

# СОРТИРОВКА И ПОИСК

---

Массивы, указатели и константность. Указатели на функции.  
Некоторые приложения к сортировке и поиску

К. Владимиров, Syntacore, 2023  
mail-to: [konstantin.vladimirov@gmail.com](mailto:konstantin.vladimirov@gmail.com)

# СЕМИНАР 3.1

---

Линейный поиск и сортировка с локальным принятием решений

# Константность в языке C

- Распространённая языковая конструкция `const` означает неизменяемость. Неизменяемость полезна, поскольку упрощает и оптимизации и поддержку

```
int a = 4, b = 4;           // целые
int const c = 5, d = 6;    // неизменяемые целые
int const * pc = &c;       // указатель на неизменяемое целое
int * const cpa = &a;      // неизменяемый указатель на целое
```

- Применение

```
a = 6;           // ок, теперь (*cpa == 6)
c = 6;           // ошибка, неизменяемые данные
pc = &d;         // ок, теперь (*pc == 6)
cpa = &b;        // ошибка, неизменяемый указатель
```

# Давний холивар: East Const

- Модификатор `const` означает константность того, что слева от него. Но если слева от него ничего нет, то он распространяется на то, что справа от него
- `const` стоящая в нормативной позиции (справа от того что должно стать константным) называется `east const` (восточный `const`, близко к "East Coast")

```
int const x; // неизменяемое целое (east const)
const int y; // неизменяемое целое (west const)
int const *x; // указатель на неизменяемое целое (east const)
const int *x; // указатель на неизменяемое целое (west const)
int *const x; // неизменяемый указатель на целое (east const)
```

- Против `west const` есть веский аргумент от `typedefs`

# Давний холивар: East Const

- Модификатор `const` означает константность того, что слева от него. Но если слева от него ничего нет, то он распространяется на то, что справа от него
- `const` стоящая в нормативной позиции (справа от того что должно стать константным) называется `east const` (восточный `const`, близко к "East Coast")

```
typedef int * pint_t; // теперь pint_t это int *
```

```
pint_t const x; // тоже, что int * const x
```

```
const pint_t x; // не тоже, что const int * x
```

- В случаях, показанных выше, `west const` может ввести в заблуждение
- Увы, в существующем коде много таких констант и многие любят их писать и даже пишут принципиально

# Чтение сложных типов в языке C

- Основные конструкции: массив указатель и функция

```
char * const * (* next)(int * const);
```

- Читается так: начинаем от имени переменной "next это"
- Читаем "указатель" и выходим из скобок
- Читаем "на функцию, принимающую неизменяемый указатель на int"
- Читаем "и возвращающую указатель на неизменяемый указатель на char"
- Соединяем: next это указатель на функцию, принимающую неизменяемый указатель на int и возвращающую указатель на неизменяемый указатель на char
- Подробнее см. [*Linden*]

# Немного тренировки

- Прочитайте следующие определения

```
char *(*carr[10])(int **);
```

```
void (*signal(int, void (*)(int)) ) (int);
```

```
unsigned (*search)(const (int *)[2], unsigned[][4]);
```

- Разумеется умение читать такие объявления не означает что ими надо злоупотреблять

# Ваш друг typedef

- Вместо

```
void (*signal(int, void (*)(int)) ) (int);
```

- Удобно записать

```
typedef void (*ptr_to_func) (int);
```

```
ptr_to_func signal(int, ptr_to_func);
```

- Это делает тип читаемым и очевидным
- Теперь запутанные правила typedef получают объяснение: они устроены так, чтобы typedef для типа точно соответствовал c-decl для его использования

# Объявления функций с массивами

- Очень часто `const` используется в аргументах функций

```
// Эта функция может прочитать и изменить массив  
void foo(int *arr, unsigned len);
```

```
// Эта функция может только прочитать массив  
void bar(const int *arr, unsigned len);
```

- Также двойственность между массивами и указателями позволяет писать

```
void foo(int arr[], unsigned len);
```

```
void bar(const int arr[], unsigned len);
```

# Поиск в массивах

- На входе функции указатель на первый элемент **некоего** массива и длина массива, а также искомый элемент

```
unsigned search(const int *parr, unsigned len, int elem);
```

- Необходимо вернуть позицию элемента от 0 до len - 1 если он есть или len, если он не найден

```
int arr[6] = {1, 4, 10, 21, 43, 56};  
unsigned p10 = search(arr, 6, 10);  
unsigned p42 = search(arr, 6, 42);  
assert(p10 == 2 && p42 == 6);
```

# Алгоритм L – линейный поиск

```
unsigned search(const int* parr, unsigned len, int elem) {  
    unsigned i;  
    for (i = 0; i < len; ++i)  
        if (parr[i] == elem)  
            return i;  
    return len;  
}
```

# Problem MM – минимум и максимум

- На входе массив целых чисел. Ваша задача посчитать его наибольший и наименьший элементы.
- Можно ли сделать это за один проход по массиву?

# Problem FY – тасование массива

- На входе массив целых чисел.
- Ваша задача его перетасовать в случайном порядке. Для тасовки используем следующий алгоритм:
- Выбираем случайный индекс от 0 до  $N$  и меняем элемент этого индекса местами с последним элементом.
- Далее выбираем случайный индекс от 0 до  $N - 1$  и меняем элемент этого индекса местами с предпоследним элементом.
- И так далее. Это называется алгоритм Фишера-Йетса.

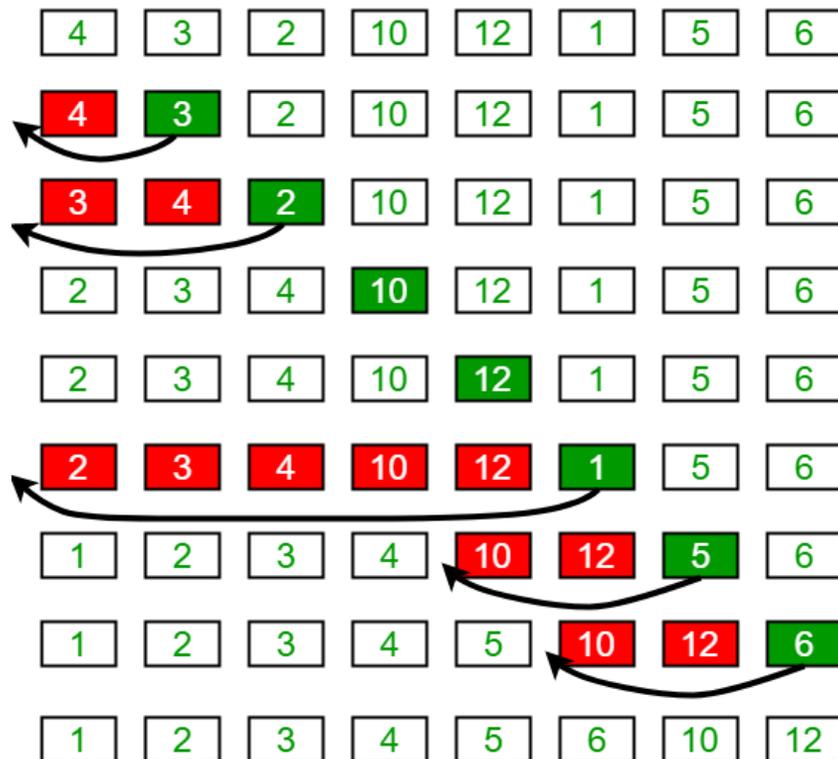
# Сортировка массива: наивный подход

- У вас есть 300 не маркированных гири разного веса и весы
- Вас просят разложить эти гири по весу в порядке возрастания
- Как вы это сделаете?



# Идея сортировки вставками

Insertion Sort Execution Example

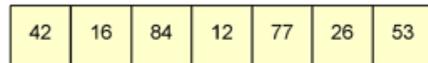


- Обычно первая идея, которая приходит человеку это отсортировать массив вставками
- Инвариант алгоритма: левая часть массива до  $n$  всегда отсортирована
- На каждом шаге  $n$  увеличивается
- При необходимости все красные элементы перемещаются
- Какая асимптотическая сложность у такого алгоритма?

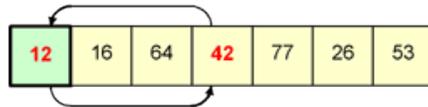
# Алгоритм I – сортировка вставками

```
unsigned moveright(int *arr, int key, unsigned last) {  
    // TODO: напишите здесь код этой функции  
}  
  
void inssort(int *arr, unsigned len) {  
    for (unsigned i = 0; i < len; ++i) {  
        int key = arr[i];  
        unsigned pos = moveright(arr, key, i);  
        arr[pos] = key;  
    }  
}
```

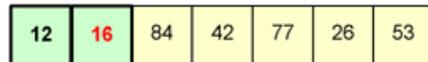
# Идея сортировки выбором



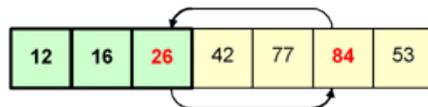
The array, before the selection sort operation begins.



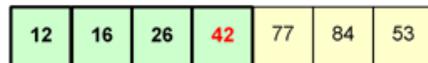
The smallest number (12) is swapped into the first element in the structure.



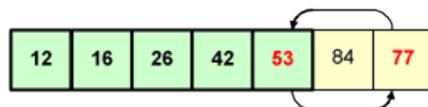
In the data that remains, 16 is the smallest; and it does not need to be moved.



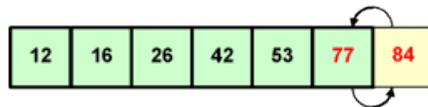
26 is the next smallest number, and it is swapped into the third position.



42 is the next smallest number; it is already in the correct position.



53 is the smallest number in the data that remains; and it is swapped to the appropriate position.



Of the two remaining data items, 77 is the smaller; the items are swapped. The selection sort is now complete.

- Вторая частая идея это сортировка выбором
- Найдём в массиве минимальный элемент и обменяем его местами с текущим
- Переместимся к следующему элементу
- Будет ли этот алгоритм асимптотически лучше вставок?

# Алгоритм SE – сортировка выбором

- Алгоритм использует уже известный алгоритм L

```
unsigned linear_search(int const * parr, unsigned len, int elem);
```

```
void swap(unsigned *v1, unsigned *v2) {  
    unsigned tmp = *v1;  
    *v1 = *v2;  
    *v2 = tmp;  
}
```

```
void selsort(int *arr, unsigned len) {  
    // TODO: напишите здесь код для сортировки выбором  
}
```

# Об одной сомнительной идее

- Может показаться заманчивым сортируя сравнивать соседние, обменивая местами, если их взаимный порядок неправилен

```
do {  
    int sw = 0;  
    for (j = len - 1; j > 0; --j)  
        if (a[j - 1] > a[j]) {  
            int tmp = a[j - 1]; a[j - 1] = a[j]; a[j] = tmp; sw = 1;  
        }  
} while (sw == 1);
```

- Это называется сортировка пузырьком (bubble sort).

# Обсуждение: bubble vs selection

- Посмотрим на простом примере, почему одинаковая асимптотика не означает одинаковое быстродействие

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

- Здесь selection sort сразу найдёт элемент и обменяет его с нужной позицией
- Bubble sort тоже это сделает, но по дороге она сделает его обмены со всеми остальными элементами
- Формально оба сделают для одного шага  $O(N)$  сравнений, но в реальности речь идёт о разнице **в разы**

# Минимум про typedef

```
typedef int myint_t;           // myint_t становится синонимом int
myint_t x = 2;                // ок, x имеет тип int

typedef int myint3_t[3];      // тип myint3_t это int[3]
myint3_t y = {1, 2, 3};      // ок, y это массив

// тип myintf_t это указатель на функцию, которая
// берёт два целых числа и возвращает целое
typedef int (*myintf_t)(int, int);

int plus(int x, int y) { return x + y; }

myintf_t z = &plus;
int a = z(2, 3); // ок, теперь a == 5
```

# Концепция `void` и `void pointer`

- Ключевое слово `void` в языке C используется сразу для нескольких вещей
- Отсутствия аргументов или результата у функции

```
void bar(void);
```

- Указателя на неопределённую память

```
void *pv = (void *) &i;
```

- Такой указатель не может быть разыменован, с ним также не работает адресная арифметика
- Всё что с ним можно сделать осмысленного это привести к типизированному указателю или передать в функцию

# Обсуждение

- Сортировать целые числа это весело и интересно, но что делать, если хочется сортировать произвольные объекты?
- Для этого можно использовать `void*` и размер объекта в памяти
- Для того, чтобы сравнивать такие объекты, можно передавать указатель на функцию-компаратор

```
typedef int (*cmp_t)(void const * lhs, void const * rhs);
```

- Свою функцию компаратор можно написать для своих типов и передать в обобщённую функцию сортировки

# Компаратор для целых чисел

- Обобщённый компаратор

```
int int_less(void const * lhs, void const * rhs) {  
    int const * lhsi = (int const *) lhs;  
    int const * rhsi = (int const *) rhs;  
    return (*lhsi < *rhsi);  
}
```

- Он работает так, что `int_less(&a, &b)` возвращает то же самое, что и сравнение `(a < b)`

```
int a = 2, b = 3;
```

```
assert(int_less(&a, &b) == 1);
```

# Вызываем qsort

- Стандартный способ отсортировать массив произвольных объектов это вызвать qsort.

```
void qsort(void *ptr, size_t count, size_t size,  
          int (*comp)(const void *, const void *));
```

- Например:

```
int arr[5] = {3, 4, 1, 2, 5};  
qsort(arr, 5, sizeof(int), int_less);
```

- Давайте забенчмаркаем?

# Problem CSE – обобщение alg SE

- Для одного шага сортировки выбором теперь понадобится чуть больше параметров

```
typedef int (*cmp_t)(void const * key, void const * elt);  
  
// eltsize – размер элемента  
// numelts – количество элементов  
// nsorted – позиция последнего отсортированного  
int selstep(void const * arr, int eltsize, int numelts,  
            int nsorted, cmp_t cmp) {  
    // напишите тут код  
}
```

- Ваша задача реализовать этот шаг. Обратите особое внимание на swap

# Обсуждение

- Все простые сортировки: insertion, selection и bubble имеют довольно плохую асимптотику  $O(N^2)$  потому что в основном принимают только **локальные** решения
- Они элемент за элементом наращивают отсортированную часть массива
- Гораздо более интересные результаты можно получить, если для сортировки так или иначе разбивать массив на две части
- Группа методов, которая так работает имеет общее название Divide & Conquer

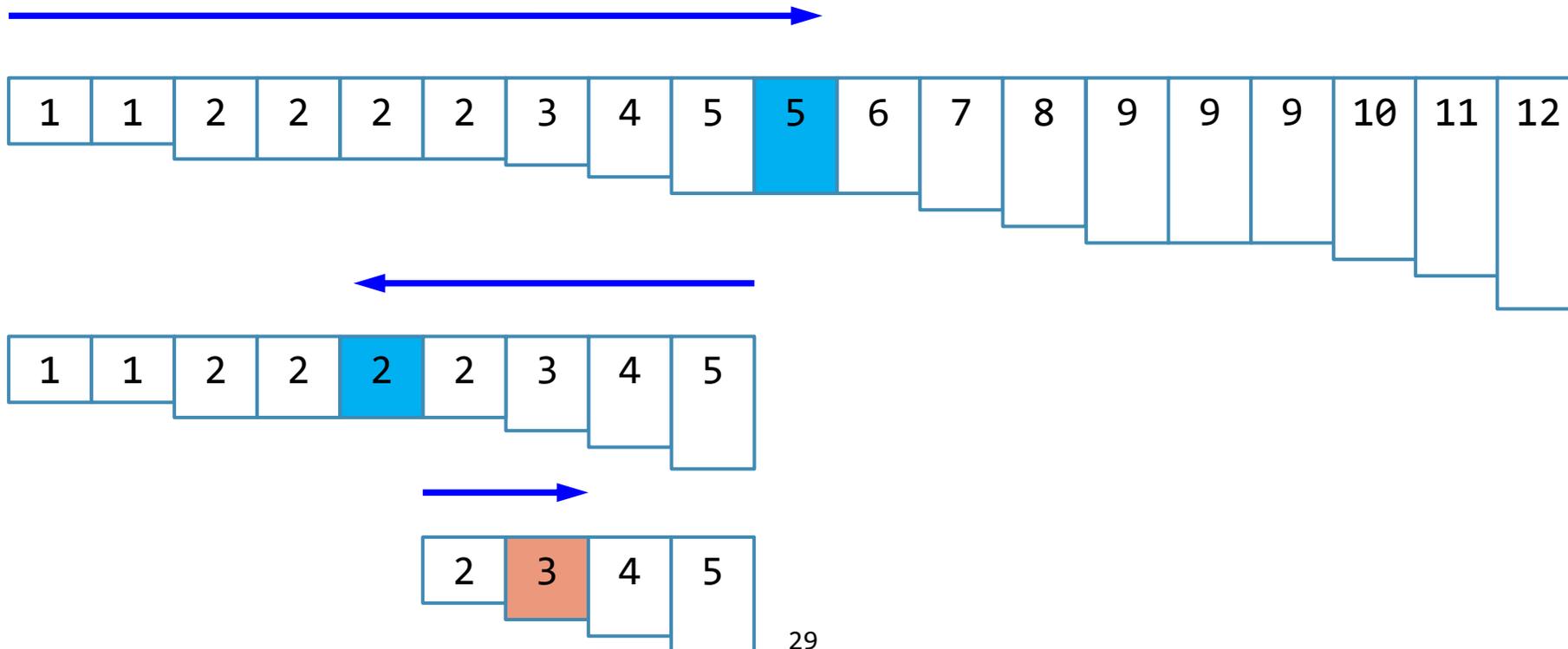
# СЕМИНАР 3.2

---

Стратегия разделяй и властвуй

# Стратегия разбиения пополам

- Также известна как "divide and conquer"

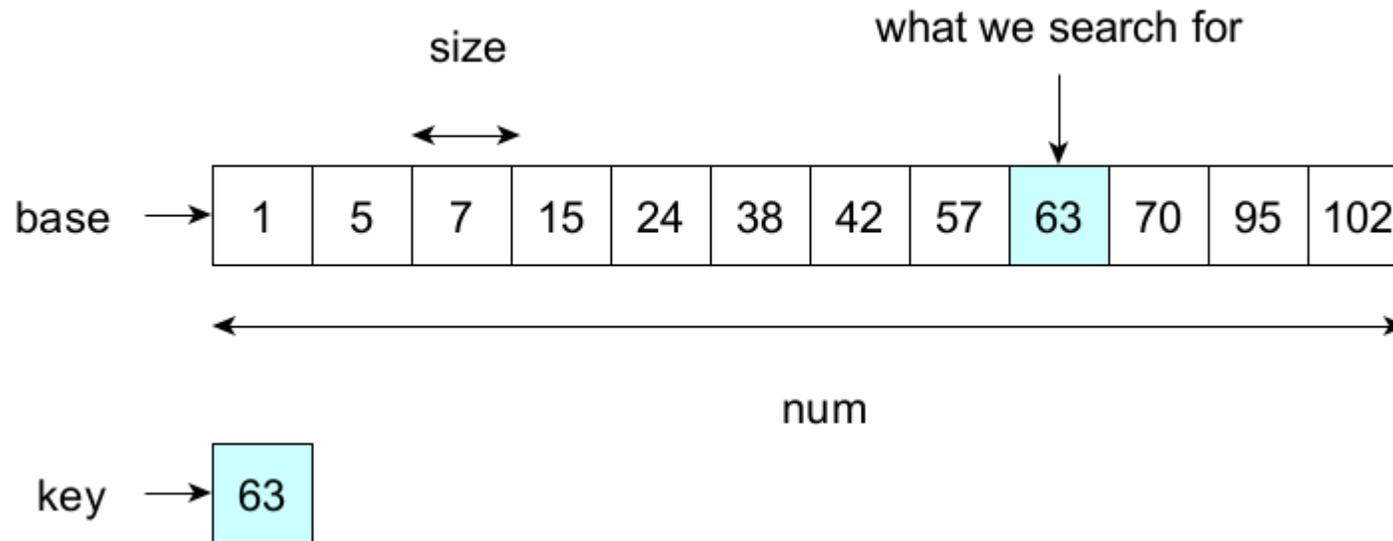


# Алгоритм В – бинарный поиск

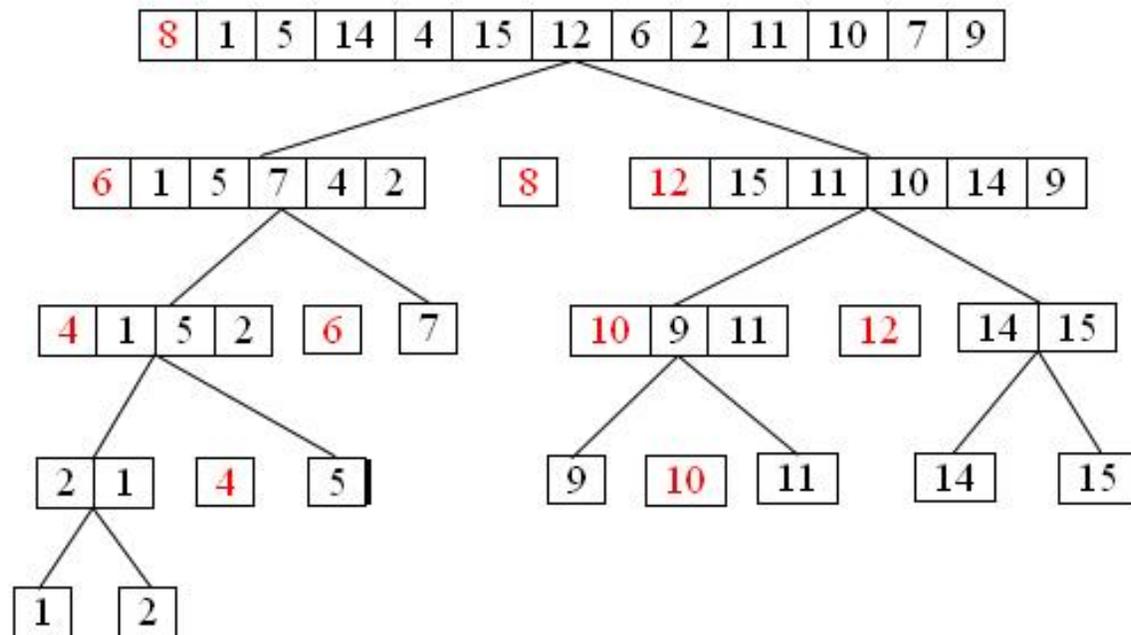
```
unsigned
binary_search(int const * parr, unsigned len, int elem) {
    int l = 0; int r = len - 1;
    while (l <= r) {
        int m = l + (r - l) / 2;
        if (parr[m] == elem) return m;
        if (parr[m] < elem) l = m + 1;
        if (parr[m] > elem) r = m - 1;
    }
    return len;
}
```

# Problem SCB – общий бинарный поиск

```
typedef int (*cmp_t)(void const * lhs, void const * rhs);  
void *cbsearch(void const * key, void const * base,  
              int num, int size, cmp_t cmp);
```



# D&C подход: быстрая сортировка



- Массив делится на две части по pivot (можно всегда выбирать первый)
- Далее результаты разбиения сортируются отдельно тем же способом
- В итоге массив собирается из отсортированных подмассивов

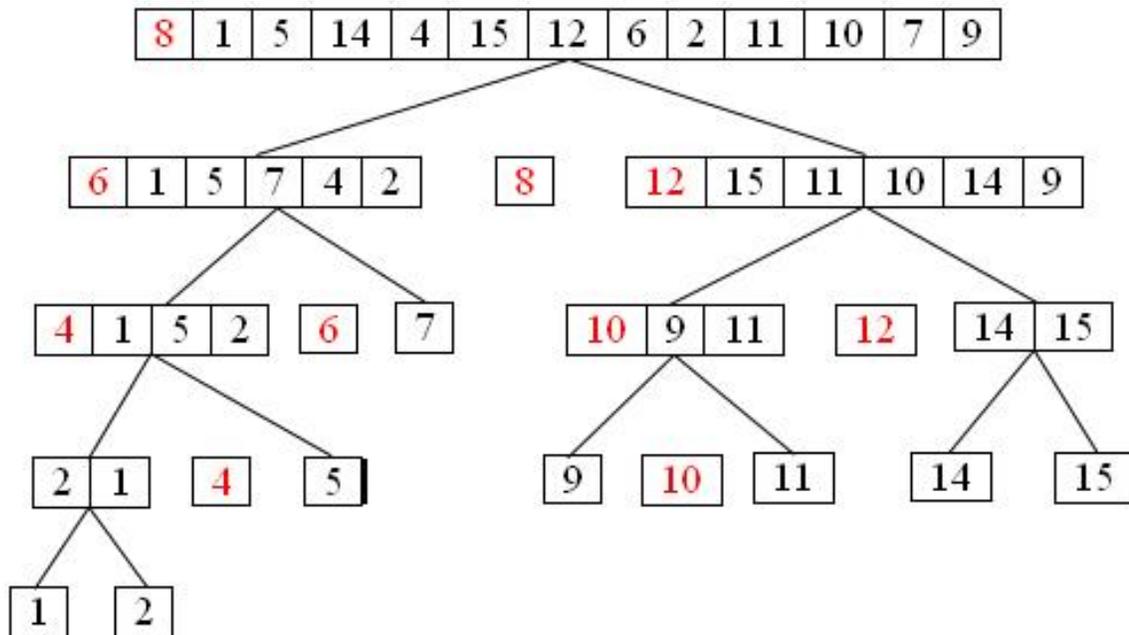
# Алгоритм Q – быстрая сортировка

```
unsigned partition(int *arr, unsigned low, unsigned high);  
void qsort_impl(int *arr, unsigned low, unsigned high) {  
    if (low >= high)  
        return;  
    unsigned pi = partition(arr, low, high);  
    if (pi > low) qsort_impl(arr, low, pi - 1);  
    qsort_impl(arr, pi + 1, high);  
}  
  
void qsort(int *arr, unsigned len) {  
    qsort_impl(arr, 0u, len - 1);  
}
```

# Средний и худший случай

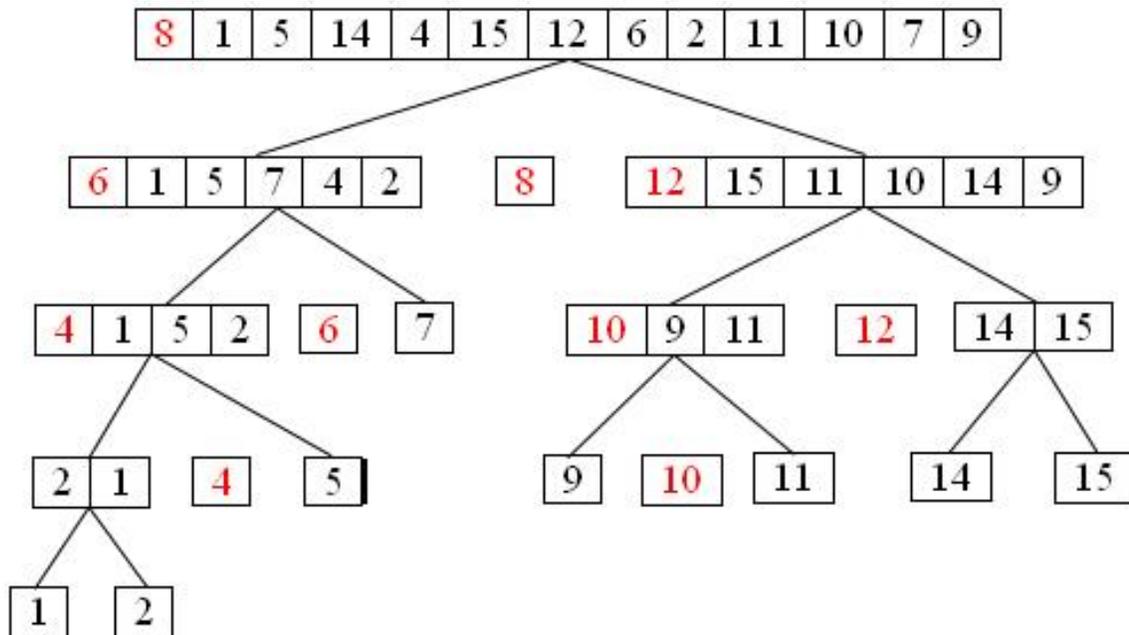
- На картинке ниже представлен средний случай

- Как нарисовать картинку худшего случая?

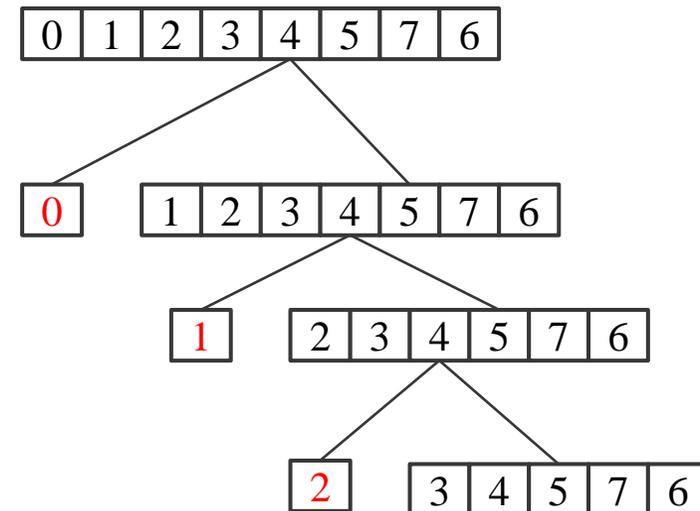


# Средний и худший случай

- На картинке ниже представлен средний случай



- Как нарисовать картинку худшего случая?

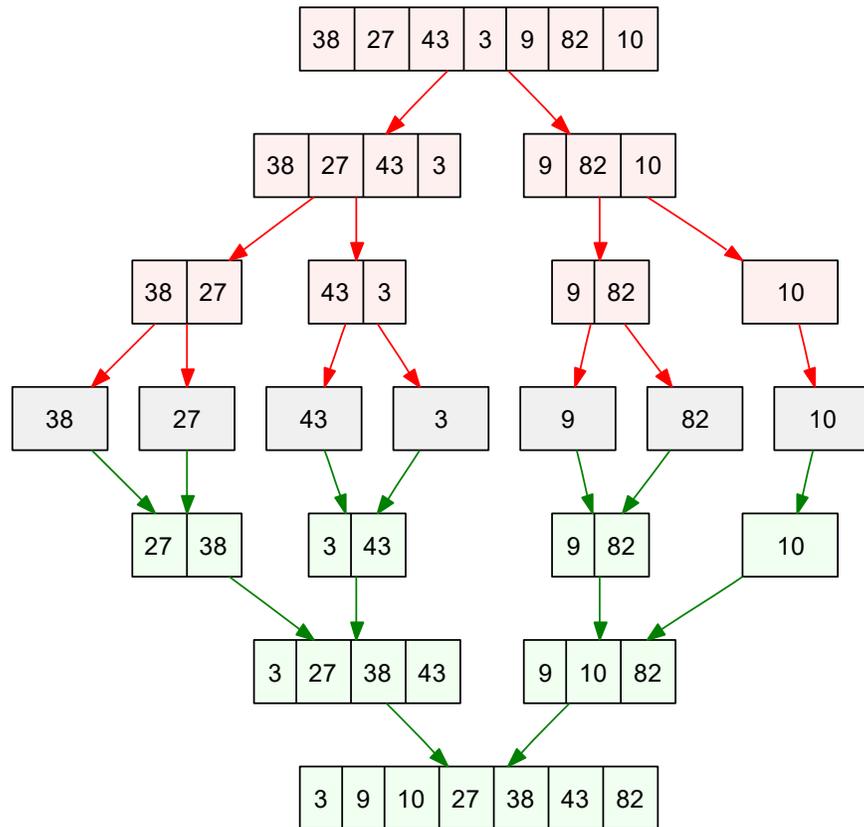


- Какая асимптотика у худшего случая?

# Обсуждение

- Быстрая сортировка в худшем случае всё ещё ведёт себя как  $O(n^2)$
- Есть остроумные способы избежать на входе почти отсортированных массивов: например случайно перемешивать массив перед сортировкой
- Можно ли придумать сортировку, которая **всегда** работает как  $O(n \log n)$ ?

# Сортировка слиянием



- Делим массив на каждом шаге примерно пополам
- Вовсе не обязательно при этом реально выделять новые массивы, можно просто хранить индексы
- Далее сливаем получившиеся подмассивы и получаем отсортированные подмассивы
- У нас нет худшего случая!

# Алгоритм М – сортировка слиянием

- Вам, предлагается реализовать ключевой шаг: функцию слияния

```
// сливает arr[l..m] и arr[m+1..r]
void merge(int *arr, int l, int m, int r) {
    // TODO: ваш код здесь
}

void merge_sort_imp(int *arr, int l, int r) {
    if (l >= r) return;
    int m = (l + r) / 2;
    merge_sort_imp(arr, l, m);
    merge_sort_imp(arr, m + 1, r);
    merge(arr, l, m, r);
}
```

# Стратегия "разделяй и властвуй"

- И алгоритм В, и проблема ME имеют нечто общее: каждая итерация алгоритма уменьшает размер рассматриваемых данных вдвое

- Для бинарного поиска:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

- Для сортировки слиянием:

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

- Это распространённый подход

- Как оценить асимптотическую сложность таких рекуррентностей?



# Master theorem

- Применяется для решения рекуррентностей возникающих при D&C подходе.
- Пусть  $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$
- Тогда решения зависят от соотношения  $d$  и  $\log_b a$ .
- Если  $d > \log_b a$ , то  $T(n) = O(n^d)$
- Если  $d = \log_b a$ , то  $T(n) = O(n^d \log n)$
- Если  $d < \log_b a$ , то  $T(n) = O(n^{\log_b a})$
- Тут можно провести простое доказательство на доске, если его не было на лекциях.

# Анализ бинарного поиска

- Примените Master theorem к рекуррентности:  $T(n) = T\left(\frac{n}{2}\right) + O(1)$
- Если  $d > \log_b a$ , то  $T(n) = O(n^d)$
- Если  $d = \log_b a$ , то  $T(n) = O(n^d \log n)$
- Если  $d < \log_b a$ , то  $T(n) = O(n^{\log_b a})$

# Анализ быстрой сортировки и слияния

- Примените Master theorem к рекуррентности:  $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$
- Если  $d > \log_b a$ , то  $T(n) = O(n^d)$
- Если  $d = \log_b a$ , то  $T(n) = O(n^d \log n)$
- Если  $d < \log_b a$ , то  $T(n) = O(n^{\log_b a})$

# Перемножение полиномов

- Пусть даны  $A(x) = x^3 + 3x^2 + 4x + 7$  и  $B(x) = x^3 + 5x^2 + x + 4$
- Чему равно  $A(x) * B(x)$ ?
- Постарайтесь осознать КАК вы это подсчитали?

# Problem MP – перемножение полиномов

- Пусть даны  $A(x) = a_0x^n + \dots + a_n$  и  $B(x) = b_0x^m + \dots + b_m$
- Вам необходимо подсчитать их произведение самым простым и очевидным способом: последовательно перемножая коэффициенты

```
struct Poly { unsigned n; unsigned *p; };
```

```
struct Poly mult(struct Poly lhs, struct Poly rhs) {  
    struct Poly ret = { rhs.n + lhs.n - 1, NULL };  
    ret.p = calloc(ret.n, sizeof(unsigned));  
    // TODO: ваш код здесь  
    return ret;  
}
```

# Перемножение: алгоритм Карацубы

- Многие, в т. ч. Колмогоров, считали, что  $O(n^2)$  это нижняя граница. До тех пор, пока тогда ещё студент Карацуба не принёс Колмогорову лучшее решение
- Основная идея такая: пусть  $A(x) = A_1x^{n/2} + A_0$  и  $B(x) = B_1x^{n/2} + B_0$
- Тогда  $C(x) = A(x)B(x) = A_1B_1x^n + (A_0B_1 + A_1B_0)x^{n/2} + A_0B_0$
- Примените Master Theorem к рекуррентности  $T(n) = 4T\left(\frac{n}{2}\right) + O(n)$
- Но:  $C(x) = A_1B_1x^n + ((A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0)x^{n/2} + A_0B_0$
- Примените Master Theorem к рекуррентности  $T(n) = 3T\left(\frac{n}{2}\right) + O(n)$

# Problem ME – поиск большинства

- На входе функции указатель на первый элемент **произвольного** массива и длина массива (решение этой задачи не предполагает сортировки)

```
int majority_element(int const * parr, unsigned len);
```

- Необходимо определить, есть ли в массиве элемент, который встречается больше  $len/2$  раз и вернуть его значение (или -1 если никакой элемент не образует большинства).

```
int arr[5] = {3, 2, 9, 2, 2};  
int x = majority_element(arr, 5);  
assert(x == 2);
```

- Напишите тело функции.

# Problem CM – обобщение слияния

- Можно даже замахнуться на сортировку объектов разных размеров
- Предположим, что у вас есть реализованный кем-то компаратор типа `xcmp_t`

```
// сравнивает два объекта разных длин
typedef int (*xcmp_t)(void const * lhs, int lsz,
                    void const * rhs, int rsz);
```

- Ваша задача реализовать любую эффективную сортировку (проще всего слияние) с набором последовательных в памяти элементов разного размера

```
// nelts количество элементов и их размеров, mem общая память
void xmsort(void * mem, int * sizes, int nelts, xcmp_t cmp);
```

- Вы можете попробовать разные алгоритмы сортировки, например быструю, но это может быть неожиданно сложно

# СЕМИНАР 3.3

---

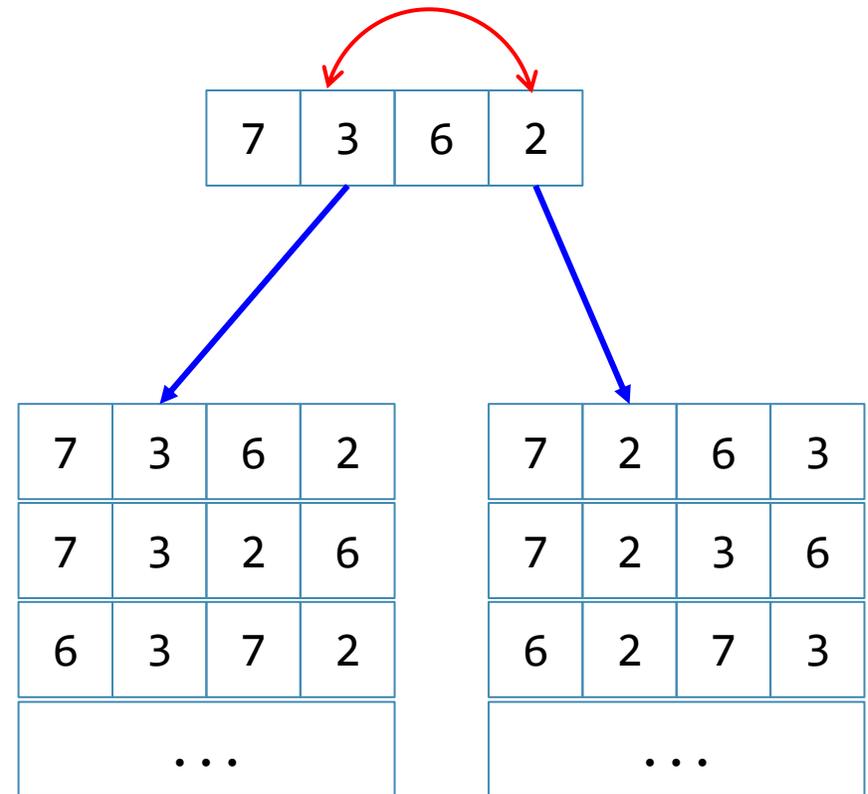
Цифровой поиск

# Обсуждение: сортировки сравнением

- Пока что ни одна из наших обобщённых сортировок не получилась по асимптотике лучше, чем  $O(n \cdot \log n)$ .
- Чтобы понять почему это так, давайте рассмотрим пространство состояний.
- Оно состоит из перестановок.
- У нас есть изначальный массив длины  $N$  и из всех его  $N!$  перестановок мы должны выбрать упорядоченную.
- С этим связан известный алгоритм Bogosort: мы просто выбираем случайную перестановку и проверяем упорядоченность. Но это скорее шутка.
- А если серьёзно, то какая стратегия тут возможна?

# Основная идея сравнений

- Когда мы сравниваем два элемента мы фиксируем два класса перестановок: все в котором эти элементы стоят в одном порядке и все где в другом.
- Всего у нас  $N!$  перестановок.
- Сколько шагов нам понадобится чтобы дойти до искомой, если мы каждый раз делим пространство пополам?
- В среднем  $O(\log(N!))$ . А сколько это?



# Железная логика и странные идеи

- По формуле Стирлинга

$$O(\log(N!)) = O\left(\log\left(\sqrt{2\pi n} \left(\frac{n}{e}\right)^n\right)\right) = O(1) + O(\log\sqrt{n}) + O\left(n \cdot \log\left(\frac{n}{e}\right)\right)$$

- Откуда немедленно следует, что

$$O(\log(N!)) = O(n \cdot \log n)$$

- В итоге кажется, что математика не пускает нас в лучшие (в среднем) сортировки. Сам процесс сравнения требует в среднем не менее чем  $n \cdot \log n$  сравнений.
- И тут нас посещает одна странная идея.

# Сортировка подсчётом

- Сначала формируем бакеты.

```
for (i = 0; i < n; ++i)  
    // TODO: как сформируем?
```

```
out = 0;
```

- Далее складываем подсчитанные элементы в массив.

```
for (i = 0; i < nb; ++i)  
    for (j = 0; j < b[i]; ++j)  
        a[out++] = i;
```

7	3	6	2	6	2	6	6
---	---	---	---	---	---	---	---

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

2	2	3	6	6	6	6	7
---	---	---	---	---	---	---	---

# Problem CST -- сортировка подсчётом

- С потока ввода приходит длина, а далее один за другим элементы.
- Ваша задача -- вывести на stdout массив бакетов.

• Пример: 

8	7	3	6	2	6	2	6	6
---	---	---	---	---	---	---	---	---

• Вывод: 

0	0	2	1	0	0	4	1
---	---	---	---	---	---	---	---

- Есть ли у сортировки подсчётом очевидные проблемы?

# Проблемы сортировки подсчётом

- Рассмотрим массив:

126	348	343	432	316	171	556	670
-----	-----	-----	-----	-----	-----	-----	-----

- Чтобы отсортировать его подсчётом потребуется 670 бакетов. Многовато на восемь элементов.
- Вторая идея: отсортировать поряждно.
- Мы можем сначала сгруппировать вместе по последней цифре, потом по предпоследней и так далее.

# Интермедия: стабильность

- Стабильной называется сортировка которая оставляет равные но не эквивалентные элементы на своём месте.

• Массив: 

126	670	343	432	316	173	556	348
-----	-----	-----	-----	-----	-----	-----	-----

- Критерий сортировки: нулевой разряд.

• Стабильный результат: 

670	432	343	173	126	316	556	348
-----	-----	-----	-----	-----	-----	-----	-----

• Нестабильный результат: 

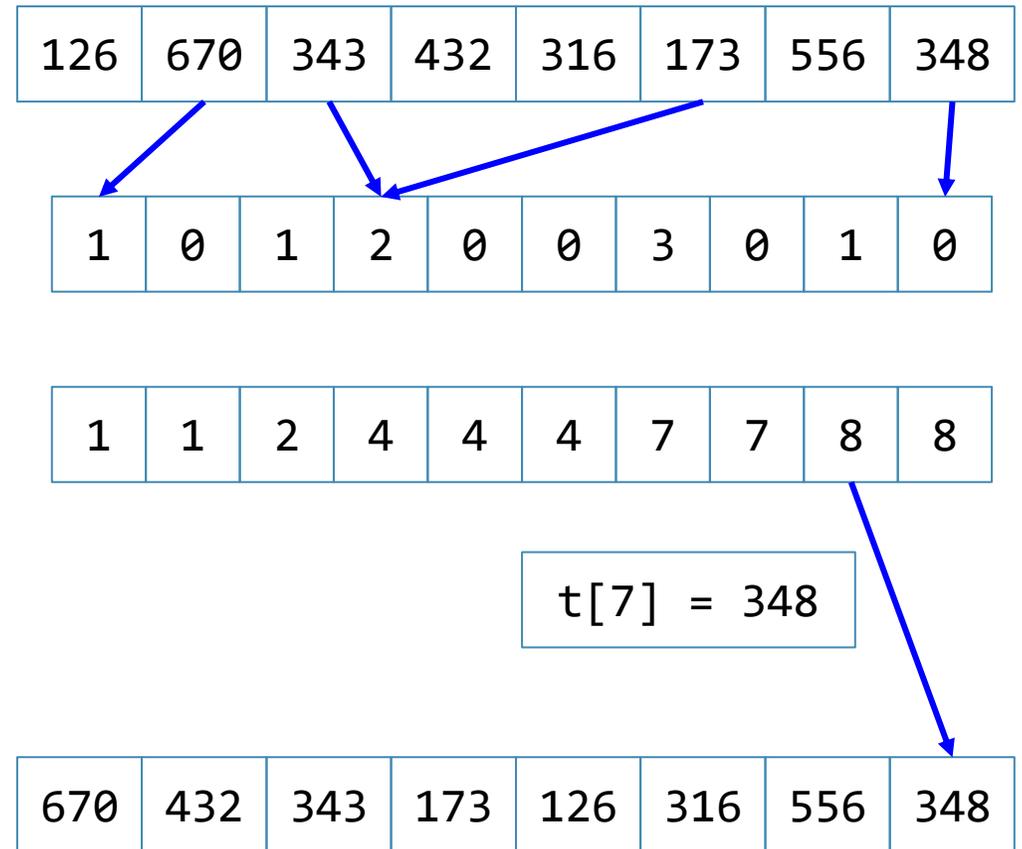
670	432	173	343	126	316	556	348
-----	-----	-----	-----	-----	-----	-----	-----

# Поразрядная сортировка

```
int expn = 1;
for (i = 0; i < ndigits; ++i) {
    int b[10] = {0};
    for (j = 0; j < n; ++j)
        b[(a[j] / expn) % 10] += 1;

    for (j = 1; j < 10; ++j)
        b[j] += b[j - 1];

    // переставить a[???] по b[???]
    expn *= 10;
}
```



# Problem RS -- поразрядная сортировка

- Ваша задача совершить над данным массивом одну итерацию стабильной **поразрядной** сортировки по заданному вам разряду.

- Пример

• Вход: 

8
---

126	670	343	432	316	173	556	348
-----	-----	-----	-----	-----	-----	-----	-----

0
---

• Выход: 

670	432	343	173	126	316	556	348
-----	-----	-----	-----	-----	-----	-----	-----

- Массив отсортирован по нулевому разряду. Сортировка стабильная.

# Многомерные массивы

- Два принципиально разных типа двумерных массивов:
- Непрерывный двумерный массив

```
int twodim[10][10] = {{0, 1}, {2, 3}};
```

```
twodim[2][3] = 100; // *(&twodim[0][0] + 2*10 + 3) = 100;
```

- Массив указателей

```
int *twodim[3] = { malloc(40), malloc(40), malloc(40) };
```

```
twodim[2][3] = 100; // *(* (twodim[0] + 2) + 3) = 100;
```

- Ситуацию усложняет то, что обращения к `arr[x][y]` выглядят в коде одинаково

# Указатели на массивы

- Подобно указателям на функции, существуют указатели на массивы

```
int *arrptrs[10]; // array of 10 pointers to int
```

```
int (*ptrarr)[10]; // pointer to array of 10 ints
```

- Основная разница проявляется при инкременте

```
int arr[10][10] = {{0, 1}, {2, 3}};
```

```
arrptrs[0] = &arr[0][0]; arrptrs[0] += 1; // +4
```

```
ptrarr = &arr[0]; ptrarr += 1; // +40
```

- Указатель на массив имеет применение когда мы хотим передать непрерывный массив в функцию

# Problem PX – матрицы в степень

- Используйте идею из алгоритма POWM (см. также [TAOCP Algorithm 4.6.3A])
- Напишите функцию для возведения в любую степень матриц NxN заданных как указатели на массивы

```
void powNxN (unsigned (*n)[N], unsigned x) {  
    // TODO: ваш код здесь  
    // вы можете предполагать NxN матрицу  
    // необходимо модифицировать n, возведя её в степень x  
}
```

- Оцените асимптотическую сложность этого алгоритма

# Домашнее задание HWS – Timsort

- Многие современные алгоритмы для лучшей производительности комбинируют слияние и вставки
- Прочитайте статью <https://en.wikipedia.org/wiki/Timsort>
- Реализуйте обобщённый (для произвольных типов) описанный там алгоритм, используя применённый на этом семинаре подход с `void*`

# Литература

- [C11] ISO/IEC, "Information technology – Programming languages – C", ISO/IEC 9899:2011
- [K&R] Brian W. Kernighan, Dennis Ritchie – The C programming language, 1988
- [Linden] Peter van der Linden – Expert C Programming: Deep C Secrets , 1994
- [Cormen] Thomas H. Cormen – Introduction to Algorithms, 2009
- [TAOCP] Donald E. Knuth – The Art of Computer Programming, 2011

