Технология и методы программирования

Лекция 7. Алгоритмы. Сложность.

Повторение — мать ~~заикания~~ учения

- Алгоритм — набор инструкций, описывающих порядок действий исполнителя для достижения результата решения задачи за конечное число действий

Алгоритмы решают **вычислительные задачи**. Если алгоритм работает для любых входных данных, его называют **корректным**.

Конкретный набор входных данных называют **экземпляром задачи**.

Алгоритм называют **корректным** если он для любого **экземпляра задачи** генерирует **корректные** выходные данные.

Анализ алгоритмов

Анализ алгоритмов нужен для того, чтобы предсказать, какие ресурсы требуются для работы алгоритма.

Анализ алгоритмов

Анализ алгоритмов нужен для того, чтобы предсказать, какие ресурсы требуются для работы алгоритма. Например

- Время работы — число элементарных шагов, которые нужно алгоритму, чтобы завершиться.
- Дополнительная память — сколько дополнительных ресурсов нужно выделить?
- Простота реализации

Сортировка вставками

```
void
insert_sort(int *arr, size_t n)
{
        int i, j;
        for (i = 1; i < n; i++) {
                int key = arr[i];
                j = i - 1;
                while (j >= 0 && arr[j] > key) {
                        arr[j + 1] = arr[j];
                        j--;
                }
                arr[j + 1] = key;
        }
}
```

Сортировка вставками

6 5 3 1 8 7 2 4

```c
void
insert_sort(int *arr, size_t n)
{
        int i, j;
        for (i = 1; i < n; i++) {
                int key = arr[i];
                j = i - 1;
                while (j >= 0 && arr[j] > key) {
                        arr[j + 1] = arr[j];
                        j--;
                }
                arr[j + 1] = key;
        }
}
```

Стоимость	Кол-во раз
C_1	n
C_2	$n-1$
C_3	$n-1$
C_4	$\sum_{j=1}^{n-1}(t_j+1)$
C_5, C_6	$\sum_{j=1}^{n-1}(t_j)$
C_7	$n-1$

$$T(n)=C_1 n+(C_2+C_3+C_7)(n-1)+C_4\sum_{j=1}^{n-1}(t_j+1)+(C_5+C_6)\sum_{j=1}^{n-1}(t_j)$$

```c
void
insert_sort(int *arr, size_t n)
{
        int i, j;
        for (i = 1; i < n; i++) {
                int key = arr[i];
                j = i - 1;
                while (j >= 0 && arr[j] > key) {
                        arr[j + 1] = arr[j];
                        j--;
                }
                arr[j + 1] = key;
        }
}
```

Стоимость	Кол-во раз
C_1	n
C_2	$n-1$
C_3	$n-1$
C_4	$\sum_{j=1}^{n-1}(t_j+1)$
C_5, C_6	$\sum_{j=1}^{n-1}(t_j)$
C_7	$n-1$

Общий случай $$T(n)=Cn+3C(n-1)+C\sum_{j=1}^{n-1}(t_j+1)+2C\sum_{j=1}^{n-1}(t_j)$$

arr отсортирован $$T(n)=Cn+3C(n-1)+C(n-1)$$ Линейная функция

arr отсортирован наоборот $$T(n)=Cn+3C(n-1)+C\left(\frac{n(n+1)}{2}-1\right)+2C\left(\frac{n(n-1)}{2}\right)$$ Квадратичная функция

Время работы. Временная сложность

Время работы в худшем и в лучшем случае могут значительно различаться.

Обычно пользователя интересует время работы в худшем случае, потому что:

- С его помощью можно спрогнозировать когда алгоритм завершится.
- На практике «плохие» входные данные попадаются довольно часто.
- Время работы в среднем может быть довольно близко к времени работы в худшем случае.

Временная сложность алгоритма (в худшем случае) — это функция от размера входных данных, равная максимальному количеству элементарных операций, проделываемых алгоритмом для решения экземпляра задачи указанного размера.

Порядок роста

Для «длинных» входных данных больший вклад в функцию времени работы вносит слагаемое самого высокого порядка.

Скорость роста (или порядок роста) — учитывает слагаемое самого высокого порядка и не учитывает константные множители (коэффициенты).

Скорость роста обозначается символом **O**.

Например в случае с сортировкой вставками функция времени работы имеет формат

$T(n)=an^2+bn+c$ Время работы алгоритма в худшем случае равно $O(n^2)$

(читается O от n в квадрате)

Порядок роста. Примеры

- Найти время работы функции поиска значения в массиве

```
int
arr_search_unsorted(int *arr, int sz, int n)
{
        for (i = 0; i < sz; i++) {
                if (arr[i] == n)
                        return i;
        }
        return -1;
}
```

- Найти время работы функции поиска значения в отсортированном массиве

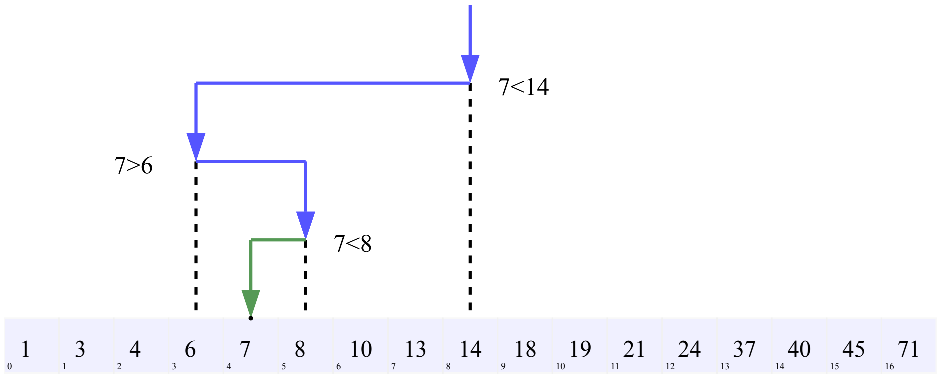
Двоичный поиск

```c
int
binsearch(int *arr, int sz, int n)
{
	int l = 0;
	int r = sz - 1;

	while (l <= r) {
		int mid = (l + r)  / 2;
		int mid_val = arr[mid];

		if (n == mid_val) {
			return mid;
		}

		if (n > mid_val)
			l = mid + 1;
		else
			r = mid - 1;
	}
	return -1;
}
```



Метод «разделяй и властвуй»

Рекурсивный метод решения задач

Метод состоит из трёх шагов:

- **Разделение** задачи на несколько подзадач, которые представляют собой меньшие экземпляры той же задачи.
- **Властвование** над подзадачами путём их рекурсивного решения. Если задача достаточно мала, она может решаться непосредственно.
- **Комбинирование** решений подзадач в решение исходной задачи.

Сортировка слиянием

- **Разделение:** Делим входной массив из **n** элементов на 2 последовательности из **n/2** элементов.
- **Властвование:** Рекурсивно сортируем эти 2 последовательности сортировкой слиянием
- **Комбинирование:** Соединяем две отсортированные последовательности

Рекурсивный запуск сортировки заканчивается, когда длина сортируемой последовательности становится равна 1 (а это уже отсортированный массив).

Сортировка слиянием. Псевдокод

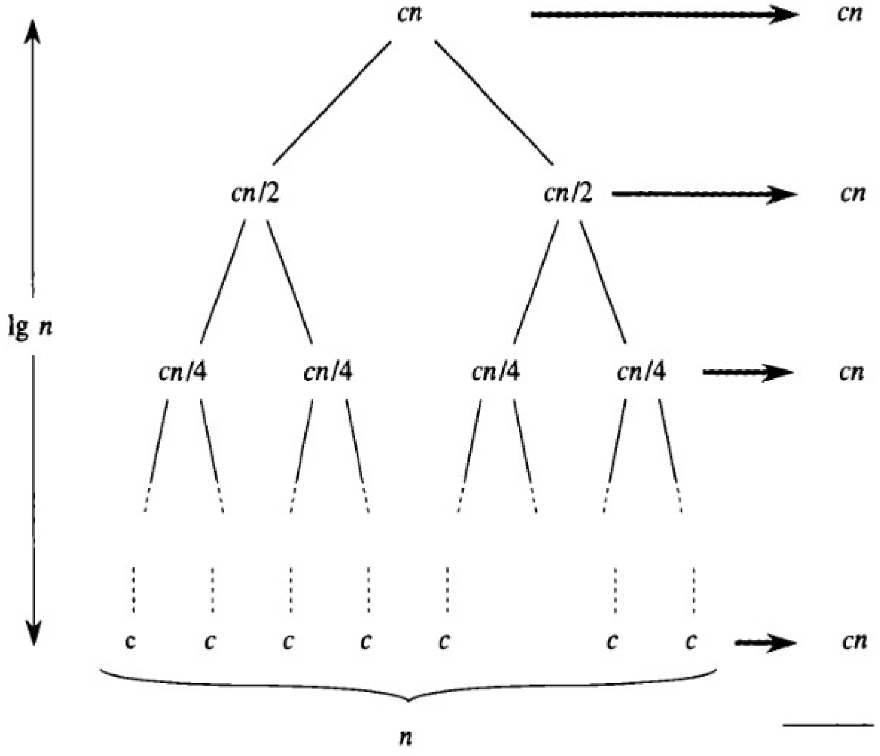
```python
def merge(a, b):
    out = []
    while True:
        if len(a) == 0:
            out.extend(b)
            break
        if len(b) == 0:
            out.extend(a)
            break
        if a[0] < b[0]:
            out.append(a.pop(0))
        else:
            out.append(b.pop(0))
    return out

def merge_sort(arr):
    l = len(arr)
    if l <= 1:
        return arr
    l2 = l / 2
    a1 = merge_sort(arr[:l2])
    a2 = merge_sort(arr[l2:])
    return merge(a1, a2)
```

Временная сложность функции merge: O(n)

Сортировка слиянием. Анализ

Сортировка слиянием. Анализ



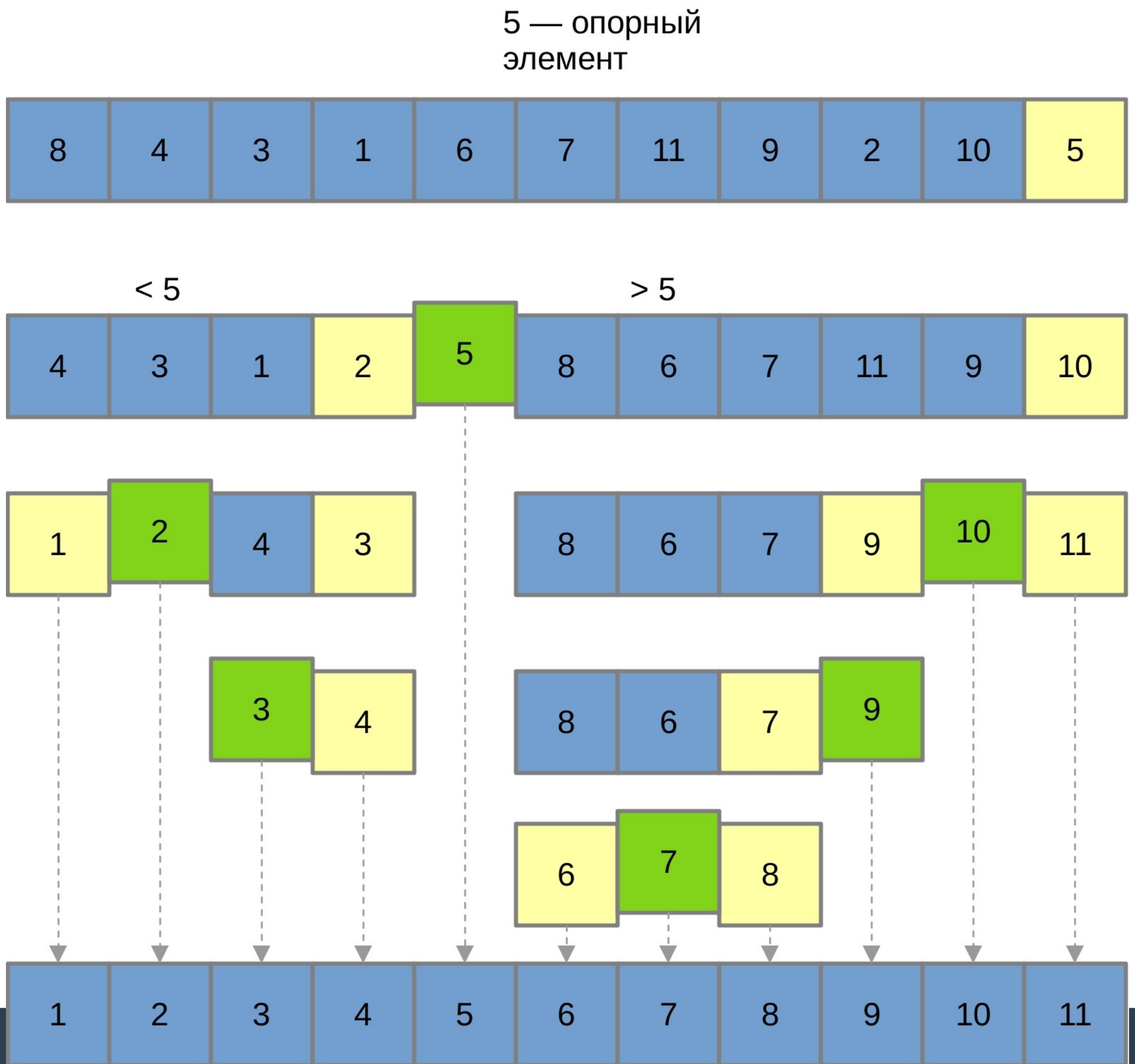
Всего:
Cn log(n) + cn

Быстрая сортировка. Идея

Основана на подходе «разделяй и властвуй»

- **Разделение:** Выбираем случайный элемент массива q. Делим массив на 3 «кучки»:
  - элементы меньше q (A)
  - элементы равные q (B)
  - элементы больше q (C)
- **Властвование:** Рекурсивно вызываем сортировку для кучек (A) и (C)
- **Комбинирование:**

Быстрая сортировка



Временная сложность:

- В среднем
  $O(n\log(n))$
- В худшем случае
  $O(n^2)$

Чего почитать

- Кормен, Т. «Алгоритмы. Построение и анализ» 3 редакция. Главы 1- 4