Московский государственный университет
имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики

Отчет по заданию №1

«Методы сортировки»

Вариант 2 / 2 / 2 / 5

Выполнил:
студент 105 группы
Домнин С. В.

Преподаватель:
Русол А. В.

Москва
2023

Содержание

Постановка задачи

Была поставлена задача реализации двух методов сортировки, а также экспериментальное их сравнение. В качестве двух методов сортировки были даны метод простого выбора (Selection sort) и метод пирамидальной сортировки (Heapsort). Каждый из методов сортировки должен быть реализован в виде отдельной функции, принимающей два параметра: массив длины N, а также саме длине этого массива N. Функция должна быть типа void, то есть не возвращать ничего, а в качестве побочного эффекта реализовывать сортировку необходимым методом.

В качестве требуемого порядка сортировки задачей была реализация сортировки в порядке невозрастания элеметнов в массиве. Для каждого массива необходимо динамически выделить память в зависимости от введенной длинны массива N и впоследствии ее очистить. Необходимо провести сравнение на одних и тех же массивах, рассмотреть массивы различной длины (N = 10, 100, 1000, 10000).

Необходиимо реализовать случайную генерацию массивов необходимой структуры, а именно провести сравнение на массивах, в которых:

- элементы упорядочены в прямом порядке,
- элементы упорядочены в обратном порядке,
- элементы не упорядочены.

Массив длины N представляет из себя набор 64-разрядных целых чисел (long long int).

В качестве результата необходимо реализовать методы сортировки, которые можно запускать, используя параметры коммандной строки, получить количество сравнений и перемещений каждой из сортировок при каждом параметре, сравнить эти значения с ассимптотическим выкладками из литературы, а также описать приведенные методы реализации кода.

Результаты экспериментов

Переходя к поставленной задаче сравнения двух методов сортировки, приведены таблицы количества сравнений и перемещений, выполненных каждой из сортировок в процессе выполнения функций. Сравнение было проведено на массивах различной длины (при N = 10, 100, 1000, 10000). Важно отметить, что в рамках тестирования массива одного типа и одной длины массивы на вход обоим функция подавались одинаковые данные. Помимо различия длины массива, было произведено тестирование на массивах различной природы (1 - полностью упорядоченный массив; 2 - полностью упорядоченный в обратном порядке массив; 3, 4 - случайные массивы).

n	**Параметр**	**Номер сгенерированного массива**				**Среднее значение**
		1	2	3	4	
10	Сравнения	45	45	45	45	45
	Перемещения	9	9	9	9	9
100	Сравнения	4950	4950	4950	4950	4950
	Перемещения	99	99	99	99	99
1000	Сравнения	499500	499500	499500	499500	499500
	Перемещения	999	999	999	999	999
10000	Сравнения	49995000	49995000	49995000	49995000	49995000
	Перемещения	9999	9999	9999	9999	9999

Таблица 1: Результаты работы Selection sort.

Сортировка методом простого выбора является простой и понятной сортировкой, которая может быть эффективной на небольших массивах данных. Однако, на больших массивах она может быть неэффективной, так как имеет сложность $O(n^2)$ в худшем, среднем и лучшем случаях. Это означает, что время выполнения этой сортировки будет расти квадратично с увеличением размера массива.

Число сравнений ключей, очевидно, не зависит от начального порядка ключей и равно $(n^2 - n)/2$ [2], так как на первом проходе нужно сделать (n-1) сравнений, на втором проходе - (n-2) сравнений и так далее, пока не останется один элемент.

Количество перемещений при сортировке методом простого выбора на отсортированном массиве размера n равно (n-1), так как на каждом проходе нужно произвести одно перемещение элемента в начало массива.

Таким образом, среднее значение количества сравнений и перемещений при сортировке методом простого выбора на случайно заполненном массиве размера n равно $(n^2)/2$, так как на каждом проходе нужно сравнить каждый элемент со всеми остальными и переместить найденный максимальный элемент в начало массива.

Эксперементальные данные сортировки методом простого выбора хорошо соотносятся с теоретическими результатами, т.к. предполагается, что количество сравнений и перемещений для массива из n элементов постоянно.

n	Параметр	Номер сгенерированного массива				Среднее значение
		1	2	3	4	
10	Сравнения	70	52	60	62	61
	Перемещения	30	21	25	26	26
100	Сравнения	1380	1132	1268	1250	1258
	Перемещения	640	516	584	575	579
1000	Сравнения	20416	17632	19144	19128	19080
	Перемещения	9708	8316	9072	9064	9043
10000	Сравнения	273912	243392	258194	258504	258501
	Перемещения	131956	116696	124097	124252	124251

Таблица 2: Результаты работы Heapsort.

Основной идеей алгоритма является построение двоичной кучи из элементов массива, в которой каждый элемент является не меньше (или не больше) своих потомков. Далее происходит поочередное извлечение корня кучи (максимального или минимального элемента, в зависимости от задачи) и перенос его в отсортированный конец массива. После каждого извлечения корня перестраивается куча, чтобы убрать из нее извлеченный элемент и восстановить свойства "кучи".

Согласно [1], сложность алгоритма пирамидальной сортировки оценивается как O(n· log2 N). Оценка количества сравнений пирамидальной сортировки 3·N·log2 N, оценка количества обменов: N·log2 N. Сравнивая теоретические значения с полученными эксперементально, можно сделать вывод, что средние значения для перемещений получились немного больше ожидаемого теоретического результата.

Можно обратить внимание на различие результатов двух методов сортировки как в теоретических значениях, так и в результате, отраженном в таблице. Количество сравнений пирамидальной сортировки при больших значениях N значительно меньше количества сравнений при сортировке методом простого выбора, но при малых значениях N в выборе пирамидальной сортировки нет смысла.

Структура программы и спецификация функций

Рассмотрим функцию сортировки методом простого выбора. Алгоритм заключается в том, чтобы проходить по всем элементам массива и находить в нём максимальный элемент. Затем этот элемент меняется местами с последним элементом неотсортированной части массива. Таким образом, на каждой итерации в неотсортированной части массива уменьшается количество элементов, среди которых нужно искать максимальный элемент. Алгоритм продолжается до тех пор, пока неотсортированная часть массива не исчезнет полностью.

```
void selectionSort(long long *mas, int N)
{
    long long cmp = 0, mov = 0;

    for (int i = 0; i < N - 1; i++)
    {
        int min_idx = i;
        for (int j = i + 1; j < N; j++)
        {
            cmp += 1;
            if (mas[j] > mas[min_idx])
            {
                min_idx = j;
            }
        }

        swap(&mas[i], &mas[min_idx]);
        mov += 1;
    }
    printf("Selection sort - %lld - %lld \n" , cmp , mov);
}
```

При каждом проходе по массиву сравниваются два элемента - текущий и максимальный. Если текущий элемент больше максимального, то максимальный элемент обновляется. Также происходит обмен найденного максимального элемента с последним элементом неотсортированной части массива. Для подсчёта количества сравнений и обменов заводятся две переменные-счетчика, которые увеличиваются соответственно при каждом сравнении и обмене элементов. В конце функции результат выводится на экран.

```
void maxHeapify(long long* arr, int i, int n,
long long* cmp, long long* moves) {
    int l = 2*i+1;
    int r = 2*i+2;
    int largest;
    if(l<n && arr[l]<arr[i]) {
        largest=l;
    } else {
        largest=i;
```

```
    }
    (*cmp)++;
    if(r<n && arr[r]<arr[largest]) {
        largest=r;
    }
    (*cmp)++;
    if(largest!=i) {
        swap(&arr[i], &arr[largest]);
        (*moves)++;
        maxHeapify(arr, largest, n, cmp, moves);
    }
}
```

Функция heapify принимает на вход массив, размерность массива N, индекс i и указатели на переменные для подсчета количества сравнений cmp и перемещений mov. Она преобразует поддерево с корнем i в двоичную кучу, если i не удовлетворяет свойству кучи. Для этого она сравнивает значение элемента с его дочерними элементами и, если необходимо, меняет их местами. Затем она рекурсивно вызывает себя для преобразования затронутого поддерева.

```
void buildMaxHeap(long long* arr, int n,
long long* cmp, long long* moves) {
    for(int i=n/2-1; i>=0; i--) {
        maxHeapify(arr, i, n, cmp, moves);
    }
}
```

Из [1] известно, что все элементы подмассива arr с arr[N/2+1] до arr[N] являются листьями дерева, поэтому каждый из них можно считать одноэлементной пирамидой, с которой можно начать процесс построения. Процедура buildMaxHeap проходит по остальным узлам и для каждого из них выполняет процедуру maxHeapify. Функция buildMaxHeap также приниимает указатели на переменные для подсчета количества сравнений cmp и перемещений mov.

```
void heapSort(long long* arr, int n) {
    long long cmp = 0, mov = 0;
    buildMaxHeap(arr, n, &cmp, &mov);
    for(int i=n-1; i>0; i--) {
        swap(&arr[0], &arr[i]);
        mov++;
        maxHeapify(arr, 0, i, &cmp, &mov);
    }
    printf("Heapsort - %lld - %lld \n" , cmp , mov);
}
```

Функция heapSort использует функцию buildMaxHeap для построения двоичной кучи из неупорядоченного массива и функцию maxHeapify для повторного преобразования кучи после каждого извлечения максимального элемента. Сначала она строит кучу путем перегруппировки элементов массива с помощью функции buildMaxHeap. Затем она последовательно извлекает макси-

мальный элемент из кучи и перемещает его в конец массива, а затем вызывает maxHeapify для уменьшенной кучи.

В функцие heapSort просиходит инициализация переменных для подсчета количества сравнений cmp и перемещений mov. После завершения сортировки функция выводит значения этих переменных в консоль.

```
void swap(long long *a, long long *b)
{
    long long temp = *a;
    *a = *b;
    *b = temp;
}
```

Для упрощения программы была реализована функция swap, которая меняет местами два элемента массива типа long long, переданных ей в качестве указателей.

```
long long *createArray(int N, char type)
{
    long long *a = (long long *)malloc(N * sizeof(long long));

    srand(time(NULL));

    for (int i = 0; i < N; i++)
    {
        a[i] = (long long) rand() * rand() * rand() * rand() * rand();
    }

    if (type == 's')
    {
        qsort(a, N, sizeof(long long), comp);
        reversed(a, N);
    }
    else if (type == 'r')
    {
        qsort(a, N, sizeof(long long), comp);
    }

    return a;
}
```

По условию для генерации исходных массивов требуется реализовать отдельную функцию, создающую в зависимости от заданного параметра и заданной длины конкретный массив, в котором:

- элементы уже упорядочены;
- элементы упорядочены в обратном порядке;
- расстановка элементов случайна.

Для этого реализована функция createArray. Она принимает на вход размер массива N и тип генерации элементов type. Она создает массив из N элементов типа long long и заполняет его случайными 64-битными целыми числами с помощью функции rand(). Если type равен 's', то функция сортирует полученный массив в порядке невозрастания с помощью функции qsort и последующим переворотом массива в обратном порядке. Если type равен 'r', то массив сортируется с помощью qsort в порядке неубывания. Наконец, функция возвращает указатель на созданный массив.

```
void reversed(long long *arr, int N)
{
    for (int i = 0; i < N / 2; i++)
    {
        swap(&arr[i], &arr[N - i - 1]);
    }
}
```

Функция reversed переворачивает массив в обратном порядке. При проходе половины массива каждый элемент меняется с соответсвующим ему элементом с конца массива.

```
int comp(const void * a, const void * b)
{
    const long long *x = (const long long *)a;
    const long long *y = (const long long *)b;
    return (*x > *y) - (*x < *y);
}
```

Для функции qsort необходимо реализовать дополнительную функцию-компоратор comp. Она возвращает реузльтат сравнения двух элементов для сортировки в порядке неубывания.

```
void printArray(long long *mas, int N)
{
    for (int i = 0; i < N; i++)
        if (i < N - 1)
            printf("%lld ", mas[i]);
        else
            printf("%lld\n", mas[i]);
}
```

Для вывода массива реализована функция printArray, которая проходит по массиву, выводит каждый элемент с пробелом, кроме последнего, и переходит на новую строку.

Отладка программы, тестирование функций

Для проверки корректности работы написанных функций сортировки было проведено обширное тестирование на массивах различных размеров и структур. Для создания тестовых массивов были использованы случайная генерация элементов, а также вручную созданные массивы, в которых элементы располагались в разном порядке.

В случаях, когда результат работы функций не соответствовал ожидаемому, была проведена пошаговая отладка, позволяющая выявить ошибки в выполнении определенных условий и операций. Благодаря доступности и обилию литературных источников, которые подробно описывают принципы работы и реализацию необходимых методов, процесс отладки был более эффективным и позволил добиться корректности работы функций.

Для метода **Selection sort** достаточно завести два счетчика и в нужных местах сортировки добавить их увеличение.

Реализация **Heapsort** для сортировки массива требует использования двух основных методов - метода создания пирамиды и метода сортировки. Так как метод создания пирамиды рекуррентный, то обычной переменной-счётчика в функции недостаточно. Поэтому адрес на него нужно передавать в качестве параметра функции.

Для тестирования функций была реализована случайная генерация массива, используя умножение функции **rand()** с явным приведением типа **(long long) rand() * rand() * rand() * rand() * rand()**. Для получения отсортированного массива в прямом и обратном порядках, использовались встроенные функции сортировки **qsort** и написанная функция переворота массива **reversed**.

Для удобства взаимодействия с программой, была реализована обработка аргументов командной строки. Это позволяет задать длину генерируемого массива, тип его генерации, вызвать справку по соответствующему параметру – **help** в командной строке и сообщить об ошибке ввода, в случае некорректных аргументов командной строки.

Анализ допущенных ошибок

В ходе выполнения работы было выевлено ошибочное использование переменных для подсчета количества сравнений и обменов. Также при проверке программы на отсортироанном массиве была выявлена опечатка, прерывающая нормальный ход программы.

Список литературы

[1] Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. Второе издание. — М.: «Вильямс», 2005.

[2] Вирт Н. Алгоритмы и структуры данных. — М.: Мир, 1989.