13. Recursion, about algorithms and their complexities on linear data structures: binary_search, optional{merge_sort, quick_sort}.

Recursion

> **Рекурсия** — это метод программирования, при котором функция вызывает сама себя для решения подзадачи, которая является частью общей задачи.

Основные свойства рекурсии:

1. *Базовый случай*: Определяет условие, при котором рекурсия прекращается (например, при достижении минимального размера задачи).
2. *Рекурсивный случай*: Определяет, как задача разбивается на более мелкие подзадачи, которые решаются рекурсивно.

!!! Важно помнить,

что когда функция вызывается рукурсивно, она потом должна вернуться туда, откуда ее вызвали.

Пусть дана задача:
Имеем телефонную книгу с номерами, они отсортированы, нужно найти определенный номер.
- *алгоритм*:
- найдем середину,
- если наше значение меньше середины - вызовем рекурсивно на левой части массива номеров,
- если больше - на правой

Binary search

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

int find(const std::vector<int>& vec, int val) {
    if (vec.empty())
        return -1;

    int midIndex = vec.size() / 2;

    if (vec[midIndex] == val)
        return midIndex;
    if (vec[midIndex] > val)
        return find(std::vector<int>(vec.begin(), vec.begin() + midIndex);
    return find(std::vector<int>(vec.begin() + midIndex + 1, val);
}
```

Это решение за $O(n)$ (вектор копируется при каждом вызове `find`).

Покажем, что $O(n)$.
Обозначим количество действий $T(n)$:

- **Рекурсивный вызов:**

        - Алгоритм делит массив пополам, вызывая сам себя на одной из половин. Это выражено как $T\left(\frac{n}{2}\right)$, то есть рекурсивная работа над половиной массива.
    - **Дополнительная работа на текущем уровне:**
        - На каждом уровне рекурсии выполняется *дополнительная работа*, которая связана с копированием половины массива в новый вектор. В данном случае, чтобы создать новый подмассив (либо левую, либо правую часть), нужно скопировать $\frac{n}{2}$ элементов, что требует $O(\frac{n}{2})$ операций.
        - Этот этап отражает необходимость *скопировать часть массива*, которая занимает $\frac{n}{2}$ действий.

$$T(n) = T\left(\frac{n}{2}\right) + \frac{n}{2}$$

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{2^2}\right) + \frac{n}{2^2}$$

$$T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right) + \frac{n}{2^3}$$

```
...
```

$$T(0) = 1$$

$$T(n) \leq \frac{n}{2} + \frac{n}{2^2} + \frac{n}{2^3} + \cdots = n$$

$$T(n) = O(n) \text{ — количество действий.}$$

Кроме того, что выполнение требует O(n) действий, еще и требуется столько же дополнительной памяти
Улучшим решение, используя итераторы

```cpp
#include <iostream>
#include <vector>

// Template function for finding an element in a sorted range using binary search
template <typename It>
It find(It start, It finish, typename It::value_type value) {
    // Base case: if the range is empty, return a default-initialized iterator
    if (finish == start) {
        return finish;
    }

    // Calculate the middle iterator
    It mid = start + (finish - start) / 2;

    // Check if the middle element is the target value
    if (value == *mid) {
        return mid;
    }

    // If the target value is smaller, search the left half
    if (value < *mid) {
        return find(start, mid, value);
    }

    // Otherwise, search the right half
    return find(mid + 1, finish, value);
}
```

Выполняется за $O(\log_2 n)$ операций:

$$T(n) = T\left(\frac{n}{2}\right) + C$$

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{2^2}\right) + C$$

$$T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right) + C$$

```
...
```

$$T\left(\frac{n}{2^k}\right) = T\left(\frac{n}{2^{k+1}}\right) + C$$

```
...
```

$$T(1) = C$$

Найдём количество C:

$$\frac{n}{2^k} \leq 1$$

$$2^k \geq n$$

$$k \geq \log_2 n$$

$$k = \lfloor \log_2 n \rfloor$$

Получаем количество $C = k$ штук, где $k = \lfloor \log_2 n \rfloor$.

? **Почему k=C:**

- k в данном случае отражает *количество итераций*/уровней рекурсии, а C — *количество операций* на уровне.
- Для упрощения анализа C принимается за пропорциональное k, так как каждое деление области увеличивает количество вычислений на фиксированное число C, пропорционально количеству уровней рекурсии.

!! binary search нельзя использовать для структур без `random access iterator`

В STL библиотеке есть такая встроенная функция `binary search`

Merge Sort

Разделяет массив на две части, сортирует их рекурсивно и затем объединяет.
Сложность: *O(nlogn)* — из-за деления массива пополам и слияния элементов.

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

std::vector<int> merge(const std::vector<int>& arr1, const std::vector<int>& arr2)
{
   std::vector<int> result;

   std::size_t index1 = 0;
   std::size_t index2 = 0;

   while (index1 < arr1.size() && index2 < arr2.size())
```

```
    {
        if (arr1[index1] < arr2[index2])
            result.push_back(arr1[index1++]);
        else
            result.push_back(arr2[index2++]);
    }

    while (index1 < arr1.size())
        result.push_back(arr1[index1++]);

    while (index2 < arr2.size())
        result.push_back(arr2[index2++]);

    return result;
}


std::vector<int> mergeSort(const std::vector<int>& arr)
{
    if (arr.size() <= 1)
        return arr;
    std::size_t mid = arr.size() / 2;

    std::vector<int> left(arr.begin(), arr.begin()+mid);
    std::vector<int> right(arr.begin() + mid, arr.end());

    left = mergeSort(left);
    right = mergeSort(right);

    return merge(left, right);
}
```

Quick Sort

```
#include <iostream>
#include <vector>
#include <algorithm>

// Partition function: rearranges elements around a pivot
// Elements less than the pivot are placed to its left, and elements greater or equal are placed to its right.
template <typename It>
It partition(It start, It finish) {
    It it = start;
    ++it;
    It wall = start;

    // Iterate through the range and partition the elements
    for (; it != finish; ++it) {
        if (*it < *start) {
            ++wall;
            std::swap(*wall, *it);
        }
    }

    // Place the pivot element in its correct position
    std::swap(*start, *wall);
```

```
    return wall;
}

// QuickSort function: sorts elements in the range [start, finish)
template <typename It>
void quickSort(It start, It finish) {
    if (start == finish) {
        return; // Base case: empty or single-element range
    }

    // Partition the range and get the pivot position
    It pivot = partition(start, finish);

    // Recursively sort the left and right partitions
    quickSort(start, pivot);
    quickSort(++pivot, finish);
}

int main() {
    std::vector<int> arr = {10, 7, 8, 9, 1, 5};
    quickSort(arr.begin(), arr.end());
    for (int num : arr) {
        std::cout << num << " ";
    }

    return 0;
}
```

Оценим сложность Quick Sort

Функция затрат описывается следующим рекуррентным соотношением:
$T(n) = T(\text{размер левой части}) + T(\text{размер правой части}) + \text{затраты на разделение (partition)}$.

- Если `pivot` попадает на *середину*, то:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$2T\left(\frac{n}{2}\right) = 2^2T\left(\frac{n}{2^2}\right) + n$$

```
...
```

Для $k = \log_2 n$, на последнем уровне размер подмассива будет равен 1, то есть:

$$\frac{n}{2^k} = 1.$$

Сумма всех затрат на каждом уровне:

$$T(n) = n + n + n + \cdots + n \cdot \log_2 n.$$

$\log_2 n$. - глубина дерева, а количество операций на каждом уровне - n.
Общее количество операций составляет:

$$T(n) = O(n \log_2 n).$$

Итог:

В лучшем случае (pivot делит массив на равные части) сложность Quick Sort равна $O(n \log_2 n)$. Это достигается, если массив делится пополам на каждом уровне.

- Если `pivot` попадает на *начало*, то:

$$T(n-1) = T(n-2) + (n-1),$$
$$T(n-2) = T(n-3) + (n-2),$$
$$T(n-3) = T(n-4) + (n-3),$$

и так далее.

Общая сумма:
На каждом уровне добавляется затрата:

$$n,\ (n-1),\ (n-2),\ \ldots,\ 1,$$

что даёт арифметическую прогрессию:

$$T(n) = n + (n-1) + (n-2) + \cdots + 1.$$

Сумма арифметической прогрессии вычисляется по формуле:

$$T(n) = \frac{n(n+1)}{2}.$$

Итоговая сложность:
Итак, общая сложность в худшем случае:

$$T(n) = O(n^2).$$