Сортировки.
Двоичный поиск.

Быстрая сортировка (Ч. Хоар, 1962)

- Идея: «разделяй и властвуй»

Среднее время работы – O(n log n)

В худшем случае – O(n^2)

В обычной реализации неустойчива

```
void Sort(vector<int> &arr, int l, int r)
{
    int i = l, j = r;
    int bar = arr[(l + r) / 2];
    while (i <= j){
        while (arr[i] < bar) i++;
        while (bar < arr[j]) j--;
        if (i <= j){
            swap(arr[i], arr[j]);
            i++, j--;
        }
    }
    if (l < j) Sort(arr, l, j);
    if (i < r) Sort(arr, i, r);
}
```

Пример:

1) 4 9 7 6 2 3 8
2) 4 3 2 6 7 9 8
3) 2 3 4 6 7 9 8
4) 2 3 4 6 7 8 9

Сортировка слиянием (Дж. Фон Нейман, 1945)

- Идеи: «разделяй и властвуй», слияние отсортированных массивов
- Время работы – O($n \log_2 n$)
- Сортировка устойчива

```cpp
void MergeSort(vector<int> &arr, int l, int r)
{
    if (l < r){
        int mid = (l + r) / 2;
        MergeSort(arr, l, mid);
        MergeSort(arr, mid + 1, r);
        Merge(arr, l, mid, r);
    }
}
```

```cpp
void Merge(vector<int> &arr, int l, int mid, int r)
{
    int n1 = mid - l + 1;
    int n2 = r - mid;
    vector<int> L(n1 + 1);
    vector<int> R(n2 + 1);
    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int i = 0; i < n2; i++)
        R[i] = arr[mid + i + 1];
    L[n1] = R[n2] = inf;
    int i = 0, j = 0;
    for (int k = l; k <= r; k++){
        if (L[i] <= R[j]) arr[k] = L[i++];
        else arr[k] = R[j++];
    }
}
```

Пример:

2 4 7 6 2 3 8 - [0;6]

2 4 7 6 2 3 8 - [0;3]

2 4 7 6 2 3 8 - [0;1]

2 4 7 6 2 3 8 - [2;3]

2 4 6 7 2 3 8 - [4;6]

2 4 6 7 2 3 8 - [4;5]

Итог: 2 2 3 4 6 7 8

Пирамидальная сортировка

- Идея: использование кучи
- Время работы – O($n\ log_2\ n$)
- Сортировка неустойчива

Сортировка разбиралась на теме:

«Структуры данных»

Сортировка подсчётом

- Идея: использование конечной длины сортируемых чисел
- Время работы – O(n)
- Сортировка устойчива

```cpp
void CountingSort(vector<int> &arr)
{
    int min = inf, max = -inf;
    for (int i = 0; i < (int)arr.size(); i++){
        min = std::min(arr[i], min);
        max = std::max(arr[i], max);
    }
    vector<int> cnt(max - min + 1, 0);
    for (int i = 0; i < (int)arr.size(); i++)
        cnt[arr[i] - min]++;
    int uk = 0;
    for (int i = min; i <= max; i++){
        for (int j = cnt[i - min]; j >= 0;  j--)
            arr[uk++] = i;
    }
}
```

Двоичный поиск

Двоичный поиск — алгоритм поиска объекта по заданному признаку в множестве объектов, упорядоченных по тому же самому признаку, работающий за логарифмическое время.

Задача

- Пусть нам дан упорядоченный массив, состоящий только из целочисленных элементов. Требуется найти позицию, на которой находится заданный элемент или сказать, что такого элемента нет.

Принцип работы

Двоичный поиск заключается в том, что на каждом шаге множество объектов делится на две части и в работе остаётся та часть множества, где находится искомый объект.

```cpp
int BinarySearch(vector<int> &arr, int val)
{
    int l = -1, r = (int)arr.size();
    while (r - l > 1){
        int mid = (l + r) / 2;
        if (arr[mid] < val) l = mid;
        else r = mid;
    }
    return (r < (int)arr.size() && arr[r] == val ? r : -1);
}
```

Задача

- Пусть у нас есть N принтеров и нам нужно напечатать К листовок. Каждый принтер печатает одну листовку за a_i секунд. Сколько нам потребуется минимальное количество времени, чтобы напечатать все листовки?
- $N = 10^5, \mathrm{K} = 10^9, a_i \leq 10^9$

- Задачу можно решить с помощью двоичного поиска по ответу.
- Будем искать двоичным методом время, за которое мы сможем напечатать все листовки.
- Для каждого время мы сможем за O(N) проверить этот факт.
- Так как функция Количество_листовок(время) монотонна, следовательно здесь применим двоичный поиск
- Получим решение за O(n log (10^{18}))

- Как сам двоичный поиск, так и метод двоичного поиска по ответу очень(!) часто встречается в задачах.

Рассмотрим ещё одну задачу:

Пусть нам задана монотонная функция и два значения аргумента x1, x2. Сказано, что на отрезке [x1;x2] имеется ровно один корень функции, т.е. $\exists!\, x\colon\ f(x) = 0$ и f(x1)*f(x2) < 0.

Нужно найти x.

Некоторые полезные советы при работе с вещественными числами

- Когда имеешь дело с вещественными числами в первую очередь нужно подумать нельзя ли от них избавиться и перейти к целым.

Пример 1:

Нам нужно сравнить два числа вида a/b, где a и b целые числа

Неправильный выбор:

If ((double)a/b < (double)c/d)

Правильный выбор:

If (a * d < c * b)

Исключение:

Когда a * d или c * b не помещаются в целочисленный тип

Пример 2:

Нам нужен цикл до sqrt(n) включительно.

Неправильный выбор:

```
for(int i = 0; i <= sqrt(n); i++)
```

Правильный выбор:

```
for(int i = 0; i * i <= n; i++)
```

Пример 3:

Сравнить расстояния между точками a,b и c,d

```
int d1 = sqr(a.x-b.x) + sqr(a.y-b.y);
int d2 = sqr(c.x-d.x) + sqr(c.y-d.y);
```

Вместо:

```
if (sqrt(d1) < sqrt(d2))
```

Лучше:

```
if (d1 < d2)
```

- Если всё-таки приходится работать с вещественными числами, то всегда нужно стараться уменьшить погрешность вычислений

Пример 1:

b/a + c/a + … = (b + c + …)/a;

Пример 2:

У нас есть прямоугольный треугольник, мы знаем длины его сторон a,b,c и один из углов A. Нужно найти sin(B)

Не лучший выбор:

```
sinb = sin(pi - A);
```

Можно так:

```
sinb = b/c;
```

- Если у нас возможно равенство вещественных чисел, то их всегда нужно сравнивать по eps
- abs(a - b) < eps <=> a == b

Eps должен быть меньше требуемой точности, но больше лучшей точности.

Обычно выбирают eps = 1e-9;

А вообще вопрос точности при работе с вещественными типами это философский вопрос ☺️

- При работе с бинпоиском, если нам нужно найти число с какой-то точностью, то почти всегда лучше это делать итерационно

Пример:

Нам нужно найти корень функции с заданной точностью

Не правильный выбор:

```
while((r – l) < eps)
```

Правильный выбор:

```
for (int i = 0; i < 100; i++)
```

Полезные ссылки

- goo.gl/KKdq1i – представление вещественных чисел в памяти компьютера