Звіт

Автор: Момот Р. КІТ-119а

Дата: 26.04.2020

ЛАБОРАТОРНА РОБОТА № 9. АЛГОРИТМИ ПРОСТИХ ПОШУКІВ

Мета: одержати навички та закріпити знання при виконанні операцій пошуку.

Індивідуальне завдання

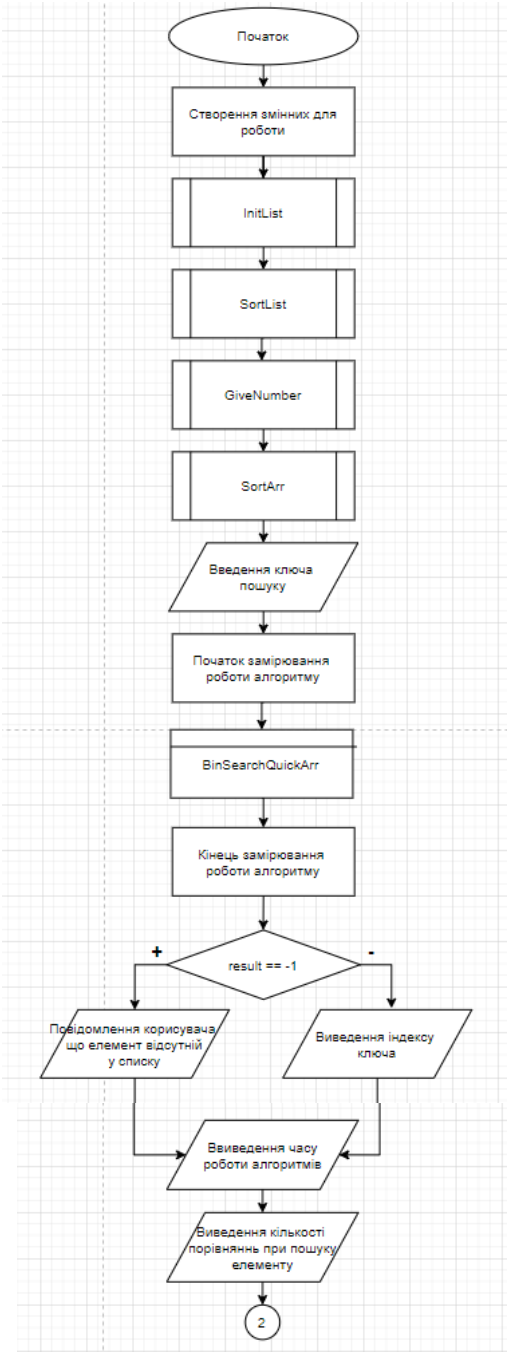
Розробити та налагодити програму, в якій реалізувати два алгоритми пошуку у відповідності до завдання. Порівняти алгоритми за часом роботи.

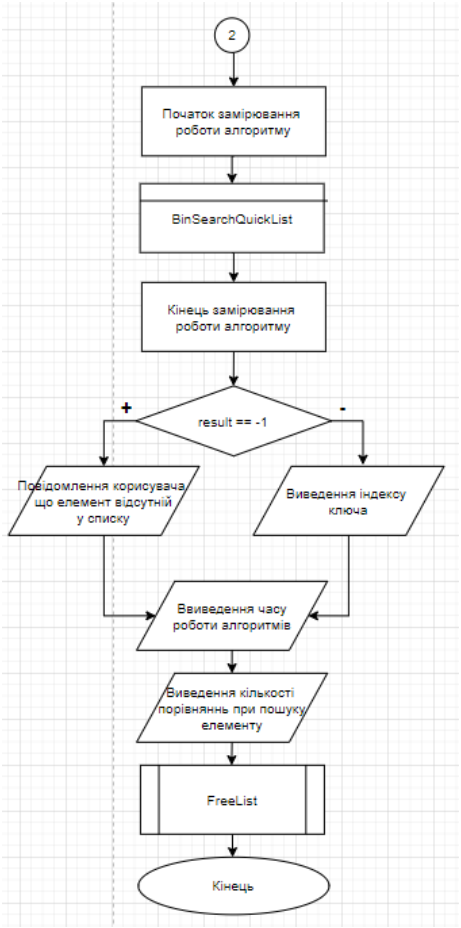
На етапі тестування для кожного з алгоритмів визначити кількість порівнянь у наборі даних з різною кількістю елементів (20, 100, 1000, 10000) визначити час пошуку, заповнити таблицю, побудувати графіки, зробити висновки.

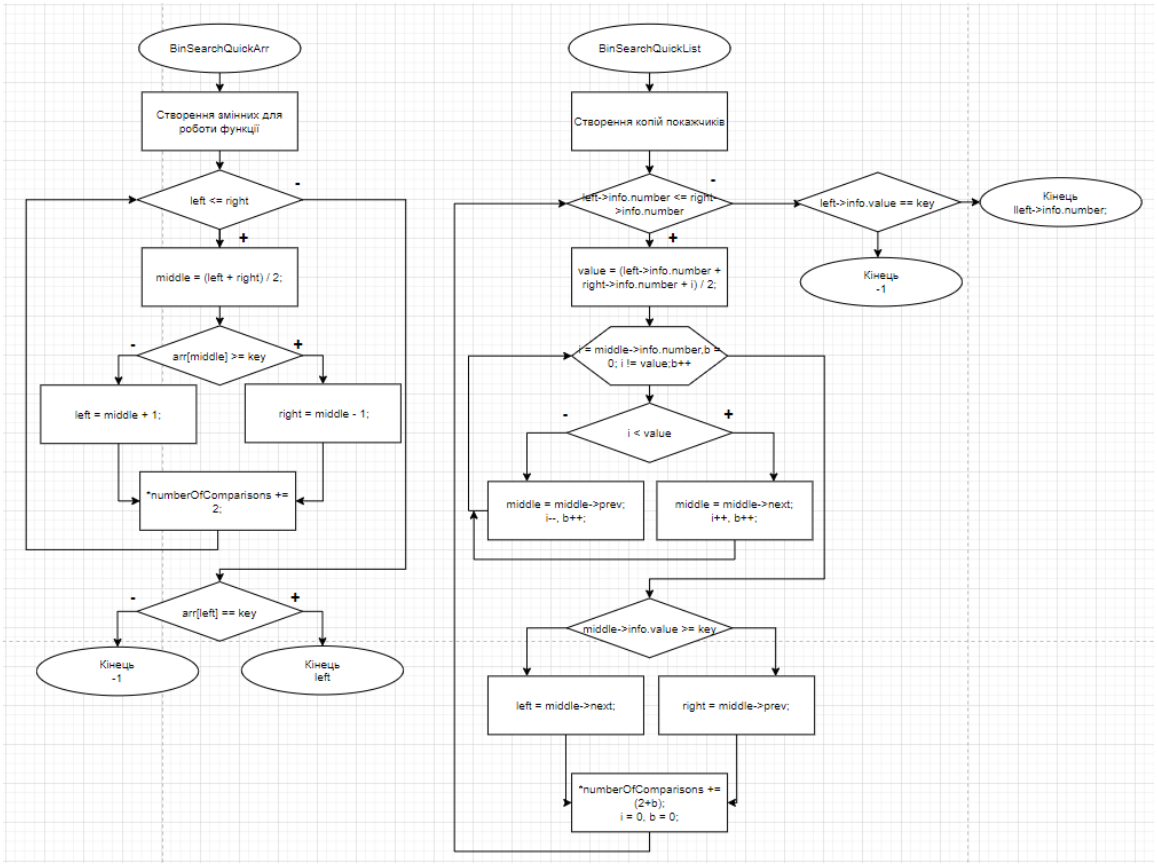
Передбачена робота з цілими числами.

13) Двійковий пошук у масиві та лінійному списку.

Блок-схема алгоритму програми






BinSearchQuickArr
Створення змінних для роботи функції
left <= right
middle = (left + right) / 2;
arr[middle] >= key
left = middle + 1;
right = middle - 1;
*numberOfComparisons += 2;
arr[left] == key
Кінець -1
Кінець left
BinSearchQuickList
Створення копій покажчиків
left->info.number <= right->info.number
left->info.value == key
Кінець lleft->info.number;
Кінець -1
value = (left->info.number + right->info.number + i) / 2;
i = middle->info.number,b = 0; i != value;b++
i < value
middle = middle->prev; i--, b++;
middle = middle->next; i++, b++;
middle->info.value >= key
left = middle->next;
right = middle->prev;
*numberOfComparisons += (2+b); i = 0, b = 0;

Текст програми

```
#define _CRT_SECURE_NO_WARNINGS
#define CRTDBG_MAP_ALLOC
#define DEBUG_NEW new(_NORMAL_BLOCK, FILE, __LINE)
#define new DEBUG_NEW
#include <crtdbg.h>
#include <stdlib.h>
#include <iostream>
#include <locale.h>
#include <chrono>

struct Element
{
	int value;		//число
	int number;		//номер в списке
};
struct List
{
	Element info;
	List* next, * prev;
};

using namespace std;

List* Init();
void InitList(List*, List*);
void CreateListAndArr(List*, int*, int);
void GiveNumber(List*);
void FreeList(List*);

void SortList(List*, List*);
void SortArr(int*, int);

int BinSearchQuickArr(int*, int, int, int*);
int BinSearchQuickList(List*,List*, int, int, int*);

void OutputArr(int*, int);
void OutputList(List*);

int main() {
	setlocale(LC_ALL, "Rus");
	struct List* h = Init();
	struct List* t = Init();
	const int SIZE = 20;		//размер массивов
	int array[SIZE];		//массив
	int* pArray = array;		//указатель на массив
	int key;			//искомое число
	int result;			//результат поиска
	int comparison = 0;		//количество сравнений
	int* pComparison = &comparison;

	InitList(h, t);
	CreateListAndArr(h, pArray, SIZE);
	SortList(h, t);
	GiveNumber(h);
	SortArr(pArray, SIZE);

	OutputList(h);
	//OutputArr(pArray, SIZE);

	cout << "Введите число, которое надо найти в массиве: ";
	cin >> key;

	auto beginClock = chrono::steady_clock::now();
```

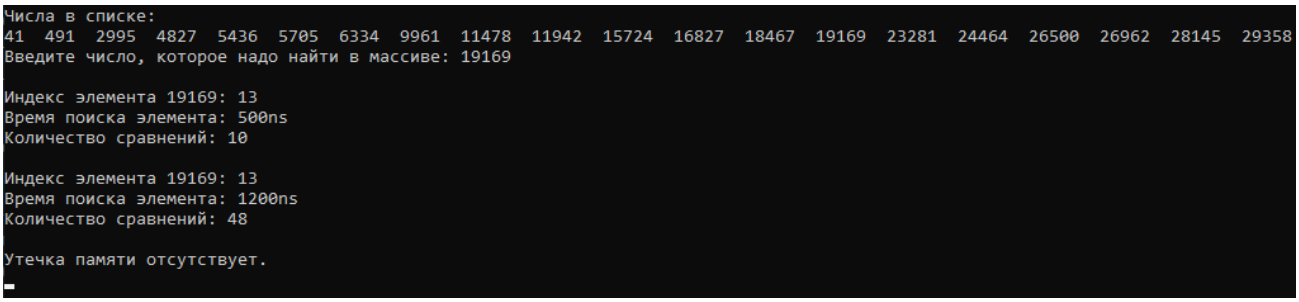
```
	result = BinSearchQuickArr(array, key, SIZE, pComparison);
	auto endClock = chrono::steady_clock::now();
	auto resultClock = chrono::duration_cast<chrono::nanoseconds>(endClock - beginClock);
	if (result == -1) cout << "Элемент в массиве отсутствует." << endl;
	else cout << "\nИндекс элемента " << key << ": " << result << endl;
	*pComparison += 2;
	cout << "Время поиска элемента: " << resultClock.count() << "ns" << endl;
	cout << "Количество сравнений: " << *pComparison << endl;

	*pComparison = 0;

	beginClock = chrono::steady_clock::now();
	result = BinSearchQuickList(h, t, key, SIZE, pComparison);
	endClock = chrono::steady_clock::now();
	resultClock = chrono::duration_cast<chrono::nanoseconds>(endClock - beginClock);
	if (result == -1) cout << "Элемент в массиве отсутствует." << endl;
	else cout << "\nИндекс элемента " << key << ": " << result << endl;
	*pComparison += 2;
	cout << "Время поиска элемента: " << resultClock.count() << "ns" << endl;
	cout << "Количество сравнений: " << *pComparison << endl;

	FreeList(h);

	if (_CrtDumpMemoryLeaks()) cout << "\nУтечка памяти обнаружена." << endl;
	else cout << "\nУтечка памяти отсутствует." << endl;

	return 0;
}


List* Init() {
	struct List* p = (struct List*)malloc(sizeof(List)); /**- Виділення пам'яті на наступний
елемент списка. */
	return p;                  /**- Повернення покажчика. */
}
void InitList(List* h, List* t) {
	h->next = t;               /**- Переміщення покажчиків. */
	h->prev = NULL;
	t->next = NULL;
	t->prev = h;
}
void CreateListAndArr(List* h,int* arr, int size) {
	struct List* p;
	int value = 0, temp;

	while (value != size)
	{
		p = (List*)malloc(sizeof(List));
		temp = rand();
		p->info.value = temp;
		p->next = h->next;
		p->next->prev = p;
		p->prev = h;
		h->next = p;

		arr[value] = temp;
		value++;
	}
}
void OutputList(List* h)
{
	struct List* p = h->next;

	cout << "Числа в списке:" << endl;
	while (p->next)
	{
```

```
			cout << p->info.value << "  ";
			p = p->next;
		}
		cout << endl;
}
void OutputArr(int* array, int size)
{
		for (size_t i = 0; i < size; i++)
		{
			cout << array[i] << "  ";
		}
		cout << endl;
}
void SortList(List* h, List* t) {
		bool pr = 0;
		struct List* p = h->next;

		do {
			pr = false;
			p = h->next;
			while (p->next)
			{
				if (p->info.value > p->next->info.value&& p->next != t)
				{
					p->next->prev = p->prev;
					p->next->next->prev = p;
					p->prev = p->next;
					p->next = p->next->next;
					p->prev->next = p;
					p->prev->prev->next = p->prev;
					pr = 1;
				}
				p = p->next;
			}
		} while (pr == 1);
}
void SortArr(int* array, int size)
{
		bool pr;
		int temp;

		do {
			pr = 0;
			for (size_t i = 0; i < size - 1; i++)
			{
				if (*(array + i) > *(array + i + 1))
				{
					temp = *(array + i);
					*(array + i) = *(array + i + 1);
					*(array + i + 1) = temp;
					pr = 1;
				}
			}
		} while (pr == 1);
}
int BinSearchQuickArr(int* arr, int key, int size, int* numberOfComparisons)
{
		int middle, left = 0, right = size;		// начальные значения границ
		while (left <= right)					//пока интервал не сузится до 0
		{
			middle = (left + right) / 2;		//середина интервала
			if (arr[middle] >= key) right = middle - 1;
			else left = middle + 1;
			*numberOfComparisons += 2;
		}
		if (arr[left] == key) return left;
```

```
	else return -1;
}
int BinSearchQuickList(List* h, List* t, int key, int size, int* numberOfComparisons)
{
	List* left = h->next;
	List* right = t->prev;
	List* middle = h->next;
	int value;
	int i = 1, b = 0;

	while (left->info.number <= right->info.number)		//пока интервал не сузится до 0
	{
		value = (left->info.number + right->info.number + i) / 2;
		for (i = middle->info.number,b = 0; i != value;b++)//находим середину интервала
		{
			if (i < value)
			{
				middle = middle->next;
				i++, b++;
			}
			else
			{
				middle = middle->prev;
				i--, b++;
			}
		}

		if (middle->info.value >= key)
		{
			right = middle->prev;
		}
		else
		{
			left = middle->next;
		}
		*numberOfComparisons += (2+b);
		i = 0, b = 0;
	}

	if (left->info.value == key)
	{
		return left->info.number;
	}
	else return -1;
}
void GiveNumber(List* h)
{
	struct List* p = h->next;
	int i = 0;

	while (p->next)
	{
		p->info.number = i;
		p = p->next;
		i++;
	}
}
void FreeList(List* h)
{
	struct List* p = h;

	while (p)
	{
		h = p;
		p = p->next;
		free(h);
```

```
        }
}
```

Результати роботи програми



Результати тестування алгоритма пошуку для масива

Кількість елементів	20	100	1000	10000
Кількість порівнянь	11	16	22	29
Час пошуку*	533	667	933	933

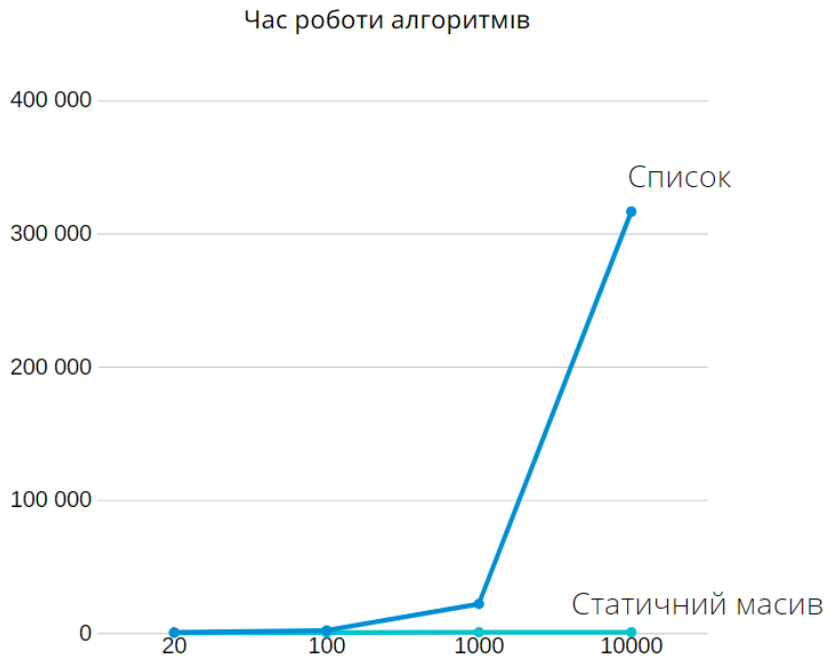
Результати тестування алгоритма пошуку для списку

Кількість елементів	20	100	1000	10000
Кількість порівнянь	148	217	2021	20026
Час пошуку*	867	2267	22167	316767

*Час пошуку вказан у наносекундах.

**У таблиці вказані середні значення після триразових перевірок даних.

Графік часу роботи алгоритмів пошуку



Висновок

У результаті роботи програми було розроблено програму, яка створює масив елементів та список розміром, вказаним користувачем та заповнює його випадковими елементами. Після цього масив та список сортуються для роботи з алгоритмом пошуку. За результатами роботи алгоритмів з масивами різних розмірів ми бачимо, що двійковий алгоритм працює повільніше у списку, тому що багато часу витрачається на переміщення покажчиків у списку, в той час, як у звичайному масиві доступ до елементів відбувається за індексом

Відповіді на питання

1. Що визначає складність алгоритму?

При визначенні складності алгоритмів пошуку за часом визначають насамперед кількість операцій порівнянь та присвоювань.

2. Яка умова повинна виконуватися при пошуку ключа цілого типу?

Ключ повинен бути цілого типу.

3. Яка умова має виконуватися при пошуку ключа дійсного типу?

Ключ повинен бути дійсного типу.

4. В алгоритмі лінійного пошуку з бар’єром, що є бар’єром?

В алгоритмі лінійного пошуку з бар’єром, бар’єром є останнє, додаткове число в масиві.

5. Перерахуйте усі відомі прості алгоритми пошуку по числовому ключу в порядку зменшення їх середнього часу пошуку.

Експонентний, лінійний з бар’єром, лінійний, бінарний, інтерполяційний.

6. Які обмеження накладаються на набір даних при лінійному пошуку з бар’єром та без нього?

При роботі з алгоритмом лінійного пошука з бар’єром останній елемент повинен бути ключом. Це означає, що масив має розмір N+1. У звичайному лінійному пошуку без бар’єра таких обмежень немає.

7. Поясніть, як виконується двійковий пошук?

Спочатку приблизно визначається запис у середині таблиці й аналізується значення ключа цього запису. Якщо воно занадто велике, то аналізується значення ключа, у середині першої половини таблиці. Зазначена процедура повторюється в цій половині доти, поки не буде знайдений необхідний запис. Якщо значення ключа занадто мале, випробується ключ, що відповідає запису в середині другої половини таблиці, і процедура повторюється в цій половині. Цей процес продовжується доти, поки не буде знайдений необхідний ключ або не стане порожнім інтервал, у якому здійснюється пошук.

8. Реалізація якого алгоритма наведена в наступному фрагменті програмного коду?

```
{   while(m[i]!= key && i<N) i++;
    if( m[i]== key) return i; else return –1;   }
```

Це **алгоритм лінійного пошуку без бар'єра**.

9. Які обмеження накладаються на дані при бінарному пошуку?

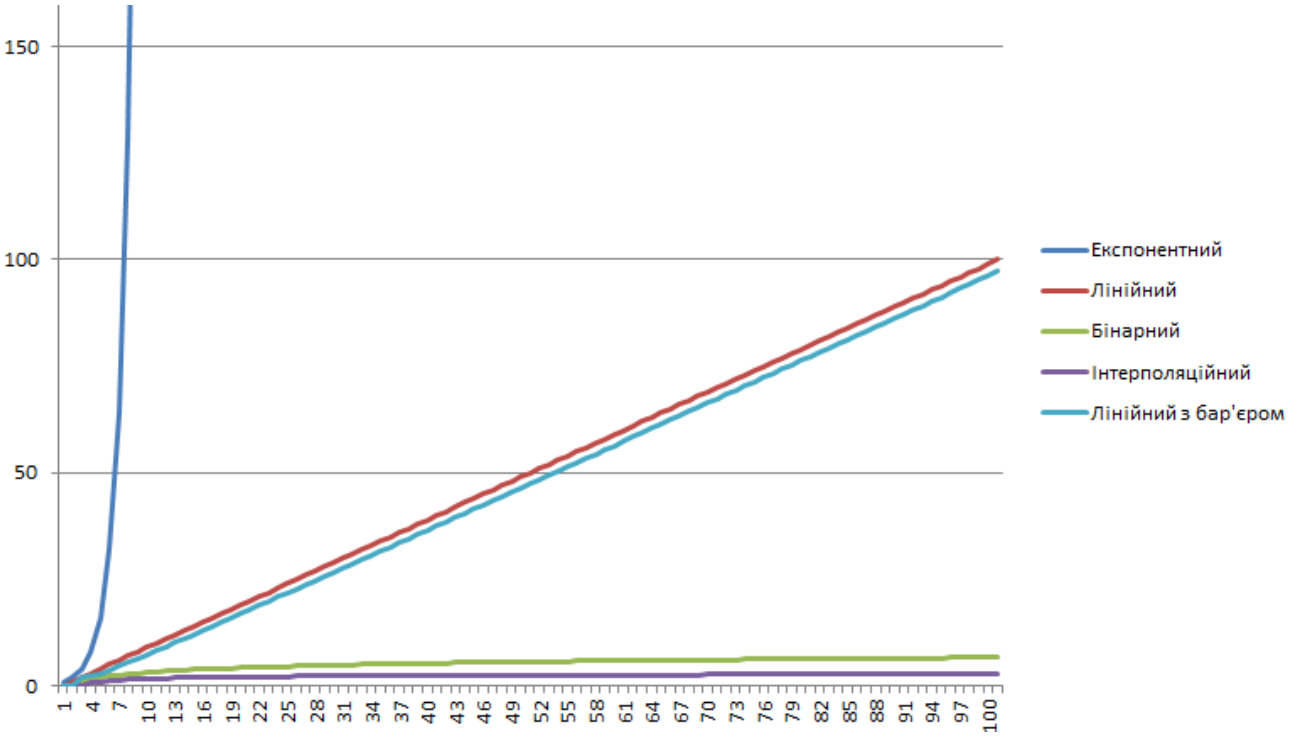
Дані для пошуку бінарним алгоритмом повинні бути упорядковані будь-яким методом сортування.

10. Реалізація якого алгоритма наведена в наступному фрагменті програмного коду?

```
{   while(m[i]!= key) i++;
    if( i!= N) return i;  else return –1;   }
```

Це **алгоритм лінійного пошуку з бар'єром**.

11. Накресліть якісний графік залежності часу пошука від кількості елементів в наборі даних для лінійного, лінійного з бар'єром, двійкового, інтерполяційного та екпоненційного алгоритмів пошуку. Дайте пояснення.



Лінійний пошук та лінійний пошук з бар'єром мають однакові порядки

алгоритму (O(N)), але пошук з бар'єром швидше через те, що в циклі перевірюється більш простий вираз.

Різниця в швидкості роботи інтерполяційного метода помітна лише тоді, коли йде робота над масивами великих обсягів. Його середня швидкість – log(log(N)).

Експоненційний алгоритм пошуку обмежений функцією 2^n ,тому його графік зростає швидше за всіх.

12. Поясніть, як виконується експоненційний пошук, в чому його відмінність від двійкового?

На першому етапі визначається діапазон елементів в масиві, де приблизно може перебувати ключ (програмний приклад 8.5). Для цього в циклі порівнюється ключ пошуку і елемент масиву. Поки ключ більше елемента масиву визначається значення індексу наступного елемента збільшенням в два рази поточного номера. Іншими словами випробовується елементи масиву з номерами 1, 2, 4, 8, 16 і т. Д. (В загальному вигляді 2^i). Звідси і назва алгоритму - експонентний.

Коли ключ пошуку виявляється менше чергового обраного елемента, настає другий етап. Тепер в знайденому інтервалі методом двійкового пошуку визначають положення ключа, якщо він в масиві є.

13. Поясніть алгоритм роботи БМ-пошука зразка в тексті.

На першому кроці будується таблиця зсувів для шуканого зразка. Далі поєднується початок рядка і зразка і починається перевірка з останнього символу зразка. Якщо останній символ зразка та відповідний йому при накладенні символ рядка не збігаються, зразок зрушується щодо рядка на величину, отриману з таблиці зміщень, і знову проводиться порівняння, починаючи з останнього символу зразка. Якщо ж символи збігаються, проводиться порівняння попереднього символу зразка і т. Д. Якщо всі символи зразка збіглися з накладеними символами рядки, значить знайшли входження і пошук закінчено. Якщо ж якийсь (не останній) символ зразка не збігається з відповідним символом рядка, зразок зсувається на один символ вправо і знову починається перевірка з останнього символу. Весь алгоритм виконується до тих пір, поки або не буде знайдено входження шуканого зразка, або не буде досягнутий кінець рядка.

14. Поясніть алгоритм роботи КМП-пошука зразка в тексті.

Цей алгоритм грунтується на тому, що починаючи кожен раз порівнювати зразок (підрядок) з самого початку, можна знищити цінну інформацію. Після часткового збігу початкової частини образу з відповідними символами тексту фактично вже відома, пройдена частина тексту, і можна обчислити деякі відомості (на основі самого образу), за допомогою яких потім виконується швидке

переміщення по тексту.

15. В чому суттєва відмінність алгоритма прямого пошука зразка в тексті від алгоритмів КМП та БМ пошуків?

При прямому пошуку в циклі індекс збільшується до першого символу, що збігається з символом в шуканому тексті. Після чого починається цикл, який перевіряє текст на відповідність тексту, який щем.

КМП пошук виконує пошук переходами по тексту, а БМ робить це ще швидше.