Podstawy Programowania

Wykład VIII

Obsługa błędów, przeszukiwanie i sortowanie tablic

Robert Muszyński
Katedra Cybernetyki i Robotyki, PWr

Zagadnienia: źródła błędów, organizacja obsługi błędów, standardowe funkcje obsługi błędów, przeszukiwanie tablic: liniowe, binarne, sortowanie tablic: przez wstawianie, drzewiaste, bąbelkowe, szybkie, przez scalanie.

Copyright © 2007–2014 Robert Muszyński

Niniejszy dokument zawiera materiały do wykładu na temat podstaw programowania w językach wysokiego poziomu. Jest on udostępniony pod warunkiem wykorzystania wyłącznie do własnych, prywatnych potrzeb i może być kopiowany wyłącznie w całości, razem ze stroną tytułową.

Ogólne uwagi o postępowaniu z błędami

- W naszych „małych” programach ilustracyjnych zazwyczaj nie martwiliśmy się o stan zakończonego programu.
- W „poważnym” programie koniecznie trzeba dbać o zwracanie sensownych i użytecznych wartości opisujących ten stan.
- Nie ma jednego, najlepszego sposobu przekazywania informacji o błędach pojawiających się w programie – można to robić zarówno centralnie jak i lokalnie.
- Do tego pojawiają się pytania: Czy jest obowiązkowe testowanie i obsługiwanie absolutnie wszystkich sytuacji nienormalnych? Czy próby wybrnięcia z nieoczekiwanych błędów mają w ogóle sens?
- Rozsądna zasada: jeśli wystąpił błąd to jesteśmy w kłopotach – lepiej nie kombinujmy za dużo, tylko starajmy się wybrnąć z całej sytuacji w najprostszy możliwy sposób. W braku lepszego pomysłu możemy WKZP (wyświetlić komunikat i zakończyć program).
- Inny wniosek: własne funkcje piszmy tak, aby w razie porażki ich użytkownik uzyskał informacje o miejscu i przyczynie powstania błędu i mógł podjąć właściwe decyzje co do sposobu dalszego postępowania.

Ogólne uwagi o postępowaniu z błędami cd.

- Ważne jest by przy wystąpieniu błędu program zwrócił odpwiednią wartość i/lub właściwie sformułowany komunikat o błędzie.
- Przyjmuje się, że zwracana przez program wartość zero świadczy o poprawnym wykonaniu programu, natomiast jakakolwiek niezerowa wartość sygnalizuje sytuację awaryjną.
- Komunikat wypisywany w przypadku błędu musi dawać użytkownikowi programu szansę na podjęcie dalszych działań. Dobra zasada: Mów to, o czym naprawdę myślisz, zdawaj sobie sprawę z tego, co mówisz, bądź zwięzły.
- Należy zadbać o to, by wypisywany komunikat nie „ugrzązł" gdzieś wśród danych wyjściowych lub w potoku.
- Rodzaje typowych błędów:
  - ⋆ błędy ochrony pamięci
  - ⋆ błędy zakresu
  - ⋆ błędy wejścia/wyjścia
  - ⋆ błędy obliczeniowe (dzielenie przez zero, przekroczenie zakresu itp.)
  - ⋆ błędy konwersji

Przykład – kopiowanie plików

```
#include <stdio.h>
main(int argc, char *argv[]) {
  FILE *fp1, *fp2;
  int c;
  fp1 = fopen(argv[1], "r");
  fp2 = fopen(argv[2], "w");
  while ((c = getc(fp1)) != EOF)
    putc(c,fp2);
}
```

Na pozór działa poprawnie, ale co będzie w przypadku jakiegoś błędu, np.:

- niepoprawnej liczby argumentów,
- niepoprawnej nazwy pliku(ów),
- braku pliku źródłowego,
- braku praw dostępu do pliku źródłowego,
- braku prawa do zapisu pliku docelowego,
- niedostępnego dysku jednego z plików,
- braku miejsca na dysku itd.

Uwzględniając błędy funkcji systemowych dostajemy:

```
main(int argc, char *argv[]) {          /* wersja 2: z wykrywaniem bledow */
  FILE *fp1, *fp2; int c;               /*            funkcji systemowych */
  if (argc != 3) {
    fprintf(stderr,"%s: wymagane 2 argumenty (podane %d)\n",argv[0],argc-1);
    exit(1);}
  if ((fp1 = fopen(argv[1], "r")) == NULL) {
    fprintf(stderr,"%s: blad otwarcia pliku %s do odczytu\n",argv[0],argv[1]);
    exit(2);}
  if ((fp2 = fopen(argv[2], "w")) == NULL) {
    fprintf(stderr,"%s: blad otwarcia pliku %s do zapisu\n",argv[0],argv[2]);
    exit(3);}
  while ((c = getc(fp1)) != EOF)
    if (putc(c, fp2) == EOF) {
      fprintf(stderr,"%s: blad zapisu na pliku %s\n",argv[0],argv[2]);
      exit(4);}
  if (ferror(fp1) != 0) {
    fprintf(stderr,"%s: blad czytania z pliku %s\n",argv[0],argv[1]);/*******/
    exit(5);}                                  /* pomijamy zamykanie plikow */
  exit(0);                                     /*    i bledy z tym zwiazane */
}                                              /*****************************/
```


Obsługa błędów – przykłady

Funkcja kopiująca napisy

```
/* kopiuj napis wskazywany przez wej do wyj */
/* PRE: poprawnie zainicjowane  wej  i  wyj */
/* POST:  skopiowanie  napisu  *wej na *wyj */
void KopiujNapis(char *wej, char *wyj)
{
  while ((*wyj++ = *wej++) != '\0');
}
```

Kopiowanie plików – całkiem podobne

```
/* kopiuj zawartosc pliku wej  do pliku wyj */
/* PRE: poprawnie zainicjowane  wej  i  wyj */
/* POST: skopiowanie strumienia wej  na wyj */
void KopiujPlik(FILE *wej, FILE *wyj)
{
  int c;
  while ((c = getc(wej)) != EOF)
    putc(c,wyj);
}
```

```
#include <stdio.h>           /* cat: sklej zawartosc plikow */
int main(int argc, char *argv[]) {
  FILE *fp;
  void KopiujPlik(FILE *, FILE *);
  char *prog = argv[0];    /* nazwa programu do komunikatow */
  if (argc == 1)        /* wywolanie bez arg.: kopiuj stdin */
    KopiujPlik(stdin, stdout);
  else
    while (--argc > 0)
      if ((fp = fopen(*++argv, "r")) == NULL) {
        fprintf(stderr, "%s: nie moge otworzyc %s\n", prog, *argv);
        exit(1);
      } else {
        KopiujPlik(fp, stdout);
        fclose(fp);
      }
  if (ferror(stdout)) {
    fprintf(stderr, "%s: wystapil blad zapisu stdout\n", prog);
    exit(2);
  }
  exit(0);
}
```


Uwagi o postępowaniu z błędami cd.

- Wszystkie błędy pojawiające się w programie można podzielić na „krytyczne” (zakończenie działania programu) i „niekrytyczne” (mimo wszystko coś się da zrobić).
- W programie można skonstruować „cetralny system obsługi błędów”, w którym to funkcje w przypadku napotkania błędu albo wywołują „centralną” funkcję obsługi błędów i przekazują jej sterowanie, albo odpowiednio przygotowują i zwracają kod błędu, który po dotarciu „na sam szczyt” obsługiwany jest przez odpowiednią funkcję obsługi błędów.
- Można także wykorzystać „rozproszony system obsługi błędów” – każda funkcja sama obsługuje napotkane błędy, wysyłając odpowiednie komunikaty i ewentualnie kończąc program.
- W każdym razie przy obsłudze błędów można wesprzeć się standardową funkcją `void perror(const char *s)`, która wypisuje tekst z tablicy `s` i zależny od implementacji komunikat o błędzie, odpowiadający wartości zmiennej `errno`, czy też funkcją `char *strerror(int nr)`, zwracającą wskaźnik do zależnego od implementacji tekstu komunikatu odpowiadającego błędowi o numerze `nr` (nagłówki `stdio.h`, `errno.h` i `string.h`).

```
main(int argc, char *argv[]) {          /* wersja 2: z wykrywaniem bledow */
  FILE *fp1, *fp2; int c;               /*            funkcji systemowych */
  if (argc != 3) {
    fprintf(stderr,"%s: wymagane 2 argumenty (podane %d)\n",argv[0],argc-1);
    exit(1);}
  if ((fp1 = fopen(argv[1], "r")) == NULL) {
    fprintf(stderr,"%s: blad otwarcia pliku %s do odczytu\n",argv[0],argv[1]);
    exit(2);}
  if ((fp2 = fopen(argv[2], "w")) == NULL) {
    fprintf(stderr,"%s: blad otwarcia pliku %s do zapisu\n",argv[0],argv[2]);
    exit(3);}
  while ((c = getc(fp1)) != EOF)
    if (putc(c, fp2) == EOF) {
      fprintf(stderr,"%s: blad zapisu na pliku %s\n",argv[0],argv[2]);
      exit(4);}
  if (ferror(fp1) != 0) {
    fprintf(stderr,"%s: blad czytania z pliku %s\n",argv[0],argv[1]);/*******/
    exit(5);}                                  /* pomijamy zamykanie plikow */
  exit(0);       /* M I E L I S M Y */         /* i bledy z tym zwiazane    */
}                                            /*****************************/
```



```
main(int argc, char *argv[]) {          /* wersja 3: wyswietlane          */
  FILE *fp1, *fp2; int c;               /*           komunikaty o bledach */
  if (argc != 3) {
    fprintf(stderr,"%s: wymagane 2 argumenty (podane %d)\n",argv[0],argc-1);
    exit(1);}
  if ((fp1 = fopen(argv[1], "r")) == NULL) {
    perror("blad otwarcia pliku do odczytu");
    exit(2);}
  if ((fp2 = fopen(argv[2], "w")) == NULL) {
    perror("blad otwarcia pliku do zapisu");
    exit(3);}
  while ((c = getc(fp1)) != EOF)
    if (putc(c, fp2) == EOF) {
      perror("blad zapisu na pliku");
      exit(4);}
  if (ferror(fp1) != 0) {
    perror("blad czytania z pliku");/*************************************/
    exit(5);}                       /* niby troche krocej, ale nie wszystko */
  exit(0);       /* M  A  M  Y */   /*            wyswietlamy co poprzednio */
}                                   /*************************************/
```

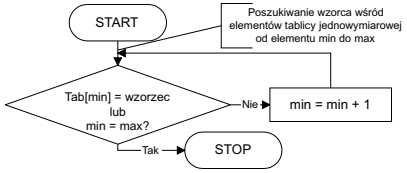
```
#include <errno.h>                                      /* wersja 4: jawnie formatowane */
int errno;                                              /*         komunikaty o bledach */
main(int argc, char *argv[]) {
  FILE *fp1, *fp2; int c;
  if (argc != 3) {
    fprintf(stderr, "%s: wymagane 2 argumenty (podane %d)\n",argv[0],argc-1);
    exit(1);
  }
  if ((fp1 = fopen(argv[1], "r")) == NULL) {
    fprintf(stderr,"%s: blad otwarcia do odczytu pliku %s, %s",argv[0],argv[1],strerror(errno));
    exit(2);
  }
  if ((fp2 = fopen(argv[2], "w")) == NULL) {
    fprintf(stderr,"%s: blad otwarcia do zapisu pliku %s, %s",argv[0],argv[2],strerror(errno));
    exit(3);
  }
  while ((c = getc(fp1)) != EOF)
    if (putc(c, fp2) == EOF) {
      fprintf(stderr, "%s: blad zapisu na pliku %s, %s", argv[0], argv[2], strerror(errno));
      exit(4);
    }
  if (ferror(fp1) != 0) {
    fprintf(stderr, "%s: blad czytania z pliku %s, %s", argv[0], argv[1], strerror(errno));
  ...           /* P O L A C Z E N I E   D W O C H   P O P R Z E D N I C H */
```


Obsługa błędów – podsumowanie

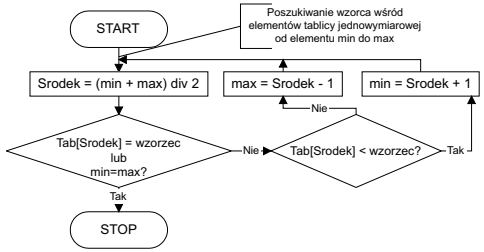
- Należy dbać o zwracanie sensownych i użytecznych komunikatów i wartości informujących o pojawiających się błędach.
- Zdecydowanie nie należy dążyć do obsłużenia wszystkich możliwych błędów.
- W przypadku interfejsu do systemu (czytanie plików, komunikacja między procesami) można ograniczyć obsługę błędów do rzeczy podstawowych – przy konstruowaniu interfejsu „do” użytkownika (menu, ręcznie wprowadzane dane) obsługa błędów powinna być bardziej rozbudowana.
- Porządny projekt programu pozwoli na uniknięcie wielu potknięć już na samym starcie – w tym zaplanowanie poprawnego systemu obsługi błędów.
- Informacje o błędach wysyłać w odpowiednie miejsca (`return`, `exit`, `fprintf(stderr,...)`).
- Wykorzystywać dostępne mechanizmy obsługi błędów (`perror`, `strerror`), pamiętając o ich właściwościach, np. o tym, że zmienna `errno` jest ustawiana przez funkcje, które wykrywają i sygnalizują sytuacje nienormalne, lecz nie zmienia wartości, gdy wynik działania funkcji jest poprawny (zatem wartość `errno` może odnosić się do wcześniejszego niż ostatnie wywołania funkcji, albo do późniejszego niż to, o które nam chodzi).

Przeszukiwanie tablic

- liniowe



- binarne



Przeszukiwanie liniowe

```
int Przeszukaj(int tab[],
               int Klucz,
               int min,max,domysl)
/* Wyszukuje wartosc Klucz w tablicy tab      */
/* pomiedzy indeksami min i max               */
/* Zwraca jej ind. lub domysl gdy !znaleziona */
{
  int znaleziony;

  znaleziony = 0;
  while ((!znaleziony) && (min <= max))
  {

    if (Porownanie(tab[min],Klucz))
      znaleziony = 1;
    else
      min := min + 1;


  }
  if (znaleziony)
    return min;
  else
    return domysl;
} /* Przeszukaj */
```

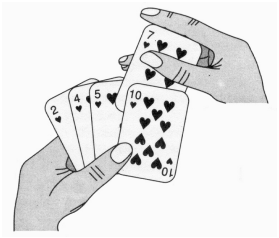
Przeszukiwanie binarne

```
/* ... POSORTOWANEJ */




srodek = (min + max) / 2;
switch (Porownanie(tab[srodek],Klucz)) {
  case  0: znaleziony = 1; break;
  case -1: max = srodek - 1; break;
  case  1: min = srodek + 1; break;
}
```

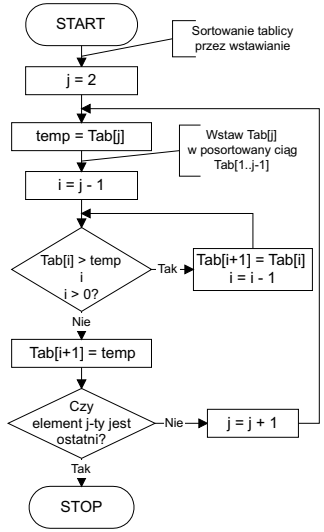
Sortowanie

- przez wstawianie
  - ⋆ przez proste wstawianie
  - ⋆ przez wstawianie połówkowe
- przez wybieranie
  - ⋆ drzewiaste
- przez zamianę
  - ⋆ bąbelkowe
  - ⋆ szybkie
- przez scalanie
- hybrydowe

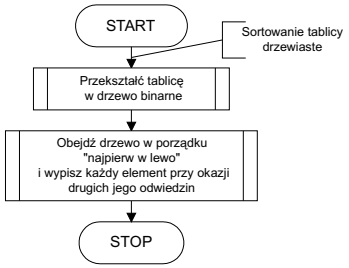
Sortowanie przez proste wstawianie



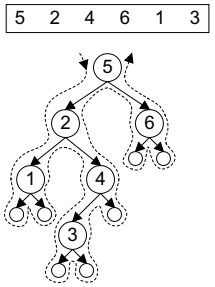
5	(2)	4	6	1	3
2	5	(4)	6	1	3
2	4	5	(6)	1	3
2	4	5	6	(1)	3
1	2	4	5	6	(3)
1	2	3	4	5	6



Sortowanie drzewiaste



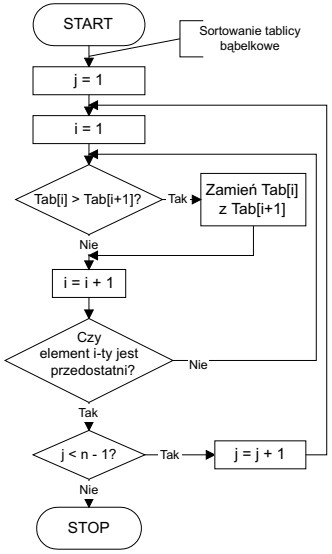
krok I

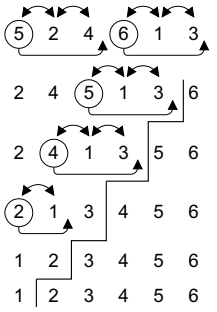


krok II

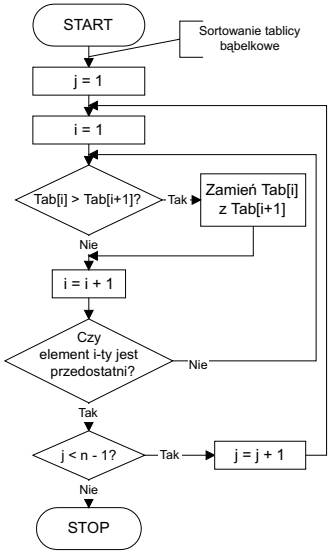
5 2 1 (1) 1 (2) 4 3 (3) 3 (4) 4 2 (5) 6 (6) 6 5

Sortowanie bąbelkowe



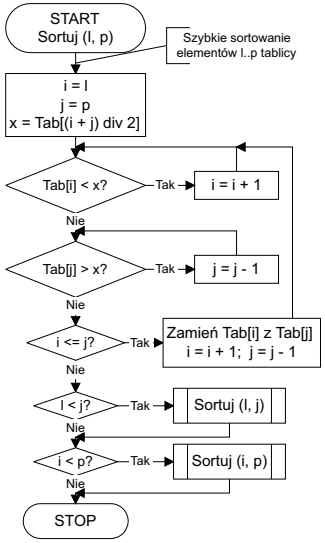


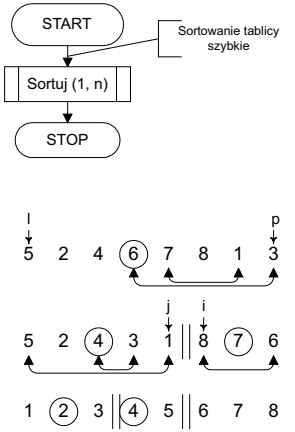
Sortowanie bąbelkowe




```
void BubbleSort(int Tab[],
                int min, max)
{
  int i, j;

  for(j = min; j < max; j++)
    for(i = min; i < max; i++)
      if (Tab[i] > Tab[i+1])
        Zamien(Tab[i], Tab[i+1]);
} /* BubbleSort */
```


Sortowanie szybkie





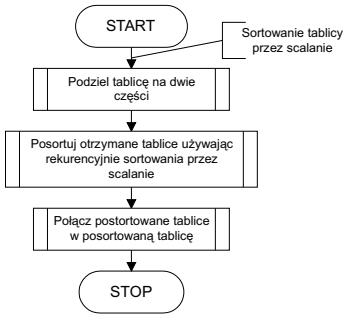
Sortowanie szybkie – funkcja standardowa

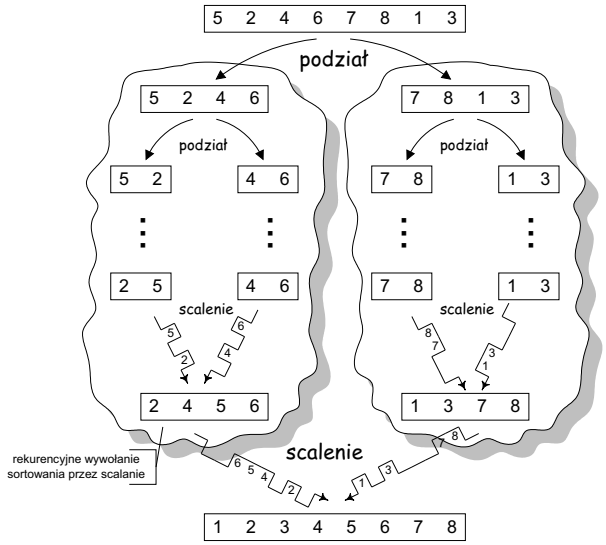
W nagłówku `stdlib.h` znajduje się funkcja

```
void  qsort(void  *base,  size_t  nel,  size_t  width,
            int (*compar)(const void *, const void *));
```

```
int porownaj_int(const void *p1, const void *p2) {
  int *i = (int *)p1;
  int *j = (int *)p2;

  if (*i > *j)  return (1);
  if (*i < *j)  return (-1);
  return (0); } /* porownaj_int */
#define TSIZE 10
int main() {
  int a[TSIZE] = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };

  qsort((void *)a, (size_t) TSIZE, sizeof(int), porownaj_int);
}
```


Sortowanie przez scalanie





Sortowanie hybrydowe

- Obserwacja:

  Sortowanie szybkie jest szybkie, ale nie dla małych tablic i nie gdy podziały w kolejnych iteracjach sortowania szybkiego są zdegenerowane

- Pomysł:

  Użyjmy w takich przypadkach innego, efektywniejszego sposobu sortowania

- Efekt – hybrydowe sortowanie introspektywne:

```
If (poziom rekurencji w sort. szybkim ponizej wyznaczonego)
  posortuj przez kopcowanie
podziel tablice w/g alg. sortowania szybkiego
If (lewa podtablica zawiera conajmniej 9 elementow)
  wywolaj rekurencyjnie proc. sortowania szybkiego dla niej
If (prawa podtablica zawiera conajmniej 9 elementow)
  wywolaj rekurencyjnie proc. sortowania szybkiego dla niej
posortuj przez wstawianie
```


Podsumowanie

- Zagadnienia podstawowe
  1. W jaki sposób można obsługiwać błędy wewnątrz programu?
  2. Porównaj wady i zalety scentralizowanego i rozproszonego systemu obsługi błędów w programie.
  3. Czy można zapisać dane do standardowego wyjścia błędów, nawet jeśli nie wystąpił żaden błąd w programie?
  4. Do czego służą funkcje `perror` i `strerror`? Jaka jest ich składnia?
  5. Jakie błędy mogą wystąpić w przedstawionej z opisem programu do przetwarzania obrazów (zobacz tabela z zadaniami na laboratorium) funkcji zapisującej obraz do pliku?
  6. Kiedy można stosować liniowe a kiedy binarne przeszukiwanie tablic?
  7. Jaka jest różnica złożoności obliczeniowej między przeszukiwaniem prostym a binarnym?
  8. Wymień wady i zalety sortowania bąbelkowego.
  9. Czym się różni sortowanie tablic od sortowania plików? Które z metod pokazanych na wykładzie mogą być użyte do sortowania tablic, a które do sortowania plików?

- Zagadnienia rozszerzające
  1. W jaki sposób można zwracać błąd wywołania funkcji (np. jak to jest zrobione w bibliotece `stdio`)?

2. Czy zawsze należy stosować sortowanie o najmniejszej złożoności obliczeniowej?
3. Przeanalizuj złożoność algorytmów sortowania tablic; jakie są mocne i słabe strony poszczególnych algorytmów?
4. Zapoznaj się z algorytmami sortowania przez kopcowanie, Shella, grzebieniowym; porównaj z algorytmami przedstawionymi na wykładzie. (wskazówka: `http://www.zgapa.pl/zga`

• Zadania

1. Przejrzyj napisane dotychczas programy pod kątem analizy możliwości wystąpienia błędów w ich działaniu. Dopisz fragmenty kodu, które zapobiegną ich wystąpieniu. Czy dopisany kod nie wymaga kolejnych zabiegów wykrywających sytuacje błędne?
2. Przeprowadź ręczną symulację algorytmu sortowania bąbelkowego dla następujących danych:
   `{1, 3, 56, 4, 90, 67, 3, 8, 12, 90}`
   `{2, 10, 2, 0, -12, 98, 67, 45, 0, 27, -12}`
3. Przeprowadź ręczną symulację algorytmu sortowania szybkiego dla następujących danych:
   `{1, 3, 56, 4, 90, 67, 3, 8, 12, 90}`
   `{2, 10, 2, 0, -12, 98, 67, 45, 0, 27, -12}`
4. Napisz funkcje implementujące (wybrane) funkcje sortowania tablicy, porównaj czas działania funkcji, liczbę porównań i zamian dla tablic wypełnionych losowymi elementami, a także wstępnie uporządkowanymi we właściwej i odwrotnej kolejności. Sprawdź, jak

zmieniają się wyniki dla różnych rozmiarów tablic.
5. Utworzyć i napisać program porządkujący ciąg n liczb całkowitych `a1, a2 ,...an` w taki sposób, aby liczby o wartościach parzystych znajdowały się na najpierw, a liczby o wartościach nieparzystych po nich. Zarówno liczby parzyste jak i nieparzyste mają być uporządkowane w kolejności rosnącej. Podać liczbę elementów nieparzystych.
6. Napisz program do przechowywania danych w postaci (data,wartość) z możliwością sortowania danych rosnąco lub malejąco, zarówno według daty, jak i wartości; wykorzystaj funkcję `qsort()`.
7. Napisz program, który znajduje najczęściej występujące elementy w ciągu danych. Na wejściu programu podawana jest pewna liczba zestawów danych (co najwyżej 1000). Każdy z zestawów ma postać: `n x1 x2 <85> xn`, gdzie `n` jest liczbą naturalną (z zakresu 1-1000), po której następuje `n` liczb całkowitych `x1 x2 <85> xn` (z zakresu od -1000 do 1000). Poszczególne liczby w zestawie zostaną rozdzielone znakiem spacji, a poszczególne zestawy znakiem nowej linii. Dla każdego z wczytanych zestawów należy wyznaczyć wartość, która w ciągu `x1 x2 <85> xn` występuje najczęściej i wypisać ją na wyjściu programu. Jeżeli takich wartości jest więcej należy je wypisać w kolejności rosnącej, rozdzielając liczby znakiem spacji. Wyniki dla poszczególnych zestawów należy rozdzielić znakiem nowej linii. Przykład:

Wejście:	Wyjście:
5 1 3 11 1 7	1
6 4 2 1 2 4 3	2 4
7 3 5 2 2 2 2 2	2
6 4 4 4 4 4 4	4

© R. Muszyński, 17 grudnia 2018