МАССИВЫ И УКАЗАТЕЛИ

Массивы и указатели в языке C: сходства, различия, двойственность.
Указатели на функции. Составные типы. Typedef.

К. Владимиров, Intel, 2018
mail-to: konstantin.vladimirov@gmail.com

Константность в языке C

- Распространённая языковая конструкция const означает readonly

```
int a = 4, b = 4;          // целые
const int c = 5, d = 6; // неизменяемые целые
const int *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

- Наиболее часто используется в объявлениях 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);
```

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

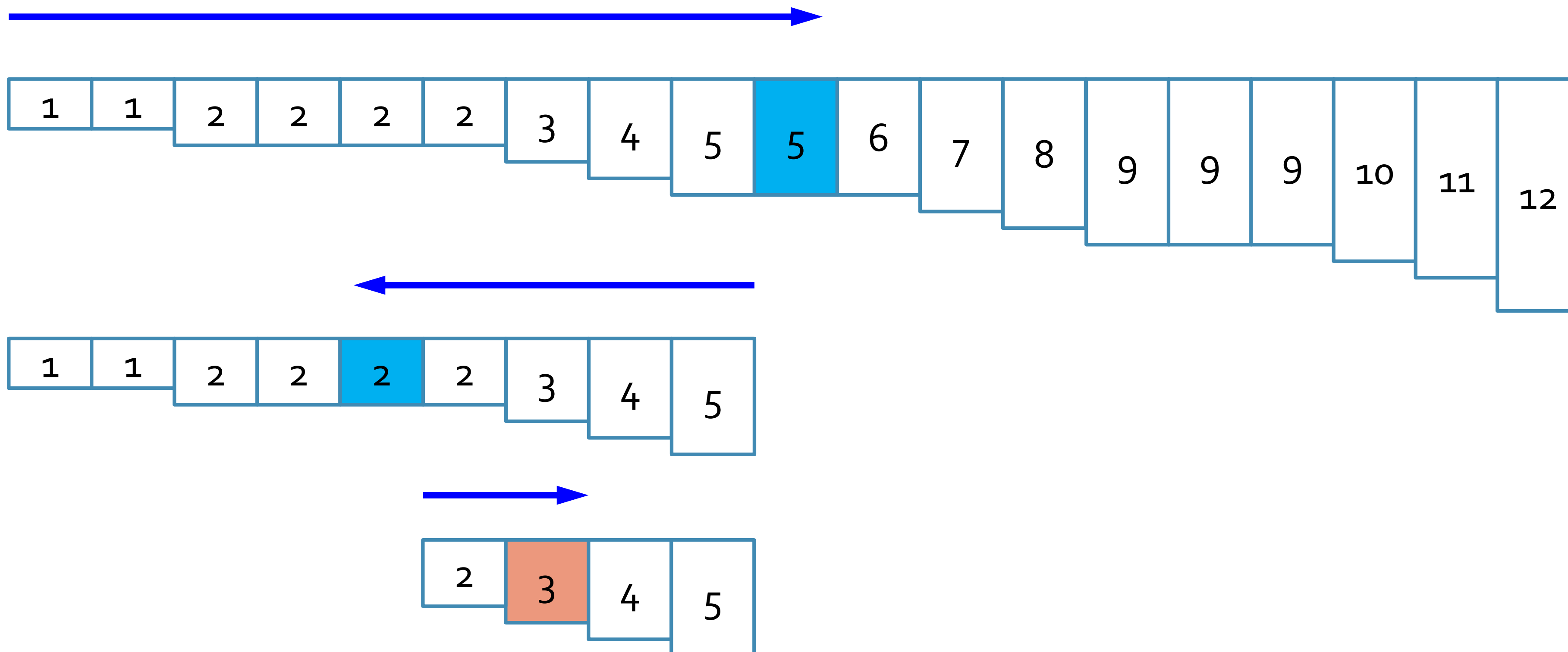
Алгоритм L: линейный поиск

```
unsigned
linear_search(const int *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", разделяй и властвуй.



Алгоритм B: бинарный поиск

```
unsigned
binary_search(const int *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)(const int *, 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

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(const int *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: необычный 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 `63`. 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(const unsigned *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:

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

- 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 $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$
- 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

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 $A(x) = x^3 + 3x^2 + 4x + 7$ и $B(x) = x^3 + 5x^2 + x + 4$
- Вам необходимо подсчитать их произведение

```
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));

  // напишите здесь код, подсчитывающий ret = lhs * rhs

  return ret;
}
```

- Оцените асимптотику умножения двух полиномов длины n

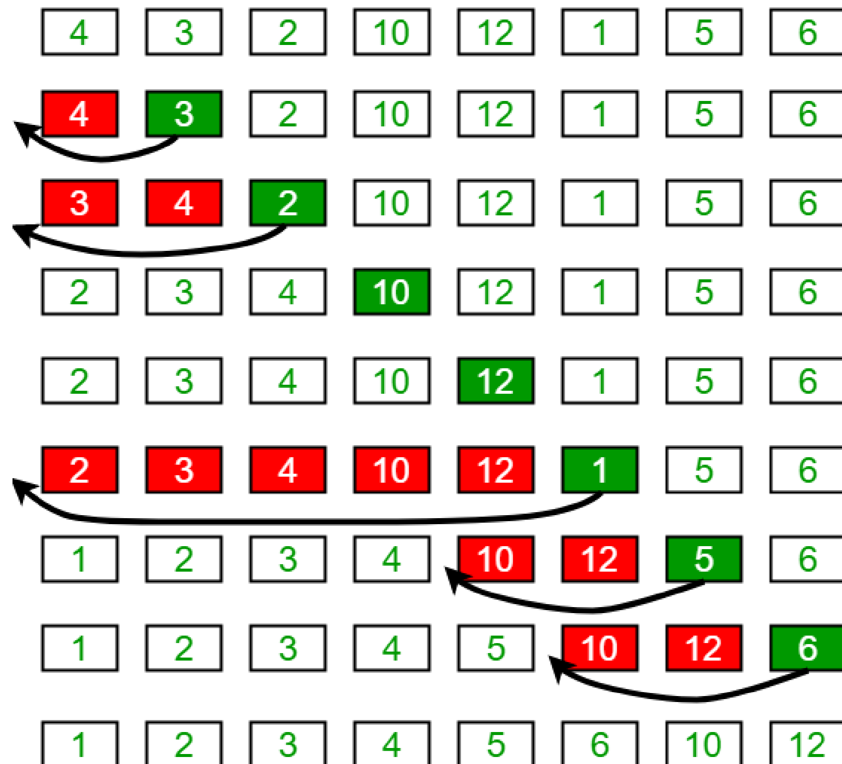
Перемножение: алгоритм Карацубы

- 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: $C(x) = A_1 B_1 x^n + \left((A_1 + a_0)(b_1 + b_0) - a_1 b_1 + a_0 b_0\right) x + a_0 b_0$
- 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



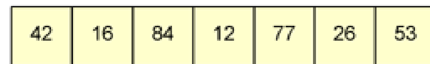
- Обычно первая идея, которая приходит человеку это отсортировать массив вставками
- Инвариант алгоритма: левая часть массива до n всегда отсортирована
- На каждом шаге n увеличивается
- При необходимости все красные элементы перемещаются
- Какая асимптотическая сложность у такого алгоритма?

Алгоритм I: сортировка вставками

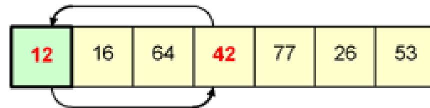
- Реализация потребует двух функций

```
unsigned moveright(int *arr, int key, unsigned last) {

// 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;
  }
}
```

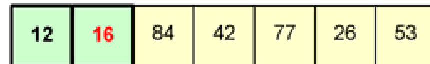
Идея сортировки выбором



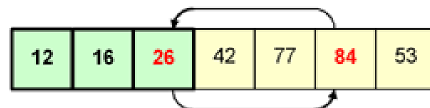
The array, before the selection sort operation begins.



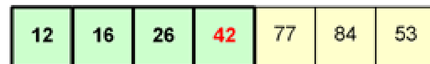
The smallest number (**12**) is swapped into the first element in the structure.



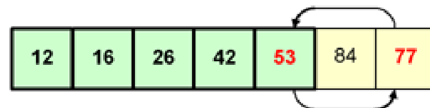
In the data that remains, **16** is the smallest; and it does not need to be moved.



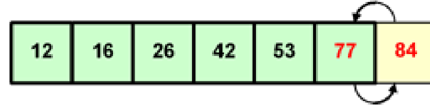
26 is the next smallest number, and it is swapped into the third position.



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.*

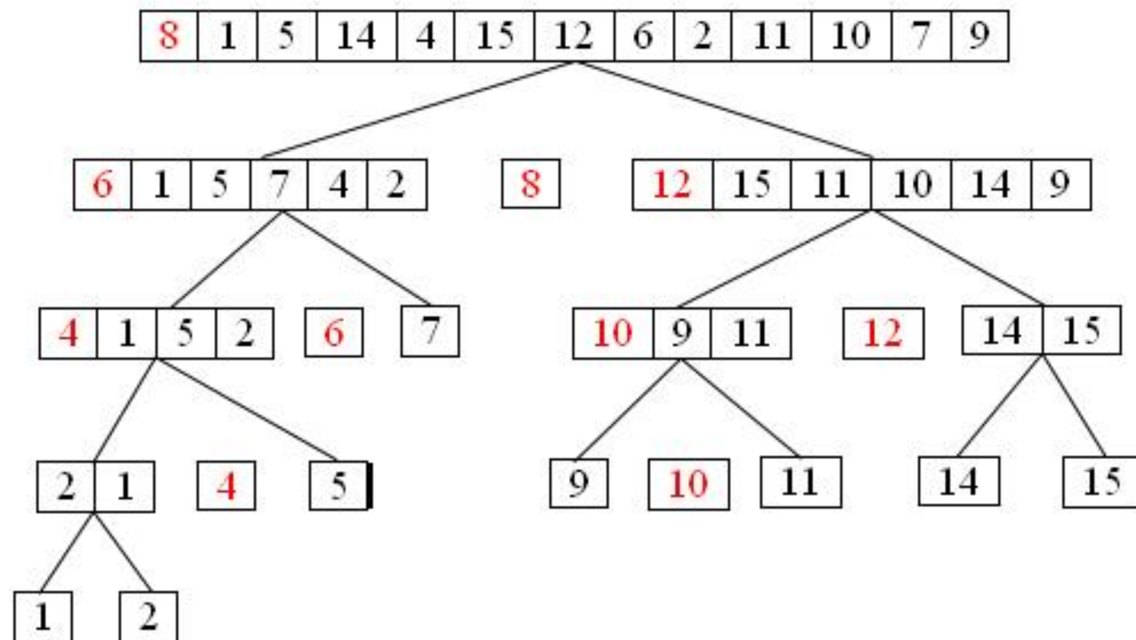
- Вторая частая идея это сортировка выбором
- Найдём в массиве минимальный элемент и обменяем его местами с текущим
- Переместимся к следующему элементу
- Будет ли этот алгоритм асимптотически лучше вставок?

Алгоритм SE: сортировка выбором

- Алгоритм использует уже известный алгоритм L

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

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

void selsort(int *arr, unsigned len) {

// напишите код для сортировки выбором

}
```

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) {
// 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);
}
```

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

Обобщение бинарного поиска

```
typedef int (*cmp_t)(const void *key, const void *elt);

void *bsearch_(const void *key, const void *base, int num, int size, cmp_t cmp) {
  const char *pivot;
  int result;

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

Problem UC: поиск в массиве double

- У вас есть массив `double *arr`
- Используйте обобщённый бинарный поиск чтобы найти в нём `double val`

```
double *search(double *sorted_arr, double val) {
// напишите здесь код вызывающий bsearch
}
```

- Для этого также напишите предикат сравнения чисел с плавающей точкой

Problem CS: обобщение alg SE

- Обобщите алгоритм SE

```
void selsort(int *arr, unsigned len);
```

- Теперь сигнатура изменится

```
typedef int (*cmp_t)(const void *key, const void *elt);

void *
selsort(const void *arr, int num, int size, cmp_t cmp) {
  // напишите тут код
}
```

- Обратите особое внимание на функцию swap

Problem CQ: обобщение сортировки

- Подобно тому как был обобщён бинарный поиск и сортировка вставками, обобщите быструю сортировку
- Проиллюстрируйте работу алгоритма сортировки с массивом double
- Это задание можно взять на дом

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

Problem POW2

- Используйте идею из алгоритма POWM (см. также $[TAOCP\ Algorithm\ 4.6.3A]$)
- Напишите функцию для возведения в любую степень матриц 2x2

```
void pow2x2 (unsigned n[2][2], unsigned k) {
  // TODO: ваш код здесь
  // код модифицирует матрицу n, возводя её в степень k
}
```

- Оцените асимптотическую сложность этого алгоритма

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
```

- Многомерные массивы и указатели на массивы редко нужны, но мы к ним ещё вернёмся чуть позже. Пока это задел на будущее.

Чтение сложных типов в языке С

- 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

HWS: Timsort (optional)

- Разберитесь в алгоритме сортировки слиянием (merge sort) и реализуйте его простейший случай (только для целых)
- Многие современные алгоритмы для лучшей производительности комбинируют слияние и вставки
- Прочитайте статью https://en.wikipedia.org/wiki/Timsort
- Реализуйте обобщённый (для произвольных типов) алгоритм Timsort используя применённый на этом семинаре подход с void*
- Все результаты отсылать на почту

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