28.10.2024

Стек. Сортировки. Реальные примеры для сортировок.

Филиппов Михаил Витальевич

m.filippov@g.nsu.ru

89232283872

Императивное программирование, 2024-2025

Новосибирский государственный университет

*НАСТОЯЩАЯ НАУКА

Давайте познакомимся

Филиппов Михаил Витальевич

- Окончил магистратуру ФФ НГУ
- Окончил аспирантуру ИТ СО РАН
- Являюсь м.н.с. ИТ СО РАН
- 7+ лет опыт в программировании C/C++

Структура лекции

Стек	Сортировки	Реальный пример для сортировки
30 минут	50 минут	10 минут

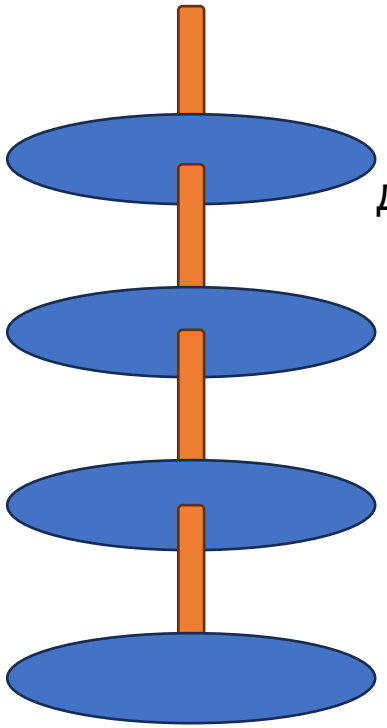
Структура лекции

Стек	Сортировки	Реальный пример для сортировки
30 минут	50 минут	10 минут

Что за стек?

Стек — важная структура данных, которая хранит свои элементы в упорядоченном порядке. Стек — это линейная структура данных, которая использует принцип, где элементы добавляются и удаляются только с одного конца, который называется TOP. Следовательно, стек называется структурой данных LIFO (Last-In-First-Out), поскольку элемент, который был вставлен последним, является первым, который будет извлечен.

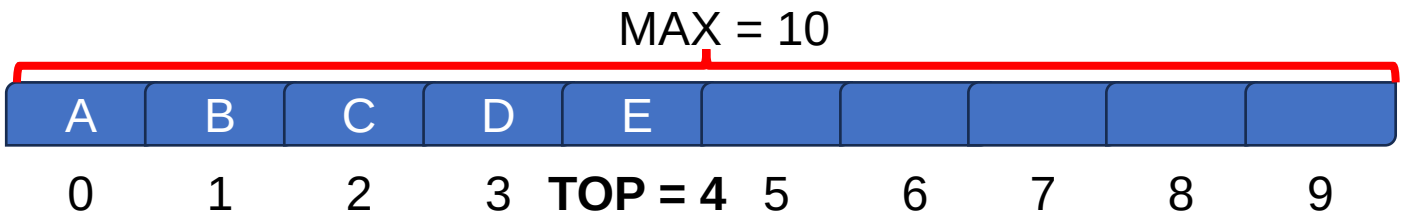
Самая верхняя пластина будет удалена первой



Другая пластина будет добавлена поверх этой пластины

Представление массива стека

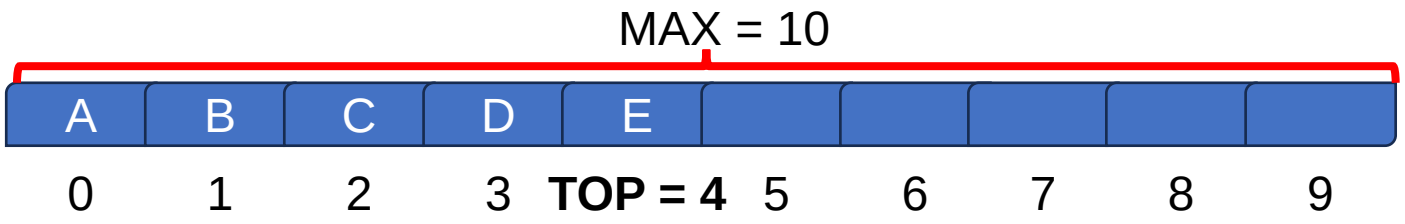
В памяти компьютера стеки можно представить как линейный массив. С каждым стеком связана переменная TOP, которая используется для хранения адреса самого верхнего элемента стека. Это позиция, в которую элемент будет добавлен или удален. Есть еще одна переменная MAX, которая используется для хранения максимального количества элементов, которые может содержать стек. Если TOP = NULL, то это означает, что стек пуст, а если TOP = MAX–1, то стек полон.




```
int st[MAX], top = -1;
```

Представление массива стека

В памяти компьютера стеки можно представить как линейный массив. С каждым стеком связана переменная TOP, которая используется для хранения адреса самого верхнего элемента стека. Это позиция, в которую элемент будет добавлен или удален. Есть еще одна переменная MAX, которая используется для хранения максимального количества элементов, которые может содержать стек. Если TOP = NULL, то это означает, что стек пуст, а если TOP = MAX–1, то стек полон.



Стек поддерживает три основные операции: push, pop и peek. Операция push добавляет элемент на вершину стека, а операция pop удаляет элемент с вершины стека. Операция peek возвращает значение самого верхнего элемента стека.

Операция push

Операция push используется для вставки элемента в стек. Новый элемент добавляется в самую верхнюю позицию стека. Однако перед вставкой значения мы должны сначала проверить, TOP=MAX–1, поскольку в этом случае стек заполнен и больше вставок быть не может. Если делается попытка вставить значение в стек, который уже заполнен, выводится сообщение OVERFLOW.

```
void push(int st[], int val)
{
    if (top == MAX - 1)
    {
        printf("\n STACK OVERFLOW");
    }
    else
    {
        top++;
        st[top] = val;
    }
}
```

Операция poop

Операция извлечения используется для удаления самого верхнего элемента из стека. Однако перед удалением значения мы должны сначала проверить, TOP=NULL, поскольку в этом случае стек пуст и дальнейшие удаления невозможны. Если попытаться удалить значение из стека, который уже пуст, выводится сообщение UNDERFLOW.

```
int pop(int st[])
{
    int val;
    if (top == -1)
    {
        printf("\n STACK UNDERFLOW");
        return -1;
    }
    else
    {
        val = st[top];
        top--;
        return val;
    }
}
```

Операция peek

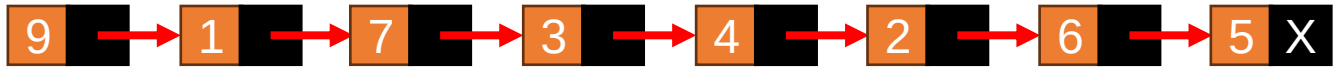
Peek — это операция, которая возвращает значение самого верхнего элемента стека, не удаляя его из стека. Однако операция Peek сначала проверяет, пуст ли стек, т. е. если TOP = NULL, то выводится соответствующее сообщение, в противном случае возвращается значение.

```
int peek(int st[])
{
    if (top == -1)
    {
        printf("\n STACK IS EMPTY");
        return -1;
    }
    else
        return (st[top]);
}
```

Связанное представление стека

Если размер массива не может быть определен заранее, то используется другая альтернатива, т. е. связанное представление. Требование к памяти связанного представления стека с n элементами составляет O(n), а типичное требование по времени для операций составляет O(1).

В связанном стеке каждый узел состоит из двух частей: одна, которая хранит данные, и другая, которая хранит адрес следующего узла. Указатель START связанного списка используется как TOP. Все вставки и удаления выполняются в узле, на который указывает TOP. Если TOP = NULL, то это означает, что стек пуст.



TOP

```
struct stack
{
    int data;
    struct stack *next;
};
struct stack *top = NULL;
```

Операция push

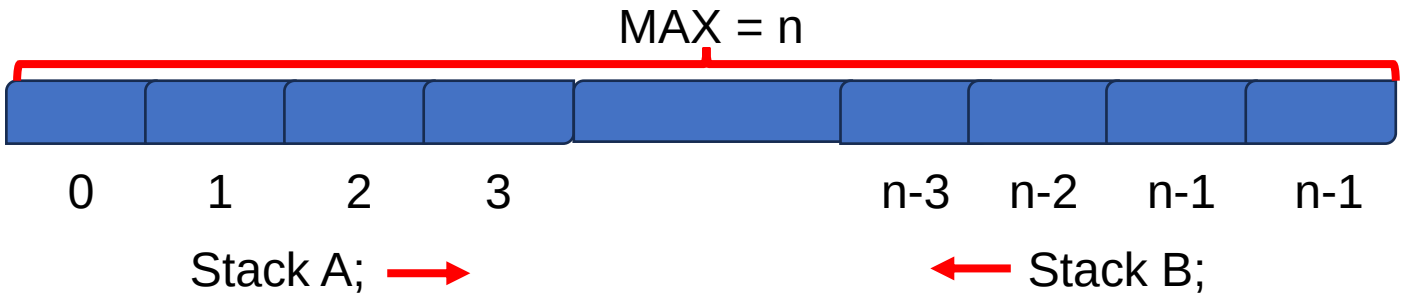
```
struct stack *push(struct stack *top, int val)
{
    struct stack *ptr;
    ptr = (struct stack *)malloc(sizeof(struct stack));
    ptr->data = val;
    if (top == NULL)
    {
        ptr->next = NULL;
        top = ptr;
    }
    else
    {
        ptr->next = top;
        top = ptr;
    }
    return top;
}
```

Операция pop

```
struct stack *pop(struct stack *top)
{
    struct stack *ptr;
    ptr = top;
    if (top == NULL)
        printf("\n STACK UNDERFLOW");
    else
    {
        top = top->next;
        printf("\n The value being deleted is: %d", ptr->data);
        free(ptr);
    }
    return top;
}
```

Multiple stacks

Существует компромисс между частотой переполнений и выделенным пространством. Поэтому лучшим решением для решения этой проблемы является наличие нескольких стеков или наличие более одного стека в одном массиве достаточного размера.



Применения стека

Мы рассмотрим следующие применения:

- Переворачивание списка
- Проверка скобок
- Преобразование инфиксного выражения в постфиксное выражение
- Оценка постфиксного выражения

Переворачивание списка

```
int main()
{
    int val, n, i,
        arr[10];
    printf("\n Enter the number of elements in the array : ");
    scanf("%d", &n);
    printf("\n Enter the elements of the array : ");
    for (i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    for (i = 0; i < n; i++)
        push(arr[i]);
    for (i = 0; i < n; i++)
    {
        val = pop();
        arr[i] = val;
    }
    printf("\n The reversed array is : ");
    for (i = 0; i < n; i++)
        printf("\n %d", arr[i]);
    return 0;
}
```

Проверка скобок

```
for (i = 0; i < strlen(exp); i++) {
    if (exp[i] == '(' || exp[i] == '{' || exp[i] == '[')
        push(exp[i]);
    if (exp[i] == ')' || exp[i] == '}' || exp[i] == ']')
        if (top == -1)
            flag = 0;
        else {
            temp = pop();
            if (exp[i] == ')' && (temp == '{' || temp == '['))
                flag = 0;
            if (exp[i] == '}' && (temp == '(' || temp == '['))
                flag = 0;
            if (exp[i] == ']' && (temp == '(' || temp == '{'))
                flag = 0;
        }
}
if (top >= 0)
    flag = 0;
if (flag == 1)
    printf("\n Valid expression");
else
    printf("\n Invalid expression");
```

Stacks может использоваться для проверки правильности скобок в любом алгебраическом выражении. Например, алгебраическое выражение является правильным, если для каждой открывающей скобки существует соответствующая закрывающая скобка. Например, выражение (A+B} является неправильным, но выражение {A + (B – C)} является правильным.

Польские записи

Инфиксная, постфиксная и префиксная записи — это три различных, но эквивалентных записи алгебраических выражений.

Инфиксная запись:

A+B

(A + B) * C

Преобразование из инфиксной в постфиксную запись:

(a) (A–B) * (C+D)

[AB–] * [CD+]

AB–CD+*

(b) (A + B) / (C + D) – (D * E)

[AB+] / [CD+] – [DE*]

[AB+CD+/] – [DE*]

AB+CD+/DE*–

Преобразование из инфиксной в префиксную запись:

(a) (A + B) * C

(+AB)*C

*+ABC

(b) (A–B) * (C+D)

[–AB] * [+CD]

*–AB+CD

Инфиксное -> постфиксное

Алгоритм преобразования инфиксной записи в постфиксную
Шаг 1: Добавить) в конец инфиксного выражения
Шаг 2: Поместить (в стек
Шаг 3: Повторять, пока не будет просмотрен каждый символ в инфиксной записи
ЕСЛИ встречается (, поместить его в стек
ЕСЛИ встречается операнд (цифра или символ), добавить его в постфиксное выражение.
ЕСЛИ встречается а), то
a. Многократно извлечь из стека и добавить его в постфиксное выражение, пока не встретится (.
b. Отбросить (. То есть удалить (из стека и не добавлять его в постфиксное выражение
ЕСЛИ встречается оператор, то
a. Многократно извлечь из стека и добавить каждый оператор (извлеченный из стека) в постфиксное выражение, которое имеет тот же приоритет или более высокий приоритет, чем
b. Поместить оператор в стек
[КОНЕЦ IF]
Шаг 4: Многократно извлечь из стека и добавить его в постфиксное выражение, пока стек не опустеет
Шаг 5: ВЫХОД

Пример

A – (B / C + (D % E * F) / G)* H
A – (B / C + (D % E * F) / G)* H)

Инфиксный символ отсканирован	Стек	Постфиксное выражение
	(	
A	(	A
-	(-	A
(	(- (	A
B	(- (	A B
/	(- (/	A B C
C	(- (/	A B C /
+	(- (+	A B C /
(	(- (+ (	A B C / D
D	(- (+ (	A B C / D
%	(- (+ (%	A B C / D E
E	(- (+ (%	A B C / D E
*	(- (+ (% *	A B C / D E F
F	(- (+ (% *	A B C / D E F * %
)	(- (+	A B C / D E F * %
/	(- (+ /	A B C / D E F * % G
G	(- (+ /	A B C / D E F * % G
)	(-	A B C / D E F * % G / +
*	(- *	A B C / D E F * % G / +
H	(- *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * -

Оценка постфиксного выражения

Шаг 1: Добавьте ")" в конец постфиксного выражения
Шаг 2: Просканируйте каждый символ постфиксного выражения и повторите Шаги 3 и 4, пока не встретится ")"
Шаг 3:

ЕСЛИ встретится операнд, поместите его в стек
ЕСЛИ встретится оператор O, то

a. Извлеките два верхних элемента из стека как A и B как A и B
b. Вычислите B O A, где A — самый верхний элемент, а B — элемент под A.
c. Поместите результат вычисления в стек

[END OF IF]

Шаг 4: УСТАНОВИТЕ РЕЗУЛЬТАТ равным самому верхнему элементу стека
Шаг 5: ВЫХОД

Инфиксная форма:
9 – ((3 * 4) + 8) / 4
Постфиксная форма:
9 3 4 * 8 + 4 /

Персонаж отсканирован	Стек
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
-	4

Структура лекции

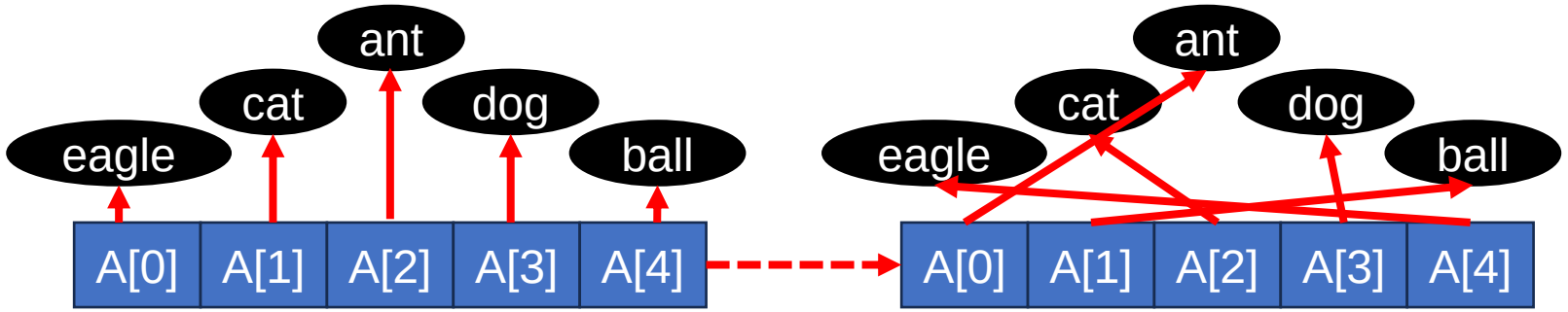
Стек	Сортировки	Реальный пример для сортировки
30 минут	50 минут	10 минут

Терминология

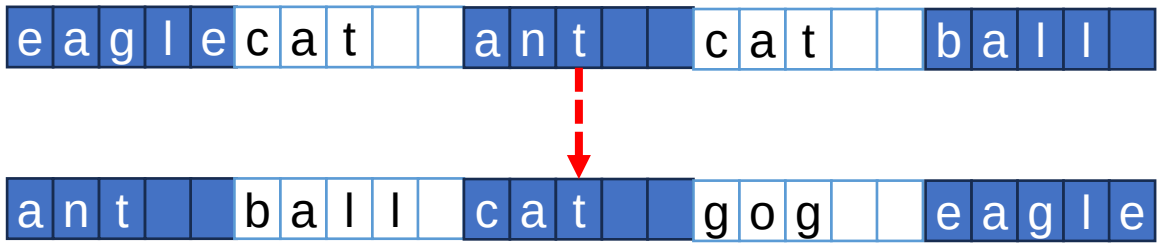
Коллекция A элементов, которые можно сравнивать один с другим, сортируется на месте (без привлечения дополнительной памяти). Мы используем обозначения A[i] и a_i для обозначения i-го элемента коллекции. По соглашению первым элементом коллекции является A[0]. Мы используем запись A[low, low+n) для обозначения подколлекции A[low]...A[low+n-1] из n элементов.
Для сортировки коллекции необходимо реорганизовать элементы A так, что если A[i] <A[j], то i<j. Если в коллекции имеются повторяющиеся элементы, то они должны быть смежными в результирующей упорядоченной коллекции; т.е. если в отсортированной коллекции A[i] =A[j], то не может быть такого k, что i<k<j и A[i] ≠ A[k]. Наконец отсортированная коллекция A должна представлять собой перестановку элементов, которые первоначально образовывали коллекцию A.

Представление

Коллекция может храниться в оперативной памяти компьютера, но может находиться и в файле в файловой системе, известной как вторичная память. Коллекция может быть заархивирована в третичной памяти (например, ленточные библиотеки или оптические диски), что может потребовать дополнительного времени только для обнаружения информации; кроме того, прежде чем она сможет быть обработана, возможно, потребуется скопировать такую информацию во вторичную память (например, на жесткий диск).
Информация, хранящаяся в оперативной памяти, обычно принимает одну из двух форм: на основе указателей или на основе значений.



Сортировка с использованием хранения указателей



Сортировка с использованием хранения на основе значений

Сравниваемость элементов

Элементы в наборе данных сравнений должны обеспечивать нестрогое упорядочение. То есть для любых двух элементов p и q из набора данных должен выполняться ровно один из следующих трех предикатов: p=q>, p<q или p>q.

Распространенные сортируемые типы-примитивы включают целые числа, числа с плавающей точкой и символы. При сортировке составных элементов на каждую отдельную часть составного элемента накладывается лексикографическое упорядочивание, сводя, таким образом, сложную сортировку к отдельным сортировкам примитивных типов.

Вопрос упорядочения далеко не так прост при рассмотрении регистров символов (“A” больше или меньше “a”?), диакритических знаков (как соотносятся “a” и “ä”?) и дифтонгов (то же для “a” и “æ”?).

Будем предполагать, что вы можете предоставить компаратор — функцию сравнения cmp, которая сравнивает элементы p и q и возвращает 0, если p=q, отрицательное число, если p<q, и положительное число при p>q.

Устойчивая сортировка

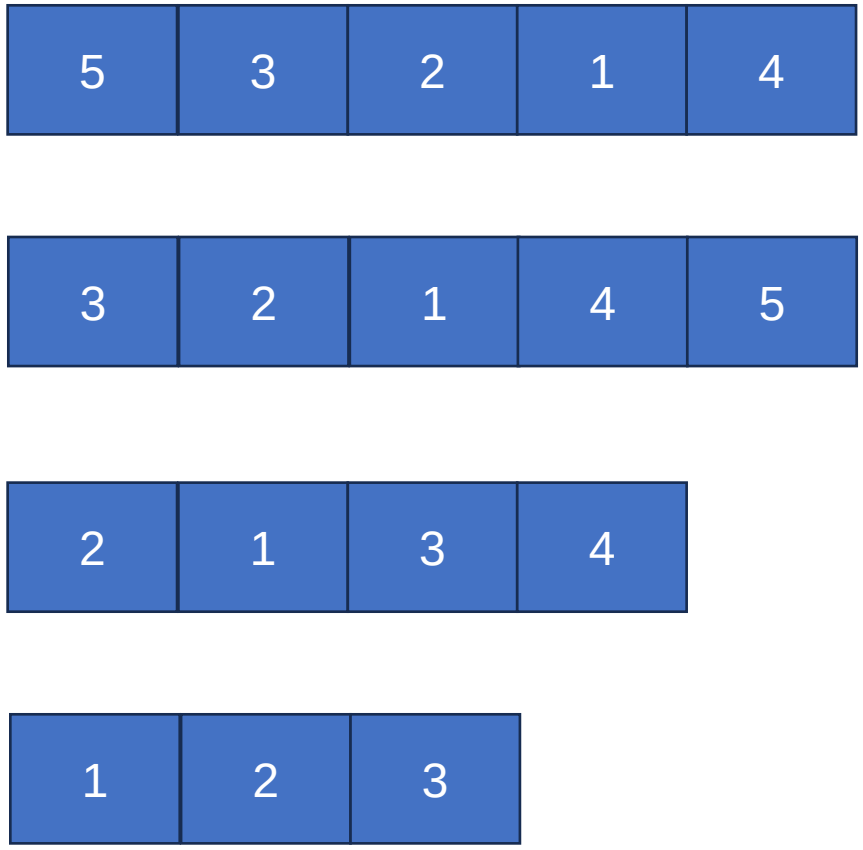
Когда функция сравнения cmp определяет, что два элемента исходной неупорядоченной коллекции ai и aj равны, может оказаться важным сохранить их относительный порядок в отсортированном множестве. Алгоритмы сортировки, которые гарантируют выполнение этого свойства, называются устойчивыми.

Город назначения	Компания	Рейс	Время вылета (в порядке возрастания)	Город назначения	Компания	Рейс	Время вылета (в порядке возрастания)
Buffalo	Air Trans	549	10:42	Atlanta	Southwest	482	13:20
Atlanta	Delta	1097	11:00	Atlanta	Delta	1097	11:00
Baltimore	Southwest	836	11:05	Atlanta	Air Trans	872	11:15
Atlanta	Air Trans	872	11:15	Atlanta	Delta	28	12:00
Atlanta	Delta	28	12:00	Atlanta	Al Italia	3429	13:50
Boston	Delta	1056	12:05	Austin	Southwest	1045	13:05
Baltimore	Southwest	216	12:20	Baltimore	Southwest	836	11:05
Austin	Southwest	1045	13:05	Baltimore	Southwest	216	12:20
Albany	Southwest	482	13:20	Baltimore	Southwest	272	13:40
Boston	Air Trans	515	13:21	Boston	Delta	1056	12:05
Baltimore	Southwest	272	13:40	Boston	Air Trans	515	13:21
Atlanta	Al Italia	3429	13:50	Boston	Air Trans	549	10:42

Критерии выбора алгоритма сортировки

Критерий	Алгоритм сортировки
Только несколько элементов	Сортировка вставками
Элементы уже почти отсортированы	Сортировка вставками
Важна производительность в наихудшем случае	Пирамидальная сортировка
Важна производительность в среднем случае	Быстрая сортировка
Элементы с равномерным распределением	Блочная сортировка
Как можно меньший код	Сортировка вставками
Требуется устойчивость сортировки	Сортировка слиянием

Сортировки перестановкой: пузырьковая



Наилучший случай = N – 1;
Наихудший случай = арифметическая прогрессия = $N - 1 + ..+ 1 = \frac{(1+N-1)N}{2} = \frac{N^2}{2} = O(N^2)$

Сортировки перестановкой: пузырьковая

```c
void sortBubble(void **ar, int n, int (*cmp)(const void *, const void *))
{
    _Bool swappedElements = 1;
    while (swappedElements)
    {
        n = n - 1;
        swappedElements = 0;
        for (int i = 1; i < n; i++)
        {
            if (cmp(ar[i], ar[i + 1]) > 0)
            {
                void *value = ar[i];
                ar[i] = ar[i + 1];
                ar[i + 1] = value;
                swappedElements = 1;
            }
        }
    }
}
```

Сортировки перестановкой: пузырьковая

Средний случай:
C(i) число сравнений, выполненных на первых i проходах
Среднее число сравнений:

$$A(N) = \frac{1}{N-1}\sum_{i=1}^{N-1} C(i)$$

$$C(i) = \sum_{j=N-1}^{i} j = \frac{i+N-1}{2}(N-i) = \frac{N^2-N-i^2+i}{2}$$

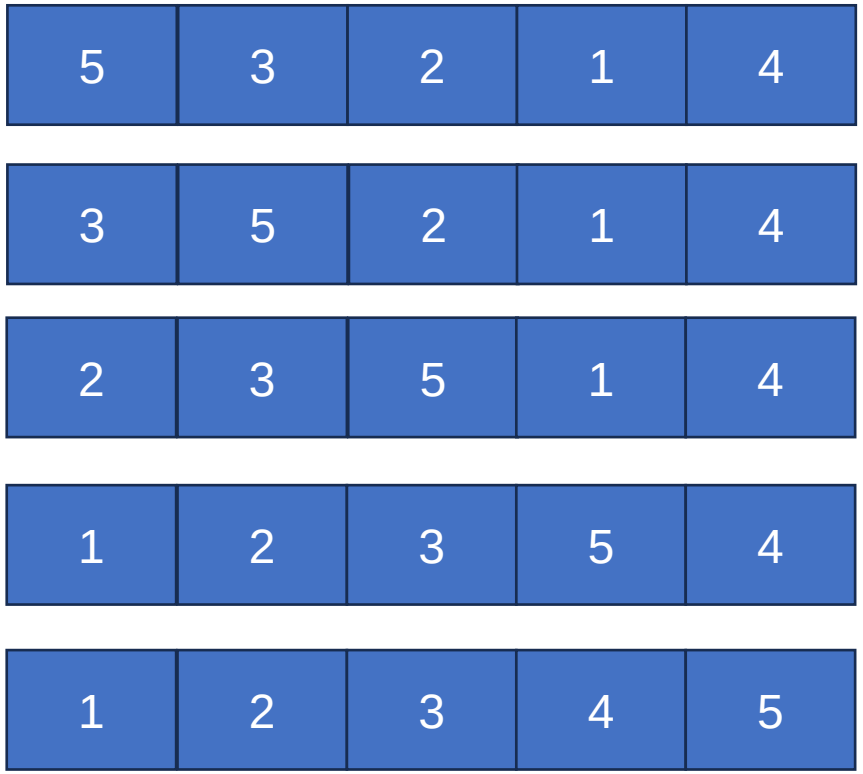
$$A(N) = \frac{1}{N-1}\sum_{i=1}^{N-1}\frac{N^2-N-i^2+i}{2} = \frac{N^2-N}{2} - \frac{1}{2}\frac{1}{N-1}\sum_{i=1}^{N-1} i^2 + \frac{1}{2}\frac{1}{N-1}\frac{1+N-1}{2}N - 1 =$$

$$= \frac{N^2-N}{2} + \frac{N}{4} - \frac{1}{2}\frac{1}{N-1}\frac{(N-1)N(2N-1)}{6} = \frac{(6-2)N^2+(-6+3+1)N}{12} =$$

$$= \frac{1}{3}N^2\left(1-\frac{1}{2N}\right) = O(N^2)$$

Сортировки перестановкой: вставками

Основная идея сортировки вставками состоит в том, что при добавлении нового элемента в уже отсортированный список его стоит сразу вставлять в нужное место вместо того, чтобы вставлять его в произвольное место, а затем заново сортировать весь список.



Наилучший случай = N – 1;
Наихудший случай = арифметическая прогрессия = $N - 1 + .. + 1 = \frac{(1+N-1)N}{2} = \frac{N^2}{2} = O(N^2)$

Сортировки перестановкой: вставками

```c
void sortPointers(void **ar, int n, int (*cmp)(const void *, const void *))
{
    int j;
    for (j = 1; j < n; j++)
    {
        int i = j - 1;
        void *value = ar[j];
        while (i >= 0 && cmp(ar[i], value) > 0)
        {
            ar[i + 1] = ar[i];
            i--;
        }
        ar[i + 1] = value;
    }
}
```

Сортировки перестановкой: вставками

Средний случай:

Сколько существует возможных положений для i-ого элемента? ***i + 1***

A(i) среднее число сравнений для вставки i-го элемента

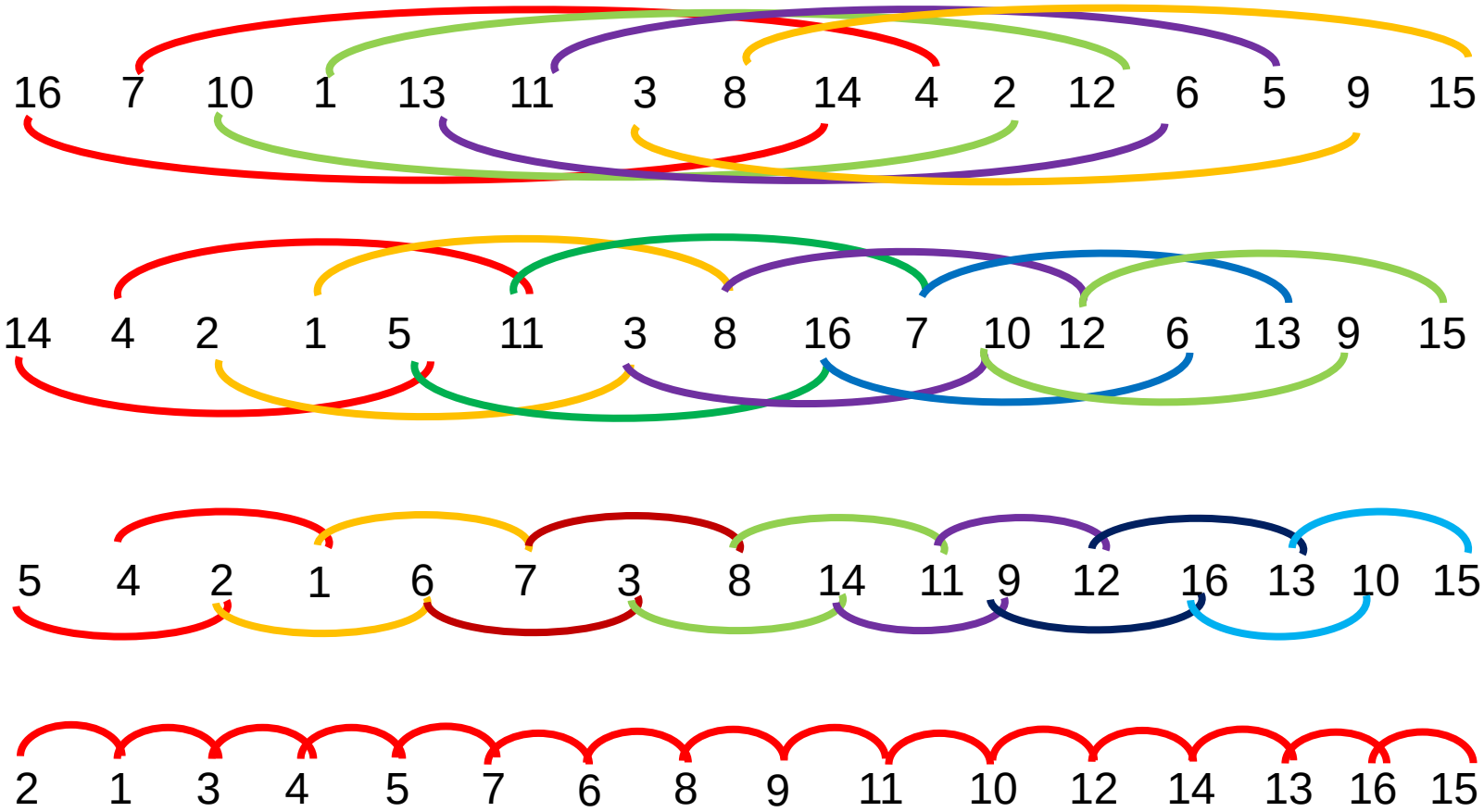
Среднее число сравнений:

$$A(i) = \frac{1}{i+1}\sum_{p=1}^{i} p + i = \frac{1}{i+1}\left(\frac{i(i+1)}{2} + i\right) = \frac{i}{2} + 1 - \frac{1}{i+1}$$

$$A(N) = \sum_{i=1}^{N-1} A(i) = \sum_{i=1}^{N-1}\left(\frac{i}{2} + 1 - \frac{1}{i+1}\right) = \frac{1}{2}\frac{N(N-1)}{2} + N - 1 - \sum_{i=1}^{N}\frac{1}{i} + 1 =$$

$$= \frac{N^2+3N}{4} - \ln(N) = \frac{N^2}{4}\left(1 + \frac{3}{4N} - \frac{\ln(N)}{4N^2}\right) = O(N^2)$$

Сортировка Шелла



Сортировка Шелла

```
void sortShell(void **ar, int n, int (*cmp)(const void *, const void *))
{
    int j;
    int passes = log(n) / log(2);
    while (passes >= 1)
    {
        int increment = pow(2, passes) - 1;
        for (int i = 1; i < increment; i++)
        {
            InsertionSort(*ar, n, i, increment);
        }
        passes--;
    }
}
```

Было доказано, что сложность этого алгоритма в наихудшем случае при выбранных нами значениях шага равна $O(N^{3/2})$

Корневая сортировка

Исходный список

310 213 023 130 013 301 222 032 201 111 323 002 330 102 231 120

Номер стопки	Содержимое			
0	310	130	330	120
1	301	201	111	231
2	222	032	002	102
3	213	023	013	323

(а) Первый проход, разряд единиц

Список, полученный в результате первого прохода

310 130 330 120 301 201 111 231 222 032 002 102 213 023 013 323

Номер стопки	Содержимое			
0	301	201	002	102
1	310	111	213	013
2	120	222	023	323
3	130	330	231	032

(б) Второй проход, разряд десятков

Список, полученный в результате второго прохода

301 201 002 102 310 111 213 013 120 222 023 323 130 330 231 032

Номер стопки	Содержимое			
0	002	013	023	032
1	102	111	120	130
2	201	213	222	231
3	301	310	323	330

(в) Третий проход, разряд сотен

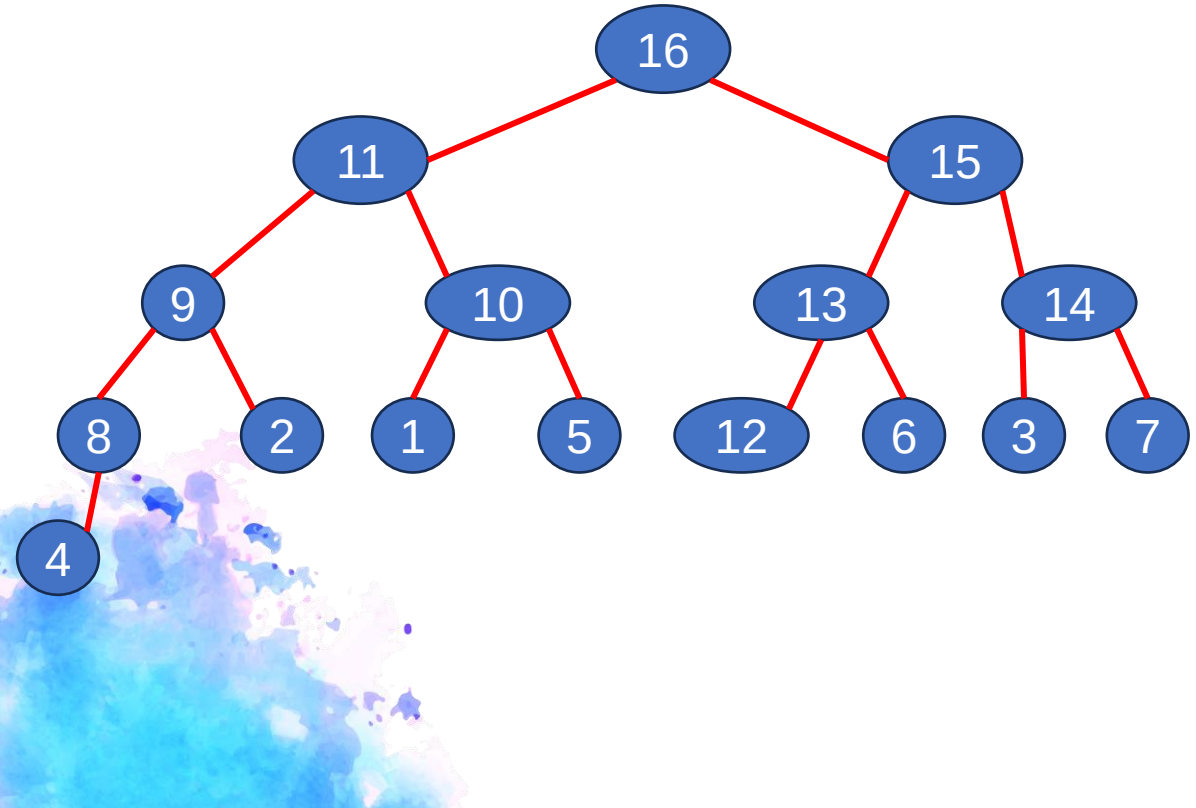
Корневая сортировка

```
void sortRadix(void **ar, int n, int (*cmp)(const void *, const void *))
{
    int shift = 1;
    for (int loop = 1; loop < keySize; loop++)
    { // keySize - количество ключей
        for (int entry = 1; entry < N; entry++)
        {
            int bucketNumber = (ar[entry].key / shift) % 10;
            Append(bucket[bucketNumber], ar[entry]);
        }
        ar = CombineBuckets();
        shift *= 10;
    }
}
```

+ Скорость O(NM), M – количество различных символов
- Много требует памяти

Пирамидальная сортировка

В основе пирамидальной сортировки лежит специальный тип бинарного дерева, называемый пирамидой



01	02	03	04	05	06	07	16	09	10	11	12	13	14	15	08
01	02	03	04	05	06	15	16	09	10	11	12	13	14	07	08
01	02	03	04	05	13	15	16	09	10	11	12	06	14	07	08
01	02	03	04	11	13	15	16	09	10	05	12	06	14	07	08
01	02	03	16	11	13	15	08	09	10	05	12	06	14	07	04
01	02	15	16	11	13	14	08	09	10	05	12	06	03	07	04
01	16	15	09	11	13	14	08	02	10	05	12	06	03	07	04
16	11	15	09	10	13	14	08	02	01	05	12	06	03	07	04

Пирамидальная сортировка (1/2)

```
static void heapify(void **ar, int (*cmp)(const void *, const void *), int idx, int max)
{
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;
    int largest;/* Поиск максимального элемента среди A[idx], A[left] и A[right]. */
    if (left < max && cmp(ar[left], ar[idx]) > 0) {
        largest = left;
    }
    else {
        largest = idx;
    }
    if (right < max && cmp(ar[right], ar[largest]) > 0) {
        largest = right;
    }/* Если наибольший элемент не является родительским,* выполняем обмен и переход далее. */
    if (largest != idx) {
        void *tmp;
        tmp = ar[idx];
        ar[idx] = ar[largest];
        ar[largest] = tmp;
        heapify(ar, cmp, largest, max);
    }
}
```

Пирамидальная сортировка (2/2)

```
static void buildHeap(void **ar, int (*cmp)(const void *, const void *), int n)
{
    int i;

    for (i = n / 2 - 1; i >= 0; i--)
    {
        heapify(ar, cmp, i, n);
    }
}
void sortPointers(void **ar, int n, int (*cmp)(const void *, const void *))
{
    int i;
    buildHeap(ar, cmp, n);
    for (i = n - 1; i >= 1; i--)
    {
        void *tmp;
        tmp = ar[0];
        ar[0] = ar[i];
        ar[i] = tmp;
        heapify(ar, cmp, 0, i);
    }
}
```

Пирамидальная сортировка

Анализ наихудшего случая

Этап построения:

1. Число сравнений пирамиды глубины D ≤ $2D^2$
2. Глубина пирамиды N элементов ≤ $C \log_2 N$

Число узлов на проходе W(N):

$$W(N) = \sum_{i=0}^{D-1} 2(D-i)2^i = \frac{2D(1-2^D)}{1-2} - 2\left((D-2)2^D + 2\right) = 2^{D+2} - 2D - 4$$

D = $\log_2 N$

$$W(N) = 4N - 2\log_2 N - 4 = O(N)$$

Этап основного цикла алгоритма:

Выполнение алгоритма при K узлах, по 2 сравнения на узел:

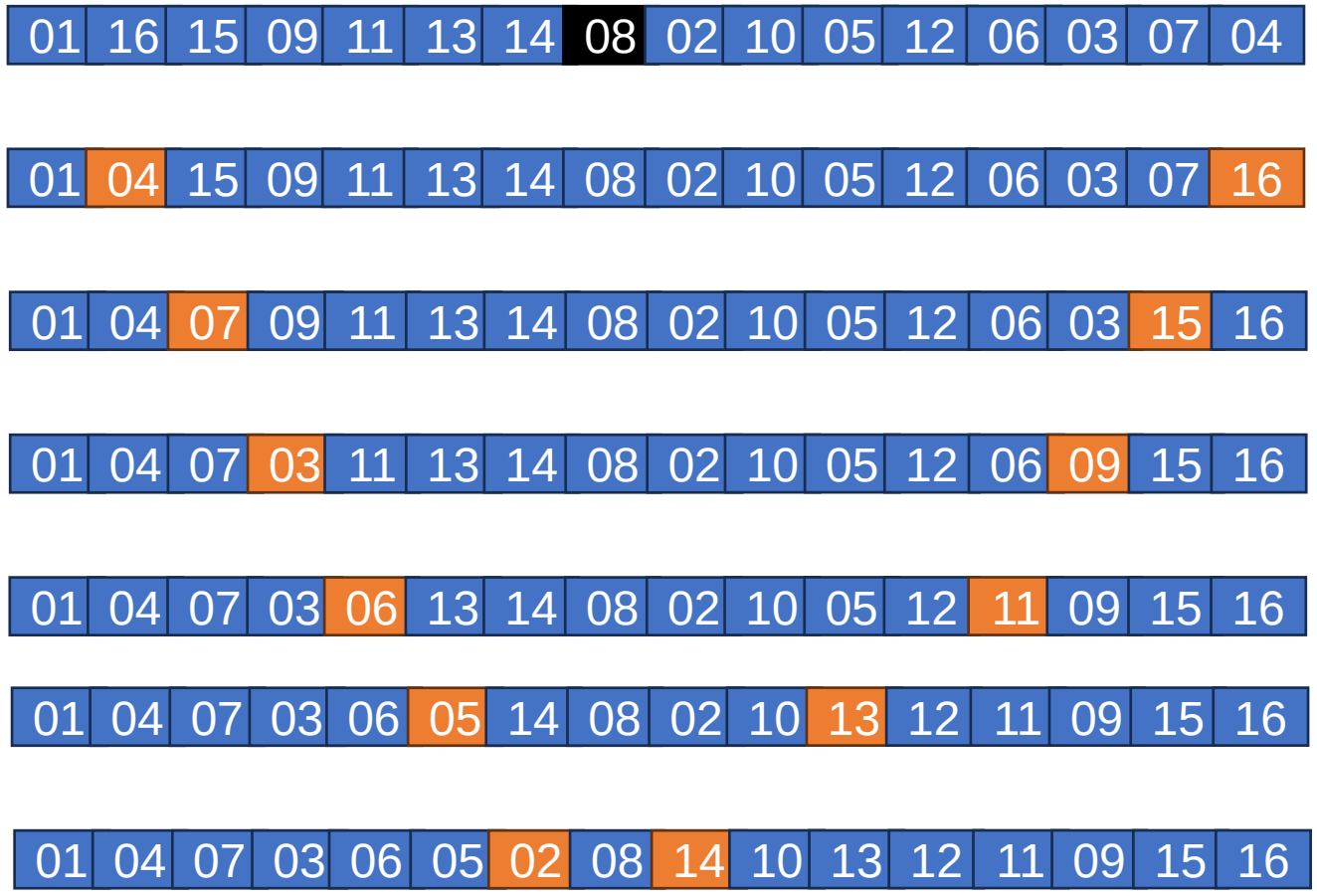
$$W_{loop} = 2\sum_{k=1}^{N-1} \log_2 K$$

Зная что узлов при глубине k ~ 2^k, получаем (d = $\log_2 N$):

$$W_{loop} = 2\sum_{k=1}^{d-1} k2^k + d\left(N - 2^d\right) = 2\left((d-2)2^d + 2\right) + d\left(N - 2^d\right) =$$

$$= 2dN - 2^{d+2} + 4 = 2N\log_2 N - 4N + 4 = O(N\log_2 N)$$

Быстрая сортировка



8 – опорный элемент

Проверка

<8 левая

>8 правая

Наихудший случай $\frac{(N-1)N}{2} = O(N^2)$

Быстрая сортировка

```
void do_qsort(void **ar, int (*cmp)(const void *, const void *), int left, int right)
{
    int pivotIndex;
    if (right <= left)
        return;
    /* Разбиение */
    pivotIndex = selectPivotIndex(ar, left, right);
    pivotIndex = partition(ar, cmp, left, right, pivotIndex);
    if (pivotIndex - 1 - left <= minSize)
        insertion(ar, cmp, left, pivotIndex - 1);
    else
        do_qsort(ar, cmp, left, pivotIndex - 1);
    if (right - pivotIndex - 1 <= minSize)
        insertion(ar, cmp, pivotIndex + 1, right);
    else
        do_qsort(ar, cmp, pivotIndex + 1, right);
}
/** Вызов быстрой сортировки */
void sortPointers(void **vals, int total_elems, int (*cmp)(const void *, const void *))
{
    do_qsort(vals, cmp, 0, total_elems - 1);
}
```

Быстрая сортировка

```
int partition(void **ar, int (*cmp)(const void *, const void *),
              int left, int right, int pivotIndex) {
    int idx,
        store;
    void *pivot = ar[pivotIndex]; /* Перемещаем pivot в конец массива */
    void *tmp = ar[right];
    ar[right] = ar[pivotIndex];
    ar[pivotIndex] = tmp; /* Все значения, не превышающие pivot, перемещаются в начало
     * массива, и pivot вставляется сразу после них. */
    store = left;
    for (idx = left; idx < right; idx++) {
        if (cmp(ar[idx], pivot) <= 0) {
            tmp = ar[idx];
            ar[idx] = ar[store];
            ar[store] = tmp;
            store++;
        }
    }
    tmp = ar[right];
    ar[right] = ar[store];
    ar[store] = tmp;
    return store;
}
```

Быстрая сортировка

Анализ среднего случая

1. Количество сравнений перед разбиением N - 1
2. Рекурсивный вызов функций, разбитых на P - 1 и N - P элементов

Число узлов на проходе A(N):

$$A(N) = N - 1 + \frac{1}{N}\sum_{i=1}^{N}(A(i-1) + A(N-i)) = N - 1 + \frac{2}{N}\sum_{i=1}^{N-1} A(i)\,;$$

$$A(1) = A(0) = 0$$

$$\mathrm{N}\cdot\ A(N) = (N-1)N + 2A(N-1) + 2\sum_{i=1}^{N-2} A(i)$$

$$\mathrm{N}\cdot \mathrm{A(N)} - (\mathrm{N}-1)\cdot \mathrm{A(N}-1) = 2A(N-1) + (N-1)N\ - (N-1)(N-2) =$$
$$= 2A(N-1) + 2N - 2$$

$$A(N) = \frac{(N+1)A(N-1) + 2N - 2}{N} \approx 1.4(N+1)\log_2 N = O(N\log_2 N)$$

Сортировка слиянием

```
void mergesort(void **copy_ar, void **ar, int
(*cmp)(const void *, const void *), int left, int
right)
{
    if (right - left < 2)
        return;
    if (right - left == 2)
    {
        if (copy_ar[left] > copy_ar[left + 1])
        {
            void *temp = copy_ar[left];
            copy_ar[left] = copy_ar[left + 1];
            copy_ar[left + 1] = temp;
        }
        return;
    }
    int mid = (right + left) / 2;
    mergesort_array(copy_ar, ar, cmp, left, mid);
    mergesort_array(copy_ar, ar, cmp, mid, right);
    // слияние левой и правой сторон Ar
    int i = left, j = mid, idx = left;
    while (idx < right)
    {
        if (j >= right || (i < mid && ar[i] < ar[j]))
        {
            copy_ar[idx] = ar[i];
            i += 1;
        }
        else
        {
            copy_ar[idx] = ar[j];
            j += i;
        }
        idx += 1
    }
}
```

Скорость во всех случаях

$W(N) = N\log_2 N - N + 1 = O(N\log_2 N)$.

Структура лекции

Стек	Сортировки	Реальный пример для сортировки
30 минут	50 минут	10 минут


Как наиболее эффективно
посчитать контрольную сумму
(летящие биты по Wi-Fi)

Пример

Самое простое, что приходит

```
uint64_t CountBitsSlow(uint64_t x) {  // O(n)
  uint64_t count = 0;
  while (x) {
    count += x & 1u;
    x >>= 1;
  }
  return count;
}
```

Пример

Теперь поинтереснее

```
uint64_t CountBitsFast(uint64_t x) {  // O(n_1)
  uint64_t count = 0;
  while (x) {
    ++count;
    x &= (x - 1);
  }
  return count;
}
```

Пример

Ну как же без треша

```
uint64_t CountBitsFaster(uint64_t x) {
  x = (x & 0x5555555555555555) + ((x >> 1) & 0x5555555555555555);
  x = (x & 0x3333333333333333) + ((x >> 2) & 0x3333333333333333);
  x = (x & 0x0f0f0f0f0f0f0f0f) + ((x >> 4) & 0x0f0f0f0f0f0f0f0f);
  x = (x & 0x00ff00ff00ff00ff) + ((x >> 8) & 0x00ff00ff00ff00ff);
  x = (x & 0x0000ffff0000ffff) + ((x >> 16) & 0x0000ffff0000ffff);
  x = (x & 0x00000000ffffffff) + ((x >> 32) & 0x00000000ffffffff);
  return x;
}
```