid - 1; // Ищем в левой половине
        }
    }
    return -1; // Не найдено
}
```

Критический баг: Overflow

Классическая ошибка новичка (и даже профессионалов): `int mid = (left + right) / 2;` Если `left` и `right` — большие положительные числа (близкие к `INT_MAX`), их сумма может переполниться и стать отрицательной. **Правильный вариант:** `int mid = left + (right - left) / 2;`

Анализ сложности

Время (Time Complexity)

$$O(\log_2 N)$$

На каждом шаге область поиска уменьшается в 2 раза. Для 1 000 000 элементов требуется всего ≈ 20 сравнений. Для 4 миллиардов (`uint32`) — всего 32 сравнения.

Память (Space Complexity)

- **Итеративная версия:** $O(1)$.
- **Рекурсивная версия:** $O(\log N)$ на стек вызовов.

Вариации (STL)

В C++ `<algorithm>` есть готовые функции, которые часто полезнее простого поиска:

1. `std::binary_search`: Возвращает `true`/`false`.
2. `std::lower_bound`: Возвращает итератор на **первый** элемент, который $\geq$ `target`.
3. `std::upper_bound`: Возвращает итератор на **первый** элемент, который $>$ `target`.

Это позволяет находить диапазоны равных элементов или место для вставки.

Бинпоиск по ответу (Binary Search on Answer)

Мощная техника решения задач, где нужно найти «минимальное значение X, при котором выполняется условие».

Условие применимости: Функция проверки `check(x)` должна быть **монотонной**. Например: `[False, False, False, True, True, True]`. Нам нужно найти первую `True`.

Пример: Задача «Коровы в стойлах» или «Дипломы». Мы не знаем ответ, но можем проверить, подходит ли число X, за $O(N)$. Тогда мы запускаем бинпоиск по диапазону возможных ответов.

Сложность: $O(N \times \log(\text{AnswerRange}))$.

```
// Шаблон бинпоиска по ответу (поиск первого True)
// Диапазон [L, R]
while (R - L > 1) {
    int mid = L + (R - L) / 2;
    if (check(mid)) {
        R = mid; // mid подходит, пробуем меньше (левее)
    } else {
        L = mid; // mid не подходит, нужно больше
    }
}
// Ответ в R (или L, зависит от задачи)
```

16. Пирамида (Binary Heap)

Двоичная куча (пирамида) — это **почти полное** бинарное дерево, которое удовлетворяет **свойству кучи**.

Обычно реализуется на базе массива.

Основные свойства

1. **Структурное свойство:** Дерево заполняется по уровням слева направо. У него нет «дырок».
2. **Свойство кучи (Heap Property):**
   - Для **Min-Heap:** Значение в любой вершине **меньше или равно** значений её детей. (Корень — минимум).
   - Для **Max-Heap:** Значение в любой вершине **больше или равно** значений её детей. (Корень — максимум).

Хранение в памяти (Array Mapping)

Так как дерево почти полное, нам не нужны указатели `left` и `right`. Мы используем арифметику индексов.

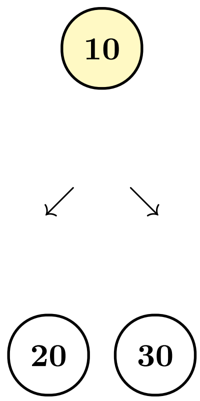
Нумерация вершин (0-based):

- Корень: индекс 0.
- Для узла с индексом i:
  - **Левый ребенок:** $2i + 1$
  - **Правый ребенок:** $2i + 2$
  - **Родитель:** $\frac{i-1}{2}$ (целочисленное деление)

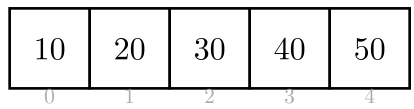
Визуализация (Дерево ↔ Массив)

Рассмотрим **Min-Heap**: `[10, 20, 30, 40, 50]`

Логическая структура



Физическая структура (Массив)



$$i = 1(\text{знач}\ \ 20) \to \text{Родитель} = \frac{1-1}{2} = 0(\text{знач}\ \ 10)$$

Основные операции

Вся магия кучи держится на двух процедурах восстановления баланса.

1. Sift Up (Просеивание вверх / Всплытие)

Когда: При **вставке** нового элемента. **Суть:** Мы добавляем элемент в самый конец массива (чтобы сохранить структуру дерева). Если он меньше родителя (в Min-Heap), мы нарушили порядок. Меняем их местами (`swap`). Повторяем, пока элемент не «всплывет» на свое место или не станет корнем.

Сложность: $O(\log N)$ (высота дерева).

2. Sift Down (Просеивание вниз / Погружение)

Когда: При **удалении корня** (extract min/max) или построении кучи. **Суть:** Мы удаляем корень, а на его место ставим **последний** элемент массива (чтобы не было дырок). Свойство кучи нарушено сверху. Мы выбираем **меньшего** из детей и меняем элемент с ним местами. Повторяем, пока элемент не «утонет» до нужного уровня.

Сложность: $O(\log N)$.

Реализация (C++ Min-Heap)

```cpp
class MinHeap {
    vector<int> heap;

    void siftUp(int i) {
        while (i > 0) {
            int parent = (i - 1) / 2;
            if (heap[i] < heap[parent]) {
                swap(heap[i], heap[parent]);
                i = parent;
            } else {
                break;
            }
        }
    }

    void siftDown(int i) {
        while (2 * i + 1 < heap.size()) {
            int left = 2 * i + 1;
            int right = 2 * i + 2;
            int j = left; // Предполагаем, что левый меньше

            // Если правый существует и он меньше левого, выбираем его
            if (right < heap.size() && heap[right] < heap[left]) {
                j = right;
            }

            // Если текущий элемент уже меньше детей — стоп
            if (heap[i] <= heap[j]) break;

            swap(heap[i], heap[j]);
            i = j;
        }
    }

public:
    // Добавление (Insert)
    void push(int x) {
        heap.push_back(x);
        siftUp(heap.size() - 1);
    }

    // Извлечение минимума (Extract Min)
    int pop() {
        int res = heap[0];
        heap[0] = heap.back(); // Ставим последний в корень
        heap.pop_back();       // Удаляем последний
        siftDown(0);           // Чиним порядок
        return res;
    }
};
```

Построение кучи (Build Heap)

Если у нас уже есть массив чисел, мы можем превратить его в кучу.

1. **Наивный способ:** N раз вызвать `push`. Сложность $O(N \log N)$.
2. **Оптимальный способ (Heapify):** Пройтись от середины массива к началу и для каждого элемента вызвать `siftDown`. Сложность: $\boldsymbol{O(N)}$. *Доказательство:* Элементов на нижних уровнях много, но просеивать их некуда (высота 0). Элементов наверху мало, просеивать глубоко. Сумма ряда сходится к линейной сложности.

Математическое обоснование $O(N)$ для Build Heap

Пусть h — высота узла (расстояние до листа).

- Листьев ($h = 0$) примерно $\frac{N}{2}$. Для них спуск занимает 0 операций.
- Узлов на высоте $h = 1$ примерно $\frac{N}{4}$. Спуск занимает 1 операцию.
- Узлов на высоте h примерно $\frac{N}{2^{\{h+1\}}}$.

Общее количество операций S:

$$S = \sum_{h=0}^{\log N} \frac{N}{2^{h+1}} \cdot h = N \sum_{h=0}^{\log N} \frac{h}{2^{h+1}}$$

Вынесем $\frac{N}{2}$ за скобку:

$$S = \frac{N}{2} \sum_{h=0}^{\log N} \frac{h}{2^h}$$

Это **арифметико-геометрическая прогрессия**. Чтобы оценить сверху, заменим верхний предел на ∞ (бесконечный ряд):

$$S < \frac{N}{2} \sum_{h=0}^{\infty} \frac{h}{2^h}$$

Известно, что сумма ряда $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$ (при $|x| < 1$). Подставим $x = \frac{1}{2}$:

$$\sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h = \frac{\frac{1}{2}}{(1 - \frac{1}{2})^2} = \frac{0.5}{0.25} = 2$$

Итого:

$$S < \frac{N}{2} \cdot 2 = N$$

Суть доказательства в одной фразе: «Сумма смещений сходится, потому что количество узлов на каждом уровне уменьшается в 2 раза, а глубина увеличивается только линейно (+1). Экспонента в знаменателе побеждает линейный числитель.»

Вывод: Сложность построения кучи — $O(N)$.

Преимущества и недостатки

Плюсы	**Минусы**
Поиск минимума/максимума за $O(1)$.	Поиск **произвольного** элемента за $O(N)$.
Эффективная память (нет указателей, просто массив).	Слияние двух куч — дорогая операция (если это не биномиальная куча).
Гарантированное $O(\log N)$ на вставку и удаление.	

17. Пирамидальная сортировка (Heap Sort)

Алгоритм сортировки, основанный на структуре данных «Двоичная куча» (Binary Heap).

Основная идея:

1. Превратить входной массив в **Max-Heap** (где корень — максимальный элемент).
2. Поменять местами корень (максимум) с последним элементом массива. Теперь максимальный элемент стоит на своем законном месте в конце.
3. Уменьшить размер кучи на 1 (забыть про последний, уже отсортированный элемент).
4. Восстановить свойство кучи для нового корня (операция `siftDown`).
5. Повторять шаги 2-4, пока куча не исчезнет.

Визуализация процесса

Массив: `[4, 10, 3, 5, 1]`

1. Построение Max-Heap:
`[10, 5, 3, 4, 1]` (10 — корень, 5 и 3 — дети).

10	5	3	4	1

↓

2. Swap(Max, Last):
Меняем 10 и 1 местами. 10 «заморожена» (сортирована).

1	5	3	4	10

↓ SiftDown(0)

3. Восстановление кучи (для размера 4):
Новый корень 1 проваливается вниз. Max-Heap восстановлен: `[5, 4, 3, 1]`.

5	4	3	1	10

↓

4. Swap(Max, Last):
Меняем 5 (корень) и 1 (последний в куче).

1	4	3	5	10

Реализация (C++)

Используем вспомогательную функцию `siftDown` (иногда называемую `heapify`), которую мы разобрали в предыдущей теме.

Важно: Для сортировки по возрастанию мы используем **Max-Heap**, чтобы выталкивать большие элементы в конец массива.

```cpp
void siftDown(std::vector<int>& arr, int n, int i) {
    int largest = i;   // Инициализируем наибольший элемент как корень
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    // Если левый дочерний элемент больше корня
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // Если правый дочерний элемент больше, чем самый большой на данный момент
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // Если самый большой элемент не корень
    if (largest != i) {
        std::swap(arr[i], arr[largest]);

        // Рекурсивно просеиваем затронутое поддерево
        siftDown(arr, n, largest);
    }
}

void heapSort(std::vector<int>& arr) {
    int n = arr.size();

    // 1. Построение кучи (Build Heap)
    // Начинаем с последнего родителя и идем до корня
    for (int i = n / 2 - 1; i >= 0; i--)
        siftDown(arr, n, i);

    // 2. Один за другим извлекаем элементы из кучи
    for (int i = n - 1; i > 0; i--) {
        // Перемещаем текущий корень (максимум) в конец
        std::swap(arr[0], arr[i]);

        // Вызываем siftDown на уменьшенной куче (размер i)
        siftDown(arr, i, 0);
    }
}
```

Анализ сложности

Время (Time Complexity)

$$O(N \log N)$$

- **Build Heap:** $O(N)$ (как мы доказали ранее).
- **Sorting:** Мы делаем $N-1$ извлечений. Каждое извлечение требует `siftDown`, который работает за высоту дерева $O(\log N)$.
- Итого: $O(N + N \log N) = O(N \log N)$.

Это справедливо для лучшего, среднего и худшего случаев. Деградации до $O(N^2)$, как у QuickSort, здесь не бывает.

Память (Space Complexity)

$$O(1)$$

Алгоритм работает **in-place**. Мы просто переставляем элементы внутри исходного массива.

Сравнение с конкурентами

Алгоритм	**Heap Sort**	**Quick Sort**	**Merge Sort**
Время (Worst)	$O(N \log N)$	$O(N^2)$	$O(N \log N)$
Память	$O(1)$	$O(\log N)$	$O(N)$
Устойчивость	Нет	Нет	Да
Кэш-локальность	Плохая	Отличная	Средняя

Почему Heap Sort часто медленнее Quick Sort на практике? Из-за скачков по индексам (`i`, `2i+1`, `2i+2`). Элементы, которые логически связаны в дереве (родитель-ребенок), в массиве могут находиться очень далеко друг от друга. Это приводит к частым промахам кэша процессора (Cache Misses).

18. Приоритетная очередь (Priority Queue)

Абстрактная структура данных (ADT), похожая на обычную очередь или стек, но с одним ключевым отличием: у каждого элемента есть **приоритет**.

Элемент с **наивысшим** (или наинизшим) приоритетом всегда извлекается первым, независимо от того, когда он был добавлен.

Аналогия: Очередь в травмпункте. Пациента с тяжелой травмой (высокий приоритет) примут раньше, чем того, кто пришел час назад с насморком (низкий приоритет).

Визуализация (Концепт)

В отличие от FIFO (Queue), здесь нет строгого «хвоста». Элементы попадают в общую кучу, но на выход всегда встает «Король».

Вход (Push)
`push(15)`, `push(50)`, `push(10)`
↓
Черный ящик (Сортировка / Куча)

Выход (Pop)
(Max-Priority)
↑
50
(Самый большой)

Реализация

Хотя приоритетную очередь можно реализовать на массиве или связном списке, это неэффективно. Стандартом де-факто является **Двоичная куча (Binary Heap)**.

Структура	**Insert (Push)**	**Extract Max (Pop)**	**Peek (Top)**
Несортированный массив	$O(1)$	$O(N)$	$O(N)$
Отсортированный массив	$O(N)$	$O(1)$	$O(1)$
Двоичная куча	$O(\log N)$	$O(\log N)$	$O(1)$

Куча обеспечивает идеальный баланс между скоростью добавления и скоростью извлечения.

Интерфейс в C++ (STL)

В C++ есть готовый контейнер `std::priority_queue` в библиотеке `<queue>`. По умолчанию это **Max-Heap**.

```
#include <queue>
#include <vector>
#include <iostream>

int main() {
    // 1. Max-Heap (по умолчанию)
    std::priority_queue<int> pq;

    pq.push(10);
    pq.push(30);
    pq.push(20);
    pq.push(5);

    std::cout << pq.top(); // Выведет 30 (максимум)
    pq.pop();              // Удалит 30

    // 2. Min-Heap (для алгоритма Дейкстры и др.)
    // Используем компаратор std::greater
    std::priority_queue<int, std::vector<int>, std::greater<int>> min_pq;

    min_pq.push(10);
    min_pq.push(30);

    std::cout << min_pq.top(); // Выведет 10 (минимум)
}
```

Применение

Приоритетные очереди — сердце многих жадных алгоритмов и системных процессов.

1. **Алгоритм Дейкстры:** Поиск кратчайшего пути в графе. Очередь хранит вершины, отсортированные по текущему расстоянию до них.
2. **Алгоритм Прима:** Поиск минимального остовного дерева (MST).
3. **Сжатие Хаффмана:** Построение оптимального префиксного кода (дерево строится, объединяя символы с наименьшими частотами).
4. **Планировщик задач в ОС:** Процессы с высоким приоритетом получают квант времени процессора раньше фоновых задач.
5. **K-th Largest Element:** Поиск K-го максимального элемента в потоке данных (держим Min-Heap размером K).

Сложность (Итог)

- **Время:** Все модифицирующие операции занимают $O(\log N)$. Получение экстремума — $O(1)$.
- **Память:** $O(N)$ для хранения элементов.

19. Двоичное дерево поиска (Binary Search Tree - BST)

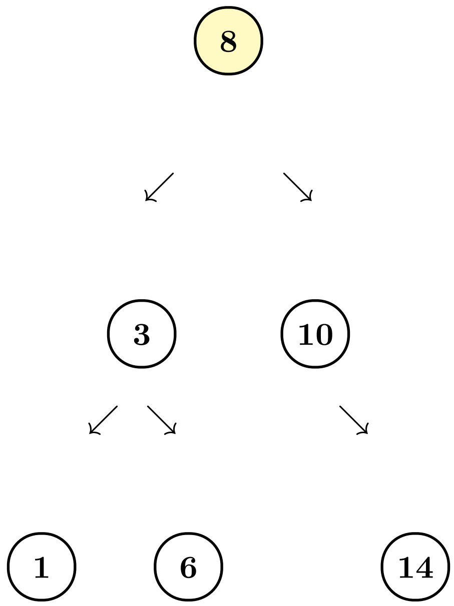
Это двоичное дерево, обладающее следующим фундаментальным свойством (инвариантом):

Для любого узла X:
1. Все ключи в **левом** поддереве меньше ключа X.
2. Все ключи в **правом** поддереве больше ключа X.
3. Левое и правое поддеревья также являются BST.

Примечание: Обычно предполагается, что ключи уникальны. Если дубликаты разрешены, условие меняется на $\leq$ или $\geq$.

Визуализация

Корректное BST:



Проверка:
- Узел 3: слева 1 (< 3), справа 6 (> 3). Ок.
- Корень 8: все слева (1, 3, 6) меньше 8. Все справа (10, 14) больше 8. Ок.

Основные операции

Все операции зависят от высоты дерева h.

1. Поиск (Search)

Идем от корня. Сравниваем искомый ключ K с текущим узлом.

- $K ==$ Node.key $\to$ Нашли.
- $K <$ Node.key $\to$ Идем влево.
- $K >$ Node.key $\to$ Идем вправо.
- Уперлись в `nullptr` $\to$ Элемента нет.

2. Вставка (Insert)

Аналогично поиску. Спускаемся вниз, пока не найдем свободное место (`nullptr`), и прикрепляем туда новый лист.

3. Удаление (Delete) — Самое сложное

Три случая:

1. **Узел — лист (нет детей):** Просто удаляем ссылку у родителя.
2. **Один ребенок:** Заменяем удаляемый узел его единственным ребенком.
3. **Два ребенка:**
   - Находим **преемника** (Successor) — минимальный элемент в **правом** поддереве.
   - Копируем значение преемника в текущий узел.
   - Рекурсивно удаляем преемника (у него гарантированно нет левого ребенка, см. п. 2).

Реализация (C++)

```
struct Node {
    int key;
    Node *left, *right;
    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};

Node* search(Node* root, int key) {
    if (root == nullptr || root->key == key)
        return root;

    if (key < root->key)
        return search(root->left, key);

    return search(root->right, key);
}

Node* insert(Node* root, int key) {
    if (root == nullptr) return new Node(key);

    if (key < root->key)
        root->left = insert(root->left, key);
    else if (key > root->key)
```

```
        root->right = insert(root->right, key);

    return root;
}
```

Обходы дерева (Tree Traversals)

Существует три классических способа обойти все узлы (DFS):

1. **Pre-order (Прямой):** Корень → Лево → Право. (Используется для копирования дерева).
2. **In-order (Центрированный):** Лево → Корень → Право. **Важно:** In-order обход BST дает отсортированную последовательность ключей!
3. **Post-order (Обратный):** Лево → Право → Корень. (Используется для удаления дерева).

Анализ сложности

Это критический момент. Сложность операций BST напрямую зависит от его формы (топологии).

Случай	**Высота (h)**	**Сложность операций**
Лучший / Средний	$\log_2 N$	$O(\log N)$
Худший (Вырожденное)	N	$O(N)$

Проблема деградации

Если вставлять элементы в отсортированном порядке (`1, 2, 3, 4, 5`), BST вырождается в связный список («палку»). Высота становится N, и все преимущества перед обычным массивом теряются.

Решение: Использовать **самобалансирующиеся** деревья (AVL, Red-Black, B-Tree), которые гарантируют высоту $\log N$.

Преимущества и недостатки

1. **Плюсы:** Эффективный поиск и вставка (в среднем), поддержка упорядоченности (range queries), динамический размер.

- **Минусы:** В худшем случае работает медленно ($O(N)$), оверхед на хранение указателей, отсутствие произвольного доступа по индексу.

20. Динамическое программирование (Dynamic Programming)

Метод решения задач, где мы разбиваем сложную проблему на перекрывающиеся подзадачи, решаем каждую один раз и запоминаем ответ («мемоизация»).

1. Задача о кузнечике (Grasshopper)

Условие: Кузнечик стоит на ступеньке 0. Цель — добраться до ступеньки N. За один ход можно прыгнуть на расстояние от 1 до K. Сколькими способами можно это сделать?

Формула перехода

Пусть $dp[i]$ — количество способов добраться до i-й ступеньки.

$$dp[i] = \begin{cases} 1 & \text{если } i = 0 \quad \text{(база)} \\ \sum_{j=1}^{k} dp[i-j] & \text{если } i > 0 \quad \text{(сумма всех предыдущих шагов)} \end{cases}$$

Важное условие реализации: При суммировании проверяем, что $i - j \geq 0$.

Реализация (C++)

```
// Сложность: O(N * K)
vector<int> dp(n + 1, 0);
dp[0] = 1;

for (int i = 1; i <= n; ++i) {
    // Перебираем длину прыжка j от 1 до K
    for (int j = 1; j <= k; ++j) {
        if (i - j >= 0) {
            dp[i] += dp[i - j];
        }
    }
}
```

—

2. Задача о рюкзаке (0/1 Knapsack Problem)

Условие: Дано N предметов. i-й предмет имеет вес w_i и стоимость v_i. Рюкзак вмещает вес W. Найти максимальную суммарную стоимость предметов, которые влезут в рюкзак.

Формула перехода

Пусть $dp[i][current\_w]$ — макс. стоимость, используя подмножество первых i предметов с лимитом веса $current\_w$.

$$dp[i][w] = \begin{cases} dp[i-1][w] & \text{если } w_i > w \quad \text{(предмет не влезает)} \\ \max(dp[i-1][w],\ dp[i-1][w-w_i] + v_i) & \text{если } w_i \le w \end{cases}$$

Пояснение: Максимум из двух вариантов:

1. Не берем предмет (остаемся с результатом предыдущего шага).
2. Берем предмет (добавляем его цену v_i к лучшему результату для оставшегося места $w - w_i$).

Оптимизация памяти до $O(W)$

Нам не нужна вся матрица, достаточно знать только предыдущую строку. Важно идти по весу в обратном порядке!

```
// Сложность: Time O(N * W), Space O(W)
vector<int> dp(W + 1, 0);

for (int i = 0; i < n; ++i) {          // Перебираем предметы
    for (int w = W; w >= weight[i]; --w) { // Идем с конца!
        dp[w] = max(dp[w], dp[w - weight[i]] + value[i]);
    }
}
```

3. Наибольшая возрастающая подпоследовательность (LIS)

Условие: Дан массив чисел. Найти длину самой длинной подпоследовательности, где элементы идут строго по возрастанию.

Подход 1: Классическая динамика ($O(N^2)$)

$dp[i]$ — длина НВП, которая **обязательно заканчивается** элементом $arr[i]$.

$$dp[i] = 1 + \max_{0 \le j < i} \{dp[j] \mid arr[j] < arr[i]\}$$

Если меньших элементов слева нет, то $dp[i] = 1$.

Подход 2: Динамика + Бинпоиск ($O(N \log N)$)

Чтобы ускорить процесс, мы изменим определение состояния динамики.

Пусть $tails[k]$ — это **минимальное** число, на которое может заканчиваться возрастающая подпоследовательность длины $k+1$.

Инвариант: Массив $tails$ всегда будет отсортирован по возрастанию. Это позволяет использовать бинпоиск.

Алгоритм: Проходим по всем числам x из входного массива:

1. Если x больше всех элементов в $tails \to$ мы можем удлинить самую длинную последовательность. Дописываем x в конец $tails$.
2. Если x не больше последнего $\to$ мы ищем в $tails$ первое число, которое $\ge x$, и **заменяем** его на x. *Смысл замены:* Мы нашли более перспективный (меньший) конец

для последовательности той же длины. Это дает нам больше шансов продолжить её в будущем.

Пример: `[10, 9, 2, 5, 3, 7, 101, 18]`

Шаг (число)	**Действие**	**Массив tails**
10	Пусто, вставляем	[10]
9	$9 < 10$, заменяем 10	[9]
2	$2 < 9$, заменяем 9	[2]
5	$5 > 2$, добавляем	[2, 5]
3	$3 < 5$, заменяем 5	[2, 3] (теперь для длины 2 у нас конец 3, это круче чем 5)
7	$7 > 3$, добавляем	[2, 3, 7]
101	$101 > 7$, добавляем	[2, 3, 7, 101]
18	$18 < 101$, заменяем 101	[2, 3, 7, 18]

Длина массива `tails` (4) — это и есть длина НВП.

Реализация ($O(N \log N)$)

```
int lengthOfLIS(vector<int>& nums) {
    if (nums.empty()) return 0;

    vector<int> tails;

    for (int x : nums) {
        // lower_bound ищет первый элемент, который >= x
        auto it = std::lower_bound(tails.begin(), tails.end(), x);

        if (it == tails.end()) {
            // Если x больше всех, расширяем последовательность
            tails.push_back(x);
        } else {
            // Иначе обновляем существующий конец на более оптимальный (меньший)
            *it = x;
        }
    }
    return tails.size();
}
```