МАССИВЫ И УКАЗАТЕЛИ

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

К. Владимиров, Intel, 2019
mail-to: konstantin.vladimirov@gmail.com

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

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(int const *arr, unsigned len);
```

- Также двойственность между массивами и указателями позволяет писать

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

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

Поиск в массивах

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

```
unsigned search(int const *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);
```

- Наивным (но часто единственным) подходом будет просматривать каждый элемент последовательно

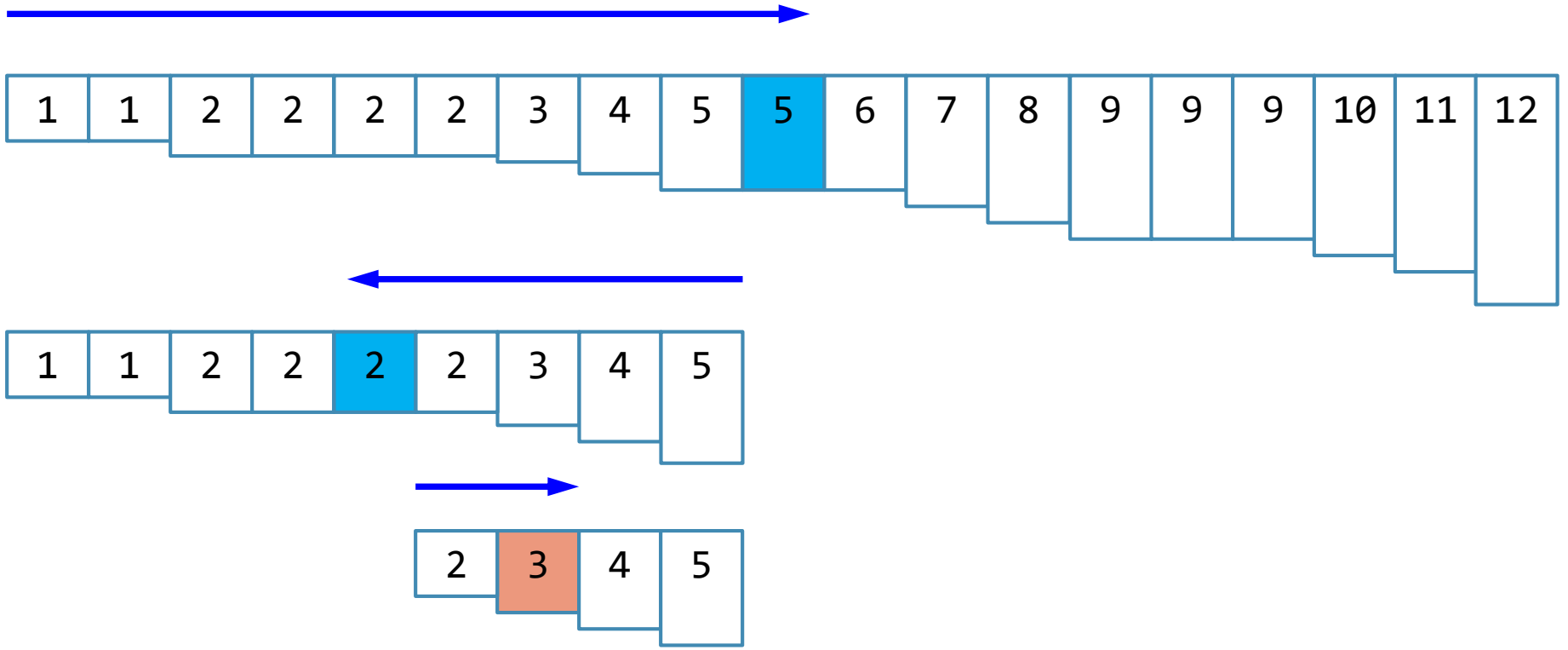
Алгоритм L – линейный поиск

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

- Алгоритм достойный и заслуженный, но если массив на входе отсортирован, то можно действовать лучше

Стратегия разбиения пополам

- Также известна как "divide and conquer"



Алгоритм B – 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;
}
```

- Сравните линейный и бинарный поиск: выделите в куче достаточно большой, последовательно заполненный массив и поищите `10000` раз случайные значения

Асимптотика

- Оцените асимптотику алгоритма B и алгоритма L
- Вам приносят реальные замеры времени поиска на некоторых данных и вы видите, что алгоритм L вдвое быстрее чем B. Что вы можете сказать об этих данных?

Смена стратегии поиска

- Функции линейного и бинарного поиска имеют одинаковую сигнатуру

```
typedef unsigned (*search_t)(int const *, unsigned, int);
```

- Теперь можно завести переменную такого типа (указатель на функцию) и вызывать линейный или бинарный поиск во время выполнения

```
search_t func = &linear_search;
func(unordered_arr, 10, -2); // вызвана linear_search
func = &binary_search;
func(ordered_arr, 20, 5); // вызвана binary_search
```

- Скоро идея указателя на функцию будет использована для некоторых обобщений поиска и сортировки

Запутанные правила typedef

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

typedef int myint3_t[3]; // тип myint3_t это int[3]
myint3_t y = {1, 2, 3};  // ok, 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); // ok, теперь a == 5
```

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

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 (необязательная задача повышенной сложности: сделайте это без рекурсии)

Problem MSB – битовый поиск

- Напишите функцию, ищущую старший установленный бит в числе

```
unsigned msb(unsigned x);
```

- На входе функции число типа unsigned
- На выходе номер её старшего бита, первый бит имеет номер `1`. Для `0x7fff` это `15`. Если ни один бит не установлен, то `0`.
- Задача со звёздочкой (опционально): совместите эту функцию и поиск, чтобы найти номер старшего установленного бита в массиве

```
unsigned arrmsb(unsigned char const * parr, unsigned len);
```

- Например для массива `{0, 3, 26}` результатом будет `66`

Стратегия "разделяй и властвуй"

- И алгоритм B, 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 мажорирующего элемента:

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

- Это распространённый подход
- Как оценить асимптотическую сложность таких рекуррентностей?

Master theorem

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

Перемножение полиномов

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

Problem MP – перемножение полиномов

- Пусть даны $A(x) = a_0 x^n + \cdots + a_n$ и $B(x) = b_0 x^m + \cdots + b_m$
- 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;
}
```

- Оцените асимптотику получившегося алгоритма. Можно ли сделать лучше?

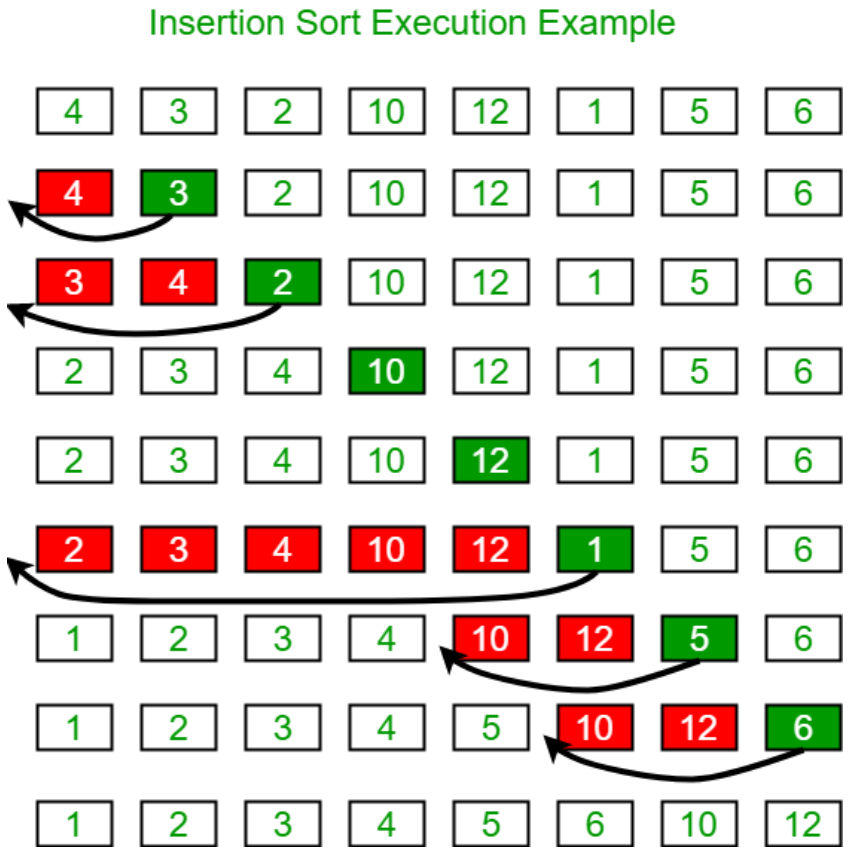
Перемножение: алгоритм Карацубы

- 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 $A(x) = A_1 x^{n/2} + A_0$ и $B(x) = B_1 x^{n/2} + B_0$
- Тогда $C(x) = A(x)B(x) = A_1 B_1 x^n + (A_0 B_1 + A_1 B_0) x^{n/2} + A_0 B_0$
- Примените Master Theorem к реккурентности $T(n) = 4T\left(\frac{n}{2}\right) + O(n)$
- Но: $C(x) = A_1 B_1 x^n + \big((A_1 + A_0)(B_1 + B_0) - A_1 B_1 - A_0 B_0\big) x^{n/2} + A_0 B_0$
- Примените Master Theorem к реккурентности $T(n) = 3T\left(\frac{n}{2}\right) + O(n)$
- Домашнее задание: реализуйте алгоритм Карацубы для problem MP

Сортировка массива: наивный подход

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

Идея сортировки вставками



- 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?

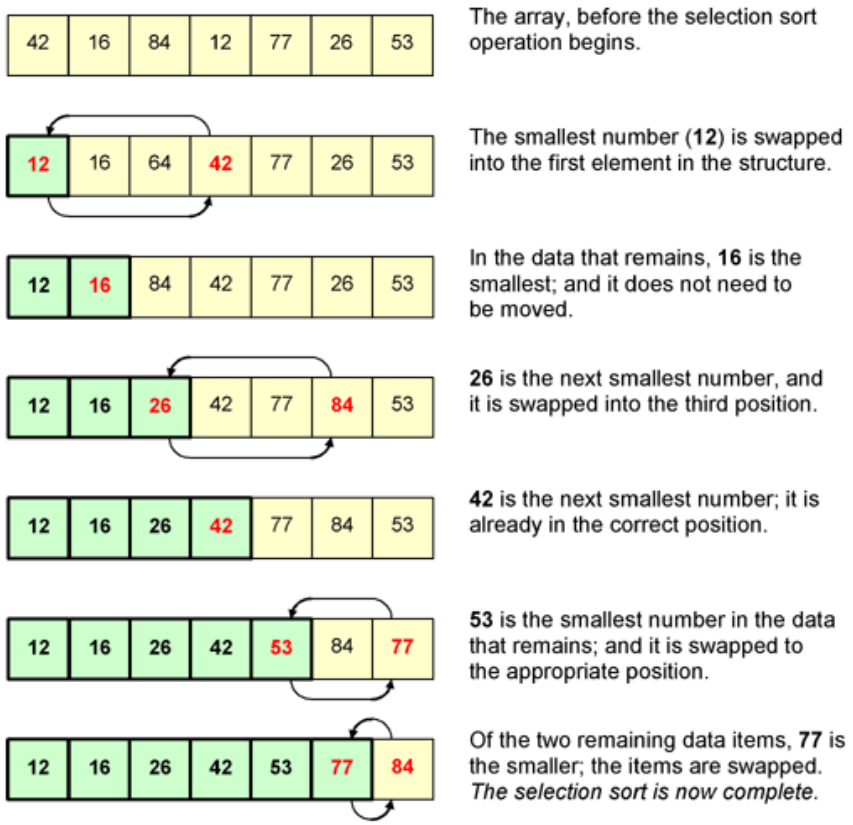
Алгоритм I – сортировка вставками

- Реализация потребует двух функций, ключевой из которых является `moveright`, делающая сдвиг массива вправо, то есть к большим индексам, чтобы освободить позицию для вставки

```
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



- 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

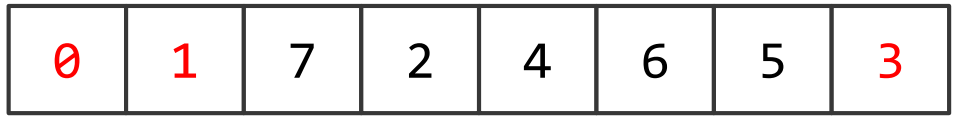
- Может показаться заманчивым сортируя сравнивать соседние, обменивая местами, если их взаимный порядок неправилен

```
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)
- Формально у него в среднем такая же асимптотика $O(N^2)$, как у вставок и выбора, но на практике он почти всегда **крайне плох**

Обсуждение: bubble vs selection

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



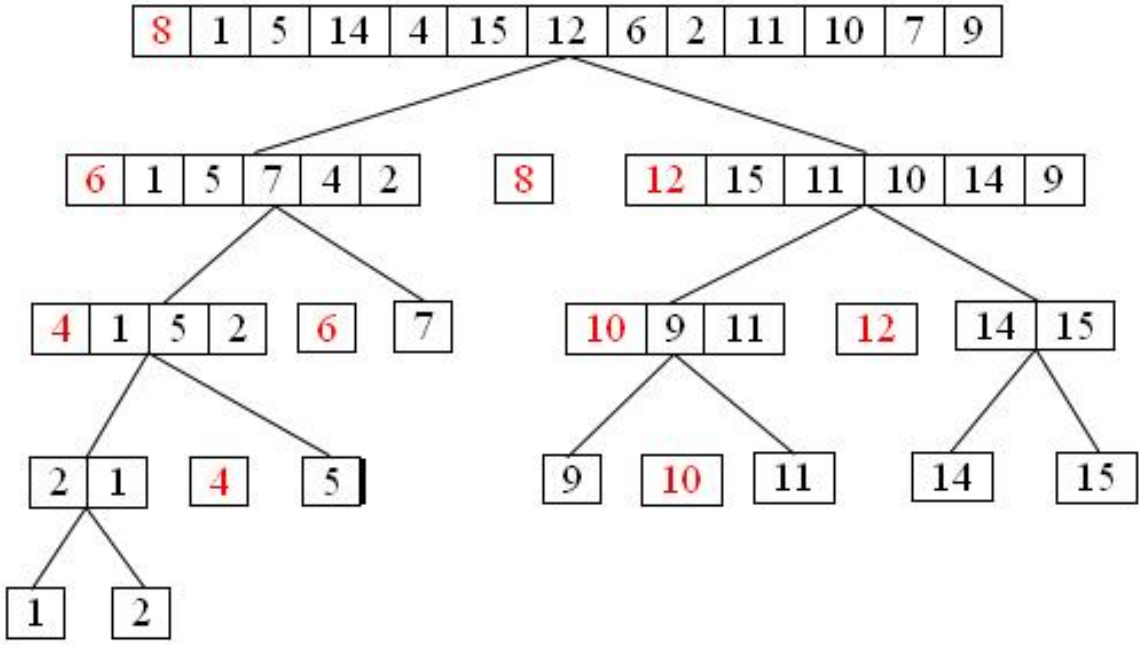
- Здесь selection sort сразу найдёт элемент и обменяет его с нужной позицией
- Bubble sort тоже это сделает, но по дороге она сделает его обмены со всеми остальными элементами
- Формально оба сделают для одного шага $O(N)$ сравнений, но в реальности речь идёт о разнице в разы

Ссылка на сравнение пузырька с insertion sort

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

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

- Примените Master theorem к рекуррентности: $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$
- Что можно сказать об асимптотике быстрой сортировки по сравнению с вставками?

Алгоритм Q – быстрая сортировка

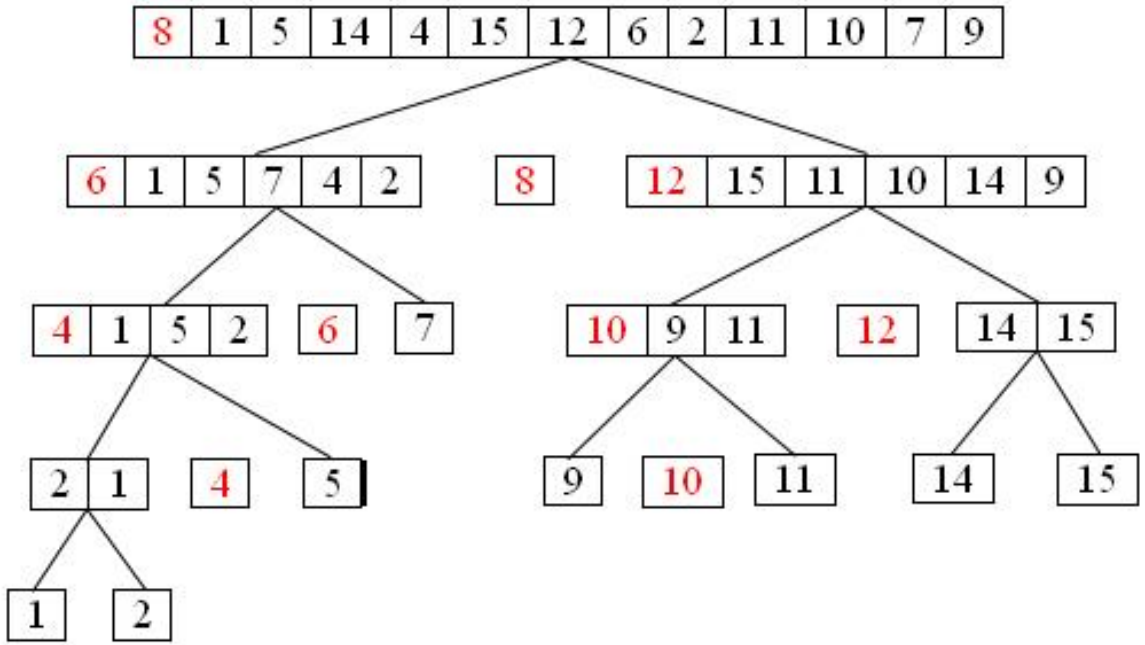
```
unsigned partition(int *arr, unsigned low, unsigned high) {
  // TODO: напишите код функции partition
}

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

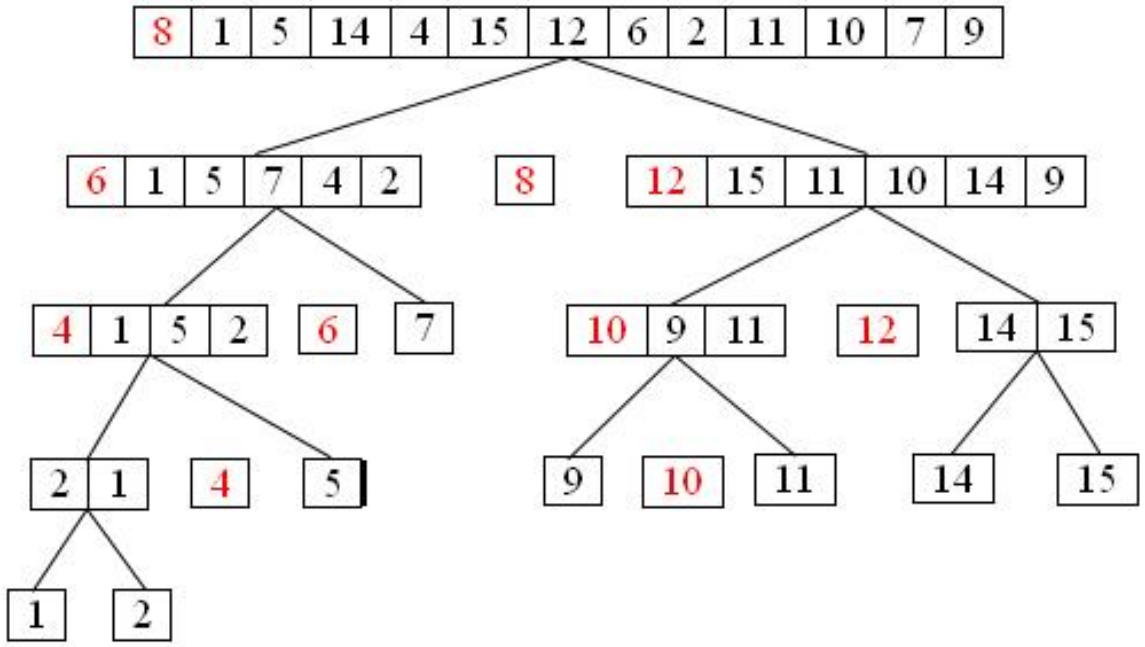
- На картинке ниже представлен средний случай



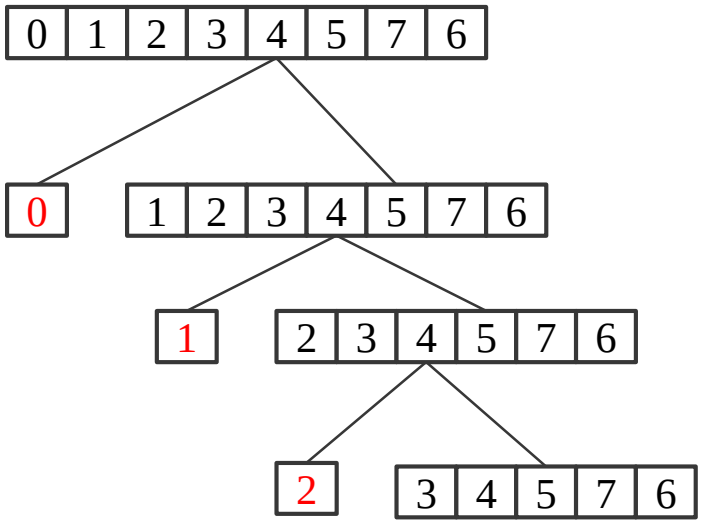
- Как нарисовать картинку худшего случая?

Средний и худший случай

- На картинке ниже представлен средний случай



- Как нарисовать картинку худшего случая?

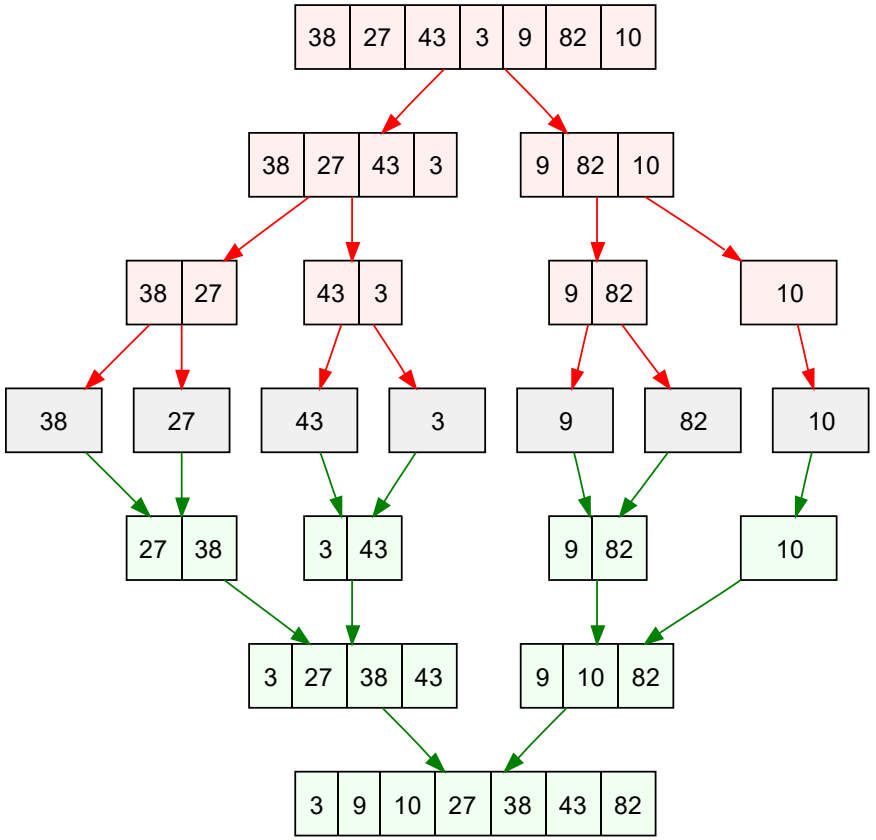


- Какая асимптотика у худшего случая?

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
- В отличии от сортировки выбором, у нас нет худшего случая, всегда $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

Алгоритм M – 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);
}
```

Обсуждение

- Сортировать целые числа это весело и интересно, но что делать, если хочется сортировать произвольные объекты?
- Например можем ли мы написать такую функцию, которая могла бы отсортировать и массив целых чисел и массив структур?

Минимум о приведении

- В языке C специальный синтаксис "`val = (T)other`" обозначает приведение одного типа к другому. Мы уже пользовались им раньше

```
double d = 49.0;
int i = (int) d; // теперь i == 49
char c = (char) i; // теперь c == '1'
```

- Точно так же можно приводить указатели

```
char *pc = (char *) &i; // теперь pc указывает на первый байт i
```

- Будьте осторожны с приведением указателей. Правила строгого алиасинга (strict aliasing rules) требуют чтобы в программе одновременно не существовало указателей двух разных типов на одну и ту же локацию в памяти
- Из этих правил есть два исключения. Тип `char*` и тип `void*`

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);
```

Алгоритм СВ – 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) {
  char const * pivot;
  int result;

  while (num > 0) {
    pivot = (char const *) base + (num / 2) * size;
    result = cmp(key, (void const *) pivot);
    if (result == 0)
      return (void *) pivot;

    // TODO: допишите здесь обобщённый бинарный поиск

  }
  return NULL;
}
```

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

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

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*`

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

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

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