МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»
Кафедра вычислительной техники

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №3
ПО ДИСЦИПЛИНЕ «ИНФОРМАТИКА»
«Сортировка и поиск»

Факультет: АВТ
Группа: ДТ-460а
Студент: Пантюхин Артём Евгеньевич

Преподаватель:
Копылова Оксана Андреевна

Новосибирск, 2025 г.

Содержание

Задание

Алгоритм сортировки реализовать в виде функции, возвращающей в качестве результата характеристику трудоемкости алгоритма (например, количество сравнений). Получить трудоемкость для различных значений N=1000,5000,10000. Сравнить с теоретической оценкой.

8. Сортировка «демоном Максвелла»:

Ёмкость с газом разделена на две половины непроницаемой стенкой. В стенке есть отверстие, которое снабжено специальным устройством, которое назовём демоном Максвелла. Демон устроен так, что через отверстие он из левой части в правую пропускает только быстрые (горячие) молекулы газа, а из правой в левую — медленные (холодные) молекулы.

Теоретические сведения

Сортировка «демоном Максвелла» действительно относится к бинарным сортировкам, исходя из идеи о дроблении массива на множество подмассивов. Наиболее близким к данному алгоритму из известных нам является алгоритм быстрой сортировки (quick sort).

Алгоритм быстрой сортировки имеет трудоёмкость O(n^2) в худшем случае, и O(n log n) в среднем.

Идея алгоритма быстрой сортировки заключается в том, чтобы, взяв какое-либо из чисел исходного массива за точку опоры, менять числа слева и справа от него местами до тех пор, пока все числа слева не окажутся меньше эталонного, а все числа справа – больше (или равны).

После этого, части массива слева и справа от числа выносятся в отдельные массивы, для которых алгоритм вызывается рекурсивно, до тех пор, пока подмассив состоит более чем из одного значения.

Далее отсортированные сегменты исходного массива собираются с сохранением порядка, образуя отсортированный массив.

В рамках реализации алгоритма дополнительных массивов не создаётся – вместо этого область работы ограничивается индексами начального и конечного элемента, в результате чего пространственная сложность алгоритма составляет O(n).

Программа

В силу того, что обработка пользовательского ввода не является частью алгоритма сортировки, её исходный код вынесен в отдельный файл и прокомментирован отдельно.

Основные идеи алгоритма

Реализация алгоритма отличается от обычного алгоритма быстрой сортировки отсутствием «отправной точки». Вместо этого, мы располагаем элементы слева и справа от «центрального» индекса, основываясь на сравнении максимального значения из левой части с минимальным значением из правой. В случае, если число из левой части оказывается больше – меняем эти числа местами. Повторяем цикл до тех пор, пока все числа из левой части не будут меньше или равны каждому из чисел правой части, после чего рекурсивно применяем данннный алгоритм для сортировки подмассивов.

Например, для массива {1,5,8,6,4} работа алгоритма будет выглядеть так:
{1,5,8} {6,4}, 4 < 8
{1,5,4} {6,8}, 6 > 5
{1,5} {4}, 4 < 5 _и_ {6} {8}, 8 > 6
{1} {4} {5} {6} {8}
{1,4,5,6,8}

Переменные

actions – счётчик, служит для оценки трудоёмкости алгоритма.
values – входной массив, amount – длина массива.
low, high, low_end, high_start – индексы границ области работы.
v1, v2 – указатели, использующиеся для замены чисел по адресу в памяти.
В рамках min_ и max_value – value – искомое значение, out – указатель на v1 или v2 для вывода не только значения, но и его адреса.
Алгоритм получает на вход массив чисел с плавающей точкой (values), индекс первого элемента (low) (в случае первого вызова - 0), и индекс последнего элемента (high) (в случае первого вызова – длина массива-1).

Для рекурсивного вызова используются переменные low_end и high_start, обозначающие конец левой части массива и начало правой соответственно.

Расчёт трудоёмкости алгоритма

Взяв размерность массива N = 2^k, цикл выполнится **n** раз.
В рамках цикла, функции нахождения максимального и минимального значения выполнятся $\mathbf{\log_2(n)}$ раз в сумме (каждую итерацию рассчётный массив делится на 2).
Функции нахождения максимума и минимума имеют трудоёмкость **3n** (множитель будет отброшен далее)
Функция обмена значений имеет трудоёмкость **3**, независимо от набора значений.

Таким образом, получаем трудоёмкость **N*(3N+3)*log2(n)+3**, на бесконечности:

$\mathbf{O^{max}(N^2\log_2(N))}$.

Сопоставив теоретические данные с полученными в ходе работы программы, отметим:

N	10	1000	5000	10000	100000
Рассчётная	332	9965784	307192809	1328771237	166096404744
Полученная	384	1553826	38055100	150099252	15009628117

Видно, что трудоёмкость алгоритма стремится к $\mathbf{N^2}$. Предположу, что причиной такого поведения является то, что теоретическую трудоёмкость мы рассчитывали для худшего из возможных сценариев.

«Составные части» программы

Форматированный вывод массива:

```
void print_array(const double* const values, ull amount) {
    printf("{ ");
    for (ull i = 0; i < amount; i++) {
        if (i) {
            printf(", ");
        }
        printf("%0.3lf", values[i]);
    }
    printf("}\n");
}
```

Определение максимального и минимального числа на заданном промежутке:

```
double max_value(double* const values, ull start, ull end, double** out) {
    double value = values[start];
    *out = &(values[start]);
    for (ull i = start+1; i <= end; i++) {
        if (values[i] > value) {
            value = values[i];
            *out = &(values[i]);
        }
    }
    return value;
}

double min_value(double* const values, ull start, ull end, double** out) {
    double value = values[start];
    *out = &(values[start]);
    for (ull i = start+1; i <= end; i++) {
        if (values[i] < value) {
            value = values[i];
            *out = &(values[i]);
        }
    }
    return value;
}
```

Замена значений по адресам:

```
void swap_values(double* v1, double* v2) {
    double buffer = *v1;
```

```
    *v1 = *v2;
    *v2 = buffer;
}
```

Рекурсивная функция сортировки:

```
void demon_sort(double* const values, ull low, ull high) {
    ull low_end = (high+low)/2;
    ull high_start = low_end+1;
    double *v1, *v2;
    while (min_value(values, high_start, high, &v1) < max_value(values, low, low_end, &v2)) {
        swap_values(v1, v2);
    }
    if (high > high_start) {
        demon_sort(values, high_start, high);
    }
    if (low_end > low) {
        demon_sort(values, low, low_end);
    }
}
```

Оценка тродоёмкости алгоритма: (модификация сортировки)

```
ull demon_sort(double* const values, ull low, ull high) {
    ull low_end = (high+low)/2;
    ull high_start = low_end+1;
    double *v1, *v2;
    ull actions = 2;
    while (min_value(values, high_start, high, &v1) < max_value(values, low, low_end, &v2)) {
        swap_values(v1, v2);
        actions += 3*(high-low+4);
    }
    //3*(end-start+1) - трудоёмкость min_, max_value в худшем случае. == 3N
    //Внутри цикла добавим три действия для замены переменной.
    actions += 3*(high-low+1);
//Соберём информацию из рекурсивных вызовов

    if (high > high_start) {
        actions += demon_sort(values, high_start, high);
    }
    if (low_end > low) {
        actions += demon_sort(values, low, low_end);
    }
    actions += 2;
    return actions;
}
```

Исходный код программы с комментариями.

[main.c - см. приложение 1.]

[parsing.h, parsing.c - см. приложение 2.]

Пример работы программы

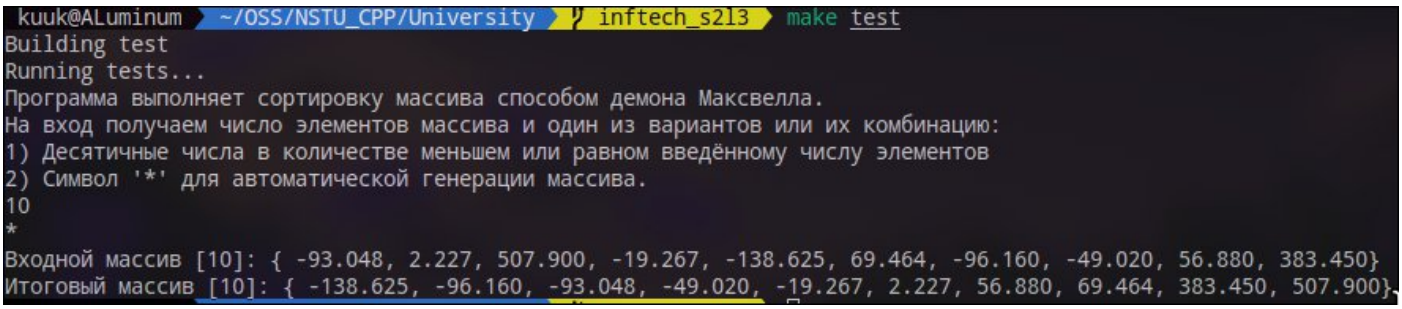

Рисунок 1 – Пример работы программы, автогенерация данных.

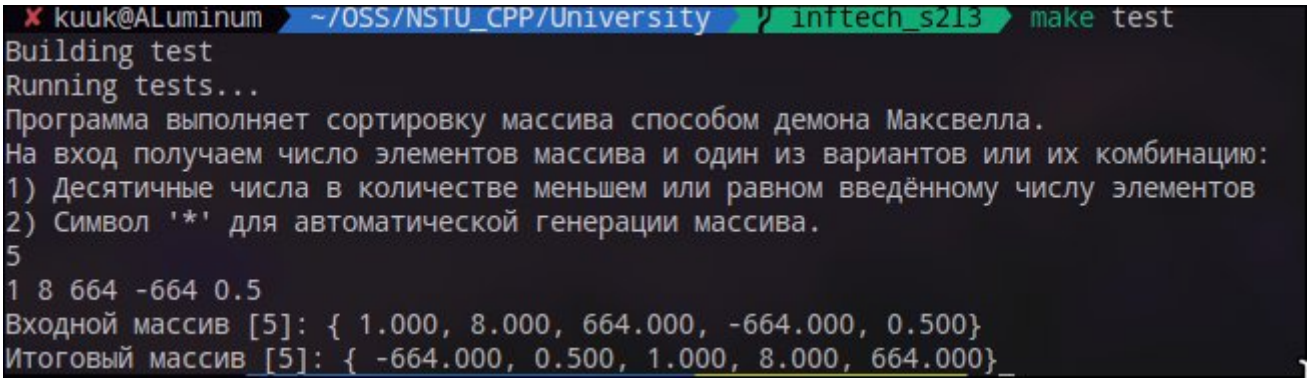

Рисунок 2 - Пример работы программы, ручной ввод.

Рисунок 3 – Пример работы программы, ввод из файла, автогенерация недостающего.

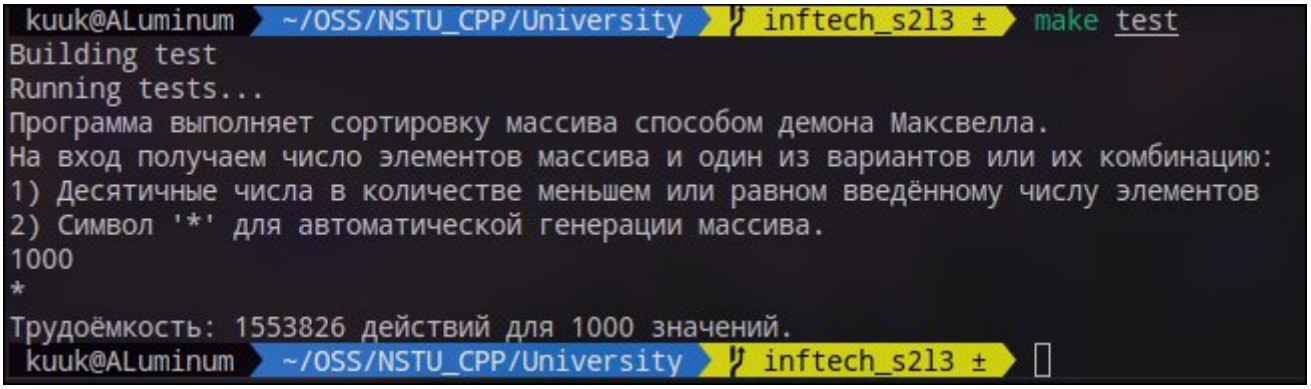

Рисунок 4 – Трудоёмкость для 1000 элементов.

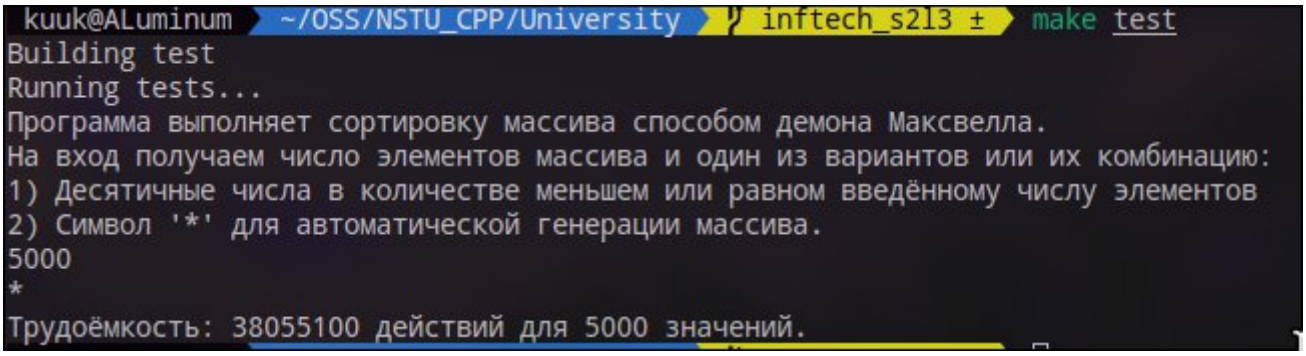

Рисунок 5 – Трудоёмкость для 5000 элементов.

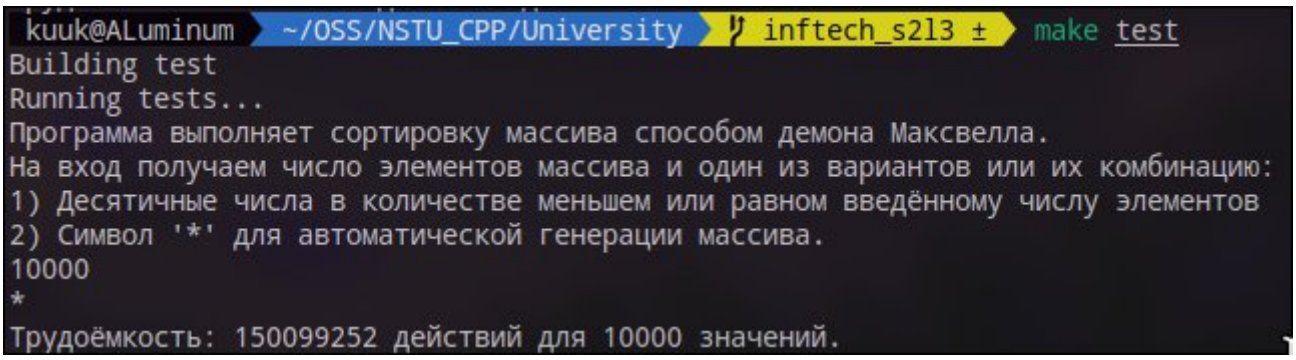

Рисунок 6 – Трудоёмкость для 10000 элементов.

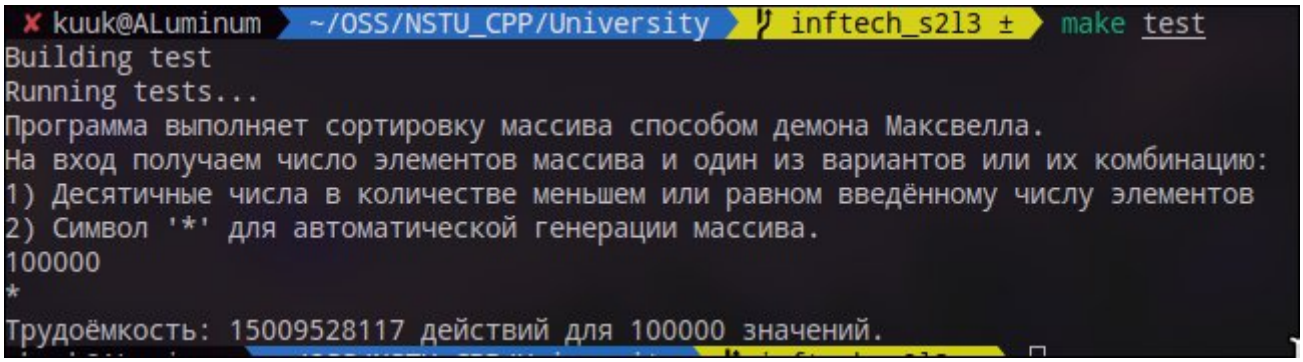

Рисунок 7 – Трудоёмкость для 100000 элементов.

Выводы

В процессе выполнения лабораторной работы были изучены алгоритмы сортировки переносом, обменом, распределением, разделением/слиянием а так же алгоритмы для сортировки ветвлений.

В рамках реализации алгоритма получен практический опыт использования целиком и/или составных частей сортировок разделением/слиянием, сортировок обменом.

Трудоёмкость, полученная в ходе рассчётов, отличается от полученных при выполнении программы значений. Предположительно, причиной является то, что по мере увеличения количества элементов входного массива, всё менее и менее вероятным становится возникновение «худшего случая».

Тем не менее, алгоритм не показал ни теоретических, ни практических преимуществ над алгоритмом «быстрой» сортировки, несмотря на использование схожих техник.

Список литературы

1. Подбельский В.В., Фомин С.С. Программирование на языке Си: Учеб.пособие. – 2-е доп. Изд. – М.: Финансы и статистика, 2004. – 600 с.

2. Романов Е. Л. Си/Си++. От дилетанта до профессионала. Электронное учебное пособие по дисциплинам "Информатика", "Программирование", "Технология программирования" для студентов 1–2 курсов направления 230100 : учеб. пособие / Е. Л. Романов. – Новосибирский государственный технический университет, № гос регистрации 0321000528, 2010. - 581 с.

3. Си/Си++ от дилетанта до профессионала [Электронный ресурс]. URL: http://ermak.cs.nstu.ru/cprog/HTML/index.htm (дата обращения 01.10.2022).

Приложение 1. Исходный код main.c

```
#include <locale.h>
#include <stdio.h>

#include "parsing.h"

#define GREETINGS_STRING \
"Программа выполняет сортировку массива способом демона Максвелла.\n\
На вход получаем число элементов массива и один из вариантов или их комбинацию:\n\
1) Десятичные числа в количестве меньшем или равном введённому числу элементов\n\
2) Символ '*' для автоматической генерации массива.\n"

//Все индексы и длины будут храниться в ull, чтобы мы могли больше вместить.
//Прозвище этому типу назначаем просто для удобства.
typedef unsigned long long ull;

/*
max_ и min_value абсолютно идентичны по своей структуре:
0) Они обе небезопасны, и им обоим индифферентно, что в них суют - каждая триггернёт сегфолт при неправильном использовании
1) На входе получаем некоторый массив, индексы начала и конца и указатель,
чтобы не только выдать значение, но и его адрес в памяти.
2) Первый элемент служит инициализатором для наших внутренних переменных
3) Все остальные до последнего включительно проверяются в рамках, собственно,
нахождения максимума/минимума.
*/
double max_value(double* const values, ull start, ull end, double** out) {
    double value = values[start];
    *out = &(values[start]);
    for (ull i = start+1; i <= end; i++) {
        if (values[i] > value) {
```

```
            value = values[i];
            *out = &(values[i]);
        }
    }
    return value;
}

double min_value(double* const values, ull start, ull end, double** out) {
    double value = values[start];
    *out = &(values[start]);
    for (ull i = start+1; i <= end; i++) {
        if (values[i] < value) {
            value = values[i];
            *out = &(values[i]);
        }
    }
    return value;
}

//Меняем значения по двум указателям, что тут скажешь.
void swap_values(double* v1, double* v2) {
    double buffer = *v1;
    *v1 = *v2;
    *v2 = buffer;
}

/*
Алгоритм сортировки на деле практически не отличается от qsort,
разве что вместо опоры на конкретное значение мы утверждаем, что в
серединке должно быть, условно,
среднее, относительно которого и будут меняться элементы.

Сортировка рекурсивная, идея следующая:
Ставим условный барьер в серединке нашей области работы (массив
делится на две половинки, потом ещё на две и т.д.)
Далее "ждём"
Если исходить из идеи демона Максвелла -
```

в это время все наиболее энергонасыщенные частицы перетекают в одну область,
наименее энергонасыщенные - в другую.

Мы же точно знаем, что число элементов в массиве не изменится,
поэтому с чистой совестью сразу меняем лучших кандидатов между собой
(В каком-то смысле, энергонасыщенная частица движется куда активнее,
так что это даже логично)

Как только мы достигли баланса в разрезе наших двух половинок,
радостно делим их ещё раз,
и так пока каждая частичка не будет окружена этими "барьерами"

В результате кучи перемещений получаем отсортированный массив.
*/
```
ull demon_sort(double* const values, ull low, ull high) {
    //Определяем "серединку" нашего массива
    //Для чётных массивов всё понятно, для нечётных - нижняя часть будет на единичку больше
    ull low_end = (high+low)/2;
    //Мы не хотим, чтобы при дроблении массива в обоих частях оказался один и тот же элемент.
    ull high_start = low_end+1;
    double *v1, *v2;
    ull actions = 2;
    //Расшифровываем, что происходит:
    /*
    Пока _наименьшее_ из значений правой части массива меньше, чем наибольшее из левой части -
    меняем их местами.
    */
    while (min_value(values, high_start, high, &v1) < max_value(values, low, low_end, &v2)) {
        swap_values(v1, v2);
        actions += 3*(high-low+4);
    }
    //3*(end-start+1) - трудоёмкость min_, max_value  == 3N
    //Внутри цикла добавим три действия для замены переменной.
```

```
    actions += 3*(high-low+1);
    //Тут у нас получается ситуация, что в левой половинке каждое из чисел меньше, чем каждое из правой.
    /*
    Чтобы не выдумывать чудес, рекурсивно вызываем этот же алгоритм до тех пор,
    пока область работы не станет меньше двух элементов.
    */
    if (high > high_start) {
        actions += demon_sort(values, high_start, high);
    }
    if (low_end > low) {
        actions += demon_sort(values, low, low_end);
    }
    actions += 2;
    return actions;
}

//Форматированный вывод массива, используем дважды, так что почему бы и не вынести
void print_array(const double* const values, ull amount) {
    printf("{ ");
    for (ull i = 0; i < amount; i++) {
        if (i) {
            printf(", ");
        }
        printf("%0.3lf", values[i]);
    }
    printf("}\n");
}

int main(void) {
    setlocale(LC_ALL, "Russian");
    //Здороваемся, объясняем что происходит
    printf(GREETINGS_STRING);
    ull amount;
    double* values;
```

```
    //Если при обработке ввода что-то пошло не так - просто не будем пытаться.
    if (process_input(&amount, &values)) {
        printf("Входной массив [%llu]: ", amount);
        print_array(values, amount);
        //Вызов сортировки.
        /*
        amount-1 потому, что это раньше мы работали с
        "количеством элементов", и в циклах говорили что i строго меньше его.
        Теперь мы работаем с индексами, помня, что < стало <=,
        а само число на единичку меньше.
        */
        ull actions = demon_sort(values,0, amount-1);
        printf("Итоговый массив [%llu]: ", amount);
        print_array(values, amount);
        printf("Трудоёмкость: %llu действий для %llu значений.\n", actions, amount);
        /*
        В комментариях к parsing.c есть строчка про это, но напомним:
        наш массив значений выделяется в куче, это динамическая память,
        и чистить её мы должны сами.
        К слову, именно поэтому мы не чистим её в else{} - она выделялась
        во время обработки ввода, и если обработка поймала ошибку - значит и памяти выделено не было.
        Впрочем, последнее утверждение уже зависит от реализаций, но у меня - вот так.
        */
        free(values);
    } else {
        printf("Данные некорректны/Ошибка выделения памяти.\n");
    }
    return 0;
}
```

Приложение 2. Исходный код parsing

parsing.h

```
#ifndef PARSING_H
#define PARSING_H

#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int flush_to_num_or_spec(char* ch);
int process_input(unsigned long long* amount, double** values);

#endif
```

parsing,c

```
#include "parsing.h"

#define AUTO_CHAR '*'

int flush_to_num_or_spec(char* ch) {
    int sign = 0;
    while (!isdigit(*ch) && *ch != AUTO_CHAR && *ch != EOF) {
        sign = *ch == '-';
        *ch = fgetc(stdin);
    }
    //вернём 1 если увидели отрицательное число
    return sign && isdigit(*ch);
}

/*
Итак, как же работает наш чудо-парсер?
*/
int process_input(unsigned long long* amount, double** values) {
    /*
    Базово отвалидируем то, что в нас засунули,
    чтобы не поймать сегфолт по входным == NULL
    Нужно это только потому, что с поинтерами работаем.
    */
    if (!amount || !values) {
        return 0;
    }
```

```
*amount = 0;
*values = 0;
//Создаём статический буфер чаров, будем переиспользовать.
char in_buffer[BUFSIZ] = {};
int buffer_pos = 0;

//Инициализируем "грязный" чар.
char ch = 'a';
//Пропускаем весь ввод, пока не встретим число
while (!isdigit(ch) && ch != EOF) {
   ch = fgetc(stdin);
}
//Копируем всё число в наш буфер
while (isdigit(ch) && ch != EOF) {
   in_buffer[buffer_pos++] = ch;
   ch = fgetc(stdin);
}
//После числа ставим 0 ('\0'), гарантируя, что буфер - валидная строка.
in_buffer[buffer_pos] = 0;

/*
Конвертируем буфер в целое беззнаковое
У этого чуда может произойти переполнение, но тут уже увы.
В любом случае, дальше по ходу работы увидим значение.
*/
/*
   При вводе из файла, можно поймать прикол
   если в файле нет цифр или вообще ничего нет
   in_buffer окажется пустым
   strtoull выдаст нам "что-то"
   а так, оставляем 0 и радуемся.
*/
if (buffer_pos) {
   *amount = strtoull(in_buffer,NULL, 10);
}

/*
Если юзер вписал 0 - ну, тут медицина бессильна
Вообще можно и return сразу писать,
но хорошей практикой считается
один вход и один выход из функции, не считая валидации входных.
*/
if (*amount) {
   /*
```

```
ALARM! ALARM! ALARM!
Выделяем память под наш массив значений
*/
*values = malloc(sizeof(double)*(*amount));
/*
Если выделили успешно, *values будет != 0
В противном случае malloc вернёт нам NULL,
который по сути (void*) 0, и мы это проверяем,
чтобы случайно не поймать сегфолт.
*/
if (*values) {
   unsigned long long consumed = 0;
   while (consumed < *amount && ch != AUTO_CHAR && ch != EOF) {
      //юзверь ввёл число, не мусор, не попросил помочь.
      //чистим буфер, чтобы чисто было.
      buffer_pos = 0;
      memset(in_buffer, 0,BUFSIZ);
      /*
      Пропускаем весь невалидный мусор
      до тех пор, пока юзверь не введёт что ему сказали.
      */
      /*
      Если мы всё же нашли число, а не ударились в EOF || AUTO_CHAR
      То оставим знак на будущее
      */
      int negative = flush_to_num_or_spec(&ch);
      //Копируем встреченное число в буфер
      while ((isdigit(ch) || ch == '.') && ch != EOF) {
         in_buffer[buffer_pos++] = ch;
         ch = fgetc(stdin);
      }
      /*
      Тут можно было бы проверить, что в буфере что-нибудь есть,
      но на деле, раз мы вошли сюда по циферке,
      значит будет как минимум она.
      */
      if (ch != AUTO_CHAR && ch != EOF) {
         (*values)[consumed++] = strtod(in_buffer, NULL);
         if (negative) {
            (*values)[consumed-1] *= -1;
         }
      }
   }
   //Теперь на тему генерации
```

```
        //srand() обновит наш хэш для генерации
        //time() суём туда чтобы хэш был всегда разным
        srand((unsigned int) time(0));
        /*
        До тех пор, пока мы не забьём массив значениями - суём туда произвольные
числа
        Да, rand() производит только инты, поэтому мы делим ранд на ранд,
        конвертированные в (double), получая
        double в рамках rvalue, и даже с дробными числами!

        Если пользователь сдался на полпути - его ввод останется на своих местах
        Если говорить о файле, тут интереснее
        В файле символ поражения - не только *, но и EOF, и если
        мы этот самый EOF поймали, логика нисколько не меняется.
        */
        for ( ; consumed < *amount; consumed++) {
            //Генерация собственно числа
            double r1 = rand() % 8942;
            double r2 = rand() % 113;
            (*values)[consumed] = r1;
            if (r2 != 0) {
                (*values)[consumed] /= r2;
            }
            //Генерация знака этому чудесному числу
            (*values)[consumed] *= rand() % 2 ? -1.0f : 1.0f;
        }
      }
   }
   return *values != NULL;
}
```