Компаратори для сортувань

Компаратор – це функція, призначення якої порівняти два об’єкти і повернути значення в залежності від результату порівняння. Якщо позначити об’єкти **a** та **b**, то можливі три варіанти результату порівняння – перший об’єкт менше, рівний або більший другого. Тоді функція повинна повертати три різних значення в залежності від результату порівняння. Домовимося закодувати значення функції в такий спосіб:

- **a>b – 1**
- **a=b – 0**
- **a<b - -1.**

І функція тоді буде виглядати так:

```
int compare(int a, int b)
{
   if(a>b) return 1;
   if(a<b) return -1;
   return 0;
}
```

Тепер розглянемо для прикладу два об’єкти типу «час доби». Очевидно, що кожен з них має три таких властивості: години, хвилини та секунди. Порівняння їх виконується в наступному порядку: спочатку порівнюються години без урахування хвилин та секунд, при рівності годин виконується порівняння хвилин і тільки при рівності годин та хвилин порівнюються секунди. Таке порівняння об’єктів називається порівнянням по трьох ключах і зберігати ці властивості зручно у масиві з трьох елементів: перший – години, другий – хвилини і третій – секунди.

З урахування алгоритму порівняння часу, описаного вище, функція буде виглядати так:

```
int compare(int a[], int b[])
{
   //Порівняння годин
   if(a[0]>b[0]) return 1;  //Години об’єкта a більше годин об’єкту b
   if(a[0]<b[0]) return -1;
   //Години рівні -> порівняння хвилин
   if(a[1]>b[1]) return 1;  //Хвилини об’єкта a більше годин об’єкту b
   if(a[1]<b[1]) return -1;
   // Години та хвилини рівні -> порівняння секунд
   if(a[3]>b[3]) return 1;  //Секунди об’єкта a більше годин об’єкту b
   if(a[3]<b[3]) return -1;
   return 0;
}
```

З точки зору реалізації сортування деякі з порівнянь є зайвими, оскільки нас цікавлять тільки дві ситуації:

- перша – переміну місцями двох елементів у масиві потрібно виконувати,
- друга – переміна місцями двох елементів у масиві не потрібна.

Тоді компаратор можна переписати таким чином, щоб він повертав логічне значення **`true`** або **`false`** у відповідності до першої або другої ситуації:

```
bool compare(int a[], int b[])
{
   if(a[0]>b[0]) return true;   //Переміна потрібна
   if(a[0]<b[0]) return false;  //Переміна не потрібна
   if(a[1]>b[1]) return true;   //Переміна потрібна
   if(a[1]<b[1]) return false;  //Переміна не потрібна
   if(a[3]>b[3]) return true;   //Переміна потрібна
   return false;
}
```

Тепер розглянемо використання наведеного компаратора для сортування. Для прикладу візьмемо базове сортування бульбашкою.

```
void bubble(int mas[][3], int n)
{
    for(int i = 1; i < n; ++i)
        for(int j = 0; j < n - i; ++j)
            if(compare(mas[j], mas[j+1]))
                swap (mas[j], mas[j+1]);
}
```

Як бачите, замість порівняння двох елементів масиву ставиться виклик функції **`compare`**.

Приклади компараторів для різних задач.

Задача №1. *Сортування модулів*.

Умова задачі: Потрібно відсортувати елементи масиву за неспаданням їх модуля. При рівності модулів елементи сортуються в звичайному порядку, тобто спочатку менші, потім більші.

Наприклад, якщо дано такий масив

```
4 -5 6 -4 5 7 8 0 -3
```

то після сортування буде отримано такий масив:

```
0 -3 -4 4 -5 5 6 7 8
```

Очевидно, що компаратор повинен спочатку порівнювати модулі чисел, а у випадку їх рівності значення.

```
bool compare(int a, int b)
{
   if(abs(a)>abs(b)) return true;   //Переміна потрібна
   if(abs(a)<abs(b)) return false;  //Переміна не потрібна
   //Модулі рівні
   if(a>b) return true;            //Переміна потрібна
   return false;
}
```

Задача №2. *Сортування по сумі цифр*.

Умова задачі: Потрібно відсортувати елементи масиву за неспаданням суми цифр числа. При рівності суми цифр елементи сортуються в звичайному порядку, тобто спочатку менші, потім більші.
Наприклад, якщо дано такий масив
4 -5 61 -4 5 17 8 0 -31
то після сортування буде отримано такий масив:
0 -31 -4 4 -5 5 61 8 17
На відміну від попереднього випадку стандартної функції, що знаходить суму цифр числа, не існує, а тому, по-перше, потрібно написати відповідну функцію.

```
int sum_dig(int N)
{
   int sum=0;
   N=abs(N);
   while(N != 0)
   {
     sum+=N%10;
     N/=10;
   }
   return sum;
}
```

А тепер компаратор. Він повинен спочатку порівнювати суми цифр чисел, а у випадку їх рівності значення чисел.

```
bool compare(int a, int b)
{
   if(sum_dig(a)>sum_dig(b)) return true;   //Переміна потрібна
   if(sum_dig(a)<sum_dig(b)) return false;  //Переміна не потрібна
   //Суми цифр рівні
   if(a>b) return true;            //Переміна потрібна
   return false;
}
```

Задача №3. *Сортування парні-непарні*.

Умова задачі: Потрібно відсортувати елементи масиву таким чином: спочатку всі парні числа за неспаданням, а потім всі непарні числа за неспаданням.
Наприклад, якщо дано такий масив
4 -5 61 -4 5 17 8 0 -31
то після сортування буде отримано такий масив:
-4 0 4 8 -31 -5 5 17 61
Для написання компаратора для цієї задачі потрібно розглянути всі можливі випадки розташування елементів у масиві. Поруч можуть знаходитися елементи в такому порядку:

- парний – непарний

- непарний – парний
- парний – парний
- непарний – непарний

У першому випадку переміну робити не потрібно, у другому – робиться безумовно, а ось у двох останніх – тільки якщо перше число більше другого. Таким чином, компаратор буде виглядати так:

```
bool compare(int a, int b)
{
   //Парність різна
   if(a%2 == 0 && b%2 != 0) return false;   //Переміна не потрібна
   if(a%2 != 0 && b%2 == 0) return true;   //Переміна потрібна
   //Парність однакова
   if(a>b) return true;            //Переміна потрібна
   return false;
}
```

Інший варіант визначення однакової парності:

```
bool comp(int a , int b) {
    if (a % 2 != 0 && b % 2 == 0) return true; //Перший парний, другий ні
    //Однакова парність
    if ((a+b) % 2 == 0 and a > b) return true;
    return false;
}
```

Компаратор, який використовує бітову арифметику для визначення парності:

```
bool compare (int a, int b)
{
  if(a & 1 && !(b & 1)) return true;
  if(b & 1 && !(a & 1)) return false;
  if(a > b) return true;
  return false;
}
```

Задача №4. *Сортування від’ємні-додатні*.

Умова задачі: Потрібно відсортувати елементи масиву таким чином: спочатку всі від’ємні числа за неспаданням, а потім всі додатні числа за незростанням.

Наприклад, якщо дано такий масив

```
4 -5 61 -4 5 17 8 0 -31
```

то після сортування буде отримано такий масив:

```
-31 -5 -4 0 61 17 8 5 4
```

Алгоритм розв’язання задачі схожий на попередній: потрібно розглянути всі можливі випадки розташування елементів у масиві. Поруч можуть знаходитися елементи в такому порядку:

- від’ємний – додатній
- додатній – від’ємний

- додатній – додатній
- від’ємний – від’ємний

У першому випадку переміну робити не потрібно, у другому – робиться безумовно, а ось у двох останніх так: для від’ємної пари – тільки якщо перше число більше другого, а для додатної – якщо перше число менше другого. Таким чином, компаратор буде виглядати так:

```
bool comp(int a,int bn2)
{
     if (a>0 && b<=0) return true;  //перше додатнє, а друге недодатнє – переміна потрібна
     if (a<=0 && b>0) return false; //перше недодатнє, а друге додатнє – переміна не потрібна
     if (a>0 && b>0 && a<b) return true; //обидва додатні і перше менше другого – переміна потрібна
    if (a<=0 && b<=0 && a>b) return true; //обидва недодатні і перше більше другого – переміна потрібна
    return false;
}
```

На цій задачі учні можуть отримати багато різних варіантів написання компараторів, які записуються іншими способами, не змінюючи при цьому сутності порівнянь. Можна, навіть, організувати змагання на отримання компаратора з найменшою кількістю порівнянь. Наприклад, нижче пропонується 7 варіантів різних компараторів, які виконують порівняння саме за умовою цієї задачі.

Варіант №1.

```
bool compare(int a, int b)
{
    if (a > 0 && b > 0 && a < b) return true;
    if (a <= 0 && b <= 0 && a > b) return true;
    if (a * b <= 0 && a > b) return true;  //добуток a*b буде недодатнім, якщо або одне число дорівнює нулю, або числа мають різні знаки
    return false;
}
```

Варіант №2.

```
bool compare(int a, int b)
{
      if ((a > 0) && (b <= 0)) return true;
      if ((a <= 0) &&  (b > 0)) return false;
      if (abs(a) < abs(b)) return true;
      return false;
}
```

Варіант №3.

```
bool compare(int a , int b )
{
    if(a>0 && b>0)
        return !(a>b);
    return (a>b);
}
```

Варіант №4.

```
bool compare(int a, int b){
    if (a >= 0 && b <= 0) return true;
    if (a < 0 && b < 0 && a > b) return true;
    if (a > 0 && b > 0 && a < b) return true;
    return false;
}
```

Варіант №5. //Один з тих, що складно сприймаються

```
bool compare (int num1, int num2)
{
    bool sgn_num1, sgn_num2;
    sgn_num1=(num1>0);
    sgn_num2=(num2>0);
    return !(!(sgn_num1 & sgn_num2)^(num1>num2));
}
```

Варіант №6.

```
bool compare(int a, int b) {
    if (a * b < 0 && a > b) return true;
    if (a * b > 0 && a > 0 && a < b) return true;
    if (a * b > 0 && a < 0 && a > b) return true;
    if (a * b == 0 && a > b) return true;
    return false;
}
```

Варіант №7. // «Найелегантніший»

```
bool comp(int a,int b)
{
    if(a>0&&b>0) return a<b;
    return a>b;
}
```