ЛАБОРАТОРНА РОБОТА №1

Назва роботи: алгоритм; властивості, параметри та характеристики складності алгоритму.

Мета роботи: проаналізувати складність заданих алгоритмів.

1. Загальні відомості.

Здавна найбільшу увагу приділяли дослідженням алгоритму з метою мінімізації часової складності розв'язання задач. Але зміст складності алгоритму не обмежується однією характеристикою. В ряді випадків не менше значення має складність логіки побудови алгоритму, різноманітність його операцій, зв'язаність їх між собою. Ця характеристика алгоритму називається програмною складністю. В теорії алгоритмів, крім часової та програмної складності, досліджуються також інші характеристики складності, наприклад, місткісна, але найчастіше розглядають дві з них – часову і програмну. Якщо у кінцевому результаті часова складність визначає час розв'язання задачі, то програмна складність характеризує ступінь інтелектуальних зусиль, що потрібні для синтезу алгоритму. Вона впливає на витрати часу проектування алгоритму.

Вперше значення зменшення програмної складності продемонстрував аль-Хорезмі у своєму трактаті "Про індійський рахунок". Алгоритми реалізації арифметичних операцій, описані аль-Хорезмі у словесній формі, були першими у позиційній десятковій системі числення. Цікаво спостерігати, як точно і послідовно описує він алгоритм сумування, користуючись арабською системою числення і кільцем (нулем). В цьому опису є всі параметри алгоритму. Це один з перших відомих у світі вербальних арифметичних алгоритмів.

Схема розроблення будь-якого об'єкту складається з трьох операцій: синтез, аналіз та оптимізація *(Рис.1.)*.

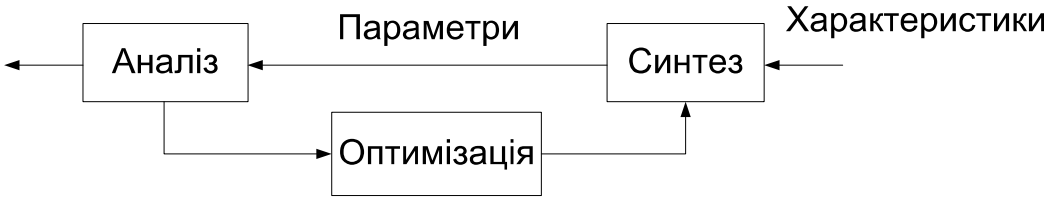


Рис. 1.

Існує два види синтезу: структурний і параметричний. Вихідними даними для структурного синтезу є параметри задачі – сформульоване намагання і набори вхідних даних. В результаті структурного синтезу отримують алгоритм, який розв'язує задачу в принципі.

Параметричний синтез змінами параметрів створює таку його структуру, яка дозволяє зменшити часову складність попередньої моделі. Існує багато способів конструювання ефективних алгоритмів на основі зміни параметрів. Розглянемо спосіб зміни правила безпосереднього перероблення на прикладі задачі знаходження найбільшого спільного дільника двох натуральних чисел..

Алгоритм знаходження найбільшого спільного дільника, яким ми користуємось для цієї цілі і донині, був запропонований Евклідом приблизно в 1150 році до н.е. у геометричній формі, в ньому порівняння величин проводилося відрізками прямих, без

використання арифметичних операцій Алгоритм розв'язку передбачав повторювання віднімання довжини коротшого відрізка від довжини довшого.

Опис алгоритму. Маючи два натуральні числа *a* та *b*, повторюємо *обчислення* для пари значень *b* та залишку від ділення *a* на *b* (тобто *a* mod *b*). Якщо *b*=0, то *a* є шуканим НСД.

Обчислення

Ітераційна версія:

```
НСД( a, b )
      поки b ≠ 0
           c = остача від ділення a на b
           a = b
           b = c
      повернути a
```

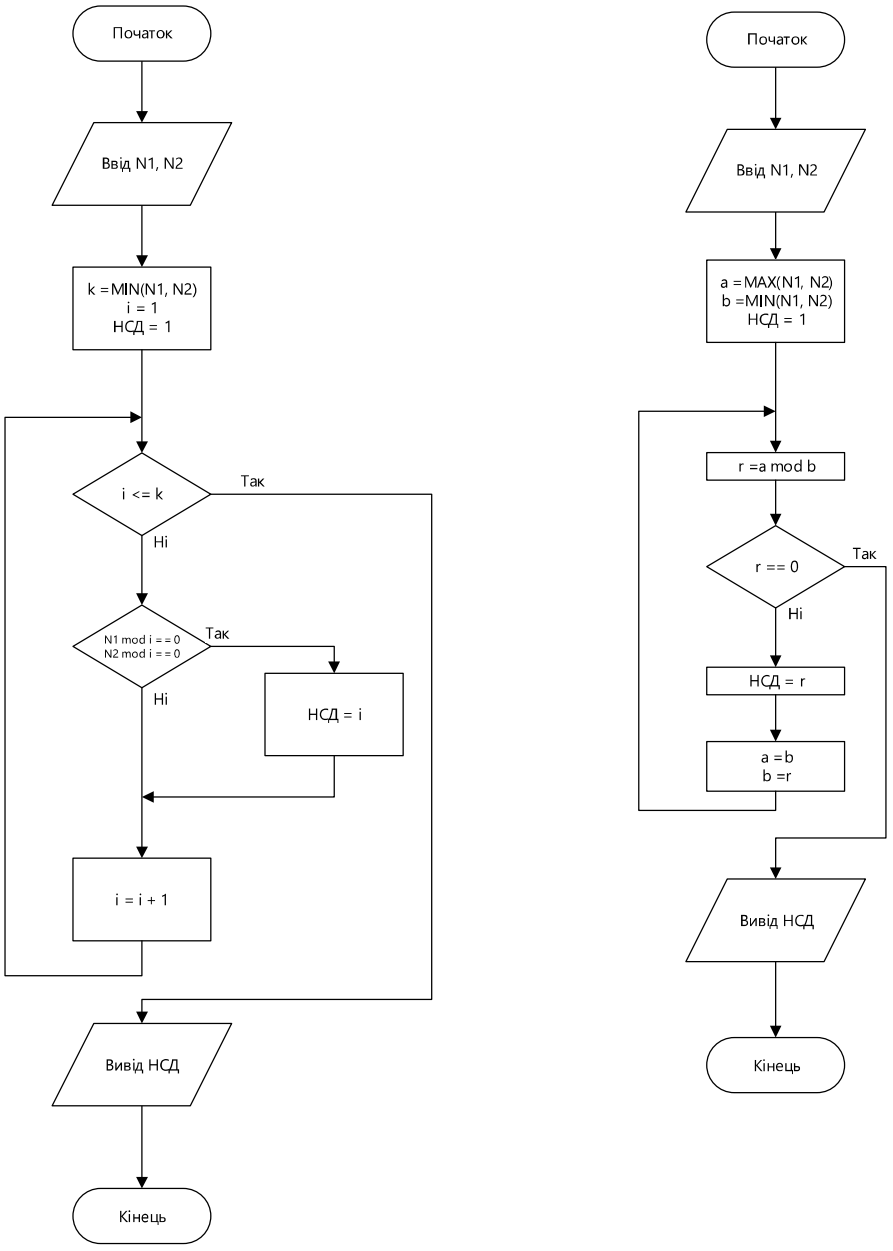
Рекурсивна версія:

```
НСД( a, b )
      якщо b == 0
         поверни a
      інакше
         повернути НСД( b, остача від ділення a на b )
```

Для того, щоб довести ефективність алгоритму, потрібно порівняти його з таким, який приймається за неефективний. Прикладом такого неефективного алгоритму є процедура послідовного перебору можливих розв’язань задачі. Будемо вважати, що алгоритм перебору утворений в результаті структурного синтезу, на основі вхідних даних та намагання знайти серед всіх допустимих чисел такє, що є найбільшим дільником двох заданих чисел.

Ефективність, як правило, визначається такою характеристикою як часова складність, що вимірюється кількістю операцій, необхідних для розв’язання задачі.

Дослідимо розв’язання задачі знаходження найбільшого спільного дільника двох цілих чисел ($N_1>0$, $N_2>0$, $N_1 \geq N_2$) алгоритмом перебору і алгоритмом Евкліда. Алгоритм перебору заснований на операції інкременту змінної n від одиниці до меншого (N_2) з двох заданих чисел і перевірці, чи ця змінна є дільником заданих чисел. Якщо це так, то значення змінної запам’ятовується і операції алгоритму продовжуються. Якщо ні, то операції алгоритму продовжуються без запам’ятовування. Операції алгоритму закінчуються видачею з пам’яті знайденого останнім спільного дільника. Блок-схема алгоритму приведена на *рис.2(а)*.



а) б)

Рис2. Блок-схема алгоритму перебору (а) і Евкліда (б).

Адаптований до сучасної арифметики алгоритм Евкліда використовує циклічну операцію ділення більшого числа на менше, знаходження остачі (r) і заміну числа, яке було більшим, на число, яке було меншим, а меншого числа на остачу. Всі перераховані операції виконуються в циклі. Операції циклу закінчуються, коли остача дорівнює нулю. Останній дільник є найбільшим спільним дільником. Блок-схема алгоритму приведена на *рис.2(б)*.

Аналіз цих двох алгоритмів показує, що часова складність алгоритму перебору значно перевищує часову складність алгоритму Евкліда. Для обох алгоритмів часова складність є функцією від вхідних даних, а не їх розміру. В таких випадках при порівнянні ефективності алгоритмів користуються порівнянням часових складностей визначених для найгіршого випадку. Часова складність для найгіршого випадку (Lmax) представляє собою максимальну часову складність серед всіх вхідних даних розміру N.

Часова складність $L_{\max}$ для алгоритму перебору:

$$L_{\max} = C \cdot N_2 \qquad (1)$$

де C – константа , яка дорівнює кількості операцій в кожній ітерації.

Для цілих чисел n $(1 \le n < r_i)$ алгоритм Евкліда знаходження найбільшого спільного дільника має найбільшу часову складність для пари чисел r_{i-1} і r_{i-2} , де 1 , 2 , 3 , … , r_{i-2}, r_{i-1} , r_i – числа Фібоначчі.

Алгоритм Евкліда є ефективним за часовою складністю у порівняні з алгоритмом перебору. Мінімізація часової складності дозволяє за всіх інших рівних умова збільшити продуктивність розв'язання задачі.

2. Приклад програми.

У *лістингу 2.1* показана реалізація трьох алгоритмів(алгоритму перебору, ітераційної версії алгоритму Евкліда та рекурсивної версії алгоритму Евкліда) знаходження НСД.

Лістинг 2.1

Знаходження НСД.

Алгоритм перебору.

```
#define DATA_TYPE volatile unsigned long long int
DATA_TYPE f1_GCD(DATA_TYPE variableN1, DATA_TYPE variableN2) {
        DATA_TYPE returnValue = 1;

        for (DATA_TYPE i = 1, k = MIN(variableN1, variableN2); i <= k; i++) {
                if (!(variableN1 % i || variableN2 % i)) {
                        returnValue = i;
                }
        }

        return returnValue;
}
```

Ітераційна версія алгоритму Евкліда.

```
#define DATA_TYPE volatile unsigned long long int
DATA_TYPE f2_GCD(DATA_TYPE a, DATA_TYPE b) {
        for (DATA_TYPE aModB;
                aModB = a % b,
                a = b,
                b = aModB;
                );

        return a;
}
```

Рекурсивна версія алгоритму Евкліда.

```
#define DATA_TYPE volatile unsigned long long int
DATA_TYPE f3_GCD(DATA_TYPE a, DATA_TYPE b) {
        if (!b) {
                return a;
        }

        return f3_GCD(b, a % b); // else
}
```

У *лістингу 2.2* показаний приклад програми, яка дозволяє провести порівняння трьох наведених алгоритмів знаходження НСД за характеристикою часової складності.

Лістинг 2.2

```
//to measure time it is better to run with Linux
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define DATA_TYPE volatile unsigned long long int
#define N1 18
#define N2 27
```

```
#define MIN(a,b) (((a)<(b))?(a):(b))
#define MAX(a,b) (((a)>(b))?(a):(b))

#define REPEAT_COUNT 1000000
#define REPEATOR(count, code) \
for (unsigned int indexIteration = (count); indexIteration--;){ code; }

double getCurrentTime() {
	clock_t time = clock();
	if (time != (clock_t)-1) {
		return ((double)time / (double)CLOCKS_PER_SEC);
	}
	return 0.; // else
}
DATA_TYPE f1_GCD(DATA_TYPE variableN1, DATA_TYPE variableN2) {
	DATA_TYPE returnValue = 1;

	for (DATA_TYPE i = 1, k = MIN(variableN1, variableN2); i <= k; i++) {
		if (!(variableN1 % i || variableN2 % i)) {
			returnValue = i;
		}
	}

	return returnValue;
}
DATA_TYPE f2_GCD(DATA_TYPE a, DATA_TYPE b) {
	for (DATA_TYPE aModB;
		aModB = a % b,
		a = b,
		b = aModB;
		);

	return a;
}
DATA_TYPE f3_GCD(DATA_TYPE a, DATA_TYPE b) {
	if (!b) {
		return a;
	}

	return f3_GCD(b, a % b); // else
}

int main() {
	DATA_TYPE a = MAX(N1, N2), b = MIN(N1, N2), returnValue;

	double startTime, endTime;

	// f1_GCD
	startTime = getCurrentTime();
	REPEATOR(REPEAT_COUNT, returnValue = f1_GCD(a, b););
	endTime = getCurrentTime();
	printf("f1_GCD return %lld \r\nrun time: %dns\r\n\r\n",
		returnValue,
		(unsigned int)((endTime - startTime) * (double)(1000000000 / REPEAT_COUNT)));

	// f2_GCD
	startTime = getCurrentTime();
	REPEATOR(REPEAT_COUNT, returnValue = f2_GCD(a, b););
	endTime = getCurrentTime();
	printf("f2_GCD return %lld \r\nrun time: %dns\r\n\r\n",
		returnValue,
		(unsigned int)((endTime - startTime) * (double)(1000000000 / REPEAT_COUNT)));

	// f3_GCD
	startTime = getCurrentTime();
	REPEATOR(REPEAT_COUNT, returnValue = f3_GCD(a, b););
	endTime = getCurrentTime();
	printf("f3_GCD return %lld \r\nrun time: %dns\r\n\r\n",
		returnValue,
		(unsigned int)((endTime - startTime) * (double)(1000000000 / REPEAT_COUNT)));

	printf("Press Enter to continue . . .");
	(void)getchar();

	return 0;
}
```

3. Завдання.

Скласти програму (C/C++), яка дозволяє провести порівняння двох алгоритмів за характеристикою часової складності.

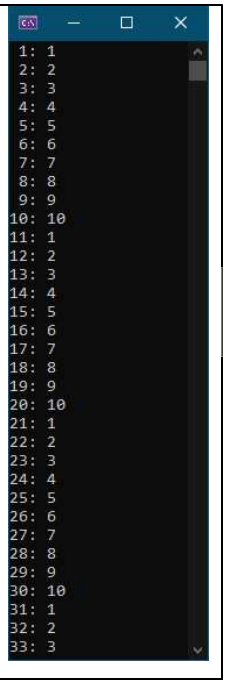
Вибір варіанту

(Nж - 1)%10 + 1

де: Nж – порядковий номер студента в групі

Співвідношення між порядковим номером студента в групі та варіантом завдання можна також відобразити за допомогою такої програми мовою С:

```c
#include "stdio.h"
#define V 10
#define S 33
#define GET_V(Nj) (((Nj) - 1)%V + 1)
int main() {
	int index = 1;
	for (;printf("%2d: %d\r\n", index, GET_V(index)), ++index <= S;);
	return 0;
}
```



Варіанти завдань

Варіант	Вхідні дані	
	Алгоритм 1	**Алгоритм 2**
1	**Сортування бульбашкою**	**Сортування вибором**
2	**Сортування бульбашкою**	**Сортування включенням**
3	**Сортування бульбашкою**	**Сортування Шелла**
4	**Сортування бульбашкою**	**Швидке сортування**
5	**Сортування вибором**	**Сортування включенням**
6	**Сортування вибором**	**Сортування Шелла**
7	**Сортування вибором**	**Швидке сортування**
8	**Сортування включенням**	**Сортування Шелла**
9	**Сортування включенням**	**Швидке сортування**
10	**Сортування Шелла**	**Швидке сортування**

У *лістингу 2.3* наведений приклад реалізації цих алгоритмів сортування, а приклад порівняння алгоритмів знаходження НСД за характеристикою часової складності було показано раніше в *лістингу 2.2* .

Лістинг 2.3

```c
#include <stdio.h>
#include <stdlib.h>
#define DATA_SIZE 32

#define sort bulbSort
//#define sort choiceSort
//#define sort insertSort
//#define sort shellSort
//#define sort quickSort

void bulbSort(int * array, int leftIndex, int rightIndex){
```

```
	int index, temp;

	if (!array){
		return;
	}
	++rightIndex;
	while (leftIndex < --rightIndex){
		for (index = leftIndex; index < rightIndex; ++index){
			if (array[index] > array[index + 1]){
				temp = array[index];
				array[index] = array[index + 1];
				array[index + 1] = temp;
			}
		}
	}
}

void bulbSort1(int * array, int leftIndex, int rightIndex){
	int index;

	if (!array){
		return;
	}
	for (index = leftIndex; index < rightIndex ? rightIndex : (index = leftIndex, --
rightIndex); ++index){
		if (array[index] > array[index + 1]){
			array[index] ^= array[index + 1];
			array[index + 1] ^= array[index];
			array[index] ^= array[index + 1];
		}
	}
}

void choiceSort(int * array, int leftIndex, int rightIndex){
	int iIndex, jIndex;

	for (iIndex = leftIndex; iIndex <= rightIndex; ++iIndex){
		int kIndex = iIndex;
		int temp = array[iIndex];
		int exch = 0;
		for (jIndex = iIndex + 1; jIndex <= rightIndex; ++jIndex){
			if (array[jIndex] < temp){
				kIndex = jIndex;
				temp = array[jIndex];
				exch = 1;
			}
		}
		if (exch){
			array[kIndex] = array[iIndex];
			array[iIndex] = temp;
		}
	}
}

void insertSort(int * array, int leftIndex, int rightIndex){
	int iIndex, jIndex;

	for (iIndex = leftIndex + 1; iIndex <= rightIndex; ++iIndex){
		int temp = array[iIndex];
		jIndex = iIndex - 1;
		for (; jIndex >= leftIndex && array[jIndex] > temp; --jIndex){
			array[jIndex + 1] = array[jIndex];
		}
		array[jIndex + 1] = temp;
	}
```

```
}

void shellSort(int * array, int leftIndex, int rightIndex){
	int dIndex, iIndex, jIndex;

	int sortingSize = rightIndex - leftIndex + 1;
	for (dIndex = leftIndex + sortingSize / 2; dIndex >= leftIndex + 1; dIndex /= 2){
		for (iIndex = dIndex; iIndex < sortingSize; iIndex++){
			for (jIndex = iIndex; jIndex >= dIndex && array[jIndex - dIndex] >
				array[jIndex]; jIndex -= dIndex){
				int temp = array[jIndex];
				array[jIndex] = array[jIndex - dIndex];
				array[jIndex - dIndex] = temp;
			}
		}
	}
}

void quickSort(int * array, int leftIndex, int rightIndex){
	int iIndex = leftIndex;
	int jIndex = rightIndex;
	int xElemntValue = array[(leftIndex + rightIndex) / 2];
	do{
		while ((array[iIndex] < xElemntValue) && (iIndex < rightIndex)) {
			++iIndex;
		}
		while ((xElemntValue < array[jIndex]) && (jIndex > leftIndex)) {
			--jIndex;
		}
		if (iIndex <= jIndex){
			int temp = array[iIndex];
			array[iIndex] = array[jIndex];
			array[jIndex] = temp;
			++iIndex;
			--jIndex;
		}
	} while (iIndex <= jIndex);
	if (leftIndex < jIndex) {
		quickSort(array, leftIndex, jIndex);
	}
	if (iIndex < rightIndex) {
		quickSort(array, iIndex, rightIndex);
	}
}

void printVector(void * data, int count/* 0 - full DATA_SIZE*/){
	int index = 0;
	for (index = 0; (!count || index < count) && index < DATA_SIZE; index++){
		printf("%d ", ((int *)data)[index]);
	}
	printf("\n");
}
int main(void){
	int data[DATA_SIZE] = { 1, 20, 29, 28, 10, 26, 25, 24,
		23, 22, 21, 30, 19, 18, 17, 16,
		15, 14, 13, 12, 11, 27, 9, 8,
		7, 6, 5, 4, 3, 2, 0, 31 };
	sort(data, 0, DATA_SIZE - 1);
	printVector(data, 0);
	(void)getchar();
	return EXIT_SUCCESS;
}
```

4. Зміст звіту

- Титульний лист;
- Завдання;
- Алгоритм рішення завдання;
- Код програми;
- Екранна форма з результатами роботи програми;
- Висновки.