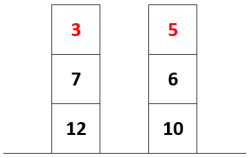
Scalam, więc sortuję

#sortowanie_przez_scalanie
#merge_sort
#dziel_i_zwyciężaj

Interesującym sposobem sortowania jest *sortowanie przez scalanie* (**merge sort**, wymawiaj: *merdż sort*) – rekurencyjny algorytm oparty na zasadzie **divide and conquer** (*dziel i zwyciężaj*), którą poznaliśmy przy okazji algorytmu Karacuby.

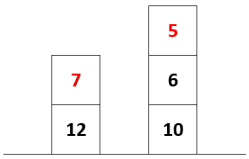
Wyobraźmy sobie, że mamy na stole dwa stosiki kart z liczbami (leżące liczbami do góry), każdy z nich oddzielnie posortowany, tak że na wierzchu widzimy kartę z najmniejszą wartością, na przykład:



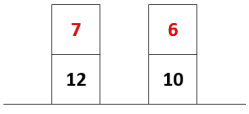
Nasze przykładowe stosiki mają tę samą wysokość, ale nie ma to większego znaczenia. Zadanie polega na scaleniu obydwu stosików w jeden posortowany ciąg liczb/kart.

W każdym kroku algorytmu na wierzchu widzimy dwie karty (liczby oznaczone na czerwono), z których wybieramy tę z mniejszą liczbą. No, chyba że stosik się skończy, a na drugim są jeszcze karty, więc z tego drugiego bierzemy karty „jak leci”.

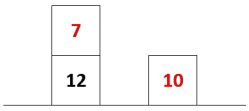
U nas pójdzie tak – bierzemy 3, odsłania się liczba 7:



Bierzemy 5, odsłania się 6:



Bierzemy 6, odsłania się 10:



Bierzemy 7, odsłania się 12:

Bierzemy 10, prawy stosik się skończył:

Bierzemy 12 jako ostatnią kartę – stół jest pusty.

Uzbieraliśmy 3, 5, 6, 7, 10 oraz 12 – cały zbiór został posortowany. Poszło łatwo, ale mieliśmy już sporo przygotowane: oddzielnie posortowane stosiki. Lewy czy prawy podzbiór mogły być posortowane rekurencyjnie, przez kolejne instancje funkcji sortującej. Czyli że algorytm powinniśmy zacząć od podziału zbioru danych na dwie części, każdą z nich podzielić jeszcze na dwie – i tak dalej, aż któraś instancja dostanie do posortowania jednoelementowy podzbiór, którego już sortować nie trzeba. Potem powinno nastąpić scalanie posortowanych podzbiorów/stosików w coraz większe stosy, aż w końcu połączymy dwa ostatnie stosy w cały zbiór.

Metoda zstępująca

Mówiliśmy o tym już w przypadku programowania dynamicznego: mamy do rozwiązania „duży" problem: posortowanie całej tablicy (wektora) liczb całkowitych (oznaczymy ją przez X). Podzielimy ten problem z grubsza na dwie połowy (ang. **split**) i do każdej z nich zastosujemy ten sam algorytm – rekurencyjnie, do spodu. Czyli czekają nas kolejne podziały, aż któreś tam instancje funkcji sortującej dostaną pojedyncze elementy wyjściowej tablicy. Tych oczywiście sortować nie trzeba, natomiast trzeba zająć się łączeniem otrzymanych posortowanych odcinków tablicy.

Jeśli przyjrzymy się przykładowi z dwoma stosikami kart, widać, że wygodnie nam będzie posłużyć się pomocniczą tablicą (oznaczymy ją przez Y), gdzie przechowamy tymczasowe wyniki obliczeń, po czym skopiujemy już uporządkowany odcinek do X. Nagłówek takiej funkcji może wyglądać tak:

```
void merge_split(vector<int> &X, int start, int finish, vector<int> &Y)
{
  . . .
}
```

Jak zwykle, przed argumentami funkcji, które są większymi strukturami, dajemy symbol referencji & (inaczej na potrzeby funkcji tworzona byłaby kopia całego wektora, co jest czasochłonne, a poza tym istnieje konieczność zmiany oryginału wektora X z wnętrza funkcji), a zmienne *start* i $finish$ wyznaczają zakres indeksów i dla danego odcinka danych: $start \leqslant i < finish$.

Funkcja `merge_split()` powinna:

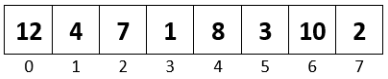
- Sprawdzić różnicę pomiędzy $finish$ i *start*: jeśli odcinek danych składa się z mniej niż dwóch liczb, wtedy funkcja powinna zakończyć działanie, bo nie ma co sortować.
- W przeciwnym przypadku należy wyznaczyć środkowy indeks (oznaczmy go *middle*).
- Wywołać swego potomka dla zakresu indeksów od *start* do *middle*.
- Wywołać swego potomka dla zakresu indeksów od *middle* do $finish$.
- Połączyć (scalić) rezultaty otrzymane przez potomków.

No to do dzieła:

```
void merge_split(vector<int> &X, int start, int finish, vector<int> &Y)
{
  if(finish - start < 2) return;
  int middle = (start + finish) / 2;
  merge_split(X, start, middle, Y);
  merge_split(X, middle, finish, Y);
  merge_parts(X, start, middle, finish, Y);
}
```

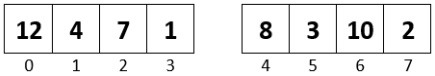
Na razie nie wygląda to zbyt skomplikowanie. Komentarza może wymagać sposób obliczania indeksu *middle*: a co jeśli suma $start + finish$ jest liczbą nieparzystą? No nic, po prostu podział odcinka danych na połowy nie będzie równy: jedna z „połówek" będzie o 1 dłuższa od drugiej.

Zobaczmy, jak będzie wyglądało to dzielenie dla przykładowego 8-elementowego zestawu danych (12, 4, 7, 1, 8, 3, 10, 2):*



Liczby od 0 do 7 oznaczają numery komórek wektora X.

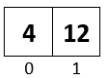
Po pierwszym podziale dostajemy dwa odcinki – każdy po 4 elementy ($0 \div 3$, $4 \div 7$):



Kolejne podziały dzielą wszystko na odcinki po 2 elementy:

Ostatnia seria podziałów daje w wyniku pojedyncze elementy wektora:

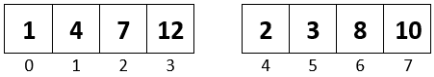
Jako żywo, na razie nic nie jest posortowane, ale teraz musimy zacząć scalać poszczególne odcinki wektora – tak samo, jak robiliśmy to ze stosikami kart na początku tego podrozdziału. Nasze stosiki nie są zbyt okazałe (każdy ma tylko po jednej karcie), ale to nic, zaraz urosną. Na przykład dwa pierwsze z lewej (o indeksach 0 oraz 1) możemy połączyć w uporządkowaną parę:



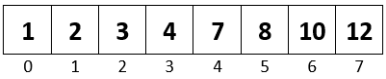
Podobnie uzyskujemy pozostałe pary:

*Użycie potęgi liczby 2 nie jest przypadkowe, bo wtedy dostajemy ładny symetryczny obrazek, ale algorytm działa oczywiście dla dowolnej ilości danych.

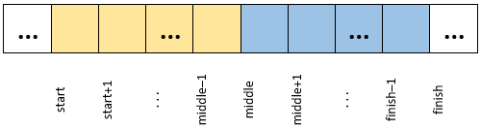
. . . które możemy połączyć w czwórki:



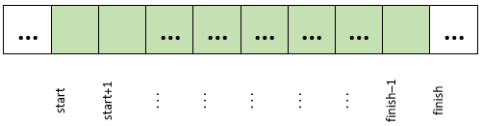
. . . a te następnie łączymy w całość:



No i udało się! Teraz trzeba to tylko „zaklepać” w C++! Kluczem do sukcesu jest funkcja `merge()`, która cichcem chyłkiem pojawiła się w naszym kodzie. Jej zadaniem będzie scalenie dwóch sąsiadujących odcinków wektora X o indeksach $start \div (middle - 1)$ (odcinek lewy, beżowy) oraz $middle \div (finish - 1)$ (odcinek prawy, niebieski):



Scalimy je zapisując dane w pomocniczym wektorze Y na odcinku o zakresie indeksów $start \div (finish - 1)$ (odcinek zielony):



Następnie przekopiujemy posortowany odcinek z powrotem do wektora X.

Najprościej będzie zrobić to przy pomocy trzech zmiennych indeksowych, które będą numerować odpowiednio:

- `i_left` – elementy w lewym odcinku wektora X,
- `i_right` – elementy w prawym odcinku wektora X,
- `i_target` – elementy w docelowym odcinku wektora Y.

Początkowe wartości tych zmiennych powinny być następujące:

```
int i_left = start, i_right = middle, i_target = start;
```

Scalanie będzie przebiegać w pętli, która powinna się kręcić, póki `i_target` jest nie większe od ilości danych, czyli $finish - 1$:

```
while(i_target < finish)
  . . .
```

Głównym elementem treści pętli musi być instrukcja warunkowa, która sprawdza, który element należy przepisać do wektora Y. Jeśli skończył się lewy odcinek, wtedy bierzemy element z prawego, jeśli skończył się prawy odcinek, wtedy bierzemy element z lewego odcinka. Jeśli w obydwu odcinkach są jeszcze elementy, wtedy bierzemy mniejszy z nich.

Wyczerpanie elementów z lewego odcinka rozpoznamy po zachodzeniu warunku:

```
i_left == middle
```

Analogicznie zakończenie prawego odcinka sygnalizuje prawdziwość warunku:

```
i_right == finish
```

Wybrany element z wektora X jest przepisywany w odpowiednie miejsce wektora Y, po czym należy powiększyć odpowiednie indeksy – w szczególności zmienną `i_target`.

Wspomniana instrukcja warunkowa będzie dość rozbudowana:

```
if(i_left == middle)
{
  Y[i_target] = X[i_right];
  i_target++;
  i_right++;
}
else
  . . .
```

Widać, że opłaca się tutaj skorzystać ze skrótowego zapisu i wstawić operatory powiększania `++` do pierwszej instrukcji:

```
if(i_left == middle)
  Y[ i_target++ ] = X[ i_right++ ];
else
  . . .
```

A oto i cała instrukcja warunkowa:

```
if(i_left == middle)
  Y[ i_target++ ] = X[ i_right++ ];
else if(i_right == finish)
  Y[ i_target++ ] = X[ i_left++ ];
else if(X[i_left] <= X[i_right])
  Y[ i_target++ ] = X[ i_left++ ];
else
  Y[ i_target++ ] = X[ i_right++ ];
```

Po zakończeniu pętli `while` kopiujemy dane z powrotem do wektora X:

```
for(int i = start; i < finish; i++)
  X[i] = Y[i];
```

Czas na podsumowanie i konsolidację funkcji `merge()`:

```
void merge_parts(vector<int> &X, int start, int middle, int finish,
           vector<int> &Y)
{
  int i_left = start, i_right = middle, i_target = start;
  while(i_target < finish)
    if(i_left == middle)
      Y[ i_target++ ] = X[ i_right++ ];
    else if(i_right == finish)
      Y[ i_target++ ] = X[ i_left++ ];
    else if(X[i_left] < X[i_right])
      Y[ i_target++ ] = X[ i_left++ ];
    else
      Y[ i_target++ ] = X[ i_right++ ];
    for(int i = start; i < finish; i++)
      X[i] = Y[i];
}
```

Przed uruchomieniem całej procedury sortowania należy jeszcze zadeklarować pomocniczy wektor Y o odpowiednim rozmiarze (takim samym jak X), na przykład:

```
int n = X.size();
vector<int> Y(n);
merge_split(X, 0, n, Y);
```

Pominęliśmy tutaj wczytywanie danych i wypisywanie wyniku – to chyba nie problem dla Czytelnika?

Jaka jest złożoność takiego algorytmu sortowania? Całkiem niezła: ilość poziomów podziałów to mniej więcej $\lg_2 n$, a ilość odcinków wektora na każdym poziomie nie przekracza n, co dawałoby oszacowanie złożoności na $n \lg_2 n$. Zupełnie przyzwoity wynik!

Jednak ta metoda sortowania wymaga dodatkowej tablicy Y, czyli jej złożoność pamięciowa jest większa niż na przykład sortowania szybkiego (ang. **quick sort**). Jeśli nam to nie przeszkadza, to wtedy OK, możemy ją stosować.[†]

Metoda wstępująca

Zajmiemy się obecnie wersją wstępującą tego algorytmu, czyli zaczniemy porządkowanie wektora X od krótkich odcinków, które będziemy stopniowo wydłużać (za każdym razem o czynnik 2), aż dojdziemy do całej długości wektora. (O takim podejściu jest mowa w podrozdziale o programowaniu dynamicznym.) Napiszemy nową funkcję `merge_up()`, która upora się z tym zadaniem – działanie tej funkcji będzie się opierać na iteracji, a nie na rekurencji:

[†]Istnieje wersja tego algorytmu nie wymagająca pomocniczej tablicy, ale o tym może kiedy indziej.

```
void merge_up(vector<int> &X, vector<int> &Y)
{
  int n = X.size();
  for(int part = 1; part < n; part *= 2)
    for(int i = 0; i < n; i += 2 * part)
      merge_parts(X, i, min(i + part, n), min(i + 2 * part, n), Y);
}
```

Zmienna *part* zawiera długość rozważanych w danym momencie odcinków – pojedynczy fragment do scalenia ma długość $2 \cdot part$. W roli zmiennej *start* występuje liczba i, w roli zmiennej $middle$ – mniejsza z liczb $i+part$ oraz n, zaś w roli zmiennej $finish$ – mniejsza z liczb $i+2 \cdot part$ oraz n. Takie zabezpieczenie (funkcja `min()`) jest potrzebne, aby nasza funkcja nie sięgała poza koniec wektora X oraz Y.

I to w zasadzie wszystko, można jeszcze wspomnieć, że złożoność tej metody jest taka sama, jak metody zstępującej, opisanej w poprzedniej sekcji.