Rozdział 4.

Algorytmy sortowania

Tematem tego rozdziału będzie opis kilku bardziej znanych metod sortowania danych. O użyteczności tych zagadnień nie trzeba chyba przekonywać; każdy programista prędzej czy później musi mieć do czynienia z tym zagadnieniem. Algorytmy sortowania są znakomitym sposobem na poznawanie algorytmiki i zagadnień związanych ze złożonością obliczeniową oprogramowania.

Pokazane w tym rozdziale opisy algorytmów sortowania będą dotyczyły głównie tzw. *sortowania wewnętrznego*, używającego tylko pamięci operacyjnej komputera. W tym wydaniu książki zdecydowałem się również omówić problematykę tzw. *sortowania zewnętrznego*. Sortowanie zewnętrzne dotyczy sytuacji, z którą większość Czytelników być może nigdy się nie zetknie w praktyce programowania: ilość danych do posortowania jest tak ogromna, że niemożliwe jest ich umieszczenie w pamięci operacyjnej i użycie jednej z metod sortowania wewnętrznego. Mimo to zagadnienia omawiane w kontekście sortowania zewnętrznego są na tyle ciekawe od strony praktycznej, że już samo pobieżne zapoznanie się z nimi może wzbogacić naszą wiedzę o programowaniu o zupełnie nowe obszary.

W praktyce jednak pamięć komputerowa (tzw. RAM) i dyski twarde systematycznie tanieją i do większości zagadnień sortowanie wewnętrzne jest wystarczające[1]. Od kilku lat coraz głośniej zaczyna być o bazach danych całkowicie rezydujących w pamięci (dla zwiększenia sprawności), rzecz niewyobrażalna kilka lat temu w praktyce.

Potrzeba sortowania danych jest związana pośrednio z typowo ludzką chęcią gromadzenia (chcemy mieć dużo) i porządkowania (a jak już dużo posiadamy, to zaczyna się problem z kontrolowaniem, co i gdzie mamy). A sortowanie rozwiązuje w dużym stopniu ten ostatni problem.

Istotnym problemem w dziedzinie sortowania danych jest ogromna różnorodność algorytmów wykonujących to zadanie. Początkujący programista często nie jest w stanie samodzielnie dokonać wyboru algorytmu sortowania najodpowiedniejszego do konkretnego zadania. Jedno z możliwych podejść do tematu polegałoby zatem na krótkim opisaniu każdego algorytmu, wskazaniu jego wad i zalet oraz podaniu swoistego rankingu jakości. Wydaje mi się jednak, że tego typu prezentacja nie spełniłaby dobrze swojego zadania informacyjnego, a jedynie sporo zamieszała w głowie Czytelnika. Skąd to przekonanie? Z pobieżnych obserwacji wynika, że programiści raczej używają dobrze sprawdzonych klasycznych rozwiązań, takich jak np. sortowanie przez wstawianie, sortowanie bąbelkowe, sortowanie szybkie niż równie dobrych (jeśli nie lepszych) rozwiązań, które służą głównie jako tematy artykułów czy też przyczynki do badań porównawczych z dziedziny efektywności algorytmów.

---

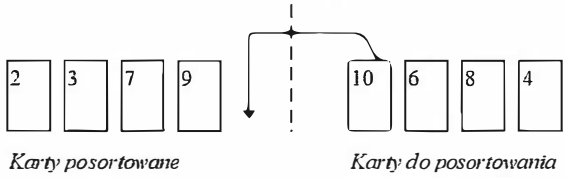
[1] Mój pierwszy prywatny komputer osobisty typu IBM PC XT miał 1 MB RAM-u i dysk twardy 20 MB!

Aby nie powiększać entropii wszechświata, skoncentrujemy się na szczegółowym opisie tylko kilku dobrze znanych, wręcz wzorcowych metod. Będą to algorytmy charakteryzujące się różnym stopniem trudności (rozpatrywanej w kontekście wysiłku poświęconego na pełne zrozumienie idei) i mające odmienne parametry czasowe. Wybór tych właśnie, a nie innych metod jest dość arbitralny i pozostaje mi tylko żywić nadzieję, że zaspokoi on potrzeby jak największej grupy Czytelników.

Sortowanie przez wstawianie, algorytm klasy O(N²)

Metoda sortowania przez wstawianie jest używana bezwiednie przez większość graczy podczas układania otrzymanych w rozdaniu kart. Rysunek 4.1 przedstawia sytuację widzianą z punktu widzenia gracza będącego w trakcie tasowania kart, które otrzymał on w dość podłym rozdaniu:

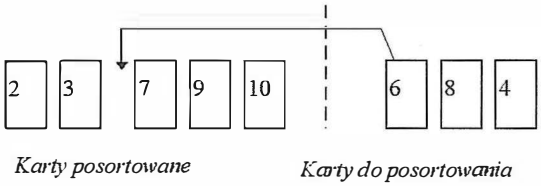
Rysunek 4.1. *Sortowanie przez wstawianie (1)*



Idea tego algorytmu opiera się na następującym niezmienniku: w danym momencie trzymamy w ręku karty posortowane[2] oraz karty pozostałe do posortowania. W celu kontynuowania procesu sortowania bierzemy pierwszą z brzegu kartę ze sterty nieposortowanej i wstawiamy ją na właściwe miejsce w pakiecie już wcześniej posortowanym.

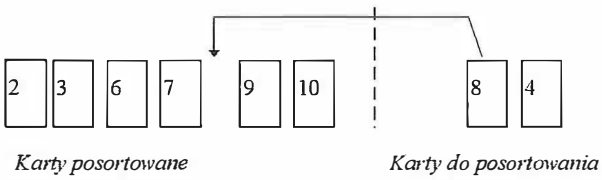
Popatrzmy na dwa kolejne etapy sortowania. Rysunek 4.2 obrazuje sytuację już po wstawieniu karty 10 na właściwe miejsce, kolejną kartą do wstawienia będzie 6.

Rysunek 4.2. *Sortowanie przez wstawianie (2)*



Tuż po poprawnym ułożeniu szóstki otrzymujemy układ z rysunku 4.3.

Rysunek 4.3. *Sortowanie przez wstawianie (3)*



Widać już, że algorytm jest nużąco jednostajny i... raczej dość wolny.

[2] Na samym początku algorytmu możemy mieć puste ręce, ale dla zasady twierdzimy wówczas, że trzymamy w nich zerową liczbę kart.

Ciekawa odmiana tego algorytmu realizuje *wstawianie* poprzez przesuwanie zawartości tablicy w prawo o jedno miejsce w celu wytworzenia odpowiedniej luki, w której następnie umieszcza ów element. Skąd mamy wiedzieć, czy kontynuować przesuwanie zawartości tablicy podczas poszukiwania luki? Podjęcie decyzji umożliwi nam sprawdzanie warunku sortowania (sortowanie w kierunku wartości malejących, rosnących czy też wg innych kryteriów).

Popatrzmy na tekst programu[3]:


Listing

insert.cpp

```
void InsertSort(int *tab)
{
for(int i=1; i<n; i++)
 {
 int j=i; // fragment [0..., i-1] jest już posortowany
 int temp=tab[j];
 while ((j>0) && (tab[j-1]>temp))
    {
    tab[j]=tab[j-1];
    j--;
    }
 tab[j]=temp;
 }
}
```

Algorytm sortowania przez wstawianie charakteryzuje się dość wysokim kosztem: jest on bowiem klasy $O(N^2)$, co eliminuje go w praktyce z sortowania dużych tablic. Niemniej jeśli nie zależy nam na szybkości sortowania, a potrzebujemy algorytmu na tyle krótkiego, by się w nim na pewno nie pomylić — to wówczas jest on idealny w swojej niepodważalnej prostocie.


Uwaga

> Dla prostoty przykładów będziemy analizować jedynie sortowanie tablic liczb całkowitych. W rzeczywistości sortowaniu podlegają najczęściej tablice lub listy rekordów; kryterium sortowania odnosi się wówczas do jednego z pól rekordu. (Patrz również rozdział 5. i zagadnienie sortowania list).

Sortowanie bąbelkowe, algorytm klasy O(N²)

Podobnie jak sortowanie przez wstawianie, algorytm sortowania bąbelkowego charakteryzuje się olbrzymią prostotą zapisu. Intrygująca jego nazwa wzięła się z analogii pęcherzyków powietrza ulatujących w górę tuby wypełnionej wodą — o ile postawioną pionowo tablicę potraktować jako pojemnik z wodą, a liczby jako pęcherzyki powietrza. Najszybciej ulatują do góry „bąbelki" najlżejsze — liczby o najmniejszej wartości (przyjmując oczywiście sortowanie w kierunku wartości niemalejących). Oto pełny tekst programu:


Listing

bubble.cpp

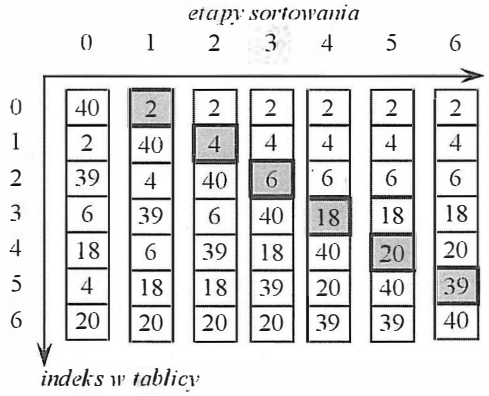
```
void bubble(int *tab)
{
for (int i=1; i<n; i++)
   for (int j=n-1;j>=i;j--)
```

[3] W tym i dalszych przykładach zakładam, że w kodzie jest umieszczona instrukcja `const int n=jakaś wartość` określająca rozmiar tablicy przeznaczonej do posortowania — pomijam to w wydrukowanych listingach.

```
        if (tab[j]<tab[j-1])
          { //zamiana
          int tmp=tab[j-1];
          tab[j-1]=tab[j];
          tab[j]=tmp;
          }
  }
```

Przeanalizujmy dokładnie sortowanie bąbelkowe pewnej 7-elementowej tablicy. Na rysunku 4.4 element zacieniowany jest tym, który w pojedynczym przebiegu głównej pętli programu „uleciał" do góry jako najlżejszy. Tablica jest przemiatana sukcesywnie od dołu do góry (pętla zmiennej `i`). Analizowane są zawsze dwa sąsiadujące ze sobą elementy (pętla zmiennej `j`): jeśli nie są one uporządkowane (u góry jest element „cięższy"), to następuje ich zamiana. W trakcie pierwszego przebiegu na pierwszą pozycję tablicy (indeks `0`) ulatuje element „najlżejszy", w trakcie drugiego przebiegu drugi najlżejszy wędruje na drugą pozycję tablicy (indeks `1`) i tak dalej, aż do ostatecznego posortowania tablicy. Strefa pracy algorytmu zmniejsza się zatem o *1* w kolejnym przejściu dużej pętli — analizowanie za każdym razem całej tablicy byłoby oczywistym marnotrawstwem!

Rysunek 4.4.
Sortowanie „bąbelkowe"



Nawet dość pobieżna analiza prowadzi do kilku negatywnych uwag na temat samego algorytmu:

- Dość często zdarzają się „puste przebiegi" (nie jest dokonywana żadna wymiana, bowiem elementy są już posortowane).
- *Algorytm* jest bardzo wrażliwy na konfiguracje danych. Oto przykład dwóch niewiele różniących się tablic, z których pierwsza wymaga jednej zamiany sąsiadujących ze sobą elementów, a druga będzie wymagać ich aż sześciu:
  - **wersja 1:** *4 2 6 18 20 39 40*,
  - **wersja 2:** *4 6 18 20 39 40 2*.

Istnieje kilka możliwości poprawy jakości tego algorytmu — nie prowadzą one co prawda do zmiany jego klasy (w dalszym ciągu mamy do czynienia z $O(N^2)$), ale mimo to dość znacznie go przyspieszają. Ulepszenia te polegają odpowiednio na:

- zapamiętywaniu indeksu ostatniej zamiany (walka z „pustymi przebiegami");
- przełączaniu kierunków przeglądania tablicy (walka z niekorzystnymi konfiguracjami danych).

Tak poprawiony algorytm sortowania bąbelkowego nazwiemy sobie po polsku *sortowaniem przez wytrząsanie* (ang. *shaker-sort*). Jego pełny tekst jest zamieszczony poniżej, lecz tym razem już bez tak dokładnej analizy, jak poprzednio:

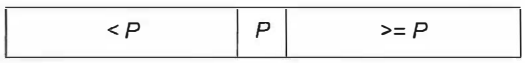
Listing *shaker.cpp*

```
void ShakerSort(int *tab)
{
int left=1, right=n-1, k=n-1;
do
{
for(int j=right; j>=left; j--)
   if(tab[j-1]>tab[j])
   {
   swap(tab[j-1],tab[j]);
   k=j;
   }
left=k+1;
for(int j=left; j<=right; j++)
   if(tab[j-1]>tab[j])
   {
   swap(tab[j-1],tab[j]);
   k=j;
   }
right=k-1;
}
while (left<=right);
}
```

Quicksort, algorytm klasy O(N log N)

Jest to słynny algorytm[4], zwany również po polsku sortowaniem szybkim. Należy on do tych rozwiązań, w których poprzez odpowiednią dekompozycję osiągnięty został znaczny zysk szybkości sortowania. Procedura sortowania dzieli się na dwie zasadnicze części: część służącą do właściwego sortowania, która nie robi w zasadzie nic oprócz wywoływania samej siebie, oraz procedury rozdzielania elementów tablicy względem wartości pewnej komórki tablicy służącej za oś (ang. *pivot*) podziału. Proces sortowania jest dokonywany przez tę właśnie procedurę, natomiast rekurencja zapewnia poskładanie wyników cząstkowych i w konsekwencji posortowanie całej tablicy.

Jak dokładnie działa procedura podziału? Otóż w pierwszym momencie odczytuje się wartość elementu osiowego `P`, którym zazwyczaj jest po prostu pierwszy element analizowanego fragmentu tablicy. Tenże fragment tablicy jest następnie dzielony, tak aby elementy mniejsze od 'P' znalazły się po lewej stronie, a większe — po prawej[5]. Ten nowy układ jest symbolicznie przedstawiony na rysunku 4.5.



Rysunek 4.5. *Podział tablicy w metodzie Quicksort*

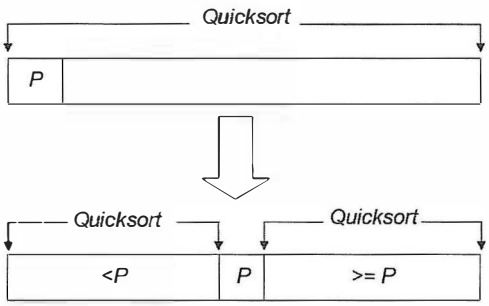
Kolejnym etapem jest zaaplikowanie procedury `Quicksort` na lewym i prawym fragmencie tablicy, czego efektem będzie jej posortowanie. To wszystko!

Na rysunku 4.6 są przedstawione symbolicznie dwa główne etapy sortowania metodą *Quicksort* (`P` oznacza tradycyjnie komórkę tablicy służącą za oś).

Jest chyba dość oczywiste, że wywołania rekurencyjne zatrzymają się w momencie, gdy rozmiar fragmentu tablicy wynosi 1 — nie ma już bowiem czego sortować.

[4] Patrz C.A.R. Hoare — „Quicksort" w Computer Journal, 5, 1(1962).

[5] Elementy tablicy są fizycznie przemieszczane, jeśli zachodzi potrzeba.



Rysunek 4.6. *Zasada działania procedury Quicksort*

Przedstawiona powyżej metoda sortowania charakteryzuje się olbrzymią prostotą, wyrażoną najdoskonalej przez zwięzły zapis samej procedury (prawdziwy kod znajduje się nieco dalej):

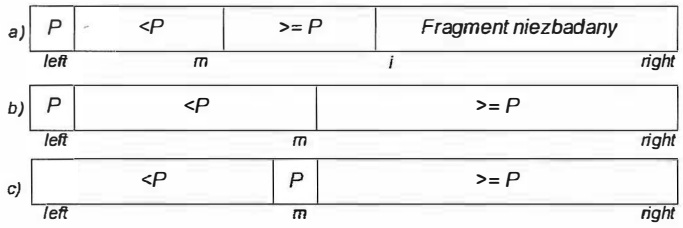
```
void Quicksort(int *tab, int left, int right)
{
if (left < right)
   {
   int m;
   // Podziel tablicę względem elementu osiowego 'P'
   // ('P' może być np. pierwszym elementem tablicy, czyli tab[left])
   // Pod koniec podziału element 'P' zostanie przeniesiony do komórki o numerze 'm'
   Quicksort(tab,left,m-1);
   Quicksort(tab,m+1,right);
   }
}
```

Jak najprościej zrealizować fragment procedury sprytnie ukryty za komentarzem? Jego działanie jest przecież najistotniejszą częścią algorytmu, a my jak dotąd traktowaliśmy go dość ogólnikowo. Takie postępowanie wynikało z dość prozaicznej przyczyny — implementacji algorytmu *Quicksort* jest mnóstwo i różnią się one właśnie realizacją procedury podziału tablicy względem wartości osi.

Oszczędzając Czytelnikowi dylematów dotyczących wyboru właściwej wersji, zaprezentujemy poniżej — zgodnie z prawdziwymi zasadami współczesnej demokracji — tę najwłaściwszą.

Kryteriami wyboru były: piękno, szybkość i prostota — tych cech można niewątpliwie doszukać się w rozwiązaniu przedstawionym w **[Ben92]**.

Pomysł opiera się na zachowaniu dość prostego niezmiennika w aktualnie rozdzielanym fragmencie tablicy (patrz rysunek 4.7).



Rysunek 4.7. *Budowa niezmiennika dla algorytmu Quicksort*

Oznaczenia:

`left` — lewy skrajny indeks aktualnego fragmentu tablicy;

`right` — prawy skrajny indeks aktualnego fragmentu tablicy;

`P` — wartość osiowa (zazwyczaj będzie to `tab[left]`);

`i` — indeks „przechadzający się" po indeksach tablicy od `left` do `right`;

`m` — poszukiwany indeks komórki tablicy, w której umieścimy element osiowy.

Przemieszczanie się po tablicy służy do poukładania jej elementów w taki sposób, aby po lewej stronie `m` znajdowały się wartości mniejsze od elementu osiowego, po prawej zaś — większe lub równe (rysunek 4.7a). W tym celu podczas przemieszczania indeksu `i` sprawdzamy prawdziwość warunku `tab[i]>P`. Jeśli jest on fałszywy, to poprzez inkrementację `i` wymianę wartości `tab[m]` i `tab[i]` przywracamy porządek. Gdy zakończymy ostatecznie przeglądanie tablicy (rysunek 4.7b) w pogoni za komórkami, które nie chciały się podporządkować niezmiennikowi, zamiana `tab[left]` i `tab[m]` doprowadzi do oczekiwanej sytuacji (rysunek 4.7c).

Nic już teraz nie stoi na przeszkodzie, aby zaproponować ostateczną wersję procedury `Quicksort`. Omówione wcześniej etapy działania algorytmu zostały połączone w jedną procedurę:

Listing

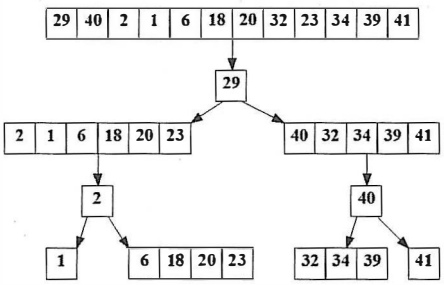
quick.cpp

```
void qsort(int *tab, int left, int right)
{
if (left < right)
   {
   int m=left;
   for (int i=left+1;i<=right;i++)
     if (tab[i]<tab[left])
        swap(tab[++m],tab[i]);
   swap(tab[left],tab[m]);
   qsort(tab,left,m-1);
   qsort(tab,m+1,right);
  }
}
```

W celu dobrego zrozumienia działania algorytmu spróbujmy posortować za jego pomocą ręcznie małą tablicę, np. zawierającą następujące liczby:

```
29, 40, 2, 1, 6, 18, 20, 32, 23, 34, 39, 41.
```

Rysunek 4.8 przedstawia efekt działania tych egzemplarzy procedury `Quicksort`, które faktycznie coś robią.



Rysunek 4.8. *Sortowanie metodą Quicksort na przykładzie*

Widać wyraźnie, że przechodząc od skrajnie lewej gałęzi drzewa do skrajnie prawej i odwiedzając w pierwszej kolejności lewe jego odnogi, przechadzamy się w istocie po posortowanej tablicy! W naszym programie taki spacer realizują wywołania rekurencyjne procedury `qsort`. Algorytm *Quicksort* stanowi dobry przykład techniki programowania zwanej „dziel i zwyciężaj", która zostanie dokładniej omówiona w rozdziale 9.

Tutaj zapowiem jedynie, że chodzi o taką dekompozycję problemu, aby osiągnąć zysk czasowy wykonywania programu (a przy okazji uproszczenie rozwiązywanego zadania). Algorytm *Quicksort* wzorowo spełnia oba te wymagania!

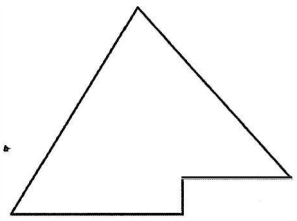
Heap Sort — sortowanie przez kopcowanie

Algorytm sortowania przez kopcowanie (zaraz wyjaśnię ten przedziwny termin!) jest klasy *O(N log N)*. Używa on ciekawej struktury danych zwanej kopcem, której charakterystykę zaraz omówię. Zaletą tego algorytmu jest właśnie ta struktura danych, która działa jednocześnie jak kolejka priorytetowa (patrz rozdział 5.).

Przedstawmy zatem nową strukturę danych, trochę jako element kolejnego rozdziału poświęconego właśnie strukturom danych. Gdyby Czytelnik miał kłopoty terminologiczne, zachęcam co najmniej do pobieżnego przejrzenia kolejnego rozdziału, gdzie struktura ta jest omówiona dokładniej, zaprezentowana jest także kolejna wersja samego algorytmu w notacji obiektowej. W tym rozdziale zaprezentuję bardziej zwięzłą wersję tego samego algorytmu, gdyż chciałbym skoncentrować się bardziej na algorytmie sortowania niż na strukturze danych.

Kopiec czy też, jak niektórzy wolą, sterta jest drzewem binarnym zawierającym liczby lub dowolne inne elementy dającego się porządkować zbioru. Cechą kopca jest kształt przedstawiony na rysunku 4.9 — jest to piramidka, w której podstawa z prawej strony jest „nadgryziona".

Rysunek 4.9. *Kopiec i jego własności (1)*



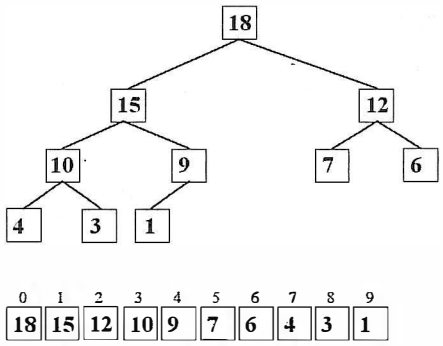
Oprócz kształtu kluczową cechą kopca jest uporządkowanie — każda wartość w węźle poniżej danego węzła jest od niej mniejsza. W notacji tablicowej warunek ten można zapisać jako: $T[ojciec(i)] \geq T[i]$.

Zawartość kopca łatwo reprezentujemy w zwykłej tablicy `T` wg następujących reguł:

- Wierzchołek kopca wstaw do `T[0]`.
- Dla dowolnego (jeśli istnieje) węzła w `T[i]` jego lewy syn jest w `T[2i+1]`, prawy w `T[2i+2]`.

Przykład takiej struktury znajdziemy na rysunku 4.10.

Rysunek 4.10. *Kopiec i jego własności (2)*



Algorytm sortowania przez kopcowanie zapisuje się następująco:

1. Ułóż dane w kopiec — niech znajdą się one w tablicy o rozmiarze `rozmiar`.
2. Usuń wierzchołek z kopca (jest to największy element) poprzez zamianę go z ostatnim liściem drzewa (`rozmiar--`).

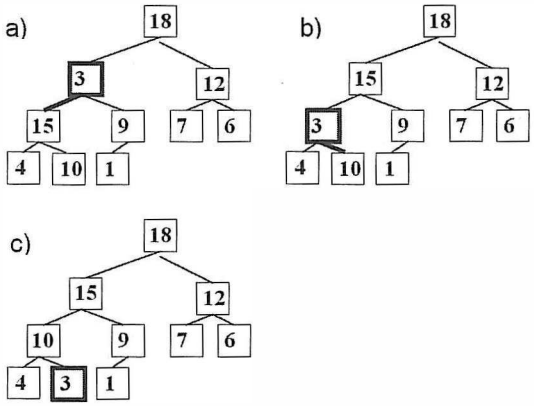
3. Przywróć własność kopca dla pozostałej (oprócz usuniętego elementu) części kopca.
4. Idź do pkt. 2.

Procedura z pkt. 3. jest zaimplementowana w następujący sposób:

1. Jeśli wierzchołek jest większy od obojga dzieci, wyjdź.
2. Zamień wierzchołek z większym dzieckiem.
3. Przywróć własność kopca w tej części kopca, w której nastąpiła zamiana.

Rysunek 4.11 pokazuje efekt działania procedury z pkt. 3. zaaplikowanej na pogrubionym węźle, który zaburza warunek $T[ojciec(i)] \geq T[i]$. W kolejnych krokach a), b) i c) poprzez zamianę wierzchołka z największym dzieckiem doprowadzamy do uporządkowania struktury.

Rysunek 4.11. *Przywracanie własności kopca na przykładzie*



Prosta realizacja algorytmu sortowania przez kopcowanie przedstawiona jest poniżej.

Listing

heap.cpp

```
void przywroc(int T[], int k, int n)
{
int i,j;

i = T[k-1];
while (k <= n/2)
   {
   j=2*k;
   if ((j<n) && ( T[j-1]<T[j]) ) j++;
   if (i >= T[j-1])
      break;
      else
         {
         T[k-1] = T[j-1];
         k=j;
         }
   }
T[k-1]=i;
}
// sortowanie tablicy T
void heapsort(int T[], int n)
{
int k, swap;
for(k=n/2; k>0; k--) przywroc(T, k, n);
```

```
do
   {
   swap=T[0];
   T[0]=T[n-1];
   T[n-1]=swap;
   n--;
   przywroc(T, 1, n);
   } while(n > 1);
}
int main()
{
int i, T[14]={12,3,-12,9,34,23,1,81,45,17,9,23,11,4};
for (i=0; i<14; i++)
   cout << "  " << T[i];

cout << endl;

heapsort(T,14);

for (i=0;i<14;i++)
   cout << "  " << T[i];
}
```

Scalanie zbiorów posortowanych

Prosty algorytm omówiony w tym punkcie zajmuje się sortowaniem dość szczególnego przypadku: tablice wejściowe $T_1[N]$ i $T_2[M]$ są uporządkowane i naszym zadaniem jest utworzenie tablicy $T_3[N+M]$, która także będzie uporządkowana.

Sam algorytm wydaje się prosty: należy pobierać dane z obu tablic, aż do ich „wyczerpania", przenosząc do tablicy wynikowej zawsze element najmniejszy (jeśli interesuje nas taki porządek danych). Algorytm „nie traci czasu" na odszukiwanie elementów do porównania, gdyż zakładamy, że dane wejściowe są faktycznie posortowane. Jest to o tyle ważne, iż w procedurze, którą pokażę dalej, nie jest w żadnym miejscu dokonywana weryfikacja tego założenia!

Popatrzmy na sam program:


Listing

scalaj.cpp

```
void scalaj(int T1[], int T2[], int T3[])
{ // T1, T2 — tablice wejściowe o rozmiarze M i M, T3 — tablica wynikowa
for (int i=0, j=0, k=0; k < N+M; k++)
   {
   if (i==N) // osiągnięto koniec zbioru T1, kopiujemy resztę
      {
      T3[k]=T2[j++]; continue;
      }
   if (j==M) // osiągnięto koniec zbioru T2, kopiujemy resztę
      {
      T3[k]=T1[i++]; continue;
      }
   if (T1[i]<T2[j])
      T3[k]=T1[i++];
   else
      T3[k]=T2[j++];
   }
}
```

Zauważmy, że dostęp do danych jest sekwencyjny, co przypomina nieco... pliki, zazwyczaj odczytywane w taki właśnie sposób!

W ramach drobnego ćwiczenia proponuję przerobić kod procedury operującej na tablicach na jego odpowiednik plikowy. Nie jest to aż takie trywialne, gdyż pełna analogia tablicowa nie może być zastosowana (zakładamy nieznane nam rozmiary plików wejściowych). Jeśli masz kłopoty z używaniem funkcji operujących na plikach, spójrz do dodatku A, gdzie podany jest program odczytujący dane z pliku.

Sortowanie przez scalanie

Kolejną metodą sortowania, którą omówimy, jest — podobnie jak *Quicksort* — metoda rekurencyjna z gatunku „dziel i zwyciężaj" (omówiona szerzej w rozdziale 9.). Algorytm jest prosty:

- Dzielimy *n*-elementowy ciąg na dwa podciągi po *n/2* elementów każdy.
- Sortujemy metodą przez scalanie każdy z podciągów osobno.
- Łączymy posortowane podciągi w jeden posortowany ciąg.

Oczywiście cała zabawa polega w tej metodzie na zgrabnym napisaniu procedury scalającej. Poniżej zamieściłem jedną z możliwych implementacji takiej procedury:


Listing

merge.cpp

```
const int N = 10;
void Scalaj(int T[], int p, int mid, int k)
// p — początek, k— koniec, mid— środek
// procedura łączy 2 posortowane tablice T[p..mid] i T[mid+1...k]
{
int T2[N];                          // tablica pomocnicza
int p1 =  p,     k1  = mid; // podtablica 1
int p2 =  mid+1, k2  = k;   // podtablica 2
// aż do wyczerpania tablic dokonaj scalenia przy pomocy tablicy pomocniczej
int i = p1;
while((p1 <= k1) && (p2 <= k2))
   {
   if(T[p1] < T[p2])
      {
      T2[i] = T[p1];
      p1++;
      }
   else
      {
      T2[i] = T[p2];
      p2++;
      }
   i++;
}

while(p1 <= k1)
   {
   T2[i] = T[p1];
   p1++;
   i++;
   }
while(p2 <= k2)
{
   T2[i] = T[p2];
   p2++;
   i++;
}
```

```
//skopiuj z tablicy tymczasowej do oryginalnej
for(i = p; i <= k; i++)
   T[i] = T2[i];
}
```

Sama procedura sortująca jest oczywiście powtórzeniem podanego wyżej schematu rekurencyjnego:

```
void MergeSort(int T[], int p, int k)
{
if(p < k)
   {
   int mid = (p + k) / 2;  //środek
   MergeSort(T, p, mid);   //sortuj lewą połowę
   MergeSort(T, mid+1, k); //sortuj prawą połowę
   Scalaj(T, p, mid, k);   //scalaj
   }
}
int main()
{
int T[N] = {4, 6, 4, 12, -3, 6, -6, 1, 8, '2'};
cout << "Przed sortowaniem:\n";
for(int x=0; x<N; x++) cout << T[x] << " "; cout << endl;
MergeSort(T, 0, N-1);
cout << "Po sortowaniu:\n";
for(int x=0; x<N; x++) cout << T[x] << " "; cout << endl;
}
```

Wyniki:

```
Przed sortowaniem:
4 6 4 12 -3 6 -6 1 8 50
Po sortowaniu:
-6 -3 1 4 4 6 6 8 12 50
```

Sortowanie zewnętrzne

Sortowanie zewnętrzne występuje, gdy rozmiar danych znacznie przekracza pojemność pamięci. Czy taka sytuacja może wystąpić w praktyce? Owszem, ale warto od razu zwrócić uwagę, że systemy informatyczne, w których można napotkać takie przypadki, są dość niszowe: np. wielkie bankowe systemy billingowe lub bankowe systemy finansowe. Czy jednak bank lub firma telekomunikacyjna użyje wiedzy opisanej w tym podrozdziale? Odpowiedź jest przewrotna: NIE.

Oto powody:

- Wielkie systemy informatyczne używają od dawna architektury *n*-warstwowej, której elementem jest serwer (lub farma serwerów) bazy danych. W bazie danych ciężar sortowania może zostać przerzucony na motor bazy danych, np. obok tabel danych zakłada się tzw. indeksy, które mogą zostać użyte do przeglądania danych w wersji posortowanej.
- Niektóre systemy (np. bankowe, oparte na komputerach mainframe) stosują COBOL, język programowania wykorzystywany do tworzenia dużych aplikacji biznesowych, opracowany w latach 60-tych, który pozwala na sprawne operowanie na plikach, także sortowanie.

Mimo to teoretyczny model sortowania zewnętrznego jest dość ciekawy i warto rzucić na niego okiem, choćby tylko w celu zapoznania się z problemami, które muszą być rozwiązane przez programistę.

Jak zauważymy, samo sortowanie zewnętrzne może być traktowane jako pewna *technika* programowania, gdyż do właściwej operacji sortowania, tej wykonywanej w pamięci operacyjnej, są wykorzystywane klasyczne, szybkie algorytmy sortowania wewnętrznego, a cały ciężar algorytmów spoczywa na umiejętnym manipulowaniu plikami roboczymi.

Nasze założenia modelu sortowania zewnętrznego zakładają następujące warunki i ograniczenia:

- Występują bardzo duże dysproporcje rozmiaru pamięci operacyjnej komputera (jest ona *ograniczona*) i zbioru danych do posortowania (teoretycznie *nieograniczony* lub ogólnie bardzo duży).
- Utrudniony jest dostęp do elementów przeznaczonych do sortowania, gdyż znajdują się one np. w plikach, które trzeba odczytywać przy pomocy mechanizmów systemu operacyjnego, zazwyczaj sekwencyjnie lub co najwyżej blokami.
- Czasy przetwarzania algorytmu sortującego w pamięci są pomijalne w porównaniu z czasami odczytu i zapisu danych z i do plików zewnętrznych.
- Fizyczne ograniczenia struktury dysków sprawiają, że korzystne jest odczytywanie danych z pliku wejściowego w większych blokach (stronach) pamięci. W związku z tym zawartość pliku dyskowego można traktować jako swego rodzaju listę złożoną z serii poszczególnych bloków.
- Do wykonywania sortowania fragmentów pliku mieszczących się w pamięci operacyjnej używa się serii buforów wejściowych, czyli zarezerwowanego fragmentu pamięci operacyjnej, w której system operacyjny umieszcza odczytany z dysku blok danych lub z których pobiera blok danych do zapisania na dysku.

Posiadając taką moc przerobową, możliwości odczytywania i zapisywania plików oraz dysponując pewnym szybkim klasycznym algorytmem do sortowania *wewnętrznego*, można zaproponować np. metodę sortowania polegającą na *rozdzieleniu i scalaniu połączonym z sortowaniem*:

- Dzielimy duży plik wejściowy na mniejsze, cząstkowe pliki, takie, które bez problemu mieszczą się w pamięci głównej.
- Sortujemy pliki cząstkowe znaną nam szybką metodą, np. Quicksort, i zapisujemy je już w wersji posortowanej. W praktyce ten punkt polega na sekwencyjnym zapisie do pamięci porcji pliku wejściowego i gdy zostanie przekroczony pewien umowny rozmiar, sortujemy w pamięci operacyjnej odczytaną zawartość i zapisujemy wynik jako kolejny plik cząstkowy. Postępujemy tak aż do pełnego podzielenia pliku wejściowego na mniejsze, już posortowane.
- W kolejnych krokach (przebiegach) scalamy te mniejsze i posortowane pliki w większe, aż do momentu posortowania całego pliku wejściowego.

Jak może działać operacja scalania plików posortowanych? Nie jest to tak trudne, jak samo sortowanie, gdyż dane wejściowe są już uporządkowane. Prosty algorytm scalający dane posortowane rosnąco mógłby wyglądać tak:

- Otwórz *pliki wejściowe*.
- Odczytuj sekwencyjnie dane z obu plików i przenoś wartość mniejszą do *pliku wyjściowego*, aż do momentu gdy któryś z plików się zakończy.
- Dopisz resztę na koniec pliku wyjściowego.

Oczywiście umowna prostota wcale nie oznacza krótkiego kodu w C++, poprawna implementacja, badająca warunki końca plików i wykonująca szereg porównań, może zająć nawet kilkadziesiąt linii kodu. Pewną alternatywą, wcale nie taką niemądrą, jest sklejenie plików razem i posortowanie przy pomocy komend systemu operacyjnego, np. tak jak to robi listing poniżej.


Listing

systemsort.cpp

```
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
ifstream plik_W1 ("input1.txt"); //plik wejściowy 1
ifstream plik_W2 ("input2.txt"); //plik wejściowy 2
ofstream plik_WYJ ("output.txt");//plik wynikowy
string s; //złączamy pliki ze sobą, używając C++
while (getline(plik_W1,s))
   plik_WYJ << s << endl;
while (getline(plik_W2,s))
   plik_WYJ << s << endl;
plik_WYJ.close(); //zwalniamy plik wynikowy
// Oczywiście złączanie można ewentualnie wykonać przez polecenie
// systemowe: system("copy input1.txt+input2.txt output.txt");
system("sort output.txt /O output.txt"); //sortujemy plik wynikowy
// przy pomocy komendy systemu operacyjnego
}
```

Wady sortowania systemowego są oczywiste:

- Tracimy kontrolę nad efektywnością (proces sortowania może mieć tak niski priorytet, że wykona się bardzo wolno).
- Program w C++ przestaje być przenośny (np. aby powyższy kod zadziałał pod Linuxem, należy dostosować składnię komendy `sort`, zamienić `copy` na `cp`).
- Sortowanie systemowe domyślnie nie rozróżnia liczb i znaków, np. gdyby nasz plik zawierał liczby, to mogłoby się okazać, że „111" jest mniejsze od „22".

Jak zatem posortować pliki z ciągami liczb, ale zapisanych jako znaki (w edytorze tekstowym)? W systemie Unix (Linux) możemy zmusić komendę `sort` do interpretowania wartości numerycznych (przełącznik `-n`), co jednak zrobić w DOS-ie (Windows)?

Moja sugestia jest następująca: proszę skopiować zawartość plików dyskowych do np. jednej tablicy liczb całkowitych, posortować ją jakąś znaną nam metodą i posortowaną tablicę znowu zapisać do pliku.

Jedyny kłopot mogą nam sprawić konwersje z formatu tekstowego na format liczby całkowitej. W C++ można do tego wykorzystać prostą instrukcję:

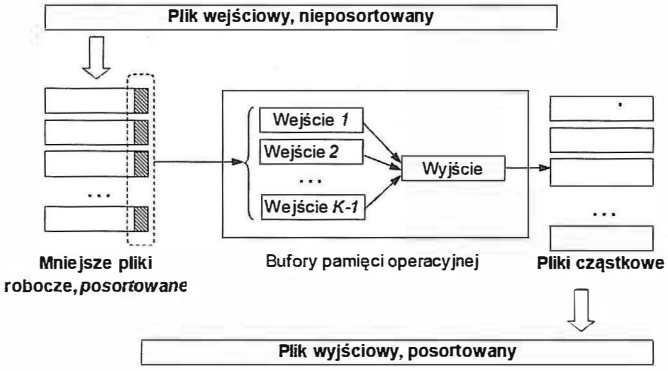
```
int jakaś_zmienna_int = atoi(jakaś_zmienna_string.c_str());
```

Zamiast w programie porównywać odczytywane znaki (typ `string`), konwertujemy je do zmiennej typu całkowitego i jej używamy w instrukcjach porównania.

Jak łatwo zauważyć, sortowanie zewnętrzne bardziej dotyka konkretnych problemów systemowych (pliki, konwersje danych) niż zagadnień czysto algorytmicznych. Wróćmy jednak na chwilę do modelu teoretycznego.

Pewien uogólniony schemat systemu sortowania zewnętrznego jest pokazany na rysunku 4.12. Zakładamy, że plik wejściowy jest N-stronicowy, a do jego sortowania możemy używać, oprócz mocy procesora, umownych obszarów pamięci:

- *K–1* buforów *wejściowych*, służących do przetwarzania danych z plików wejściowych.
- *1* bufor wyjściowy, służący do zapisu do plików zewnętrznych (roboczych lub wynikowych).



Rysunek 4.12. *Ogólny model sortowania zewnętrznego*

Suma tych K buforów nie może oczywiście przekraczać pojemności dostępnej pamięci operacyjnej.

Proces sortowania będzie oczywiście wieloprzebiegowy, bowiem w pojedynczym przebiegu możemy przetworzyć wyłącznie fragment dużego pliku wejściowego. W pierwszym, „zerowym" przebiegu trzeba podzielić plik wejściowy na mniejsze pliki robocze i posortować je. Rozmiar tego „mniejszego" pliku zależy od dostępnej pamięci operacyjnej — im większy, tym lepiej.

Następnie, przy użyciu naszych umownych buforów w pamięci operacyjnej, możemy odczytywać początkowe (a zatem posortowane po „zerowym" przebiegu) fragmenty plików posortowanych (patrz lewa część rysunku), sortować je przez złączanie w pamięci komputera (przy użyciu buforów pokazanych w środkowej części rysunku) i zapisywać do plików cząstkowych wynikowych pokazanych z prawej strony. (W jakimś prawdziwym systemie operacyjnym te umowne pliki z prawej strony mogą być dynamicznie tworzone lub też mogą być używane ponownie poprzednie pliki do zapisywania nowych rezultatów cząstkowych).

Proces sortowania przez scalanie z użyciem buforów i zapisu do pliku (plików) wynikowego (wynikowych) kontynuujemy aż do momentu, gdy pliki z lewej strony i same bufory staną się puste. Tak jak już wcześniej wspomniałem, koszt sortowania algorytmu jest związany z liczbą wykonywanych przebiegów, gdyż operacje plikowe stanowią największy narzut czasowy. Ponieważ wąskim gardłem jest sam podział pliku na N-bloków i liczba zastosowanych K-buforów w pamięci głównej, można obliczyć, że koszt sortowania zewnętrznego wyniesie:

$$2N \cdot \left\lceil 1 + \log_{K-1} \left\lceil \frac{N}{K} \right\rceil \right\rceil$$

Prawy człon tego iloczynu stanowi tak naprawdę liczbę przebiegów, wartości są zaokrąglane do najbliższej wartości całkowitej w górę (symbol $\lceil\ \rceil$).

Uwagi praktyczne

Kryteria wyboru algorytmu sortowania mogą być zebrane w kilka łatwych do zapamiętania zasad:

- Do sortowania małej liczby elementów nie używaj superszybkich algorytmów, takich jak np. *Quicksort*, gdyż zysk będzie znikomy.
- Część znanych z literatury i prasy fachowej algorytmów sortowania nie jest nigdy — lub jest bardzo rzadko — stosowana praktycznie. Powodem ich stworzenia były prace akademickie, które nigdy nie zostały wykorzystane w praktyce programistycznej. Zresztą, trzymając się dobrze znanych metod, mamy większą pewność, iż nie popełnimy jakiegoś nadprogramowego błędu!

Podczas programowania warto również uważnie czytać, czy w bibliotekach standardowych używanego kompilatora nie ma już zaimplementowanej funkcji sortującej. Przykładowo: w wielu kompilatorach[6] istnieje gotowa funkcja o nazwie... `qsort` o następującym nagłówku:

```
void qsort(void *base, size_t nmemb, size_t size,
      int (*compar)(const void *, const void *))
```

Tablica do posortowania może być dowolna (typ `void`), ale musimy dokładnie podać jej rozmiar: liczbę elementów `nmemb` o rozmiarze `size`. Funkcja `qsort` wymaga ponadto jako parametru *wskaźnika do funkcji porównawczej*. Przy omawianiu list jednokierunkowych dokładnie omówiono pojęcie wskaźników do funkcji; tutaj w celu zilustrowania podam tylko gotowy kod do sortowania tablicy liczb całkowitych (przeróbka na sortowanie tablic innego typu niż `int` wymaga jedynie modyfikacji funkcji porównawczej `comp` i sposobu wywołania funkcji `qsort`):


Listing

qsort2.cpp

```
int comp(const void *x, const void *y)
{
int xx=*(int*)x; //jawna konwersja z typu
int yy=*(int*)y; //void* do int*
if ( xx <  yy)
   return -1;
if ( xx == yy)
   return 0;
else
   return 1;
}
const int n=12;
int tab[n]={40,29,2,1,6,18,20,32,34,39,23,41};
int main()
{
for (int i=0; i<n; i++)
   cout << tab[i] <<"   ";
cout << endl;
qsort(tab, n, sizeof(int), comp);
for (int i=0; i<n; i++)
   cout << tab[i] <<"   ";;
cout << endl;
}
```

Funkcja porównawcza `comp` zmienia się w zależności od typu danych sortowanej tablicy. Przykładowo: dla tablicy wskaźników ciągów znaków użylibyśmy jej następująco:

```
int comp(const void* a, const void* b)
{
return(strcmp((char*)a,(char*)b));
}

int main()
{
char s[5][4]={"aaa", "ccc", "ddd", "zzz", "fff" };
qsort((void*)s, 5, sizeof(s[0]), comp);
for (int i=0; i<5; i++)
   cout << s[i] << endl;
}
```

Wadą stosowania gotowej funkcji bibliotecznej jest brak dostępu do kodu źródłowego: dostajemy kota w worku i musimy się do niego przyzwyczaić.

Pisanie własnej procedury sortującej ma tę zaletę, że możemy ją zoptymalizować pod kątem naszej własnej aplikacji. Już wbudowanie funkcji `comp` wprost do procedury sortującej powinno nieco poprawić jej parametry czasowe, nie zmieniając jednak klasy algorytmu!

[6] Przykładowo: Borland C++ Builder, GNU C++, Visual C++.