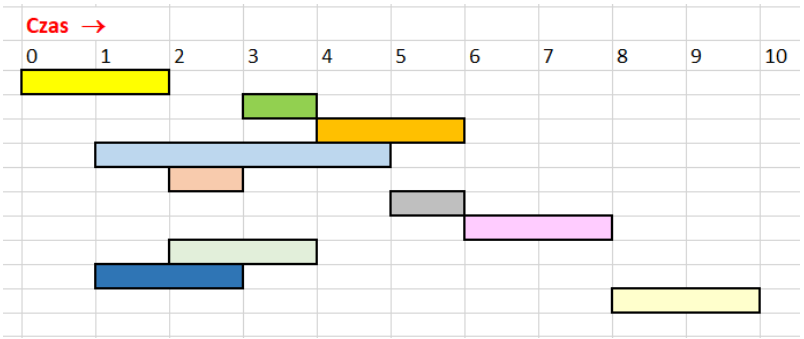
Układanie rozkładu zajęć

#algorytm_zachłanny #wydawanie_reszty
#C++11 #pętla_foreach
#auto

Wyobraźmy sobie następujący problem: w pewnej sali wykładowej planowane są zajęcia dla studentów, jednak wstępny rozkład przewiduje zbyt wiele wykładów, tak że niektóre z nich kolidują ze sobą. Godziny rozpoczęcia i zakończenia zajęć są ustalone i dla uproszczenia przyjmiemy, że są to zawsze pełne godziny.

Naszym zadaniem jest wybranie jak największej ilości zajęć, przy czym nie jest ważne, czy są to wykłady jedno czy dwugodzinne (lub dłuższe) – liczy się tylko ilość zajęć, które można wcisnąć bezkolizyjnie w grafik na ten dzień. Możemy założyć, że nie będzie przerw pomiędzy zajęciami i najwcześniejsze zajęcia zaczynają się o godzinie 0. Dla ustalenia uwagi skupmy się na następującej liście wykładów:



A oto tabelka z godzinami rozpoczęcia i zakończenia poszczególnych zajęć:

Lp.	Początek	Koniec
1	0	2
2	3	4
3	4	6
4	1	5
5	2	3
6	5	6
7	6	8
8	2	4
9	1	3
10	8	10

Jest to ewidentnie problem typowo optymalizacyjny – trzeba wszakże znaleźć najlepszy (najliczniejszy) podzbiór zajęć. Należy się spodziewać, że może istnieć więcej niż jedno optymalne rozwiązanie, ale na pewno jedna rzecz jest jednoznacznie określona: maksymalna ilość niekolidujących zajęć.

Nasuwającym się sposobem rozwiązania jest metoda programowania dynamicznego, która na bank da poprawny rezultat.* Jednak można postawić sobie pytanie, czy musimy wytaczać aż taką armatę na rozwiązanie tego problemu? Nie można zastosować czegoś prostszego (i potencjalnie szybszego)?

W tym przypadku odpowiedź jest twierdząca – możemy zastosować *algorytm zachłanny* (ang. **greedy**, wymawiaj: *gridi*, z długim pierwszym *i*). Metoda ta polega na dokonaniu wyboru (cząstkowego), który na danym etapie rozwiązania wydaje się być najlepszy. Widać tu różnicę w stosunku do programowania dynamicznego, gdzie sprawdzaliśmy wszystkie cząstkowe rozwiązania.

W przypadku układania grafiku metoda zachłanna polega na wybraniu zajęć, które kończą się jak najwcześniej. Następnie szukamy kolejnych najwcześniejszych zajęć, które mogą się odbyć (czyli nie kolidują ze wcześniejszymi) i tak dalej, i tak dalej. W ten sposób dostajemy optymalne rozwiązanie, ponieważ im zajęcia kończą się wcześniej, tym szybciej zwalniana jest sala na kolejny wykład.

Widoczne jest, że przed przystąpieniem do dopasowywania zajęć, należy ich listę posortować niemalejąco względem czasu zakończenia. Ponieważ każda pozycja na liście to para liczb (czas rozpoczęcia, czas zakończenia), więc będziemy mieć do czynienia z sortowaniem wektora par, o czym była mowa w podrozdziale *Kto lepszy?*. Zaczniemy może od napisania funkcji `main()`, która powinna wczytać dane oraz wywołać procedurę `schedule()`, która zajmie się rozwiązaniem problemu. Na koniec, abyśmy mogli nacieszyć się wynikiem, wybrane zajęcia zostaną wypisane na ekranie.

Zakładamy następujący format danych wejściowych: w pierwszym wierszu będzie znajdować się liczba naturalna N – ilość zajęć na liście, zaś w kolejnych N wierszach podana będzie godzina rozpoczęcia i godzina zakończenia kolejnych zajęć (oddzielone pojedynczym odstępem). Jedna uwaga techniczna: ponieważ wektor par będzie sortowany, zatem lepiej wpisać czas zakończenia jako składową `first` pary, gdyż sortowanie par standardowo odbywa się względem właśnie tej składowej.

Oto przykładowa funkcja `main()`:

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

void schedule(vector<pair<int, int>> V, vector<pair<int, int>> A)
{ . . . }

int main()
{
  int N; cin >> N;
  vector<pair<int, int>> V(N), A;
  for(int i = 0; i < N; i++)
    cin >> V[i].second >> V[i].first;
  schedule(V, A);
  for(auto x : A)
    cout << x.second << " " << x.first << endl;
}
```

*Oczywiście oprócz metody *brute force*, czyli brutalnej siły.

Pojawiło się tu kilka nowych rzeczy, charakterystycznych dla standardu C++11 (i nowszych). Po pierwsze, mamy nową postać dyrektywy `#include <>`. Taki zapis powoduje, że kompilator uwzględnia *wszystkie* możliwe biblioteki i po prostu pobiera z nich te zasoby, które są mu potrzebne. Nie trzeba zatem pamiętać, co należy *inkludować*.

Po drugie, deklaracja wektora par zawiera zagnieżdżony typ danych:

```
vector<pair<int, int>>
```

W starszej wersji języka (a dokładniej biblioteki STL) dwa zamykające nawiasy kątowe `>>` należałoby napisać z odstępem:

```
vector<pair<int, int> >
```

Obecnie nie jest to już konieczne, można je pisać łącznie.

Po trzecie mamy tu nowy rodzaj pętli `for`, zwyczajowo nazywany **foreach** (wymawiaj: *foricz*, z akcentem na *i*), czyli „dla każdego". Chodzi o to, że zmienna x przebiega po wszystkich elementach wektora A – w tym wektorze procedura `schedule()` umieściła swój wynik działania. Ten typ pętli stosuje się, gdy nie są istotne numery elementów kontenera, po którym przebiega iteracja.

Po czwarte, słowo kluczowe `auto` oznacza, że typ danych zmiennej x zostanie wybrany automatycznie – w tym przypadku będzie to `pair<int, int>`.

Musimy wszakże pamiętać, by ustawić opcje kompilacji tak, by system korzystał z dobrodziejstw standardu C++11.

Pozostaje już tylko dopisać funkcję `schedule()`:

```
void schedule(vector<pair<int, int>> &V, vector<pair<int, int>> &A)
{
  sort(V.begin(), V.end());
  int n = V.size();
  A.push_back(V[0]);
  int k = 0;
  for(int m = 1; m < n; m++)
    if(V[m].second >= V[k].first)
    {
      A.push_back(V[m]);
      k = m;
    }
}
```

Kod funkcji rozpoczyna się od sortowania wektora V z propozycjami zajęć. Następnie do wektora wynikowego A dopisujemy w ciemno zajęcia kończące się najwcześniej, reprezentowane przez $V[0]$. Zmienna k oznacza numer ostatnich zaakceptowanych zajęć (na początku 0). Zmienna m spaceruje po kolejnych zajęciach, aż do końca wektora V. Jeśli m-te zajęcia nie kolidują z k-tymi zajęciami, wtedy dołączamy je do wektora A i uaktualniamy wartość k.

Powyższy kod można by zapisać także przy pomocy nowo poznanej pętli `foreach`, na przykład:

```
void schedule(vector<pair<int, int>> &V, vector<pair<int, int>> &A)
{
  sort(V.begin(), V.end());
  pair<int, int> last = V[0];
  A.push_back(last);
  for(auto x : V)
    if(x.second >= last.first)
    {
      A.push_back(x);
      last = x;
    }
}
```

Zmienna *last* przechowuje ostatnie zaakceptowane zajęcia.

Tym razem nie widać jakiegoś znaczącego uproszczenia kodu, a poza tym jego działanie różni się pewnym drobiazgiem od poprzedniej wersji (jakim?).

Nawiasem mówiąc, efektem ubocznym działania procedury `schedule()` jest posortowanie oryginału wektora V w funkcji `main()` – czasem może to być objaw niepożądany (może nam zależeć na zachowaniu oryginalnej kolejności jego elementów). Niech Czytelnik zastanowi się, co należy zmienić w definicji funkcji `schedule()`, aby tego uniknąć.

To nie zawsze działa

Algorytm zachłanny w opisanym tutaj przykładzie spisuje się bardzo dobrze: daje poprawny wynik i ma optymalną złożoność (liniową), czyli lepszą niż ewentualny *dynamik*. Jednak w wielu sytuacjach podejście zachłanne nie daje poprawnego rezultatu – należy zatem dokładnie przeanalizować postawiony problem i wybrać pomiędzy podejściem dynamicznym (wolniejszym, ale pewnym) oraz zachłannym. Do takich klasycznych zadań należy problem wydawania reszty.[†]

Wyobraźmy sobie, że chcemy wydać resztę o wartości powiedzmy 22 bitalarów, mając do dyspozycji po 10 monet o nominałach 2, 5 oraz 10. Interesuje nas taki sposób wydania, aby użyta była minimalna ilość monet. Możemy to zrealizować zachłannie: idziemy od wyższych nominałów do niższych. Najpierw bierzemy monetę 10 – do wydania zostaje 12, bierzemy drugą monetę 10 – do wydania zostaje 2. Nominał 10 jest już za duży, podobnie nominał 5. Bierzemy zatem monetę o nominale 2 i reszta zostaje wydana (optymalnie).

A jak wygląda sprawa przy tej samej kwocie do wydania (22), ale przy innym zestawie monet? Załóżmy, że mamy jedną monetę 10, jedną 9 i 15 monet o nominale 2. Jeśli znów będziemy iść od góry, weźmiemy monetę 10, potem monetę 9 (bo się mieści) i jesteśmy zablokowani, bo pozostałymi monetami nie dopełnimy pożądanej kwoty. Rozwiązaniem jest ominięcie monety 9 i wzięcie 6 monet o nominale 2, czyli da się (*dynamik* da dobry wynik).

[†]Podkreślamy tutaj odmienność tego problemu od zadania o tym samym tytule, które pojawiło się w *Drugich miaukotach* – tam ilość monet była dowolna, a nominały wszystkie możliwe.

Ten przykład pokazuje, że podejście zachłanne może kryć w sobie pułapkę, więc wybierając sposób rozwiązania problemu musimy go starannie sprawdzić.

Do znanych i ważnych algorytmów, gdzie podejście zachłanne działa i jest najlepszym sposobem rozwiązania, należy algorytm Dijkstry wyszukiwania ścieżek o najmniejszej wadze w grafie o krawędziach z nieujemnymi wagami oraz algorytm Kruskala znajdowania minimalnego drzewa rozpinającego. Przedstawimy te algorytmy w miaukotach poświęconym grafom.