Бюджетное учреждение высшего образования   
Ханты-Мансийского автономного округа   
«Сургутский государственный университет»

Политехнический институт

Кафедра автоматики и компьютерных систем

**Отчет**

по лабораторной работе № 3

по дисциплине «Алгоритмы и структуры данных»

Выполнил: студент группы 609-21,

Шумилов И. Д.

Принял: старший преподаватель кафедры АиКС

Назаров Е.В.

Сургут

2024 г.

**Цель работы:** изучить базовые алгоритмы поиска, исследовать их свойства, закрепить навыки структурного программирования.

**Общее задание на лабораторную работу:**

1. Разработать функции, реализующие три алгоритма поиска (прыжками обязательно, еще два – по выбору).
2. Исследовать алгоритмы поиска: построить и сравнить зависимости среднего количества сравнений в случаях успешного и неуспешного поиска для заданных алгоритмов.
3. Найти зависимость оптимальной величины «прыжка» от размера последовательности.
4. Составить отчет, в котором привести графики полученных зависимостей, анализ свойств алгоритмов и выводы по работе.

**Индивидуальное задание на лабораторную работу:**

1. Сортировка прыжками (одноуровневая и двухуровневая)
2. Сортировка Фибоначчи
3. Интерполяционная сортировка

*Листинг 1. Код разрабатываемой программы:*

*#include <iostream>*

*enum Search { Jump1, Jump2, Fib, Inter };*

*int compN;*

*int oneLvlJumpSearch(int arr[], int n, int x, int step) {*

*int prev = 0;*

*int fixStep = step;*

*while (compN++, arr[std::min(step, n) - 1] < x) {*

*prev = step;*

*step += fixStep;*

*if (prev >= n)*

*return -1;*

*}*

*while (compN++, arr[prev] < x) {*

*prev++;*

*if (prev == std::min(step, n))*

*return -1;*

*}*

*if (compN++, arr[prev] == x)*

*return prev;*

*return -1;*

*}*

*int oneLvlJumpSearch(int arr[], int n, int x) {*

*return oneLvlJumpSearch(arr, n, x, sqrt(n));*

*}*

*int twoLvlJumpSearch(int arr[], int n, int x) {*

*int step1 = (int)sqrt(n), step2 = (int)sqrt(step1);*

*int prev1 = 0, prev2 = step1;*

*while (compN++, arr[std::min(prev2, n) - 1] < x) {*

*prev1 = prev2;*

*prev2 += step1;*

*if (prev1 >= n)*

*return -1;*

*}*

*prev2 = prev1 + step2;*

*int end = prev1 + step1;*

*while (compN++, arr[std::min(prev2, end) - 1] < x) {*

*prev1 = prev2;*

*prev2 += step2;*

*if (prev1 >= end)*

*return -1;*

*}*

*while (compN++, arr[prev1] < x) {*

*prev1++;*

*if (prev1 == std::min(prev2, end))*

*return -1;*

*}*

*if (compN++, arr[prev1] == x)*

*return prev1;*

*return -1;*

*}*

*int jumpSearch(int arr[], int n, int x, int lvl) {*

*if (lvl == 1)*

*return oneLvlJumpSearch(arr, n, x);*

*else if (lvl == 2)*

*return twoLvlJumpSearch(arr, n, x);*

*return -1;*

*}*

*int fibonacciSearch(int arr[], int n, int x) {*

*int f2 = 0;*

*int f1 = 1;*

*int f = f2 + f1;*

*while (f < n) {*

*f2 = f1;*

*f1 = f;*

*f = f2 + f1;*

*}*

*int offset = -1;*

*while (f > 1) {*

*int i = std::min(offset + f2, n - 1);*

*if (compN++, arr[i] < x) {*

*f = f1;*

*f1 = f2;*

*f2 = f - f1;*

*offset = i;*

*} else if (compN++, arr[i] > x) {*

*f = f2;*

*f1 = f1 - f2;*

*f2 = f - f1;*

*} else*

*return i;*

*}*

*if (compN++, f1 && arr[offset + 1] == x)*

*return offset + 1;*

*return -1;*

*}*

*int interpolationSearch(int arr[], int low, int high, int x) {*

*int pos;*

*if (low <= high && x >= arr[low] && x <= arr[high] && high != low) {*

*pos = low + (((double)(high - low) / (arr[high] - arr[low])) \* (x - arr[low]));*

*if (compN++, arr[pos] == x)*

*return pos;*

*if (compN++, arr[pos] < x)*

*return interpolationSearch(arr, pos + 1, high, x);*

*if (compN++, arr[pos] > x)*

*return interpolationSearch(arr, low, pos - 1, x);*

*}*

*return -1;*

*}*

*int getMax(int\* arr, int N) {*

*int tmp = arr[0];*

*for (int i = 1; i < N; i++) {*

*if (tmp < arr[i])*

*tmp = arr[i];*

*}*

*return tmp;*

*}*

*void countSort(int\* arr, int N, int exp) {*

*int\* output = new int[N];*

*int count[10] = { 0 };*

*for (int i = 0; i < N; i++)*

*count[(arr[i] / exp) % 10]++;*

*for (int i = 1; i < 10; i++)*

*count[i] += count[i - 1];*

*for (int i = N - 1; i >= 0; i--) {*

*output[count[(arr[i] / exp) % 10] - 1] = arr[i];*

*count[(arr[i] / exp) % 10]--;*

*}*

*for (int i = 0; i < N; i++)*

*arr[i] = output[i];*

*}*

*void radixSort(int\* arr, int N) {*

*int max = getMax(arr, N);*

*for (int exp = 1; max / exp > 0; exp \*= 10)*

*countSort(arr, N, exp);*

*}*

*void genArr(int arr[], int n) {*

*int tmp;*

*for (int i = 0; i < n; i++) {*

*tmp = rand();*

*if (tmp % 2 == 0)*

*arr[i] = tmp;*

*else*

*arr[i] = tmp - 1;*

*}*

*radixSort(arr, n);*

*}*

*int countCompN(int arr[], int n, int x, Search s) {*

*switch (s) {*

*case Jump1:*

*compN = 0;*

*jumpSearch(arr, n, x, 1);*

*break;*

*case Jump2:*

*compN = 0;*

*jumpSearch(arr, n, x, 2);*

*break;*

*case Fib:*

*compN = 0;*

*fibonacciSearch(arr, n, x);*

*break;*

*case Inter:*

*compN = 0;*

*interpolationSearch(arr, 0, n - 1, x);*

*break;*

*}*

*return compN;*

*}*

*void getAvg(int arr[], Search s, int iters = 100, int unsuccess = 0) {*

*int total = 0;*

*for (int n = 10000; n <= 100000; n += 10000) {*

*arr = new int[n];*

*for (int j = 0; j < iters; j++) {*

*genArr(arr, n);*

*int x = arr[(int)((double)rand() / RAND\_MAX \* n)] + unsuccess;*

*total += countCompN(arr, n, x, s);*

*}*

*std::cout << total / iters << std::endl;*

*total = 0;*

*}*

*std::cout << std::endl;*

*}*

*void getMod(int arr[], int iters = 100) {*

*int total = 0;*

*for (int n = 100; n <= 500; n += 100) {*

*std::cout << "Array size: " << n << std::endl;*

*arr = new int[n];*

*for (int s = 1; s <= n / 2; s++) {*

*for (int i = 0; i < iters; i++) {*

*genArr(arr, n);*

*int x = arr[(int)((double)rand() / RAND\_MAX \* n)];*

*oneLvlJumpSearch(arr, n, x, s);*

*total += compN;*

*compN = 0;*

*}*

*std::cout << total / iters << std::endl;*

*total = 0;*

*}*

*std::cout << "\n\n" << std::endl;*

*}*

*}*

*int main(int argc, char\*\* argv) {*

*int \*arr = nullptr;*

*bool successful = false;*

*srand((unsigned)time(NULL));*

*getAvg(arr, Jump1, 500, 0);*

*getAvg(arr, Jump2, 500, 0);*

*getAvg(arr, Fib, 500, 0);*

*getAvg(arr, Inter, 500, 0);*

*std::cout << std::endl;*

*getAvg(arr, Jump1, 500, 1);*

*getAvg(arr, Jump2, 500, 1);*

*getAvg(arr, Fib, 500, 1);*

*getAvg(arr, Inter, 500, 1);*

*// getMod(arr, 1000);*

*}*

**Пояснения к программе**

Помимо необходимых функций поиска была разработана функция заполнения массива случайными четными числами (чтобы в будущем было проще делать неуспешные поиски, просто пытаясь найти нечетное число) и была взята поразрядная сортировка, разработанная в предыдущей лабораторной работе, для сортировки сгенерированного случайного массива.

**Исследование алгоритмов**

Были проведены замеры среднего количества операций для всех видов поиска при успешном и неудачном поисках для разных размеров последовательностей.

Таблица 1 – Зависимости среднего количества сравнений в случаях успешного и неуспешного поиска

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Размер массива | Среднее количество сравнений алгоритма поиска (S(n)) | | | | | | | |
| Прыжками (одноуровневая) | | Прыжками (двухуровневая) | | Фибоначчи | | Интерполяционная | |
| Успех | Неудача | Успех | Неудача | Успех | Неудача | Успех | Неудача |
| 10000 | 104 | 102 | 60 | 62 | 17 | 20 | 6 | 7 |
| 20000 | 244 | 247 | 145 | 144 | 36 | 42 | 13 | 16 |
| 30000 | 420 | 420 | 248 | 245 | 56 | 66 | 19 | 24 |
| 40000 | 621 | 627 | 367 | 359 | 75 | 89 | 25 | 33 |
| 50000 | 842 | 848 | 497 | 482 | 95 | 113 | 31 | 41 |
| 60000 | 1099 | 1101 | 640 | 621 | 114 | 138 | 37 | 50 |
| 70000 | 1364 | 1379 | 786 | 767 | 133 | 162 | 42 | 59 |
| 80000 | 1648 | 1667 | 948 | 927 | 153 | 188 | 48 | 69 |
| 90000 | 1940 | 1968 | 1118 | 1100 | 173 | 213 | 53 | 78 |
| 100000 | 2255 | 2288 | 1295 | 1273 | 193 | 238 | 59 | 87 |

Рисунок 1 – График зависимости количества операций от размера в поисках прыжками

Рисунок 2 – График зависимости количества операций от размера в поисках Фибоначчи и интерполяционном

По графикам видно, что при поиске прыжками количество операций сравнения при успешном и неуспешном поиске отличается очень незначительно. Однако при поиске Фибоначчи или интерполяционном поиске при успешном поиске количество операций значительно меньше. Также следует отметить, что при поисках прыжками временная сложность приближена к графику O(n^2), при этом двухуровневый поиск значительно быстрее одноуровневого, при поиске Фибоначчи временная сложность близится к графику O(log n), а при интерполяционном к графику O(log log n).

**Определение зависимости оптимальной величины прыжка от размера**

Рисунок 3 – График зависимости размера последовательности от величины прыжка

Используя функцию getMod() (листинг 1) были получены данные о количестве операций анализ графика зависимости на рис. 3 можно прийти к выводу, что оптимальная величина прыжка близится к значению, значения меньше этого занимают значительно больше операций, значения больше постепенно начинают занимать все больше и больше операций.

**Сравнительная оценка реализованных алгоритмов**

Таблица 2 – Сравнительная оценка реализованных алгоритмов

|  |  |
| --- | --- |
| Алгоритм поиска | Средняя сложность |
| Поиск прыжками (любой) | O() |
| Поиск Фибоначчи | O(log n) |
| Интерполяционный поиск | O(log log n) |

**Вывод:** были изучены алгоритмы одноуровневого и двухуровневого поиска прыжками, поиска Фибоначчи, интерполяционного поиска и были исследованы их свойства.