Wstawiam, więc sortuję

#sortowanie #in_situ
#short_evaluation #stabilność

W razie naglącej potrzeby posortowania danych (czyli ustawienia ich w określonej kolejności, na przykład rosnącej) zawsze możemy skorzystać z bibliotecznej funkcji sortującej `sort()`, o której piszemy w podrozdziale *Drużyna, w szeregu zbiórka!*. Nie musimy wtedy dociekać, jaki algorytm został przez tę funkcję wykorzystany – ważne, że działa i że działa szybko. Warto jednak poznać podstawowe algorytmy sortowania, nie tylko ze względów historycznych. Jeden z największych informatyków wszech czasów, Donald E. Knuth, napisał kiedyś, że poznając klasyczne metody sortowania można bardzo wiele nauczyć się o samej algorytmice – i Kocurro jak najbardziej podziela ten pogląd. Dlatego w tych *Miaukotach* poznamy rozmaite, powszechnie znane sposoby sortowania – dobre, lepsze lub jeszcze lepsze (niekoniecznie w tej kolejności).

Jednym z najbardziej znanych sposobów sortowania danych jest sortowanie przez wstawianie (ang. **insertion sort**, wymawiaj: *inserszyn sort*).* Generalnie chodzi o posortowanie tablicy *w miejscu* (łac. **in situ**), czyli bez wykorzystania dodatkowej pamięci do przechowywania danych. Dla ustalenia uwagi sortować będziemy rosnąco, dokładniej: niemalejąco (więc ewentualne jednakowe elementy znajdą się obok siebie).†

Sortowanie przez wstawianie przećwiczymy na poniższej ośmioelementowej tablicy przypadkowych liczb całkowitych (oznaczymy ją $A[\,]$):

6	5	8	2	3	2	4	7
0	1	2	3	4	5	6	7

W trakcie wykonywania algorytmu nasza tablica będzie podzielona na dwie części: po lewej znajdować się będzie część posortowana (i będzie ona stopniowo rosnąć, aż osiągnie rozmiar całej tablicy), natomiast po prawej będziemy mieć sektor jeszcze nieruszony, z oryginalną kolejnością elementów (ten sektor będzie stopniowo się kurczyć).

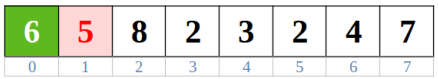
Początkowy element tablicy $A[0]$ o wartości 6 można uznać za zalążek części posortowanej (jej tło będziemy oznaczać na zielono, *pardon*, na oliwkowo):

6	5	8	2	3	2	4	7
0	1	2	3	4	5	6	7

*Po raz kolejny proszę moje koleżanki-anglistki i kolegów-anglistów o wybaczenie za taki uproszczony zapis wymowy, ale to dla dobra sprawy...

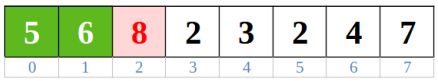
†Wzajemna kolejność jednakowych elementów bywa istotna, ale o tym za chwilę.

Na każdym kroku głównej pętli algorytmu będziemy zajmować się elementem tablicy, który znajduje się zaraz na prawo od części posortowanej. W tym przypadku jest to element $A[1]$ o wartości 5:

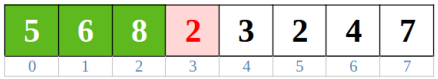


Szukamy dla niego odpowiedniego miejsca, idąc od prawej po części posortowanej, aż napotkamy element nie większy od $A[1]$, wtedy możemy wstawić $A[1]$ tuż za znalezionym elementem – w ten sposób nie zaburzymy porządku w posortowanej części tablicy.

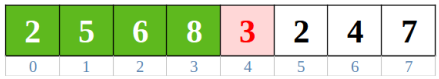
Może się jednak zdarzyć, że element, dla którego szukamy miejsca, okaże się mniejszy od wszystkich elementów w posortowanej części i trzeba będzie wstawić go na samym początku. Tak zresztą się dzieje w naszym przypadku, gdyż 5 jest mniejsze od 6. Nasz element (5) wędruje na miejsce o indeksie 0, natomiast element z tego miejsca (6) przesuwamy na miejsce o indeksie 1:



Teraz zajmujemy się elementem $A[2]$ o wartości 8. Spoglądamy w lewo i od razu napotykamy element 6, mniejszy od 8. Zatem $A[2]$ może spokojnie pozostać na swoim miejscu i posortowana część tablicy rośnie o jeden element, a my szukamy miejsca dla elementu $A[3]$ o wartości 2:

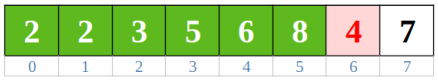


Patrzymy w lewo: jest 8 (za dużo), bardziej w lewo – jest 6 (za dużo!), jeszcze bardziej w lewo – jest 5 (nadal za dużo!!). Dalej w lewo jest już tylko ściana,[‡] więc element 2 wędruje na sam początek tablicy, elementy 5, 6 oraz 8 przesuwamy o jedno miejsce w prawo i ustawiamy celownik na element $A[4]$ o wartości 3:[§]



Wędrujemy w lewo i znajdujemy element o wartości 2 – tuż za nim możemy wstawić naszą 3, przesunąć ponownie elementy 5, 6 oraz 8 w prawo i zająć się elementem $A[5]$ o wartości 2:

Taka wartość już wystąpiła, więc tę dwójkę wstawimy na prawo od niej, przesuwając o jedną pozycję elementy 3, 5, 6 oraz 8. Obecnie na wokandzie mamy element $A[6]$ o wartości 4:

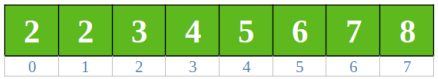


[‡] Jak w przypadku ekstremalnej lewicy w parlamencie.

[§] Proszę nie martwić się, jak przesuwa się taką całą grupę elementów – jest na to efektywny sposób, który pokażemy przy implementacji algorytmu.

Jego miejsce jest obok elementu o wartości 3, wstawiamy go, przesuwamy elementy 4, 5, 6 oraz 8 i zajmujemy się ostatnim elementem o wartości 7:

Patrzymy w lewo od niego: jest 8 (za dużo), potem jest 6 – OK, więc po tej 6 wstawiamy 7, a element 8 przesuwamy na koniec tablicy:



Mamy więc posortowaną całą tablicę, więc czas na zaimplementowanie algorytmu w formie procedury. Argumentami jej będą tablica $A[\,]$ oraz jej rozmiar, który oznaczymy przez N:[¶]

```
void insertion_sort(int A[], int N)
{
  . . .
}
```

Zauważmy, że rozmiar tablicy N występuje jako oddzielny argument i nie podaje się go bezpośrednio w nawiasach kwadratowych.

Przeglądając przedstawiony powyżej algorytm widzimy, że indeks elementu $A[i]$, dla którego poszukujemy miejsca w posortowanej części tablicy przebiega wartości od 1 do $N-1$:

```
for(int i{ 1 }; i < N; i++)
{
  int val = A[i];
  . . .
}
```

W celu zwiększenia przejrzystości oznaczyliśmy ten element przez *val*.[‖]

Dla każdego takiego elementu przeglądamy elementy tablicy $A[j]$ począwszy od $j = i - 1$. Przeglądanie kończy się, gdy znajdziemy element $A[j] \leqslant val$ i wtedy wstawiamy *val* w miejsce $A[j]$ – lub jeśli przekroczymy lewy kraniec tablicy (j osiągnie wartość ujemną), wtedy *val* wstawiamy na początkowym miejscu w tablicy.

Ważne jest, aby sprawdzać warunek nieujemności j jeszcze *przed* porównaniem $A[j]$ oraz $A[i]$, aby – Boże broń! – nie zapytać się o element spoza tablicy.[**] W języku C++ dla wyrażeń logicznych obowiązuje reguła **short evaluation**, czyli jeśli podczas obliczania wyrażenia logicznego stanie się jasne, jaką będzie miało ostateczną wartość, wówczas nie oblicza się reszty wyrażenia. Na przykład jeśli pierwszy czynnik w koniunkcji jest fałszywy, wtedy nie sprawdza się drugiego (i ewentualnie dalszych). Ta niepozorna reguła może mieć doniosłe skutki,

[¶] Jeśli tablica (*array*) przekazywana jest jako argument funkcji, wtedy mamy dostęp do oryginalnych elementów, ale nie mamy informacji o rozmiarze tablicy. Inaczej jest w przypadku choćby wektora-argumentu, gdzie możemy odczytać jego długość, ale z kolei operujemy na kopii oryginału (chyba że użyjemy symbolu referencji).

[‖] Od angielskiego **value** (wartość), wymawiaj: *walju*.

[**] W piekle jest podobno specjalne miejsce dla programistów niepilnujących tej zasady!

jeśli obliczanie kolejnych czynników ma jakieś działania uboczne, na przykład oznacza zmianę wartości jakiejś zmiennej.

No to może tak by to było:

```
for(int i{ 1 }; i < N; i++)
{
  int val = A[i];
  int j = i - 1;
  while(j >= 0 && A[j] > val)
    j--;
  A[j + 1] = val;
}
```

Pętla `while` kończy się o krok za daleko, stąd w końcowym przypisaniu (wstawieniu) występuje indeks $j + 1$.

O czymś jednak zapomnieliśmy, o czymś kluczowym. Brakuje przesuwania w prawo elementów tablicy większych od *val*! Obiecaliśmy, że zrobimy to prosto i tak właśnie będzie: wstawimy przenoszenie wartości do pętli `while`: przy każdym jej obiegu przesuwamy kolejny element $A[j]$ o oczko w prawo (przecież wiemy, że jest większy od *val*):

```
for(int i{ 1 }; i < N; i++)
{
  int val = A[i];
  int j = i - 1;
  while(j >= 0 && A[j] > val)
  {
    A[j + 1] = A[j];
    j--;
  }
  A[j + 1] = val;
}
```

Teraz możemy już zestawić cały program prezentujący działanie sortowania przez wstawianie:

```
#include <bits/stdc++.h>
using namespace std;

void insertion_sort(int A[], int N)
{
  for(int i{ 1 }; i < N; i++)
  {
    int val = A[i];
    int j = i - 1;
```

```
    while(j >= 0 && A[j] > val)
    {
      A[j + 1] = A[j];
      j--;
    }
    A[j + 1] = val;
  }
}

int main()
{
  int A[] = {6, 5, 8, 2, 3, 2, 4, 7};
  insertion_sort(A, 8);
  for(int x : A)
    cout << x << ' ';
  cout << '\n';
  return 0;
}
```

Na ekranie – zgodnie z oczekiwaniem – pojawi się lista liczb:

```
2 2 3 4 5 6 7 8
```

Jak już wspominaliśmy, jednakowe elementy (tutaj dwie dwójki) lądują obok siebie. Jednak przyjrzyjmy się uważniej: lewa dwójka pochodzi z miejsca o indeksie 3, zaś prawa uprzednio stała na pozycji 5. Zachowana została oryginalna kolejność wystąpień tych dwójek.[††] W powyższym przykładzie jest to bez znaczenia, ale proszę sobie wyobrazić, że liczba (ogólniej: klucz), według której sortujemy, jest tylko jednym z atrybutów złożonych obiektów przechowywanych w sortowanej tablicy. Wtedy może się okazać, że zachowanie oryginalnej kolejności (w ramach tej samej wartości klucza) jest przydatną cechą algorytmu. Taką własność nazywamy *stabilnością* metody sortowania.

Na koniec dodajmy, że – formalnie rzecz biorąc – złożoność obliczeniowa tego algorytmu nie jest porywająca: pesymistycznie możemy spodziewać się złożoności kwadratowej $O(N^2)$. Mamy bowiem do ustawienia $N-1$ elementów, a dla każdego z nich wychodzi w najgorszym przypadku od 0 do $N-2$ porównań/przestawień. Można ten algorytm nieco podrasować, choćby dodając wyszukiwanie binarne dla znajdowania pozycji kolejnych elementów w posortowanej części tablicy. Jeśli ten algorytm zastosujemy do sortowania struktury typu lista, wtedy odpadnie nam dość czasochłonne przesuwanie kolejnych elementów (dlaczego?).

Nawet w najprostszej wersji sortowanie przez wstawianie bywa użyteczne, gdyż jego złożoność zależy od odległości elementu od docelowej pozycji w posortowanej tablicy. Jeśli te odległości nie są duże, wówczas sortowanie przebiega w całkiem zadowalającym tempie. Wrócimy do tego tematu przy omawianiu sortowania kubełkowego.

[††] Dzieje się tak dzięki temu, że w warunku pętli `while` mamy nieostrą nierówność (dlaczego?).