Лабораторна робота №2

Розробка та реалізація програм сортування та пошуку елементів в статичному масиві. Принципи роботи з вказівниками та одновимірними динамічними масивами

Мета роботи: оволодіння навичками складання програм пошуку та сортування елементів масиву, оволодіння навичками складання програм для роботи з вказівниками та одновимірними динамічними масивами та виконання їх в середовищі програмування

Завдання

Завдання 2.1. Сформулювати математичний запис фрагмента програми та обчислити значення змінної `x` після його виконання, згідно варіанту, наведеному в таблиці 1.1. Елементи масиву обчислюються за формулою `a[i]=p[i]-50`, де `p[i+1]=(p[i]*11+7)%100`. `p[0]` дорівнює `N` – номеру варіанта за списком групи, кількість елементів у масиві дорівнює `size=15`.

Таблиця 1.1 – Варіанти завдання 2.1

№	Фрагмент програми
1–5	`for (int i=0; i< size; i++) {` `  for (int j= size-1; j>i; j--){` `  if (a[j] < a[j-1]) {` `  int t = a[j];` `  a[j] = a[j-1];` `  a[j-1] = t;` `  }` `  }` `  }` `x=a[0];`
6–10	`for (int i=1; i<size; i++) {` `  int curr = a[i];` `  int j = i-1;` `  while (j>=0 && a[j]>curr) {` `  a[j+1]=a[j];` `  j--;` `  }` `  a[j+1] = curr;` `  }` `x=a[0];`

11–15	`for (int i=0; i<size-1; i++) {` `int nmin = i;` `for (int j=i+1; j<size; j++){` `if (a[j]<a[nmin]) nmin = j;` `}` `if (i != nmin) {` `int t = a[nmin];` `a[nmin] = a[i];` `a[i] = t;` `}` `}` `x=a[0];`
16–20	`for (int i=0; i< size; i+=2){` `for (int j= size-1; j>i; j-=2){` `if (a[j] > a[j-2] ) {` `int t = a[j];` `a[j] = a[j-2];` `a[j-2] = t;` `}` `}` `}` `x=a[2];`
21–25	`for (int i=0; i<size-1; i++) {` `int nmin = i;` `for (int j=i+1; j<size; j++){` `if (a[j]>a[nmin]) nmin = j;` `}` `if (i != nmin) {` `int t = a[nmin];` `a[nmin] = a[i];` `a[i] = t;` `}` `}` `x=a[0];`
26–30	`for (int i=1; i<size; i+=2) {` `int curr = a[i];` `int j = i-2;` `while (j>=0 && a[j]>curr) {` `a[j+2]=a[j];` `j-=2;` `}` `a[j+2] = curr;` `}` `x=a[1];`

Завдання 2.2. Скласти програму сортування послідовностей чисел згідно за варіантами, які наведені в таблиці 1.2, при виконанні застосовувати **статичні** масиви. Послідовності заповнити за допомогою датчика випадкових чисел.

Таблиця 1.2 – Варіанти завдання 2.2

№	Завдання
1	Дана послідовність цілих чисел $a_1, a_2, \dots, a_{25}$. Розташувати елементи послідовності, кратні 3 за зростанням (інші елементи залишаються на своїх місцях). Метод сортування – вибір.
2	Дана послідовність дійсних чисел $a_1, a_2, \dots, a_{22}$. Розташувати додатні елементи послідовності, що стоять на непарних місцях за зростанням (інші елементи залишаються на своїх місцях). Метод сортування – вибір.
3	Дана послідовність дійсних чисел $a_1, a_2, \dots, a_{22}$. Елементи, що стоять на парних місцях, розташувати в порядку зростання, а на непарних в порядку зменшення. Метод сортування – бульбашкою.
4	Дана послідовність дійсних чисел $a_1, a_2, \dots, a_{22}$. Розташувати елементи послідовності, що стоять на парних місцях у порядку зменшення (інші елементи залишаються на своїх місцях). Метод сортування – бульбашкою.
5	Дана послідовність дійсних чисел $a_1, a_2, \dots, a_{22}$. Розташувати елементи послідовності, менші за середньоарифметичне значення, за зростанням (інші елементи залишаються на своїх місцях). Метод сортування – вставка.
6	Дана послідовність дійсних чисел $a_1, a_2, \dots, a_{22}$, відмінних від нуля. Розташувати від'ємні елементи послідовності за спаданням, а додатні – по зростанню. Метод сортування – вибір.
7	Дана послідовність додатних цілих чисел $a_1, a_2, \dots, a_{25}$. Розташувати елементи послідовності, що кратні 3, за спаданням (інші елементи залишаються на своїх місцях). Метод сортування – вставка.
8	Дана послідовність цілих чисел $a_1, a_2, \dots, a_{25}$. Розташувати парні елементи за зростанням (інші елементи залишаються на своїх місцях). Метод сортування – вибір.
9	Дана послідовність дійсних чисел $a_1, a_2, \dots, a_{22}$. Потрібно розташувати додатні елементи послідовності за спаданням (інші елементи залишаються на своїх місцях). Метод сортування – вставка.
10	Дана послідовність цілих чисел $a_1, a_2, \dots, a_{25}$. Розташувати непарні елементи за спаданням (інші елементи залишаються на своїх місцях). Метод сортування – вибір.
11	Дана послідовність дійсних чисел $a_1, a_2, \dots, a_{22}$. Розташувати елементи послідовності, які належать діапазону $[p_1, p_2]$, за спаданням (інші елементи залишаються на своїх місцях). Метод сортування – бульбашкою.
12	Дана послідовність дійсних чисел $a_1, a_2, \dots, a_{30}$. Розташувати першу половину елементів послідовності, за спаданням, інші елементи розташувати за зростанням. Метод сортування – бульбашкою.
13	Дана послідовність дійсних чисел $a_1, a_2, \dots, a_{22}$. Розташувати елементи

	послідовності, які містять цифру N, за зростанням (інші елементи залишаються на своїх місцях). Метод сортування – вставка.
14	Дана послідовність дійсних чисел $a_1, a_2, \dots, a_{20}$. Розташувати елементи послідовності, сума цифр яких дорівнює N, за спаданням (інші елементи залишаються на своїх місцях) Метод сортування – вибір.
15	Дана послідовність цілих чисел $a_1, a_2, \dots, a_{30}$. Розташувати додатні елементи послідовності, які кратні 5, за спаданням (інші елементи залишаються на своїх місцях). Метод сортування – вибір.

Завдання 2.3. Нехай є наступний фрагмент програми, поданий у варіантах (див. табл. 1.3). Поясніть, як зміниться масив після його виконання, при наступному початку програми (Вказівка: замість `N` підставити номер варіанта за списком групи).

```
#include <iostream>
#include<cmath>
using namespace std;
int* form(int&n) {  n=10+N%10;
     int*a=new int[n];
     *a=N;
     for(int i=1;i<n;i++)
          *(a+i)=*(a+i-1)+1;
     return a;
}
int main(){
```

Таблиця 1.3 – Варіанти завдання 2.3

№	Фрагмент програми
1–5	`int *a, n;  a=form(n);` `int **p=new int *[n];` `for (int i=0;i<n;i++)` `{ p[i]=(a+i);` `*p[i]=*p[i]+N;` `}`
6–10	`int *a, n,i,*p;` `a=form(n);` `for (p = &a[0], i = 0 ; i <n; i++ )` `{` `*(p+i) -=N*N;` `}`
11–15	`int *a, n,i,*p;` `a=form(n);` `for ( p = a, i = 0; p+i < a+n; i++ ){` `*(p+i) *=*(p+i);` `}`
16-20	`int *a, n,i,*p;` `a=form(n);` `for ( p = a+n, i=1; i <=n; i++ )`

	{ *(p-i) *= (int)sqrt(N); }
21–25	int *a, n,i,*aa; a=form(n); int **p=new int *[n]; aa=a+n-1; for (i=0;aa-a>=0;aa--,i++) { *(p+i)=aa; **(p+i)+=*(a+i); }
26–30	int *a, n,i,t=0,*p; a=form(n); int *aa=new int[n]; p=aa; for (i=n-1;i>=0;i--) {*p++=*(a+i); t+=*(a+i);} t=t/n; for (i=0;i<n;i++) *(a+i)=*(aa+i)+t; }

Завдання 2.4. Скласти програму, що заповнює **динамічний** одновимірний масив за допомогою випадкових чисел, потім змінює масив, згідно завдань, які наведені в таблиці 2.2, та виконати її в середовищі програмування.

Примітка:

a. Вивести масив до та після модифікації.

b. При звертанні до елементів масиву використовувати вказівники.

Таблиця 2.2 – Варіанти завдання 2.4

№	Завдання
1.	Видалити всі від’ємні двозначні елементи масиву.
2.	Видалити елемент із заданим ключем (значенням) і елемент із заданим номером.
3.	Видалити всі елементи, розташовані між першим парним елементом і елементом p, де p – це ціла частина середнього арифметичного елементів масиву.
4.	Видалити N елементів, починаючи з номера K, де N і K вводяться з клавіатури.
5.	Видалити всі парні елементи.
6.	Видалити всі елементи з парними індексами.
7.	Видалити всі непарні елементи.
8.	Видалити всі елементи з непарними індексами.
9.	Додати K простих чисел на початку масиву, де K вводиться з клавіатури.
10.	Додати K простих чисел в кінець масиву, де K вводиться з клавіатури.

11.	Додати K чисел Фібоначчі на початку масиву, де K вводиться з клавіатури.
12.	Додати K чисел Фібоначчі в кінець масиву, де K вводиться з клавіатури.
13.	Додати K елементів, починаючи з номера N, де N і K вводяться з клавіатури.
14.	Додати після кожного від'ємного елемента його абсолютне значення.
15.	Додати після кожного парного елемента, елемент зі значенням 0.

Короткі теоретичні відомості

Найбільш простий зі способів пошуку даних – лінійний пошук. Даний алгоритм порівнює кожний елемент масиву із ключем, наданим для пошуку. Алгоритм лінійного пошуку відмінно працює тільки для невеликих або неупорядкованих масивів і є абсолютно надійним.

Якщо масив містить упорядковану послідовність даних, то більш ефективним буде двійковий (бінарний) пошук.

Припустимо, що змінні *Lb* і *Rb* містять, відповідно, ліву й праву границі відрізка масиву, де перебуває потрібний елемент. Пошук завжди будемо починати з аналізу середнього елементу *M* відрізка масиву. Якщо шукане значення менше *M*, переходимо до пошуку в лівій половині відрізка, де всі елементи менші за елемент, який тільки що перевірили. Інакше кажучи, значенням *Rb* стає (*M*-1) і на наступній ітерації робота ведеться з половиною масиву. Таким чином, у результаті кожної перевірки вдвічі звужується область пошуку.

Двійковий пошук – дуже потужний метод, наприклад, якщо довжина масиву рівна 1023, тоді після першого порівняння проміжок звужується до 511 елементів, а після другого – до 255, тобто для пошуку в масиві з 1023 елементів досить 10 порівнянь.

Ідея методу бульбашкового сортування полягає в наступному: крок сортування полягає в проході знизу вгору по масиву. По дорозі проглядаються пари сусідніх елементів. Якщо елементи деякої пари знаходяться в неправильному порядку, то вони міняються місцями. Після першого проходу по масиву "вгорі" виявляється самий "легкий" елемент.

Наступний прохід робиться до другого зверху елемента, таким чином другий за величиною елемент піднімається на правильну позицію. Перегляд елементів робиться по все зменшуючій нижній частині масиву до тих пір, поки в ній не залишиться тільки один елемент. На цьому сортування закінчується, так як послідовність впорядкована за зростанням.

Основні принципи методу: середня кількість порівнянь та перестановок мають квадратичний порядок зростання, звідси можна зробити висновок, що алгоритм бульбашки дуже повільний і малоефективний. Проте, у нього є величезний плюс: він простий і його можна по-всякому покращувати.

Якщо на якомусь із проходів не відбулося жодного обміну, це означає, що всі пари розташовані в правильному порядку, тобто масив вже відсортований. Таким чином перший крок оптимізації полягає в запам'ятовуванні, чи проводився при даному проході будь-який обмін. Якщо ні – алгоритм закінчує роботу.

Ідея методу сортування вибором полягає в тому, щоб створювати відсортовану послідовність шляхом приєднання до неї одного за одним елементів в правильному порядку. Алгоритм складається з n послідовних кроків, починаючи від нульового і закінчуючи (n-1). На i-му кроці вибираємо найменший з елементів $a[i]$ ... $a[n]$ і міняємо його місцями з $a[i]$. Незалежно від номеру поточного кроку i, послідовність $a[0]$ ... $a[i]$ є впорядкованою. Таким чином, на етапі (n-1) вся послідовність, крім $a[n]$ елементу виявляється відсортована, а $a[n]$ стоїть на останньому місці по праву: все менші елементи вже пішли вліво.

Основні принципи методу: для знаходження найменшого елемента з n+1 розглядаємих елементів алгоритм робить n порівнянь. Оскільки кількість перестановок завжди буде меншою за кількість порівнянь, час сортування зростає щодо кількості елементів. Крім того, алгоритм не використовує додаткову пам'ять: всі операції відбуваються "на місці".

Сортування цим методом можна використовувати для масивів, що мають невеликі розміри.

Сортування простими вставками в чомусь схоже на методи, наведені раніше. Аналогічним чином робляться проходи по частині масиву, і аналогічним же чином на його початку "виростає" відсортована послідовність.

Однак в сортуванні бульбашкою або вибором можна було чітко заявити, що на *i*-му кроці елементи *a*[0] ... *a*[*i*] стоять на правильних місцях і нікуди вже не перемістяться. В цьому ж сортуванні подібне твердження буде більш слабким: послідовність *a*[0] ... *a*[*i*] впорядкована. При цьому по ходу алгоритму в неї будуть вставлятися нові елементи.

Розглянемо дії алгоритму на *i*-му кроці. Послідовність до цього моменту розділена на дві частини: впорядковану *a*[0] ... *a*[*i*] та невпорядковану *a*[*i*+1] ... *a*[*n*]. На наступному, (*i*+1)-му кожному кроці алгоритму беремо a[i+1]-й елемент і вставляємо на потрібне місце в відсортовану частину масиву. Пошук відповідного місця для чергового елементу вхідної послідовності здійснюється шляхом послідовних порівнянь з елементом, що стоять перед ним. Залежно від результату порівняння елемент або залишається на поточному місці (вставка завершена), або вони міняються місцями і процес повторюється.

Гарним показником сортування є дуже природна поведінка: майже відсортований масив буде упорядкован дуже швидко. Це, вкупі зі стійкістю алгоритму, робить метод хорошим вибором в відповідних ситуаціях.

Вказівки. Динамічні масиви.

Вказівник – це змінна, яка містить адресу іншої змінної. Вказівники дуже широко використовуються в мові C++. Іноді вони дають єдину можливість виразити потрібну дію, та зазвичай ведуть до більш компактних і ефективних програм.

Так як вказівник містить адресу об'єкта, це дає можливість "непрямого" доступу до цього об'єкта через вказівник. Припустимо, що `x` – змінна, наприклад, типу `int`, а `px` – вказівник, створений ще не зазначеним способом. Унарна операція `&` видає адресу об'єкта, так що оператор `px =`

`&x;` присвоює адресу `x` змінній `px`; кажуть, що `px` "вказує" на `x`. Операція `&` застосовується тільки до змінних та елементів масиву. Не можна отримати адресу регістрової змінної.

Унарна операція `*` розглядає свій операнд як адресу кінцевої мети і звертається за цією адресою, щоб витягти вміст. Отже, якщо `y` має тип `int`, то `y=*px;` присвоює `y` вміст того, на що вказує `px`. Так послідовність `px=&x; y=*px;` присвоює `y` те ж саме значення, що і оператор `y=x;`

Опис вказівника `int *px;` є новим і має розглядатися як мнемонічний; такий запис, треба читати: вказівник `px` вказує на тип `int`. Це означає, що якщо `px` з'являється в контексті `*px`, то це еквівалентно змінній типу `int`.

Вказівники можуть входити у вираз. Наприклад, якщо `px` вказує на ціле `x`, то `*px` може з'являтися в будь-якому контексті, де може зустрітися `x`. У виразах виду `y = *px + 1;` унарні операції `*` та `&` пов'язані зі своїм операндом міцніше, ніж арифметичні операції, так що такий вислів бере значення, на яке вказує `px`, додає `1` і присвоює результат змінній `y`.

Вказівники є змінними, з ними можна поводитися, як і з іншими змінними. Якщо `py` – інший вказівник на змінну типу `int`, то `py = px;` копіює вміст `px` в `py`, в результаті чого `py` вказує на те ж, що і `px`.

Використання вказівників як альтернативний спосіб доступу до змінних приховує в собі небезпеку – якщо була змінена адреса, що зберігається у вказівнику, то цей вказівник більше не посилається на потрібне значення.

Мова C++ пропонує альтернативу для більш безпечного доступу до змінних через вказівники. Оголосивши посилальну змінну, можна створити об'єкт, який, як і вказівник, буде посилатися на інше значення, але, на відміну від вказівника, постійно прив'язаний до цього значення. Таким чином, посилання на значення завжди посилається на це значення.

Синтаксис оголошення посилання:

`<ім'я типу> & <ім'я посилання> = <вираз>;`

або

`<ім'я типу> & <ім'я посилання> (<вираз>);`

Одного разу ініціалізувавши посилання йому не можна присвоювати інше значення. На відміну від вказівників, які можуть бути оголошені неініціалізовані або встановлені в нуль (`NULL`), посилання завжди посилаються на об'єкт. Для посилань обов'язкова ініціалізація при створенні і не існує аналога нульового вказівника.

Посилання не можна ініціалізувати в наступних випадках:

– при використанні в якості параметрів функції.

– при використанні в якості типу значення, що повертає функція.

– в оголошеннях класів.

Не існує операторів, що безпосередньо виконують дії над посиланнями.

Посилальні змінні використовуються досить рідко, значно зручніше використовувати саму змінну, ніж посилання на неї. Більш широке застосування посилання мають в якості параметрів функції. Посилання особливо корисні у функціях, які повертають декілька об'єктів (значень). Посилання та вказівники в якості параметрів функцій тісно пов'язані.

В C++ передача аргументів функцій здійснюється "за значенням", тобто змінні, що передаються у функцію, не можуть поміняти своє значення. Якщо треба змінити аргументи функції, тоді в якості аргументів треба використовувати посилання: `void func (int &a,int &b);`

У мові C++ існує взаємозв'язок між вказівниками та масивами. Будь-яку операцію, яку можна виконати за допомогою індексів масиву, можна виконати й за допомогою вказівників.

Опис `int a[10];` визначає масив розміру `10`. Запис `a[i]` відповідає елементу масиву через `i` позицій від початку. Якщо `pa` – вказівник цілого, описаний як `int *pa;` тоді присвоювання `pa = &a[0]` приводить до того, що `pa` вказує на нульовий елемент масиву `a`. Це означає, що `pa` містить адресу елемента `a[0]`. Присвоювання `x = *pa` буде копіювати вміст `a[0]` в `x`.

Якщо pa вказує на певний елемент масиву a, тоді pa+1 вказує на наступний елемент, а pa-i вказує на елемент, що стоїть на i позицій до елемента, на який вказує pa, pa+i вказує на елемент, що стоїть на i позицій після. Таким чином, якщо pa вказує на a[0], тоді *(pa+1) посилається на вміст a[1], pa+i – адреса a[i], а *(pa+i) – вміст a[i].

Ці зауваження справедливі незалежно від типу змінних у масиві a. Суть визначення "додавання 1 до вказівника", а також його поширення на всю арифметику вказівників, полягає в тому, що збільшення масштабується розміром пам'яті, що займає об'єкт, на який вказує вказівник. Таким чином, i в pa+i перед збільшенням множиться на розмір об'єктів, на які вказує pa.

Посилання на a[i] можна записати у вигляді *(a+i). Ці дві форми еквівалентні. Якщо застосувати операцію & до обох частин такого співвідношення еквівалентності, то ми одержимо, що &a[i] та a+i ідентичні: a+i – адреса i-го елемента від початку a. З іншого боку, якщо pa є вказівником, то у виразах його можна використовувати з індексом: pa[i] ідентично *(pa+i).

Є одна відмінність між іменем масиву та вказівником: вказівник є змінною, так що операції pa=a та pa++ мають сенс. Але ім'я масиву є константою, а не змінною: конструкції типу a=pa або a++, або p=&a будуть помилковими.

Динамічна пам'ять або пам'ять вільного зберігання відрізняється від статичної тим, що програма повинна явно запросити пам'ять для елементів, збережених у цій області, а потім звільнити пам'ять, якщо вона більше не потрібна.

За допомогою операції виділення пам'яті можна виділяти пам'ять динамічно, тобто на етапі виконання програми:

```
вказівник_на_тип_= new ім'я_типу (ініціалізатор);
```

Ініціалізатор – це необов'язковий вираз, який може використовуватися для всіх типів, крім масивів.

При виконанні оператору `int *ip = new int;` створюються 2 об'єкта: динамічний безіменний об'єкт та вказівник на нього з іменем `ip`, значенням якого є адреса динамічного об'єкта. Якщо вказівнику `ip` присвоїти інше значення, то можна втратити доступ до динамічного об'єкта. У результаті динамічний об'єкт як і раніше буде існувати, але звернутися до нього вже не можна. Такі об'єкти називаються сміттям.

При виділенні пам'яті об'єкт можна ініціалізувати: `int *ip = new int(3);` Можна динамічно розподілити пам'ять і під масив: `double *mas = new double [50];` Далі із цією динамічно виділеною пам'яттю можна працювати як зі звичайним масивом.

У випадку успішного завершення операція `new` повертає вказівник зі значенням, відмінним від нуля. Результат операції, рівний `0`, тобто нульовому вказівнику `NULL`, говорить про те, що не знайдений неперервний вільний фрагмент пам'яті потрібного розміру.

Операція звільнення пам'яті `delete` – звільняє ділянку пам'яті для подальшого використання в програмі, яка раніше була виділена операцією `new`: `delete ip;` – видаляє динамічний об'єкт типу `int`, який був створений за допомогою команди `ip = new int;` або `delete [ ] mas;` – видаляє динамічний масив який був створений за допомогою команди `double *mas = new double[50];`

Безпечно застосовувати операцію до вказівника `NULL`. Результат же повторного застосування операції `delete` до одного й того ж вказівника не визначений. Звичайно відбувається помилка, що призводить до зациклення. Щоб уникнути подібних помилок, слід виконувати перевірку `if (ip)` перед застосуванням операції `delete`.

Приклад виконання роботи

Завдання 2.1. Сформулювати математичний запис фрагмента програми та обчислити значення змінної x після його виконання, де size=22 – розмір масиву, N=31 – варіант за списком групи.

```
for (int i=0; i<size-1; i++) {
      if(a[i]%2==0)
      {int nmin = i;
        for (int j=i+1; j<size; j++){
            if (a[j]<a[nmin]&& (a[j]%2==0)) nmin = j;
        }
        if (i != nmin) {
            int t = a[nmin];
            a[nmin] = a[i];
            a[i] = t;
        }
    }
      }
x=a[3];
```

Розв’язання

Даний фрагмент програми виконує сортування парних елементів масиву за зростанням. Реалізація сортування виконана методом вибору, зміна x=−38.

Завдання 2.2. Дана послідовність цілих чисел $a_1, a_2, \dots, a_{35}$. Розташувати елементи послідовності, кратні 5 за зростанням (інші елементи залишаються на своїх місцях). Сортування виконати методом вибору.

Розв’язання

1. Постановка задачі

Дана послідовність цілих чисел $a_1, a_2, \dots, a_{35}$. Відсортувати за зростанням тільки елементи, які кратні 5.

2. Алгоритм розв’язання задачі

Алгоритм розв’язання задачі можна представити у вигляді такої послідовності дій:

Дія 1. Ввести елементи одновимірного масиву *a*.

Дія 2. Відсортувати елементи, які кратні 5.

Дія 3. Вивести упорядкований масив на екран.

3. Текст програми

```
#include <iostream>
#include <windows.h>
using namespace std;
void chooseSort(int a[], int size){
for (int i=0; i<size-1; i++) {
      if(a[i]%5==0)
      {int nmin = i;
        for (int j=i+1; j<size; j++){
            if (a[j]<a[nmin]&& (a[j]%5==0)) nmin = j;
        }
        if (i != nmin) {
            int t = a[nmin];
            a[nmin] = a[i];
            a[i] = t;
        }
    }
      }
}
void printArr(int a[], int size){
    for(int i=0; i<size; i++) {
        cout << a[i] << " ";
    }
    cout << "\n";
}
int main(){
   SetConsoleCP(1251);SetConsoleOutputCP(1251);
   int p[22], arr[22],n=22;
   p[0]=31;
   for(int i=0;i<n-1;i++)
   p[i+1]=(p[i]*11 + 7) % 100;
      for(int i=0;i<n;i++)
        arr[i]=p[i]-50;
```

```
    cout<<"До сортування:\n";
      printArr(arr, n);
      chooseSort(arr, n);
    cout<<"Після сортування:\n";
      printArr(arr, n);
      return 0;
}
```

4. Результати роботи програми

```
До сортування
-19 -2 -15 42 -31 -34 33 -30 -23 -46 1 18 5 -38 -11 -14 -47 -10
-3 -26 21 38
Після сортування
-19 -2 -30 42 -31 -34 33 -15 -23 -46 1 18 -10 -38 -11 -14 -47 5
-3 -26 21 38
```

Завдання 2.3. Пояснити, що виконує наступний фрагмента програми, де N=32.

```
int *a, n,*aa,t; a=form(n);
for (aa=a; a+n-1-aa>=0; aa+=2){
    t=*aa; *aa=*(aa+1); *(aa+1)=t; }
```

Розв’язання

Програма реалізує спочатку формування лінійного масиву

$$\{a_i\}, \text{де } i = \overline{1,12}$$

```
32 33 34 35 36 37 38 39 40 41 42 43
```

Наданий фрагмент програми переставляє місцями елементи з парними та непарними індексами.

```
33 32 35 34 37 36 39 38 41 40 43 42
```

Завдання 2.4. Видалити з масиву усі елементи, що співпадають з першим елементом. При складанні програми, використовувати динамічне розподілення пам’яті та вказівники.

Розв’язання

1. Постановка задачі

Скласти програму видалення з масиву усіх елементів, що співпадають з першим елементом на мові С++.

2. Алгоритм розв’язання задачі

Алгоритм розв’язання задачі можна представити у вигляді такої послідовності дій:

Дія 1. Виділити динамічну пам'ять для елементів масиву.

Дія 2. Заповнити масив випадковими числами.

Дія 3. Вивести елементи масиву на екран.

Дія 4. Визначити кількість елементів, що відрізняються від першого.

Дія 5. Виділити динамічну пам'ять для нового масиву.

Дія 6. У новий масив записати тільки ті елементи, що відрізняються від першого.

Дія 7. Звільнити пам'ять, що була виділена під старий масив.

Дія 8. Замінити значення вказівки

3. Текст програми

```
#include <iostream>
#include <stdlib.h>
#include <windows.h>
using namespace std;
int* form(int&n){
  cout<<"\nВведіть n=";
  cin>>n;
  int*a=new int[n];
  for(int i=0;i<n;i++)
       *(a+i)=rand()%100;
  return a;
}
void print(int*a,int n){
  for(int i=0;i<n;i++)
       cout<<*(a+i)<<" ";
  cout<<"\n";
}
int count0(int*a,int n){
```

```
int k=0,i=0;
for(;i<n;i++)
                if(*(a+i)!=*(a+0)) k++;
return k;
}
int*dell(int *a,int&n){
  int newsize=count0(a,n);
  int*b=new int [newsize];
  for(int j=0, i=0;i<n;i++)
       if(*(a+i)!=*(a+0))
       {
            *(b+j)=*(a+i);j++;
       }
    n=newsize;
    delete [] a;
    return b;
}
int main(){SetConsoleCP(1251);SetConsoleOutputCP(1251);
  int *a, n;     a=form(n);
    cout<<"До видалення:\n";print(a,n);
  a=dell(a,n);
  cout<<"Після видалення:\n";print(a,n);
    delete [] a;
  return 0;
}
```

4. Результати роботи програми

```
Введіть n=10
До видалення:
41 67 34 0 69 24 78 58 62 64
Після видалення:
67 34 0 69 24 78 58 62 64
```

Контрольні питання

1. Який основний принцип роботи алгоритму лінійного пошуку?
2. До якого масиву можна застосовувати алгоритм бінарного пошуку?

3. Який основний принцип роботи алгоритму бінарного пошуку?
4. Який основний принцип роботи алгоритму бульбашкового сортування?
5. Як працює алгоритм сортування вибором?
6. Як працює алгоритм сортування простими вставками?
7. Що таке вказівник?
8. Як працює унарна операція `&`?
9. Як працює унарна операція `*`?
10. Що таке посилальна змінна?
11. Як можна звернутися до певного елемента масиву через індекс?
12. Як можна звернутися до певного елемента масиву через вказівник?
13. Що таке динамічна пам'ять?