** **5** Dany jest ciąg f_n, określony wzorem rekurencyjnym:

$$f_n = \begin{cases} -1 & \text{dla } n = 1 \text{ lub } n = 2 \\ f_{n-1} - f_{n-2} & \text{dla } n > 2 \end{cases}$$

a. Jaki ciąg opisuje powyższy wzór?

b. Podaj wzór ogólny powyższego ciągu oraz napisz funkcję obliczającą n-ty wyraz ciągu.

** **6** Dane są: trzy liczby całkowite p, k, n spełniające warunek $p \leqslant k < n$, tablica uporządkowanych liczb całkowitych `A[0..n-1]`, liczba całkowita x oraz funkcja rekurencyjna F:

```
funkcja F(A[],p,k,x)
    jeśli p = k to zwróć A[p] = x i zakończ
    s ← (p + k) div 2
    jeśli A[s] < x to zwróć F(A[],s+1,k,x) i zakończ
    zwróć F(A[],p,s,x) i zakończ
```

a. Jaki jest wynik działania powyższej funkcji?

b. Oszacuj liczbę porównań wykonywanych przez funkcję.

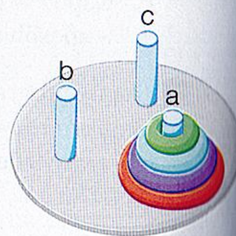
c. Napisz program, który wylosuje, posortuje i wypisze tablicę `A[0..n-1]` liczb całkowitych, następnie wczyta do zmiennej `x` dowolną liczbę całkowitą i wypisze wynik powyższej funkcji wywołanej z parametrami `F(A,0,n-1,x)`.

** **7** W pliku, który otrzymasz od nauczyciela (np. *Hanoi.pdf*), znajduje się opis rekurencyjnego rozwiązania problemu wież Hanoi. Napisz w pseudokodzie funkcję rekurencyjną rozwiązującą ten problem. Polecenie `Przenies(i,j)`, gdzie `i` oraz `j` oznaczają wieże, potraktuj jako elementarne.

** **8** W pliku, który otrzymasz od nauczyciela (np. *Hanoi.pdf*), znajduje się opis rekurencyjnego rozwiązania problemu wież Hanoi. Napisz funkcję zliczającą operacje przenoszenia pojedynczego krążka w problemie wież Hanoi dla n krążków. Zapisz rozwiązanie rekurencyjnie, a następnie iteracyjnie.

*** **9** Zapisz iteracyjnie rozszerzony algorytm Euklidesa.

*** **10** Korzystając z pliku, który udostępni ci nauczyciel (np. *Hanoi.pdf*), rozwiąż problem wież Hanoi w sposób iteracyjny.
Uwaga: Wieże ustaw na okręgu i przekładaj krążek o najmniejszej średnicy zgodnie z ruchem wskazówek zegara lub przeciwnie do tego ruchu w zależności od tego, czy liczba krążków jest parzysta czy nieparzysta.



18. Metoda zachłanna

Prawdopodobnie nie raz zdarzyło ci się korzystać z różnych automatów z napojami i słodyczami lub biletomatów. Zazwyczaj można w nich płacić kartą albo gotówką. Przy płatności gotówką wiele automatów ma możliwość wydania reszty. Czy wiesz, według jakiego algorytmu działają? Czy w ten sam sposób postępuje sprzedawca, gdy wydaje resztę w sklepie? Odpowiedzi na te pytania znajdziesz w tym temacie. Poznasz przy tym nową metodę algorytmiczną nazywaną metodą zachłanną.

Cele lekcji

- Poznasz metodę zachłanną, stosowaną do rozwiązywania problemów algorytmicznych.
- Opracujesz algorytm wydawania reszty najmniejszą liczbą monet i banknotów.
- Przygotujesz optymalny harmonogram wykorzystania sali.
- Poszukasz drogi o największej sumie pól na kwadratowej planszy wypełnionej liczbami.
- Przeanalizujesz problem pakowania plecaka.
- Wykorzystasz funkcję **sort** z biblioteki STL i napiszesz własną funkcję porównującą elementy.

18.1. Wydawanie reszty metodą zachłanną

Wyobraź sobie, że podczas zakupów kasjer wydaje ci kwotę 7,25 zł wyłącznie monetami jednogroszowymi. Z perspektywy klienta byłoby to bardzo uciążliwe, a z perspektywy sklepu nieopłacalne – wkrótce kasjerowi zabrakłoby monet jednogroszowych do wydawania kwot wymagających zwrócenia jednego grosza.

Warto więc zastanowić się nad algorytmem minimalizującym liczbę wydawanych banknotów i monet. Zastosujemy w nim **metodę zachłanną**, która polega na realizowaniu przez algorytm najlepszego wyboru w danym kroku rozwiązywania problemu. W algorytmie zachłannym nie ocenia się, jak decyzja podjęta w danej chwili wpłynie na kolejne kroki i całe rozwiązanie. Kontynuuje się rozwiązywanie problemu zgodnie z wyborem dokonanym w danym kroku – oczekuje się, że rozwiązanie globalne też będzie najlepsze.

Metoda zachłanna

Zauważ, że na początku wydawania reszty można wybrać największy nominał, który jest mniejszy od kwoty reszty lub równy tej kwocie, i wydać go tyle razy, ile to możliwe, aby nie przekroczyć reszty. Następnie można powtarzać ten krok, pamiętając, aby zmniejszać za każdym razem kwotę reszty do wydania o łączną wartość już wybranych nominałów.

Algorytm wydawania reszty

Kasjerzy i automaty przyjmujące gotówkę korzystają z prostego sposobu wydawania reszty najmniejszą liczbą banknotów lub monet. Jest to przykład podejścia zachłannego w algorytmice.

Jak wydać resztę najmniejszą liczbą banknotów lub monet?

1. Szukamy największego dostępnego nominału nie większego od reszty.
2. Wydajemy nominał tyle razy, ile można, aby nie przekroczyć kwoty reszty.
3. Pomniejszamy resztę do wydania o łączną wartość już wybranych nominałów, a następnie powtarzamy kroki 1 i 2.

Jak zwrócić resztę wynoszącą 142,43 zł?

Kwota do wydania	Nominał	Liczba banknotów lub monet
142,43 zł	100 zł	1
42,43 zł	20 zł	2
2,43 zł	2 zł	1
0,43 zł	0,20 zł	2
0,03 zł	0,02 zł	1
0,01 zł	0,01 zł	1

Kolejne kwoty są wyznaczane przez resztę z dzielenia:

```
kwota = kwota mod nominał
```

Kolejne liczby banknotów lub monet są wyznaczane przez dzielenie całkowite:

```
liczba = kwota div nominał
```

Polskie nominały

W programie realizującym ten algorytm wszystkie nominały można wyrazić w groszach i przechować jako liczby całkowite w tablicy o stałych wartościach, np.:

```
Nominaly[15] = {50000, 20000,
10000, 5000, 2000, 1000, 500,
200, 100, 50, 20, 10, 5, 2, 1}
```

Oto specyfikacja problemu wydawania reszty oraz zapis algorytmu:

Specyfikacja

Dane: `kwota` – reszta do wydania wyrażona w groszach, liczba całkowita dodatnia, `n` – liczba dostępnych nominałów, `Nominaly[0..n–1]` – tablica dostępnych nominałów, wyrażonych w groszach, uporządkowana malejąco.

Wynik: `Reszta[0..n–1]` – tablica z liczbą poszczególnych banknotów i monet o łącznej wartości `kwota`. Kolejność nominałów w tablicy `Reszta` jest zgodna z kolejnością nominałów w tablicy `Nominaly`.

Dobra rada

Zawsze dokładnie czytaj specyfikację problemu. Często narzuca ona typ danych konieczny do zastosowania w programie.

```
dla i ← 0, 1, …, n – 1 wykonuj
    Reszta[i] ← kwota div Nominaly[i]
    kwota ← kwota mod Nominaly[i]
```

W pętli wyliczamy liczbę banknotów lub monet o wartości `Nominaly[i]`, korzystając z operacji dzielenia całkowitego. Kwota pozostała do wydania to reszta z dzielenia przez dany nominał.

Ćwiczenie 1

W pliku otrzymanym od nauczyciela (np. *reszta.xlsx*) zrealizuj algorytm wydawania reszty najmniejszą liczbą banknotów lub monet.

Oto fragment kodu źródłowego programu realizującego algorytm:

```
1.  const int N=15;
2.  const int NOMINALY[]={50000,20000,10000,5000,2000,1000,
3.                        500,200,100,50,20,10,5,2,1};
4.  void WydajReszte(int kwota, int Reszta[])
5.  {
6.      for (int i=0;i<N;i++)
7.      {
8.          Reszta[i]=kwota/NOMINALY[i];
9.          kwota=kwota%NOMINALY[i];
10.     }
11. }
12. int main()
13. {
14.     int kwota, Reszta[N];
15.     cout<<"Podaj kwote w groszach: "; cin>>kwota;
16.     WydajReszte(kwota,Reszta);
17.     for (int i=0;i<N;i++)
18.         if (Reszta[i]>0)
19.             cout<<NOMINALY[i]<<": "<<Reszta[i]<<endl;
20.     return 0;
21. }
```

Fragment kodu źródłowego programu wyznaczającego liczbę poszczególnych banknotów lub monet potrzebnych do wydania reszty

Stała tablicowa

Tablice, s. 330

W linii 1 została zadeklarowana stała N, określająca liczbę dostępnych nominałów, a w liniach 2–3 **stała tablicowa** o nazwie NOMINALY, czyli tablica wypełniona stałymi (niezmiennymi) wartościami nominałów. **Tablicom** można nadawać wartości początkowe podczas ich deklaracji – wpisujemy je w nawiasach klamrowych, oddzielając przecinkami. Rozmiar tablicy wynika z liczby elementów w nawiasach klamrowych, więc nie trzeba go podawać.

Przekazanie parametru przez wskaźnik, s. 152

Zwróć uwagę na nagłówek funkcji w wierszu 4. Tablice jako parametry są **przekazywane przez wskaźnik**, więc zmiany wprowadzone w tablicy `Reszta` zostaną zachowane. Pętla w wierszach 17–19 powoduje wypisanie liczby tych nominałów, które są wykorzystywane do wydania reszty. Rysunek 18.1 przedstawia efekt działania programu.

```
Podaj kwote w groszach: 13579
10000: 1
2000: 1
1000: 1
500: 1
50: 1
20: 1
5: 1
2: 2
```

Rys. 18.1. Przykładowe wywołanie programu wyznaczającego resztę wydawaną najmniejszą liczbą banknotów lub monet

Warto wiedzieć

Rozmiar tablicy można także podać podczas określania wartości początkowych. Jeśli podamy mniejszy rozmiar niż liczba elementów w nawiasach klamrowych, będzie to błąd. W przypadku podania większego rozmiaru pozostałe elementy będą miały wartość 0.

Ćwiczenie 2

a. Uruchom program ze s. 257 i sprawdź jego działanie.
b. Popraw program tak, aby wypisywał wartości w złotówkach i groszach.

Warto wiedzieć

Optymalizacja rozwiązania danego problemu może oznaczać coś innego dla różnych problemów. W przypadku wydawania reszty oznacza ona wydanie możliwie najmniejszej liczby banknotów lub monet.

Można podać przykłady hipotetycznych systemów monetarnych, dla których metoda zachłanna nie znajdzie optymalnej (najmniejszej) liczby banknotów i monet. Załóżmy, że w polskim systemie monetarnym monetę 2 zł zastępujemy monetą 4 zł i chcemy wydać 8 zł reszty. Metoda zachłanna wyznaczy wówczas jako resztę jedną monetę 5 zł i trzy monety 1 zł, czyli razem cztery monety. Kwotę 8 zł można by jednak wydać za pomocą dwóch monet 4 zł, czyli w tej sytuacji dla podanych nominałów istnieje lepsze rozwiązanie.

A to ciekawe

Systemy monetarne a wydawanie reszty

Systemy monetarne wielu krajów są tak skonstruowane, aby metoda zachłanna działała w nich optymalnie, tzn. żeby można było wydać resztę najmniejszą liczbą banknotów i monet. Polski system monetarny jest zbudowany na bazie liczb: 5, 2, 1. Wartości banknotów i monet są wielokrotnościami tych liczb.

Choć omówiony algorytm rozwiązuje przedstawiony w specyfikacji problem, nie zawsze wskazuje najmniejszą liczbę monet, którymi można wypłacić resztę. Takie rozwiązania z wykorzystaniem metody zachłannej, o ile są łatwe w implementacji, mogą wystarczać, gdy można zaakceptować przybliżone rozwiązanie danego problemu.

Ćwiczenie 3

Podaj wymyślony przez siebie przykład systemu monetarnego, w którym wydawanie reszty metodą zachłanną nie jest optymalne.

Zapamiętaj

Metoda zachłanna polega na podejmowaniu decyzji optymalnej w danym kroku. Nie rozważa się w tym momencie, jak wybór wpłynie na kolejne kroki. Dlatego podejście zachłanne nie zawsze prowadzi do znalezienia optymalnego rozwiązania. Można je jednak zastosować, gdy wystarczy znaleźć rozwiązanie przybliżone.

18.2. Optymalne wykorzystanie sali

Rozważmy następujący problem: w szkole jest organizowany festiwal teatralny, na który każda z klas przygotowuje przedstawienie. W sali gimnastycznej została ustawiona scena na próby. Czas trwania przedstawień jest różny, klasy rozpoczynają też próby o różnych godzinach. Zadanie polega na ustaleniu harmonogramu, według którego w sali gimnastycznej odbędzie się możliwie jak najwięcej prób przedstawień. Zakładamy, że dwie próby nie mogą się odbywać w sali jednocześnie, a klasy, które nie znajdą się na liście, ćwiczą w innej sali. Tabela 18.1 przedstawia przykładowe dane.

Warto wiedzieć

Optymalizacja oznacza dążenie do znalezienia rozwiązania, które zgodnie z ustalonym kryterium jest oceniane jako najlepsze z dopuszczalnych rozwiązań.

Klasa	Godzina rozpoczęcia	Czas trwania (min)
1a	9	150
1b	10	50
2a	11	60
2b	12	50
3a	13	60
3b	14	80
4a	15	90
4b	16	100

Tabela 18.1. Przykładowe zestawienie godzin rozpoczęcia i czasu trwania prób

Warto wiedzieć

Podczas rozwiązywania problemu wykorzystania sali można wskazać maksymalną liczbę prób, które mogą się w tej sali odbyć, ale dla pewnych układów danych sam wybór klas może być różny. Na przykład jeśli dwie klasy miałyby tej samej długości próbę o tej samej porze, to można byłoby wybrać pierwszą lub drugą z nich.

Maksymalna liczba prób w sali gimnastycznej dla danych zamieszczonych w tabeli 18.1 na s. 259 wynosi 6 (klasy: 1b, 2a, 2b, 3a, 3b i 4b). Zauważ, że gdyby jako pierwsza zaczęła próbę klasa 1a, wówczas próby mogłoby odbyć tylko 5 klas (1a, 2b, 3a, 3b, 4b). W tej sytuacji uwzględnienie w harmonogramie klasy, która najwcześniej rozpoczyna próbę, nie jest dobrym wyborem.

Metoda zachłanna w tym przypadku polega na wyborze w danym kroku próby, która najwcześniej się kończy i nie koliduje z już wybranymi próbami (tymi, które zakończyły się wcześniej). Strategia ta prowadzi to wyboru optymalnego rozwiązania, czyli ustalenia maksymalnej liczby prób, które mogą się odbyć w sali gimnastycznej.

Oto specyfikacja problemu:

Specyfikacja

Dane: `n` – liczba klas, `P[0..n–1]` – tablica prób, próba reprezentowana jest przez dwie liczby całkowite dodatnie `pocz` i `czas`, określające godzinę rozpoczęcia próby i czas jej trwania w minutach.

Wynik: `lp` – maksymalna liczba prób niekolidujących ze sobą.

Aby określić maksymalną liczbę prób, które mogą się odbyć w sali gimnastycznej, wygodnie będzie uporządkować przedstawienia według czasu ich zakończenia.

Algorytm znajdujący maksymalną liczbę prób, przy założeniu uporządkowania przedstawień według czasu ich zakończenia, można zapisać w pseudokodzie następująco:

Dobra rada

Próbę możesz reprezentować z wykorzystaniem tablicy dwuwymiarowej. W pierwszej kolumnie mogą być pamiętane godziny rozpoczęcia prób, a w drugiej – czas ich trwania.

```
lp ← 1  // liczba prób wybranych do sali
ost ← 0 // indeks w tablicy ostatniej wybranej próby
dla i ← 1, 2, …, n – 1 wykonuj
    jeśli P[i].pocz * 60 ≥ P[ost].pocz * 60 + P[ost].czas to
        lp ← lp + 1
        ost ← i
```

Zauważ, że godzina rozpoczęcia próby jest mnożona przez 60, aby otrzymana wartość była wyrażona w minutach (czas trwania próby podajemy w minutach). Do reprezentacji próby (godziny rozpoczęcia i czasu trwania) wykorzystamy **strukturę**. Dzięki temu będziemy mogli skorzystać z **funkcji `sort`** z biblioteki STL w celu uporządkowania tablicy struktur zawierającej informacje o próbach.

Struktura, s. 109

Funkcja `sort`, s. 214

Oto definicja struktury proba:

```
struct proba
{
    int pocz;
    int czas;
};
```


Aby funkcja **`sort`** mogła porównać dwie próby, należy zdefiniować funkcję logiczną określającą, która z dwóch prób kończy się wcześniej. Oto kod źródłowy struktury `proba` i funkcji logicznej `Porownaj`:

```
1. struct proba
2. {
3.     int pocz, czas;
4. };
5.
6. bool Porownaj(proba a, proba b)
7. {
8.     return (a.pocz*60+a.czas<b.pocz*60+b.czas);
9. }
```

Kod źródłowy struktury `proba` **i funkcji logicznej** `Porownaj`

Dobra rada

Dane dotyczące czasu sprowadzaj do jednolitej postaci, np. zamieniając na wartości minutowe.

Obsługa plików tekstowych, s. 336–338

Dane odczytamy z **pliku tekstowego**. Przykład takich danych przedstawia rysunek 18.2. W każdym wierszu znajdują się dwie liczby oddzielone spacją, które określają godzinę rozpoczęcia i czas trwania kolejnych prób (tablica `P`). W pliku znajduje się tyle wierszy, ile wynosi wartość stałej `N`, określającej rozmiar tablicy `P` w programie. Oto kod źródłowy funkcji `MLP` (maksymalna liczba prób) oraz funkcji `main`:

```
9 150
10 50
11 60
12 50
13 60
```

Rys. 18.2. Fragment zestawu danych z pliku *dane.txt*

```
1.  int MLP(proba P[])
2.  // Maksymalna Liczba Prób
3.  {
4.      int lp=1, ost=0;
5.      for (int i=1;i<N;i++)
6.          if (P[i].pocz*60>=P[ost].pocz*60+P[ost].czas)
7.          {
8.              lp++;
9.              ost=i;
10.         }
11.     return lp;
12. }
13.
14. int main()
15. {
16.     proba P[N];
17.     ifstream we("dane.txt");
18.     for (int i=0;i<N;i++)
19.         we>>P[i].pocz>>P[i].czas;
20.     we.close();
21.     sort(P,P+N,Porownaj);
22.     cout<<"Maksymalna liczba prob: "<<MLP(P);
23.     return 0;
24. }
```

Fragment kodu źródłowego programu wyznaczającego maksymalną liczbę prób

Zmienna plikowa, s. 122

Dane liczbowe z pliku tekstowego wczytujemy tak samo jak z klawiatury. Zamiast standardowego wejścia `cin` należy użyć nazwy **zmiennej plikowej** powiązanej z danym plikiem. W linii 16 deklarowana jest *N*-elementowa tablica typu proba. Jeden element tablicy stanowi strukturę pamiętającą dwie liczby opisujące próbę. W wierszu 17 deklarowana jest zmienna we typu `ifstream`, a z nią skojarzony jest plik *dane.txt*, który jest otwarty w trybie do odczytu. Pętla w wierszach 18–19 odczytuje godziny rozpoczęcia i czas trwania prób. W następnym wierszu plik jest zamykany, ponieważ nie będzie już potrzebny. Aby skorzystać z obsługi plików, należy dołączyć bibliotekę `fstream`.

W wierszu 21 wywołana jest funkcja sortująca `sort`. Jej trzeci parametr (funkcja Porownaj) określa sposób porównania dwóch elementów. W linii 22 wywoływana jest funkcja MLP, a na ekranie wypisywana jest jej wartość, czyli maksymalna liczba prób. Aby skorzystać z funkcji `sort`, należy dołączyć bibliotekę `algorithm`.

Dobra rada

Pamiętaj, żeby plik z danymi umieścić w tym samym katalogu (folderze), w którym znajduje się plik z programem.

Ćwiczenie 4

Uruchom program i sprawdź jego działanie dla pliku:

a. przekazanego przez nauczyciela (np. *proby.txt*),

b. zawierającego przygotowane przez ciebie dane.

18.3. Maksymalna suma w kwadracie – metoda zachłanna

Szukamy podciągu spójnego o maksymalnej sumie, s. 207–210

Zajmowaliśmy się już problemem znalezienia **podciągu o maksymalnej sumie**. Teraz zapiszemy algorytm, który będzie poszukiwał maksymalnej sumy na kwadratowej planszy z liczbami.

Zadanie polega na przejściu od lewego górnego rogu planszy do prawego dolnego tak, aby suma liczb znajdujących się w poszczególnych polach była jak największa. Jedynymi dozwolonymi ruchami są przejścia na sąsiednie pola w prawo lub w dół. Liczby w polach, w omawianym przez nas przykładzie, będą tylko dodatnie.

Rysunek 18.3 przedstawia przykładową planszę z zaznaczonym na zielono podciągiem o maksymalnej sumie, która wynosi 60.

	0	1	2	3	4
0	5	5	1	3	9
1	8	6	4	9	8
2	5	1	3	9	4
3	5	6	2	8	3
4	5	6	3	6	5

Rys. 18.3. Przykładowa plansza z zaznaczonym podciągiem o maksymalnej sumie

Oto specyfikacja problemu:

Specyfikacja

Dane: n – liczba wierszy i kolumn planszy,
P[0..n–1][0..n–1] – tablica liczb całkowitych dodatnich.

Wynik: suma – maksymalna suma liczb w polach od P[0][0] do P[n–1][n–1], dozwolone ruchy to przejście na sąsiednie pole w prawo lub w dół.

Zastosowanie strategii zachłannej w tym przypadku polega na wyborze najlepszego sąsiedniego pola, czyli pola z większą liczbą. Jeśli obydwa pola mają taką samą wartość, nie ma znaczenia, które z nich zostanie wybrane.

Na poszukiwaną ścieżkę składa się dokładnie $2n - 1$ pól. Po dotarciu do dolnego wiersza można się już poruszać tylko w prawo. Analogicznie po osiągnięciu ostatniej kolumny można już wykonywać tylko ruchy w dół. Aby nie sprawdzać, czy indeks nie przekracza zakresu tablicy, można dodać do niej **wartowników**, jak na rysunku 18.4.

Warto wiedzieć

Algorytmy działające według strategii zachłannej nie biorą pod uwagę kolejnych kroków, wybierają najlepszy ruch tylko w danym kroku.

Wartownik, s. 64

	0	1	2	3	4	5
0	5	5	1	3	9	0
1	8	6	4	9	8	0
2	5	1	3	9	4	0
3	5	6	2	8	3	0
4	5	6	3	6	5	0
5	0	0	0	0	0	0

Rys. 18.4. Plansza z zaznaczonymi wartownikami

Oto zapis algorytmu zachłannego w pseudokodzie:

```
suma ← A[0][0]
w ← 0 // wiersz, w którym się znajdujemy
k ← 0 // kolumna, w której się znajdujemy
dla i ← 1, 2, …, 2 * n – 2 wykonuj
   jeśli A[w+1][k] > A[w][k+1] to
      suma ← suma + A[w+1][k]
      w ← w + 1
   w przeciwnym przypadku
      suma ← suma + A[w][k+1]
      k ← k + 1
```

Dobra rada

Algorytm poszukiwania maksymalnej sumy w kwadracie możesz również zapisać za pomocą następującej pętli:

```
dopóki (w < n – 1 lub
        k < n – 1)
wykonuj
```

Wartością początkową zmiennej suma jest liczba pamiętana w polu startowym. Zmienne w i k przechowują współrzędne pola, w którym się aktualnie znajdujemy (ostatnio wybrane pole). W pętli wybierane jest sąsiednie pole z większą liczbą oraz modyfikowane są wartości zmiennych suma oraz w lub k.

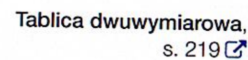
Poniżej znajduje się kod źródłowy funkcji Losuj, losującej planszę (wraz z ustawieniem wartowników), oraz funkcji MaksSuma, realizującej omówiony algorytm.

Kod źródłowy funkcji Losuj, generującej dane w tablicy, funkcji Wypisz, wypisującej liczby z planszy, oraz funkcji MaksSuma, która wyznacza maksymalną sumę liczb

```
1.  const int N=5;
2.  void Losuj(int A[][N+1])
3.  {
4.     for (int i=0;i<N;i++)
5.        for (int j=0;j<N;j++) A[i][j]=1+rand()%9;
6.     for (int i=0;i<N;i++)
7.     {
8.        A[N][i]=0; A[i][N]=0;
9.     }
10. }
11. void Wypisz(int A[][N+1])
12. {
13.    for (int i=0;i<N;i++)
14.    {
15.       for (int j=0;j<N;j++) cout<<A[i][j]<<" ";
16.       cout<<endl;
17.    }
18. }
19. int MaksSuma(int A[][N+1])
20. {
21.    int suma=A[0][0], w=0, k=0;
22.    for (int i=1;i<2*N-1;i++)
23.       if (A[w+1][k]>A[w][k+1])
24.       {
25.          suma+=A[w+1][k]; w++;
26.       }
27.       else
28.       {
29.          suma+=A[w][k+1]; k++;
30.       }
31.    return suma;
32. }
```

Dobra rada

Jeśli chcesz wylosować liczbę c z przedziału [a, b], możesz zastosować zapis: c = a + rand()%(b-a+1)


Tablica dwuwymiarowa, s. 219

Parametrem każdej z funkcji Losuj, Wypisz i MaksSuma jest **tablica dwuwymiarowa**. W języku C++ konieczne jest podanie liczby elementów drugiego wymiaru (i kolejnych wymiarów w przypadku tablic wielowymiarowych). W linii 23 sprawdzane jest, czy przejście w dół planszy daje większą sumę częściową niż przejście w prawo. Przykładowe wywołanie programu i wyznaczenie sumy dla wylosowanej tablicy liczb przedstawia rysunek 18.5.

```
3 2 2 8 9
4 7 1 5 1
2 2 7 6 2
8 5 1 5 1
7 9 2 8 4
Suma: 46
```

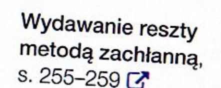
Rys. 18.5. Przykładowe wywołanie programu

Ćwiczenie 5

Napisz program poszukujący maksymalnej sumy pól w kwadracie. Wykorzystaj funkcje Losuj oraz MaksSuma. Czy za każdym razem otrzymujesz maksymalną sumę?

Zwróć uwagę, że metoda zachłanna nie zawsze znajduje maksymalną sumę pól. W przykładzie danych z tego zagadnienia (rys. 18.4, s. 263) wystarczy w polu (2,3) zamienić 9 np. na 7, aby algorytm zachłanny nie wyznaczył maksymalnej sumy.

Program działa więc zgodnie ze specyfikacją (celem jest znalezienie maksymalnej sumy) tylko dla pewnych zestawów liczb. Dla pozostałych zestawów znajduje jedynie rozwiązanie przybliżone, a więc nie realizuje specyfikacji. Podobnie jak w przypadku **problemu wydawania reszty** program zawsze znajduje rozwiązanie, ale nie zawsze jest ono optymalne.


Wydawanie reszty metodą zachłanną, s. 255–259

Ćwiczenie 6

a. Wyznacz maksymalną sumę w kwadracie z rysunku 18.4 na s. 263 po zmianie wartości w polu (2,3) z 9 na 5.

b. Zastosuj metodę zachłanną do wyznaczenia maksymalnej sumy w kwadracie z rysunku 18.4 na s. 263 po zmianie wartości w polu (2,3) z 9 na 5.

18.4. Pakowanie plecaka metodą zachłanną

Pakowanie plecaka to dla wielu z nas duże wyzwanie. Na przykład do samolotu często można zabrać tylko jedną sztukę bagażu i to o ograniczonej wadze. Na wycieczkę górską pakuje się plecak tak, aby nie ważył on zbyt wiele. Spośród przedmiotów, które powinno się zabrać, trzeba wybrać te najbardziej potrzebne tak, aby maksymalnie wykorzystać dopuszczalną wagę.

Takie zagadnienie w ujęciu informatycznym nazywamy **problemem plecakowym**. Można go sformułować w następujący sposób: mamy *n* rodzajów przedmiotów, każdy o określonej wadze i wartości (informujący o jego przydatności). Chcemy wybrać te przedmioty, które w sumie będą miały największą wartość, ale łącznie ich waga nie przekroczy wyznaczonego limitu. Zakładamy przy tym, że dysponujemy nieograniczoną liczbą przedmiotów każdego rodzaju oraz że są one niepodzielne, to znaczy nie możemy zapakować do plecaka fragmentu żadnego przedmiotu. Tak sformułowany problem nazywamy **ogólnym problemem plecakowym**.

Problem plecakowy

Ogólny problem plecakowy

Jeśli mamy do dyspozycji tylko jeden przedmiot każdego rodzaju (pakowanie ogranicza się do decyzji, czy dany przedmiot spakować czy nie), to tak sformułowany problem nazywamy **decyzyjnym problemem plecakowym**.

Decyzyjny problem plecakowy

Pakowanie plecaka

Aby możliwie najlepiej zapakować plecak tak, aby ważył maksymalnie 7 kg, każdemu przedmiotowi przypisujemy jego wagę oraz wartość, która określa, jak bardzo jest przydatny.

Jak wybrać przedmioty do spakowania?

Przedmiot	Waga [g]	Wartość	Iloraz	Łączna waga przedmiotów
Kompas	20	30	1,5	20
Telefon	200	100	0,5	220
Rękawiczki	100	30	0,3	320
Mapa	200	54	0,27	520
Powerbank	400	85	0,21	920
Obiektyw	500	100	0,2	1420
Butelka z wodą	1000	150	0,15	2420
Notes	200	29	0,145	2620
Aparat	900	100	0,11	3520
Laptop	1800	200	0,11	5320
Portfel	150	16	0,107	5470
Książka	200	20	0,1	5670
Zasilacz	800	68	0,085	6470
Linka	500	34	0,068	6970
Sweter	650	40	0,062	7620
Rondel	350	15	0,043	7970
Okulary	50	2	0,04	8020

Obliczamy stosunek wartości do wagi. Im wyższy iloraz, tym korzystniej jest spakować dany przedmiot

Przedmioty porządkujemy malejąco według ilorazów i wyliczamy narastająco łączną wagę przedmiotów. Na zielono zaznaczono najistotniejsze przedmioty, których łączna waga nie przekracza 7 kg

W dalszej części tematu zajmiemy się ogólnym problemem plecakowym z nieograniczoną liczbą przedmiotów do zapakowania.

Specyfikacja

Dane: `n` – liczba rodzajów przedmiotów, `P[0..n–1]` – tablica przedmiotów, przedmiot reprezentowany jest przez dwie liczby całkowite dodatnie `waga` i `wart`, określające jego wagę i wartość, `makswaga` – maksymalna łączna waga przedmiotów, które można spakować. Dysponujemy nieograniczoną liczbą egzemplarzy każdego rodzaju przedmiotu.

Wynik: `K[0..n–1]` – tablica liczb całkowitych nieujemnych określających, ile egzemplarzy każdego przedmiotu należy spakować tak, aby łączna wartość była jak największa, a łączna waga była mniejsza lub równa `makswaga`; `makswart` – wartość spakowanych przedmiotów.

Podobnie jak w poprzednich przykładach zastosujemy strategię zachłanną. Należy się zastanowić, według jakiego kryterium wybierać kolejne przedmioty, aby w każdym kroku podejmować najkorzystniejszą decyzję. Najlepszym wyborem wydaje się uwzględnienie zarówno wartości, jak i wagi przedmiotów. Kryterium wyboru będzie więc stosunek wartości do wagi – im wyższy, tym korzystniej będzie spakować dany przedmiot. Przedmioty porządkujemy malejąco według tego stosunku.

Oto zapis algorytmu w pseudokodzie:

```
makswart ← 0  // łączna wartość spakowanych przedmiotów
dla i ← 0, 1, …, n – 1 wykonuj
    K[i] ← makswaga div P[i].waga
    makswaga ← makswaga mod P[i].waga
    makswart ← makswart + K[i] * P[i].wart
```

Do reprezentacji pojedynczego przedmiotu (jego wartości i wagi) wykorzystamy **strukturę**. Umożliwi to uporządkowanie tablicy według określonego kryterium za pomocą **funkcji `sort`** z biblioteki STL. Oto kod źródłowy definiujący strukturę `przedmiot` i funkcję `Porownaj`:

```
1. struct przedmiot
2. {
3.     int wart, waga;
4. };
5.
6. bool Porownaj(przedmiot a, przedmiot b)
7. {
8.     return (float(a.wart)/a.waga > float(b.wart)/b.waga);
9. }
```

Warto wiedzieć
Stosunek ceny do jakości jest wykorzystywany przy tworzeniu różnego rodzaju rankingów, np. urządzeń elektronicznych.

Warto wiedzieć
Algorytm wyboru przedmiotów jest bardzo podobny do algorytmu wydawania reszty.

Dobra rada
Zauważ, że ze struktury korzystaliśmy też przy tworzeniu harmonogramu wykorzystania sali.

Struktura, s. 109
Funkcja `sort`, s. 214

Kod źródłowy struktury `przedmiot` i funkcji `Porownaj`, porównującej dwa przedmioty

W linii 8 używamy znaku większości, ponieważ chcemy posortować wartości od największej do najmniejszej. Interesuje nas też wynik dzielenia rzeczywistego, a nie całkowitego. Dlatego następuje **konwersja typu** na liczby zmiennoprzecinkowe – typ **float** oznacza takie właśnie liczby. Oto kod źródłowy funkcji `Plecak`, realizującej omówiony algorytm:

Konwersja (rzutowanie typu), s. 54

Kod źródłowy funkcji, zwracającej maksymalną wartość przedmiotów, które można zapakować do plecaka

```
1.  int Plecak(przedmiot P[], int makswaga, int K[])
2.  {
3.      int makswart=0;
4.      for (int i=0;i<N;i++)
5.      {
6.          K[i]=makswaga/P[i].waga;
7.          makswaga=makswaga%P[i].waga;
8.          makswart+=K[i]*P[i].wart;
9.      }
10.     return makswart;
11. }
```

Stała `N` oznacza liczbę rodzajów przedmiotów. Wartością funkcji jest łączna wartość spakowanych przedmiotów. Informacja o liczbie poszczególnych przedmiotów jest zwracana w tablicy `K`, przekazywanej przez wskaźnik.

Algorytm nie zawsze znajduje optymalną wartość, a odpowiedzi, które zwraca, należy traktować jak przybliżenia optymalnego rozwiązania. Dla danych z tabeli 18.2 i maksymalnej wagi 17 algorytm znajduje wartość 54, choć optymalnym rozwiązaniem jest 55.

Warto wiedzieć

Tablice domyślnie przekazywane są przez wskaźnik.
To znaczy, że wszystkie zmiany, które funkcja wprowadza w tablicy, zostają zachowane po zakończeniu działania funkcji.

Wartość	10	15	11	4	3
Waga	3	5	4	2	2
Iloraz	3,33...	3	2,75	2	1,5

Tabela 18.2. Przykładowy zestaw danych dla ogólnego problemu plecakowego

Ćwiczenie 7

a. Napisz program rozwiązujący w przybliżony sposób ogólny problem plecakowy. Dane wczytaj z pliku tekstowego, który otrzymasz od nauczyciela (np. *do_plecaka.txt*). Oto przykład wywołania programu:

```
Wartosc: 54
Liczby poszczegolnych przedmiotow:
5 o wartosci 10 i wadze 3
0 o wartosci 15 i wadze 5
0 o wartosci 11 i wadze 4
1 o wartosci 4 i wadze 2
0 o wartosci 3 i wadze 2
```

b. Które przedmioty i w jakiej liczbie należy spakować, aby dla danych z tabeli 18.2 otrzymać wartość 55?

Podsumowanie

- Metoda zachłanna polega na podejmowaniu najkorzystniejszej decyzji w danym kroku rozwiązania.
- Problem wydawania reszty polega na przedstawieniu danej kwoty reszty za pomocą jak najmniejszej liczby banknotów i monet. Aby rozwiązać ten problem metodą zachłanną, należy wybierać w danym kroku jak największy nominał nie większy od kwoty, która pozostała jeszcze do wydania.
- Problem ustalenia optymalnego harmonogramu wykorzystania sali polega na takim przypisaniu zajęć do sali, aby odbyło się ich jak najwięcej. Strategia zachłanna umożliwia optymalne rozwiązanie tego problemu: w danym kroku należy wybierać zajęcia, które kończą się najwcześniej.
- Problem plecakowy polega na takim wyborze przedmiotów do spakowania, aby ich łączna wartość była jak największa, a łączna waga nie przekroczyła dopuszczalnej.
- W ogólnym problemie plecakowym mamy do dyspozycji nieograniczoną liczbę przedmiotów każdego rodzaju. W decyzyjnym problemie plecakowym możemy zabrać tylko jeden przedmiot każdego rodzaju. Problem sprowadza się do decyzji, czy dany przedmiot spakować czy nie.
- Rozwiązując problem plecakowy metodą zachłanną, możemy nie otrzymać optymalnego rozwiązania (o największej wartości).
- Funkcja **sort** z biblioteki STL pozwala określić własne kryterium porównania dwóch elementów poprzez definicję funkcji o wartości logicznej.

Zadania

* **1** W pliku arkusza kalkulacyjnego, który otrzymasz od nauczyciela (np. o nazwie *wydaj_reszte_zlotowki.xlsx*), rozwiąż problem wydawania reszty tak, aby wyświetlane były kwoty w złotówkach i groszach.

* **2** Program poszukujący maksymalnej sumy w kwadracie zmodyfikuj tak, aby wypisywał także liczby tworzące sumę.

* **3** Zapisz w pseudokodzie funkcję realizującą algorytm wyboru przedmiotów w decyzyjnym problemie plecakowym.

** **4** Oszacuj złożoność obliczeniową algorytmów rozwiązywania problemów omówionych w tym temacie, stosujących metodę zachłanną. Przyjmij, że dane są już uporządkowane zgodnie z najlepszym kryterium dla danego problemu (nie musisz uwzględniać kosztu sortowania w szacowaniu złożoności obliczeniowej).

** **5** Napisz program znajdujący przybliżone rozwiązanie decyzyjnego problemu plecakowego metodą zachłanną.