

Lekcja 2: Rekurencja i złożoność obliczeniowa

Wstęp

To lekcja o rekurencji. Zapowiadaliśmy ją od samego początku. Teraz pokazujemy ją w szczegółach - całą anatomię wywołania rekurencyjnego. Postarajcie się jak najlepiej to prześledzić i zrozumieć. Tego nie można nauczyć się na pamięć. To trzeba zobaczyć - i się zachwycić. Programista nie będzie profesjonalistą w swoim zawodzie, jeśli nie będzie miał wyczucia w stosowaniu rekurencji. Po tym, czy umie ją stosować, można poznać prawdziwego guru...

Złożoność obliczeniowa

Zagadnienia złożoności obliczeniowej algorytmów stanowią prawie zawsze klasyczne rozdziały wstępne w podręcznikach poświęconych algorytmom i strukturom danych. Uzasadnieniem tego jest konieczność przybliżenia pojęć, które będą służyły do oceny jakości prezentowanych później szczegółowo algorytmów. Mogliście zauważyć, że już w poprzedniej lekcji nie uniknęliśmy odwoływania się do złożoności obliczeniowej, bez bliższego wyjaśniania, na czym rzecz polega. Teraz więc najwyższy czas na wprowadzenie elementarnych podstaw w tym zakresie.

Złożoność obliczeniowa algorytmu określona jest przez czas i pamięć potrzebne do jego wykonania. Mówimy więc o **złożoności czasowej** i **złożoności pamięciowej**. Zagadnienia te zwykle stają się ważne dopiero wtedy, gdy stajemy przed problemami przetwarzania dużych ilości danych. Projektując algorytm dla rozwiązania takiego zadania chcielibyśmy, by jego złożoność obliczeniowa była jak najmniejsza - by program liczył się jak najkrócej i zabierał jak najmniej pamięci. Nie zawsze takie podejście jest prawidłowe - często lepiej jest dobrać prostszy algorytm, który zrealizuje zadanie w dostatecznie krótkim czasie, niż poszukiwać algorytmu szybszego, lecz bardziej złożonego, a przez to trudniejszego w implementacji, co może być przyczyną błędów.

Zupełnie bowiem niezależną i nadrzędną sprawą są zagadnienia **poprawności algorytmów**. Nie ma sensu zajmować się złożonością algorytmu, którego poprawność budzi wątpliwości. Dbałość o poprawność algorytmu musi być więc z natury rzeczy pierwszoplanowa. Dowodzenie poprawności (w tym również tzw. poprawności częściowej) algorytmów jest jednak zagadnieniem trudnym i wykraczającym poza ramy tego podręcznika. Jednym z najbardziej oczywistych wymagań jest tzw. własność stopu algorytmu, czyli skończoności obliczeń - dla poprawnych danych algorytm powinien zatrzymać się w skończonym czasie. Ale dowodzenie własności stopu też nie jest sprawą trywialną.

Skoro dowodzenie poprawności algorytmów nie jest łatwe ani często stosowane w praktyce, najpewniejszą zasadą jest pisanie programów prostych i przejrzystych, a dopiero potem, gdy zajdzie taka potrzeba, zastanawianie się nad zmniejszeniem ich złożoności obliczeniowej.

Błędem byłoby jednak całkowite pomijanie problemu złożoności nawet przy prostych zagadnieniach. Przypomnijcie sobie z lekcji programowania proste zadanie wydrukowania elementów leżących na głównej przekątnej tablicy kwadratowej A o $n \times n$ elementach. Rozwiązaniem właściwym jest zastosowanie pojedynczej pętli for z odwołaniem się do kolejnych elementów $A[i,i]$. Zadanie to rozwiązuje się więc w czasie liniowo proporcjonalnym do n , czyli rozmiaru tablicy. Mówimy, że złożoność czasowa tego algorytmu jest rzędu $O(n)$. Za chwilę wyjaśnimy dokładniej, co to znaczy.

Zadanie pobrania elementów z głównej przekątnej tablicy może być też zrealizowane według dużo bardziej pracochłonnego algorytmu, jaki nieraz obserwujemy jako wynik prac zaliczeniowych początkujących studentów. Przechodzą oni w pętli podwójnej wszystkie elementy (i,j) tablicy, za każdym razem sprawdzając, czy może już trafili na przekątną, czyli czy indeks i jest równy indeksowi j . Zamiast wykonać n kroków - wykonują n^2 kroków, i to wymagających dodatkowo wykonania instrukcji warunkowej. Złożoność takiego "algorytmu" jest rzędu $O(n^2)$. Jak widać, ten tzw. naiwny algorytm o dużej (bo opisanej funkcją kwadratową) złożoności bardzo łatwo jest zastąpić algorytmem działającym w czasie liniowym. Inne ciekawe przykłady prostych ulepszeń algorytmów naiwnych, które obniżają złożoność z kwadratowej na liniową, znajdziecie [na tej stronie](http://wazniak.mimuw.edu.pl), w uniwersyteckim portalu informatycznym WAŻNIAK: <http://wazniak.mimuw.edu.pl>.

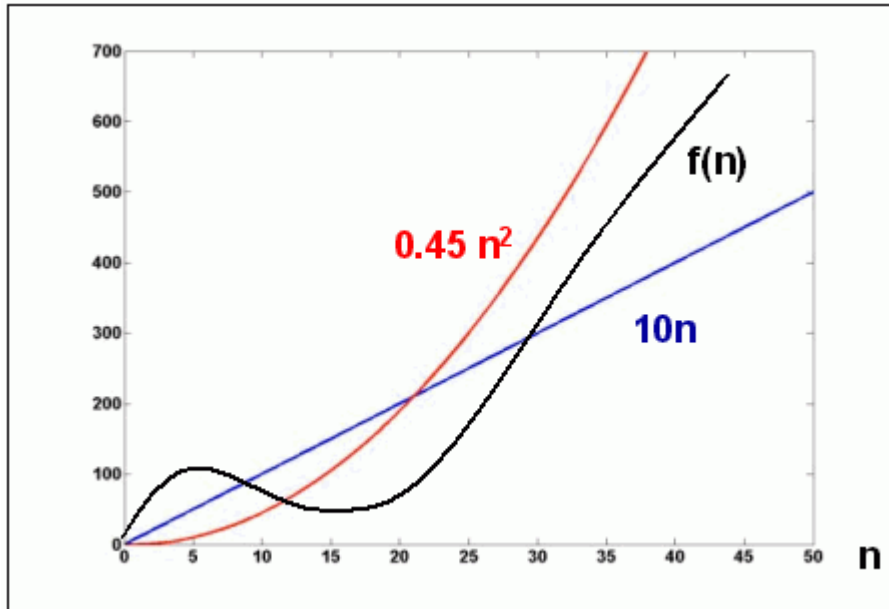
W analizie algorytmów zagadnienia złożoności czasowej są znacznie ważniejsze i znacznie częściej rozpatrywane niż zagadnienia złożoności pamięciowej, bo brak pamięci nie jest w dzisiejszych czasach sprawą krytyczną. Aby uniezależnić się od sprzętu, na którym realizowany jest algorytm, złożoność czasową mierzy się jako liczbę wykonanych operacji charakterystycznych dla algorytmu i decydujących o jego koszcie - są to tzw. **operacje dominujące**, takie jak np. porównywanie i zamiana elementów w algorytmach sortowania. Złożoność jest przy tym oceniana jako funkcja rozmiaru zadania - rozumianego jako liczba danych, na których algorytm operuje (na przykład liczba danych, które mają być posortowane).

Notacja dużego O

Wiemy już, że oceniając koszt algorytmów najczęściej posługujemy się tzw. notacją "dużego O". Zamiast powiedzieć, że algorytm wykonuje się w czasie proporcjonalnym do kwadratu wartości n , gdzie n jest liczbą danych, mówimy, że złożoność algorytmu jest rzędu $O(n^2)$. Notacja "dużego O" zdefiniowana jest następująco:

Mówimy, że funkcja f ma złożoność $O(g(n))$, jeśli istnieje pewna dodatnia stała c oraz pewna nieujemna wartość całkowita N , taka że dla wszystkich $n \geq N$ spełniona jest zależność $f(n) \leq cg(n)$.

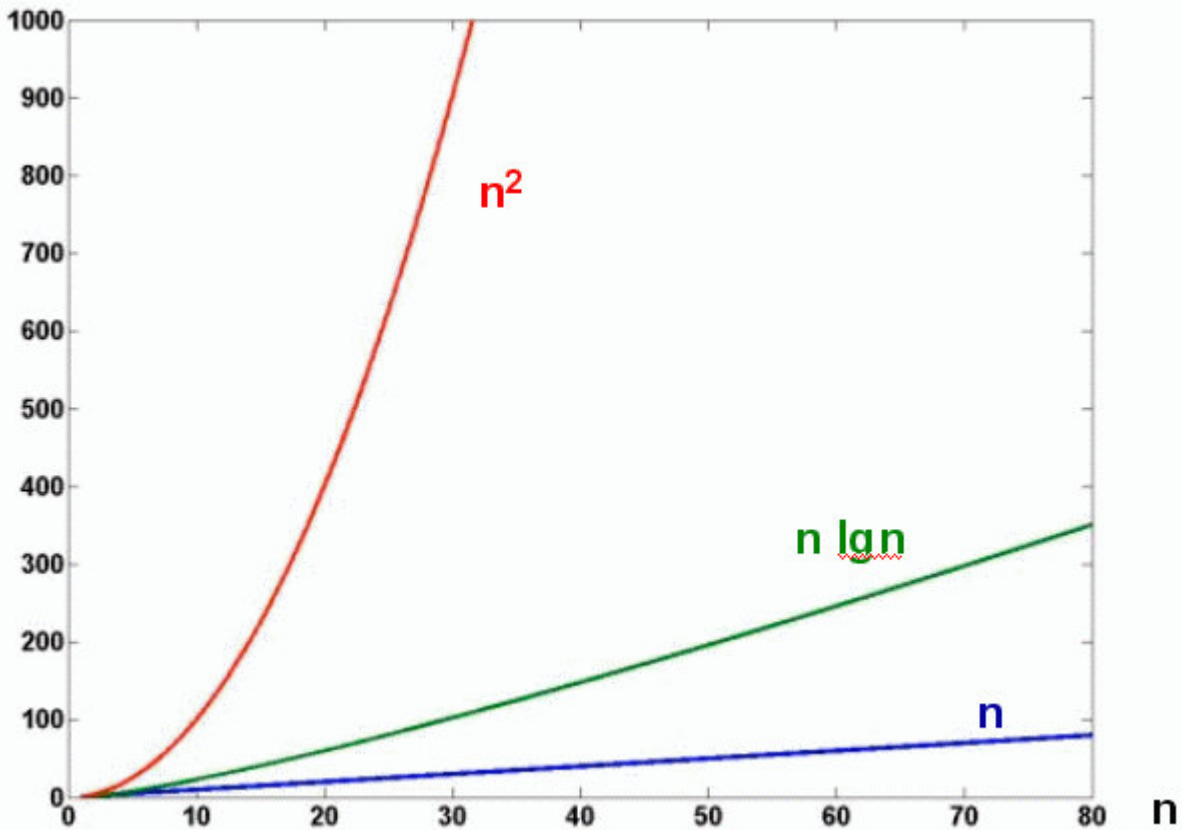
Oznacza to, że dla dostatecznie dużego n i dla pewnej stałej c wykres funkcji złożoności $f(n)$ leży poniżej wykresu funkcji $c \cdot g(n)$. Zobaczcie to na przykładzie trzech funkcji opisujących złożoność obliczeniową i przedstawionych na poniższym wykresie:



Jedna z tych funkcji jest funkcją kwadratową. Druga z nich, funkcja $f(n)$, dla małych n leży powyżej wykresu funkcji kwadratowej, ale poczynając od pewnego n leży całkowicie pod wykresem funkcji kwadratowej. Możemy więc powiedzieć, że funkcja $f(n)$ jest rzędu $O(n^2)$. Co więcej, to samo możemy powiedzieć również o funkcji liniowej, jakkolwiek w tym przypadku bardziej naturalne jest określenie jej jako funkcji $O(n)$. Dla danej funkcji f istnieje nieskończenie wiele funkcji $cg(n)$ ograniczających ją od góry, ale z reguły wybiera się tę "najmniejszą" - w tym przypadku liniową, a nie kwadratową.

Notacja "dużego O" dotyczy więc asymptotycznego kresu górnego złożoności - opisuje asymptotyczne zachowanie funkcji, jej ograniczenie od góry, dla odpowiednio dużego n . Zauważmy też, że zarówno $10n^2 + 100n$, jak i $n^2/10$ są rzędu $O(n^2)$.

Na zakończenie popatrzcie na poniższy wykres. Przedstawione są tu trzy najczęściej spotykane funkcje opisujące złożoność obliczeniową typowych algorytmów w zależności od rozmiaru zadania n . Zwróćcie uwagę na funkcję **$n \lg n$** , leżącą pomiędzy funkcją liniową a kwadratową, ale znacznie bliżej tej liniowej. To tzw. funkcja liniowo-logarytmiczna, przy czym uwaga: $\lg n$ jest to **logarytm przy podstawie 2** z wartości n . To osiągnięcie tej właśnie obniżonej złożoności, zamiast złożoności opisanej funkcją kwadratową, jest bardzo często efektem zastosowania specjalnie dobranych algorytmów - wtedy kiedy uzyskanie złożoności liniowej nie jest możliwe lub bardzo trudne. Przykładowo w następnej lekcji zajmiemy się różnymi algorytmami sortowania. Pokażemy, że proste algorytmy sortowania mają złożoność $O(n^2)$, szybkie algorytmy sortowania mają złożoność $O(n \lg n)$.



Warto też podkreślić, że bardzo niska złożoność $O(\lg n)$, możliwa do uzyskania w niektórych algorytmach (np. w przeszukiwaniu binarnym, co pokażemy w następnym rozdziale) na poniższym wykresie obrazowana byłaby funkcją prawie pokrywającą się z osią ox , trudną do narysowania. Z kolei funkcja 2^n wznosiłaby się bardzo stromo, blisko osi oy - tak wysoką złożoność mają niektóre algorytmy, które zajmiemy się jeszcze w tej lekcji.

Rekurencja

Wspomnieliśmy już wcześniej, że podprogram może wywoływać inne podprogramy. W szczególności może również wywoływać samego siebie - to prawie jak z wężem zjadającym swój własny ogon :). Wywołanie takie, jak pamiętacie z lekcji 1, nazywamy wywołaniem rekurencyjnym.

Co to jest ?

Ogólnie **rekurencją** (lub **rekursją**) nazywamy sposób pojmowania rzeczy w kategoriach odwoływania się od samych siebie. Zakrawa to oczywiście na syndrom nieskończoności i jako takie jawi nam się jako nieco niezwykle. Lecz postaramy się Wam pokazać, że ów "syndrom nieskończoności" jest jedynie paradoksem, a sama rekurencja jest bardzo elegancką konstrukcją często wykorzystywaną w programowaniu. Przy czym w tej lekcji poznacie rekurencyjne wywołania podprogramów, natomiast w lekcji 4 - najprostszą rekurencyjną strukturę danych.

Rekurencyjna funkcja odwołuje się w sposób pośredni bądź bezpośredni do samej siebie. Bezpośrednią rekurencję bardzo łatwo zaobserwować. Jeśli gdziekolwiek w kodzie programu napotkacie coś takiego:

```
void ZROB_COS () {
    ...
    ZROB_COS (); // wywołanie funkcji przez samą siebie
    ...
}
```

to właśnie oznacza, że natknęliście się na kod rekurencyjny.

Do czego wykorzystuje się rekurencję? Zanim przedstawimy Wam proste matematycznie przykłady, pokażemy, że zwykłe codzienne czynności też można zapisać w sposób rekurencyjny. Popatrzcie na taki przykład, którego pomysł jest podawany w wielu miejscach za informatykiem A.P. Jerszowem:

```
void JEDZ_KASZKĘ() {
    zjedz_łyżkę_kaszki;
    if (! talerz_pusty)
        JEDZ_KASZKĘ ();          // tu funkcja wywołuje sama siebie;
}
```

Cykliczna czynność zjadania kaszki łyżką po łyżce została tu zapisana...bez użycia pętli. Ale ta pętla istnieje, tyle że w sposób niejawni - ukryta jest właśnie w rekurencji. Każdorazowe wywołanie funkcji JEDZ_KASZKĘ powoduje zjedzenie kolejnej łyżki i ponowne jej wywołanie, czyli znowu zjedzenie jednej łyżki i kolejne wywołanie funkcji - i tak w kółko, aż do momentu, gdy talerz będzie pusty.

Można by powiedzieć: a po co tu stosować rekurencję? To samo można przecież zapisać prościej, z użyciem pętli, którą dobrze znamy:

```
void JEDZ_KASZKĘ() {
    do
        zjedz_łyżkę_kaszki;
    while (! talerz_pusty);
}
```

I rzeczywiście, w tym przypadku tak jest. Ale ten banalny zdawałoby się pomysł dobrze służy pokazaniu idei zapisu rekurencyjnego i pozwala zrozumieć jego zasadę. I zaraz zobaczycie, że nie każdą rekurencję da się tak łatwo zastąpić iteracją.

Najpierw jednak zaczniemy od przykładu bardzo w swej konstrukcji podobnego do tego powyżej. Założmy, że mamy **wczytywać i drukować kolejne liczby całkowite aż do napotkania zera** (zero też trzeba drukować). Funkcję realizującą to zadanie można bardzo łatwo napisać w sposób nierekurencyjny, przy użyciu pętli repeat:

```
1. void CZYTAJ_i_PISZ() {                      // wersja iteracyjna
2.     int x;                                  // zmienna lokalna
3.     do {
4.         cin >> x;                            // wczytanie liczby
5.         cout << x << ' ';                  // wydrukowanie liczby wczytanej
6.     } while (! x==0);                       // dopóki nie napotkano zera
7. }
```

Z rekurencyjną wersją też powinniście już sobie poradzić: trzeba to zrobić prawie tak samo, jak w przykładzie z kaszką:

```
1.
2. void CZYTAJ_i_PISZ() {                      // wersja rekurencyjna
3.     int x;                                  // zmienna lokalna
4.     cin >> x;                            // wczytanie liczby
5.     cout << x << ' ';                  // wydrukowanie liczby wczytanej
6.     if (! x==0)                          // jeśli nie wczytano zera
7.         CZYTAJ_i_PISZ();                 // funkcja wywołuje sama siebie
8. }
```

Jak to działa

Przedstawione powyżej przykłady zawierają najprostsze postaci rekurencji - tzw. rekurencję ogonową. Charakteryzuje się ona tym, że:

- wywołanie rekurencyjne następuje na samym końcu działania funkcji, tzn. w momencie wywołania funkcji nie ma już w niej nic więcej do wykonania
- wywołanie to jest jedynym wywołaniem rekurencyjnym w tej funkcji.

Można by taką funkcję rekurencyjną uznać za sztukę dla sztuki (choć w niektórych językach taki udoskonalony zapis pętli bywa przydatny czy wręcz konieczny), gdyby nie to, że ten pomysł pomoże Wam zrozumieć zasadę zapisu znacznie trudniejszych algorytmów rekurencyjnych. Założmy więc, że mamy napisać funkcję, która **wczytuje liczby aż do napotkania zera i drukuje w kolejności odwrotnej** (wstecz), poczynając od zera.

Gdybyście próbowali napisać to z pętlą, okazałoby się natychmiast, że nie da się tego już tak prosto zrobić - bo trzeba gdzieś te wczytywane liczby przechowywać. Można by zastosować tablicę - tylko trzeba by ją zdefiniować "na wyrost" i potem wykorzystać

tylko tyle miejsc, ile liczb wczytano. Należałoby więc wczytywać do tablicy liczby od przodu, a potem wypisywać poruszając się wstecz. Ale okazuje się, że za pomocą rekurencji można to napisać znacznie prościej, i to bez nadmiarowej rezerwacji pamięci. Wystarczy tylko przestawić jedną linijkę w poprzedniej funkcji - i nic więcej. Popatrzcie:

```
1. void PISZ_WSTECZ() { // wersja rekurencyjna
2.   int x; // zmienna lokalna
3.   cin >> x; // wczytanie liczby
4.   if (! x==0) // jeśli nie wczytano zera
5.     PISZ_WSTECZ(); // funkcja wywołuje samą siebie
6.   cout << x << ' '; // wydrukowanie liczby wczytanej
7. }
```

Napiszcie sobie cały program z tą funkcją (wystarczy ją po prostu wywołać z poziomu funkcji main) i sprawdźcie, że to działa, choć na pierwszy rzut oka wcale tego nie widać. Nie martwcie się, jeśli to nie jest dla Was oczywiste. Trzeba dobrze zrozumieć, co się tu dzieje i jak to wykonuje kompilator. Najważniejsze to zauważyć, że:

1. funkcja wywołując samą siebie zaczyna wszystko od początku, czyli wczytuje liczbę, sprawdza ją, po czym albo ją drukuje i kończy działanie, albo nie kończy działania, tylko ponownie wywołuje samą siebie (a więc - zaczyna wszystko od początku, czyli wczytuje liczbę...itd.)
2. w momencie, gdy funkcja wywołuje samą siebie, nie zrobiła jeszcze wszystkiego, co zawiera się w jej treści - i że to trzeba będzie kiedyś dokończyć. Kiedyś - czyli gdy zakończy się realizacja tego samowywołania, a więc po zakończeniu działania instrukcji `if x<>0 then PISZ_WSTECZ`.

Popatrzmy dla przykładu, jak wykona się funkcja po wczytaniu 3 liczb niezerowych i zera na końcu:

- wczytaj liczbę, a ponieważ nie jest ona zerem, wywołaj samą siebie, czyli...
- wczytaj liczbę, a ponieważ nie jest ona zerem, wywołaj samą siebie, czyli...
- wczytaj liczbę, a ponieważ nie jest ona zerem, wywołaj samą siebie, czyli...
- wczytaj liczbę, a ponieważ jest ona zerem, wydrukuj tę liczbę
- dokończ to, co zostało do zrobienia w poprzednim wywołaniu funkcji, czyli wydrukuj poprzednio wczytaną liczbę
- dokończ to, co zostało do zrobienia w jeszcze wcześniejszym wywołaniu funkcji, czyli wydrukuj jeszcze wcześniej wczytaną liczbę
- dokończ to, co zostało do zrobienia w jeszcze wcześniejszym wywołaniu funkcji, czyli wydrukuj jeszcze wcześniej wczytaną liczbę

Schematycznie można to zapisać następująco:

```
cin >> x; // wczytanie pierwszej liczby
cin >> x; // wczytanie drugiej liczby
cin >> x; // wczytanie trzeciej liczby
cin >> x; // wczytanie czwartej liczby
// czwarta liczba okazała się zerem
cout << x << ' '; // wydrukowanie czwartej liczby
cout << x << ' '; // wydrukowanie trzeciej liczby
cout << x << ' '; // wydrukowanie drugiej liczby
cout << x << ' '; // wydrukowanie pierwszej liczby
```

Zapis taki będzie jednak prawidłowy tylko przy założeniu, że kolejno używane zmienne x to nie są te same zmienne, tylko kopie zmiennej lokalnej x, kolejno tworzone przy każdym wywołaniu funkcji, by nie stracić poprzednio wczytanej wartości. I tak to rzeczywiście wykonuje kompilator:

- po pierwszym wywołaniu tej funkcji (czyli z poziomu funkcji main) wczytuje pierwszą liczbę
- jeśli jest ona zerem, to ją drukuje i wraca do miejsca wywołania w funkcji main
- jeśli zaś nie jest zerem, to zapamiętuje miejsce w funkcji main, do którego ma powrócić, i zapamiętuje wczytaną liczbę
- zaczyna wykonywać funkcję PISZ_WSTECZ od nowa - czyli wczytuje następną liczbę, ale używając nowej zmiennej lokalnej x do jej przechowywania
- jeśli nie jest ona zerem, to znowu zapamiętuje miejsce, do którego należy powrócić w funkcji wywołującej (czyli w funkcji PISZ_WSTECZ) i zapamiętuje kolejną wczytaną liczbę
- za każdym razem, gdy funkcja PISZ_WSTECZ wywoła samą siebie, i wczyta wartość niezerową, wszystko wykonuje się tak samo - czyli wciąż nic się nie drukuje, a tylko kolejne liczby zapamiętywane są przy użyciu kolejnych zmiennych pomocniczych będących kopiami zmiennej lokalnej x

- w końcu następuje wczytanie zera - zatem kompilator od razu przechodzi do instrukcji pisania, czyli wydrukuje tę wartość zerową
- powraca do poprzedniego wywołania funkcji, czyli odczytuje miejsce, do którego należało powrócić, zapamiętaną poprzednio wartość zmiennej lokalnej `x` - i ją drukuje
- wszystko powtarza się tak samo aż do momentu dokończenia wykonania wszystkich wywołań funkcji.

Miejsce powrotu jest określone przez adres pierwszej instrukcji za instrukcją wywołującą, czyli w naszym przykładzie adres instrukcji `cout << x << ' ';`. Może warto jeszcze na zakończenie uzmysłowić Wam, jak to jest naprawdę realizowane, bo to też ułatwi zrozumienie istoty rekurencji. Otóż odbywa się to następująco:

- każdorazowe wywołanie funkcji rekurencyjnej powoduje zapamiętanie (odłożenie) na tzw. **stosie rekursji** adresu powrotnego oraz wartości zmiennych i parametrów funkcji (tych przekazywanych przez wartość) wraz z adresem powrotu do miejsca wywołania tej funkcji
- po zakończeniu wykonania podprogramu rekurencyjnego następuje powrót do podprogramu macierzystego (wywołującego) - do miejsca wskazywanego przez adres powrotu na stosie, a zapamiętane na stosie zmienne lokalne oraz parametry stają się znowu aktywne i wykonanie funkcji macierzystej jest kontynuowane.
- jednocześnie informacje o tych zmiennych i parametrach oraz adres powrotu są zdejmowane ze stosu jako już niepotrzebne - przestają istnieć

W ten sposób powstaje stos, który najpierw rośnie, a później zanika. Informacje są zdejmowane ze stosu w kolejności odwrotnej do odkładania. *Stos* czyli **kolejkę LIFO** znacie już dobrze z lekcji 1. Dla naszego przykładu funkcji `PISZ_WSTECZ` stos ten w kolejnych fazach działania funkcji należy sobie wyobrażać następująco:



Stos najwyższy na tym rysunku odpowiada sytuacji, kiedy wczytano `x=0`. Od tej pory ze stosu będą zdejmowane kolejne informacje. Najniżej na tym stosie położony adres powrotu jest to adres powrotu do funkcji `main`, wywołującej naszą funkcję za pierwszym razem.

Skoro już wiecie, jak to wszystko działa, dużo łatwiej przyjdzie Wam zrozumieć **funkcję obliczającą silnię**. Jak wiadomo,

$$N! = 1 * 2 * 3 * \dots * (N-1) * N$$

Stosując taką definicję silni, trudno bezpośrednio zauważyć, gdzie tu ukryte jest odwoływanie się do samej siebie. Lecz silnię można także zdefiniować następująco:

$$N! = \begin{cases} 1 & \text{dla } N=1 \\ N * (N-1)! & \text{dla } N>1 \end{cases}$$

W tym przypadku widać od razu, że obliczenie silni dla `N` wymaga "jedynie" znajomości silni dla `N-1`. Innymi słowy, aby obliczyć silnię `N`, musimy najpierw obliczyć silnię `N-1` - czyli funkcja musi wywołać samą siebie. Napiszmy więc taką funkcję:

```

1. int silnia(int n)
2. {
3.     if (n > 1)
4.     {
5.         // w następnej linijce jest właśnie ukryta rekurencja. Wywołujemy
6.         // funkcję silnia z jej wnętrza
7.         return n * silnia (n-1);
8.     }
9.     else

```

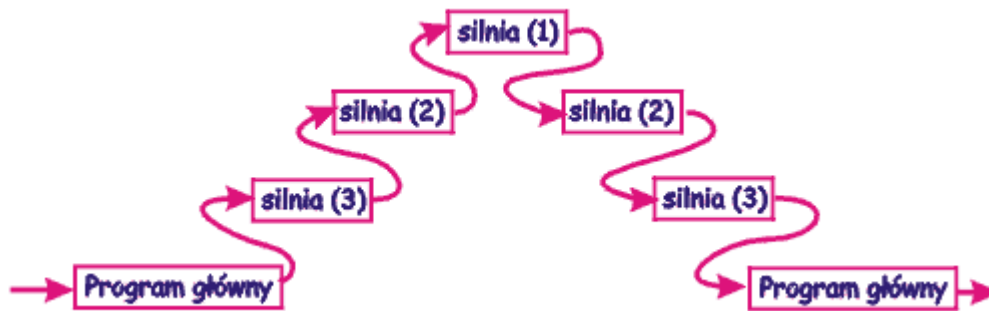


```

10. {
11.     return 1;
12. }
13. };

```

Poniżej zamieściliśmy rysunek, który obrazuje tok wykonania programu wykorzystującego tę funkcję. Każdy kolejny wywołanie to przerwanie obliczeń na danym poziomie i przejście na poziom wyższy, który zarazem obrazuje narastający stos. Dopiero po wyznaczeniu wartości `silnia(1)` kompilator powraca kolejno do przerwanych obliczeń, by wynik końcowy zwrócić do programu głównego.



Przykłady geometryczne

Rekurencyjne spojrzenie wykorzystamy teraz do rozwiązania następującego problemu geometrycznego. Załóżmy, że dany jest obszar złożony z pewnej liczby pikseli, zawierający kontur, którego wnętrze chcemy wypełnić innym kolorem niż aktualny kolor pikseli w tym wnętrzu. Należy więc tak "rozlać" nową farbę, aby dopłynęła do brzegów konturu lub do granic analizowanego obszaru. Okazuje się, że ten algorytm, powszechnie znany pod nazwą **FloodFill**, można zapisać niesłychanie prosto korzystając z czterech wywołań rekurencyjnych. Zobaczcie, jak to wygląda, przy założeniu, że analizowany obszar jest prostokątną macierzą pikseli o współrzędnych w zakresie $[0..ymax, 0..xmax]$, gdzie y rosną w dół, a x rośnie na prawo:

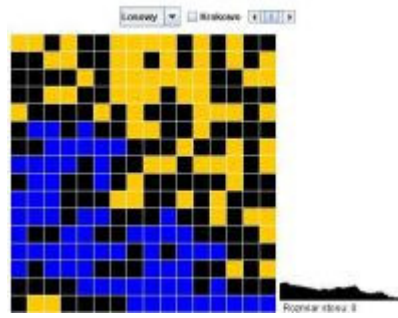
```

1. void Fill (x, y) { /
   / x, y - miejsce, od którego zaczynamy rozlewanie farby
2.
3.     if (x<0 !! y<0 !! x>xmax !! y>ymax) return; /
   / koniec, jeśli wyszliśmy poza granice obszaru
4.     if (getPixel (x, y) == oldColor) { // jeśli piksel (x,y)
       ma stary kolor
5.         putPixel (x, y, newColor); // zamaluj go nowym kolorem
6.         Fill (x, y - 1); /
       / zrób krok do góry i zacznij od początku
7.         Fill (x - 1, y); /
       / zrób krok w lewo i zacznij od początku
8.         Fill (x, y + 1); /
       / zrób krok na dół i zacznij od początku
9.         Fill (x + 1, y); /
       / zrób krok w prawo i zacznij od początku
10.    }
11. }
12. // getPixel - pobierz kolor danego piksela
13. // putPixel - zamaluj dany piksel danym kolorem

```

Żeby zobaczyć, jak to działa, uruchomcie sobie poniższy aplet, klikając w obrazek. Po uruchomieniu apletu, lewym przyciskiem myszy dodajemy czarne piksele do konturu. Możemy też wybrać z listy rozwijanej kształt konturu lub go wylosować. Kliknięcie prawym przyciskiem myszy w dowolny piksel powoduje, że piksel ten i wszystkie sąsiednie piksele w tym samym kolorze zostaną zamalowane nowym kolorem (niebieskim=`newColor`).

Jeśli więc klikniemy w piksel pomarańczowy, niebieska farba pokryje ten piksel i rozleje się na sąsiednie pomarańczowe piksele - aż do granic czarnego konturu lub do brzegu prostokątnej tablicy. Ale możemy też kliknąć prawym przyciskiem w czarny piksel - wtedy czarny kolor to `oldColor` i niebieska farba zaleje ten czarny piksel i wszystkie czarne z nim sąsiadujące. Wypróbujcie to sami; zwróćcie przy tym uwagę, że w funkcji tej w ogóle nie jest ważny kolor konturu ograniczającego, a liczy się tylko to, że ma on kolor inny niż stary kolor, wybrany myszą.

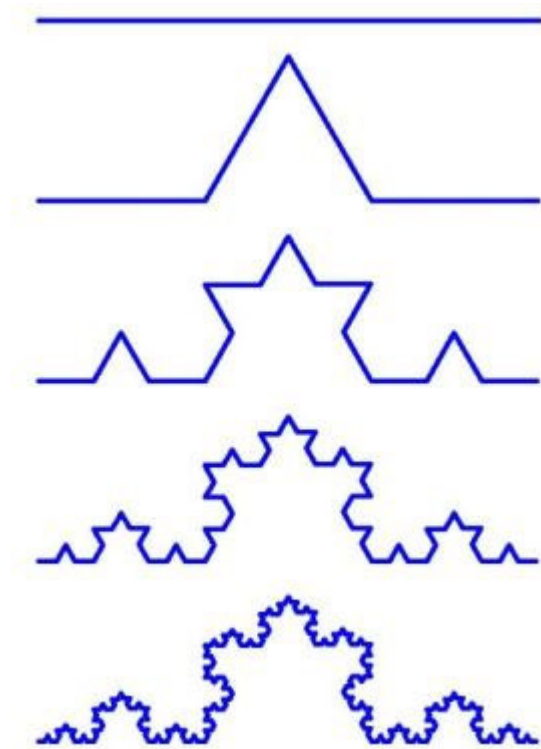


Aplet można uruchomić z bardzo małą prędkością (regulacja suwakiem) lub krokowo i dokładnie się przyjrzeć kolejności wykonywanych kroków, poczynając od wybranego myszką piksela. Na żółto jest zaznaczany aktualnie sprawdzany piksel. Zwróćcie uwagę, że jeśli funkcja wywoła samą siebie robiąc krok do góry, to zostają jej jeszcze do wykonania trzy inne wywołania rekurencyjne, które musi odłożyć na potem (zapisując aktualny stan zmiennych x, y na stosie) i do których kiedyś będzie musiała powrócić. Ale zanim powróci, będzie musiała znowu wywołać samą siebie, znowu odłożyć informacje na stos - i tak dalej - więc ten powrót do dokończenia pierwszego wywołania funkcji odbywa się na samym końcu. Ale odbyć się musi, mimo że na przykład w przypadku zamalowywania kwadratu wszystkie piksele są już dawno zamalowane - kompilator tego jednak nie wie, dopiero musi to sprawdzić, a wszystkie wywołania rozpoczęte muszą zostać w swoim czasie dokończone.

Z prawej strony obrazka możecie obserwować **stan stosu** w trakcie działającej funkcji. Na stos odkładane są wartości parametrów aktualnych, czyli współrzędnych (x, y) tego miejsca, do którego trzeba będzie powrócić - tym razem to parametry aktualne, a nie zmienna lokalna zmieniają się przy każdym nowym wywołaniu funkcji i trzeba je odkładać na stos, a potem, gdy już niepotrzebne - zdejmować. Warto popatrzeć, jak różnie przebiega odkładanie informacji na stos w zależności od miejsca startu - dla tego samego obszaru wypełnienia.

Wielokrotne sprawdzanie tych samych pikseli powoduje, że to na pewno nie jest najszybsze w sensie efektywności programu rozwiązanie - znacznie szybsze byłoby wypełnianie wnętrza linii po linii, z poszukiwaniem początku i końca zamalowywanego odcinka tej linii (lub kilku odcinków, jeśli na wypełnianym obszarze są wyspy). Ale to rekurencyjne rozwiązanie jest niesłychanie proste w zapisie, czytelne i eleganckie. Takie są właśnie podprogramy rekurencyjne, choć zwykle nie jest łatwo wpaść na sposób ich napisania. Potrzebna tu często intuicja, pomysłowość i doświadczenie w tworzeniu takich konstrukcji. Dlatego tak szczegółowo opisaliśmy Wam anatomię rekurencyjnych wywołań. A o efektywności rekurencyjnych wersji poczytajcie w następnym rozdziale, poświęconym złożoności obliczeniowej.

Drugi przykład to **krzywa Kocha** (pomysł szwedzkiego matematyka, 1904r.), będąca brzegiem fraktala zwanego płatkami śniegu. Krzywa powstaje z odcinka, który dzielimy na 3 równe części, zastępując część środkową dwoma odcinkami tej samej długości. Każdy z czterech powstałych w ten sposób odcinków znowu dzielimy na 3 części, zastępując część środkową dwoma - i tak dalej aż do uzyskania odpowiedniego poziomu szczegółowości (po kilku krokach różnice pomiędzy kolejnymi wersjami stają się nieuchwytnie). Krzywa taka w nieskończoności składa się z odcinków nieskończenie krótkich, ale długość ma nieskończoną. Kolejne krzywe utworzone w kroku 0, 1, 2, 3 i 4 możecie zobaczyć poniżej:



To rekurencyjne generowanie kształtu z natury rzeczy daje się zapisać w sposób rekurencyjny. Skoro każdy odcinek zastępujemy czterema krótszymi, funkcja musi cztery razy wywołać samą siebie. Żeby napisać w całości taką funkcję, należałoby dodać podane operacje graficzne: rysowanie odcinka, przesunięcie kursora i zmianę kierunku rysowania. I jeszcze coś bardzo ważnego: zapamiętanie położenia kursora na początku wywoływanej funkcji oraz powrót do tego położenia po zakończeniu wszystkich czterech wywołań. Bez tego krzywa składałaby się z rozdzielonych fragmentów. Po narysowaniu czterech małych "ząbków" trzeba bowiem wrócić na początek, zrobić duży krok, obrót i narysować kolejne cztery "ząbki". Ponieważ nie umiecie jeszcze pracować w trybie graficznym, na tym etapie proponujemy Wam tylko przemyślenie poniższego schematu i rysunkowe sprawdzenie jego poprawności.

```
// funkcja rysuje krzywą utworzoną z odcinka o długości bok,
// parametr ile określa liczbę wykonanych podpodziałów
void Koch (int bok, ile){
    if (ile==0)
        rysuj_odcinek_o_dlugosci (bok);
    else {
        ...// zapamiętaj położenie
        Koch (bok/3, ile-1);
        ...// przesun się o bok/3
        ...// obróć się o -60 stopni
        Koch (bok/3, ile-1);
        ...// przesun się o bok/3
        ...// obróć się o 120 stopni
        Koch (bok/3, ile-1);
        ...// przesun się o bok/3
        ...// obróć się o -60 stopni
        Koch (bok/3, ile-1);
        ...// wróć do zapamiętanego położenia
    }
}
```

Rekurencja pośrednia i warunek końca

To co widzieliście do tej pory, było rekurencją bezpośrednią. Wcześniej wspominaliśmy jednak, że istnieje jeszcze jeden rodzaj rekurencji - pośrednia. W tym przypadku funkcja nie wywołuje samej siebie w sposób jawny. Załóżmy, że mamy dwie funkcje: A i B. Jeśli funkcja A wywołuje B, a B wywołuje A, to mamy do czynienia z rekurencją pośrednią - bo to przecież w końcu A pośrednio wywołuje A - czyli samą siebie. Aby uniknąć problemu z definiowaniem funkcji - którą z nich najpierw zdefiniować, A czy B - rozwiązuje się to z wykorzystaniem deklaracji (w postaci nagłówka) jednej z nich: jeszcze jej nie definiujemy, ale już ją deklarujemy (zapowiadamy), aby kompilator wiedział, że będzie. Potrzebna jest tu dyrektywa **forward**. Taki schemat wygląda następująco:

```

1. void A(...); // tylko nagłówek, czyli deklaracja funkcji A
2.
3. void B(...) {
4.     ...
5.     A(...); // funkcja B wywołuje funkcję A
6.     ...
7. };
8.
9. void A(...) {
10.    ...
11.    B(...); // funkcja A wywołuje funkcję B
12.    ...
13. };
14.
15. int main() {
16.    ...
17.    A(...); // funkcja A wywołuje funkcję B,
18.            // czyli pośrednio samą siebie
19. }

```

Na zakończenie podsumowanie dotyczące warunku końca rekurencji. Podstawowym wymogiem związanym z każdym programem rekurencyjnym jest zakończenie w pewnym momencie jego realizacji - czego przeciwieństwem byłoby wykonywanie się w nieskończoność (w praktyce - do przepełnienia się stosu). Aby zagwarantować, że nasz podprogram kiedyś się skończy, musimy wyposażyć go w dwie niezbędne ku temu cechy:

- Po pierwsze, musi istnieć jego wywołanie nierekurencyjne, to znaczy musi w treści funkcji istnieć taka możliwość, by przy pewnych warunkach nie wołała ona samej siebie. W naszym przykładzie z silnią będzie to instrukcja `silnia:=1`, która się wykonuje przy wywołaniu funkcji z argumentem nie większym od 1.
- Drugi warunek jest trudniejszy do wytłumaczenia. Otóż każde z kolejnych rekurencyjnych wywołań musi być coraz bliższe temu nierekurencyjnemu. Odnosząc to do przykładu z silnią: w każdym kolejnym wywołaniu funkcji, `n` musi maleć. Podczas rysowania płata śniegu, musi maleć długość rysowanego odcinka. W przeciwnym przypadku nigdy nie osiągnęlibyśmy końca.

Złożoność algorytmów rekurencyjnych

Poniższe rozważania będą wstępem do efektywności obliczeniowej algorytmów w odniesieniu do rekurencji. Zaczniemy od trywialnego stwierdzenia, że prawie każdy algorytm rekurencyjny można zapisać w postaci iteracyjnej. Podobnie jak definicję silni można zapisać na dwa sposoby, istnieją również co najmniej dwie metody napisania funkcji ją obliczającej. W poprzednim segmencie zamieściliśmy wersję rekurencyjną, teraz przyszła więc pora na wersję iteracyjną:

```

1. int silnia_i(int n)
2. {
3.     int tmp = 1;
4.
5.     for (int i=1; i<=n; i++)
6.         tmp *= i;
7.     return tmp;
8. };

```

W tym akurat przypadku dosyć łatwo jest stworzyć wersję iteracyjną funkcji rekurencyjnej. I jest ona wydajniejsza niż poprzednia wersja. Wykonanie funkcji `silnia_i` wymaga wykonania n mnożeń i dwu przypisań. Natomiast wywołanie procedury `silnia` z poprzedniego segmentu oprócz wykonania n mnożeń wymaga n wywołań funkcji, co daje dodatkowy narzut czasowy. Zatem w tym wypadku wersja iteracyjna będzie wyraźnie szybsza i wymagająca znacznie mniej zasobów.

Pokusimy się tutaj o stwierdzenie, że w większości przypadków wersje iteracyjne podprogramów wykonują się szybciej niż ich odpowiedniki rekursywne. Dlaczego więc warto stosować rekurencję?

Podstawowym powodem jest elegancja rozwiązania. Podprogram rekurencyjny ułatwia człowiekowi skupienie się na rzeczywistym problemie, a dokładniej rzecz ujmując, na jego pojedynczej warstwie, bez wnikania w ów wspomniany "paradoks nieskończoności". Są problemy, których rozwiązanie iteracyjne jest niemożliwe bądź też znacznie bardziej skomplikowane niż rekurencja. Jeden z takich problemów poznać przy analizie programu kalkulatora umieszczonego w tej lekcji - jest to analizator wyrażeń matematycznych (czy też ogólniej wyrażeń regularnych) pospolicie zwany **parserem**.

Czasem niska wydajność rekurencji daje się wyeliminować poprzez modyfikacje algorytmu. Rozważmy na przykład funkcję obliczającą ciąg Fibbonaciego (kolejny częsty przykład dla rekurencji). W ciągu takim kolejny wyraz, począwszy od trzeciego, jest sumą dwu poprzednich. Czyli możemy ciąg taki zapisać następująco:

$$f(n) = \begin{cases} 1 & \text{dla } n \leq 2 \\ f(n-2) + f(n-1) & \text{dla } n > 2 \end{cases}$$

Obserwując powyższą definicję, możemy napisać funkcję rekurencyjną dokładnie ją implementującą:

```
1. // funkcja obliczająca wartość ciągu Fibonacciego.
2. int fibonacci_1(int n)
3. {
4.     // zauważcie - że n w tej funkcji i n w funkcji main to inne zmienne
5.     if (n <= 2)
6.         return 1;
7.     else
8.         return fibonacci_1(n-1) + fibonacci_1(n-2);
9. }; // koniec funkcji fibonacci_1
```

Poniżej zaś zamieściliśmy cały program wykorzystujący tę funkcję:

```
1. #include <iostream>
2. #include <cstdlib>
3.
4. using namespace std;
5.
6. // funkcja obliczająca wartość ciągu Fibonacciego.
7. int fibonacci_1(int n)
8. {
9.     // zauważcie - że n w tej funkcji i n w funkcji main to inne zmienne
10.    if (n <= 2)
11.        return 1;
12.    else
13.        return fibonacci_1(n-1) + fibonacci_1(n-2);
14. }; // koniec funkcji fibonacci_1
15.
16. // funkcja main
17. int main(int argc, char *argv[])
18. {
19.     int n;
20.
21.     cout << "Program rekurencja_3 - oblicza wartosc ciagu Fibonacciego\n\n";
22.
23.     // wczytanie liczby, dla jakiej wartość ciągu ma zostać policzona:
24.     cout << "Podaj numer liczby Fibonnaciego: ";
25.     cin >> n;
26.
27.     // i od razu wyświetlmy wyniki:
28.     cout << "\nfibonacci(" << n << ") = " << fibonacci_1(n) << endl;
29.
30.     return 0;
31. }
```

Obliczenie liczby Fibonacciego przy wykorzystaniu powyższej procedury wymaga obliczenia dwu liczb poprzednich, obliczenie każdej z nich znowu wymaga policzenia dwu poprzednich itd. Zatem ilość wykonywanych obliczeń wzrasta wykładniczo wraz z numerem liczby (czas obliczeń podwaja się wraz z każdą kolejną liczbą. W literaturze fachowej określilibyśmy złożoność tego programu jako n^n). Dlatego też nie zalecamy testowania tego programu z numerami kolejnymi ciągu większymi od 40.

Zauważcie jednak, że do obliczenia aktualnej liczby wykorzystujemy dwie poprzednie, czyli możemy zapisać:

$$f(n) = f(n-2) + f(n-1) = f(n-2) + f(n-2) + f(n-3) = \dots$$

Innymi słowy, wystarczy, że obliczymy $f(n-2)$, a następnie dodamy do tego $f(n-3)$ aby otrzymać $f(n-1)$, którego nie trzeba liczyć rekurencyjnie. Poniżej znajdziecie więc zmodyfikowaną wersję procedury:

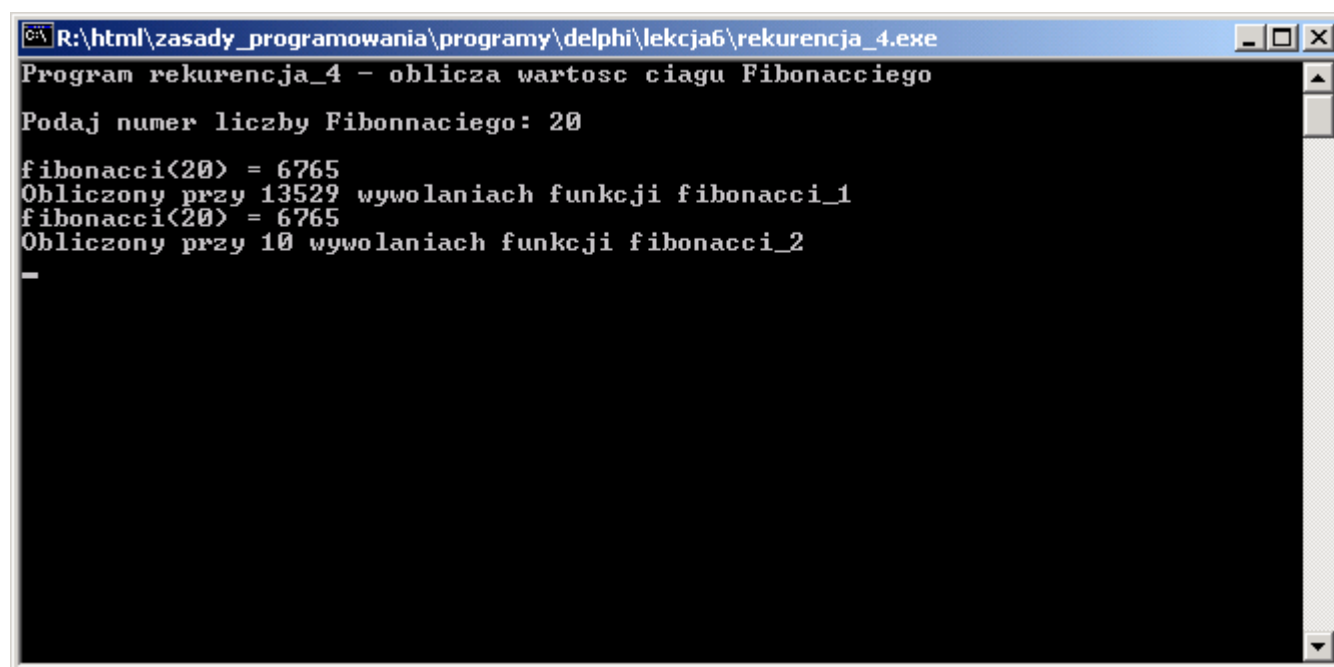
```
1. // Funkcja obliczająca wartość ciągu Fibonacciego - wersja optymalizowana.
2. // Tym razem wprowadziliśmy dwa parametry:
3. // n - numer liczby fibonacciego
4. // fn1 - wartość poprzedniego wyrazu
5. int fibonacc2(int n, int &fn1)
6. {
7.     // Pomocnicze zmienne lokalne
8.     int tf;
9.     // warunek zakończenia rekurencji
10.    if (n == 2)
11.    {
12.        fn1 = 1;
13.        return 1;
14.    }
15.    else if (n == 1)
16.    {
17.        fn1 = 0;
18.        return 1;
19.    }
20.    // jeśli nie został spełniony, obliczamy wartość funkcji
21.    else
22.    {
23.        tf = fibonacc2(n-2, fn1);
24.        fn1 += tf;
25.        return tf + fn1;
26.    };
27. }
```

Jak widzicie, funkcja ta tylko raz wywołuje samą siebie - czyli obliczenie następnej liczby wymaga jednego wywołania rekurencyjnego. Dlatego też ilość obliczeń rośnie liniowo wraz z numerem liczby. Jak istotna to różnica, możecie się przekonać uruchamiając poniższy program:

```
1. #include <iostream>
2. #include <cstdlib>
3.
4. using namespace std;
5.
6.
7. // pomocnicza zmienna globalna licznik
8. // posłuży nam do poznania liczby wywołań obu funkcji
9. static int licznik;
10.
11. // funkcja obliczająca wartość ciągu Fibonacciego.
12. int fibonacc1(int n)
13. {
14.     int rval;
15.     // zwiększenie licznika
16.     ++licznik;
17.
18.     // zauważcie - że n w tej funkcji i n w funkcji main to inne zmienne
19.     if (n <= 2)
20.         rval = 1;
21.     else
22.         rval = fibonacc1(n-1) + fibonacc1(n-2);
23.
24.     return rval;
25. }; // koniec funkcji fibonacc1
26.
27. // Funkcja obliczająca wartość ciągu Fibonacciego - wersja optymalizowana.
28. // Tym razem wprowadziliśmy dwa parametry:
29. // n - numer liczby fibonacciego
30. // fn1 - wartość poprzedniego wyrazu
31. int fibonacc2(int n, int &fn1)
```

```
32. {
33.     // zwiększenie licznika
34.     licznik++;
35.
36.     // Pomocnicze zmienne lokalne
37.     int rval, tf;
38.     // warunek zakończenia rekurencji
39.     if (n == 2)
40.     {
41.         fn1 = 1;
42.         return 1;
43.     }
44.     else if (n == 1)
45.     {
46.         fn1 = 0;
47.         return 1;
48.     }
49.     // jeśli nie został spełniony, obliczamy wartość funkcji
50.     else
51.     {
52.         tf = fibonacci_2(n-2, fn1);
53.         fn1 += tf;
54.         return tf + fn1;
55.     };
56. }
57.
58. // funkcja main
59. int main(int argc, char *argv[])
60. {
61.     int n, fn1;
62.
63.     cout << "Program rekurencja_4 - wartosc ciagu Fibonacciego po raz drugi\n\n";
64.
65.     // wczytanie liczby, dla jakiej wartość ciągu ma zostać policzona:
66.     cout << "Podaj numer liczby Fibonnacciego: ";
67.     cin >> n;
68.
69.     // zerujemy licznik
70.     licznik = 0;
71.     // liczymy i wyświetlamy wyniki:
72.     cout << "\nfibonacci(" << n << ") = " << fibonacci_1(n) << endl;
73.     cout << "  Obliczono przy " << licznik << " wywołaniach funkcji fibonacci_1\n";
74.
75.     // zerujemy licznik
76.     licznik = 0;
77.     // liczymy i wyświetlamy wyniki:
78.     cout << "\nfibonacci(" << n << ") = " << fibonacci_2(n, fn1) << endl;
79.     cout << "  Obliczono przy " << licznik << " wywołaniach funkcji fibonacci_2\n";
80.
81.     return 0;
82. }
```

Wyniki pracy programu:



```
R:\html\zasady_programowania\programy\delphi\lekcja6\rekurencja_4.exe
Program rekurencja_4 - oblicza wartosc ciagu Fibonacciego
Podaj numer liczby Fibonnaciego: 20
fibonacci(20) = 6765
Obliczony przy 13529 wywolaniach funkcji fibonacci_1
fibonacci(20) = 6765
Obliczony przy 10 wywolaniach funkcji fibonacci_2
-
```

Widać więc wyraźnie, że do obliczenia 20-ej z kolei liczby Fibonacciego przy wykorzystaniu wzoru definicyjnego musimy wykonać ponad 13500 wywołań funkcji, natomiast prosta przeróbka pozwoliła zmniejszyć tę ilość do 10 wywołań... Nawet zakładając, że funkcja `fibonacci_2` wykonuje się 4 razy wolniej, i tak zysk w szybkości jest olbrzymi: ponad 300 razy. I będzie on rósł wraz z obliczaną liczbą Fibonacciego (dla 40-ej z kolei liczby różnica wynosi już ponad 2,5 miliona razy) ...

Jak więc widzicie, czasem w miarę prosta modyfikacja algorytmu daje bardzo duże przyspieszenie wykonywania programu.

Problem skoczka szachowego

Rozważmy następujący problem znalezienia drogi skoczka szachowego. Otóż mamy szachownicę o wymiarach $n \times n$. Konik (skoczek) porusza się po polach w sposób zgodny za przyjętymi zasadami gry w szachy. Zadanie, jakie należy rozwiązać polega na wyszukaniu takiej drogi skoczka, która przechodzi przez wszystkie pola szachownicy, a konik „wskakuje” na każde pole dokładnie raz. Wynika z tego, że mając n^2 pól szachownicy, należy wykonać skoczkiem $n^2 - 1$ ruchów, gdyż pole, z którego startujemy, jest już „zaliczone”.

Rysunek przedstawia omawiany problem, przy tym ponumerowane pola szachownicy oznaczają 8 możliwych ruchów, jakie może wykonać skoczek z aktualnego położenia.

| | | | | |
|---|---|---|---|---|
| | 2 | | 3 | |
| 1 | | | | 4 |
| | |  | | |
| 8 | | | | 5 |
| | 7 | | 6 | |

Powyższy problem – jak też wiele innych zagadnień szachowych z racji tego, że w zasadzie każdy problem szachowy rozwiązuje się poprzez ciąg przejść (kroków) z jednego stanu (położenia figury na szachownicy) do drugiego - można rozwiązać stosując **algorytm z powrotami**. Omawiany tu przykład bazuje na znakomitym podręczniku Wirtha.

Ogólny zapis algorytmu dla problemu skoczka (jak również jest to dobry szablon do tworzenia algorytmów rozwiązujących inne problemy) może wyglądać następująco:

Procedura **szukajRuchu**(aktualne położenie skoczka)

```

1. repeat (kolejno dla każdego z 8 teoretycznych ruchów skoczka z aktualnego położenia) {
2.     if (ruch jest prawidłowy (tzn. prowadzi do nieodwiedzonego pola)) {
3.         zaktualizuj położenie skoczka
4.         zapamiętaj ten ruch
5.         if (istnieją pola nieodwiedzone) {
6.             szukajRuchu(aktualne położenie) // rekurencyjne wywołanie
7.             if (żaden ruch z tego miejsca nie daje rezultatu - ruch nieudany)
8.                 wykreśl ostatnio zapamiętany ruch
9.         }
10.    }
11. } until (ruch był udany lub przejrzano wszystkie 8 ruchów z poprzedniego położenia)

```

Należy w tym miejscu wyjaśnić, jaki cel mają niektóre instrukcje powyższego algorytmu. Przyjmujemy, że szachownicę odwzorowujemy na układ współrzędnych (x,y) – skoczek może znajdować się na jednym punkcie układu, a jego ruch oznacza dodanie/odjęcie wartości 1 lub 2 do wartości x, podobnie jest w przypadku współrzędnej y. Informację o tym, czy dane pole zostało odwiedzone, musimy również przechowywać – najwygodniej w tablicy o rozmiarach takich jak szachownica (nazwijmy ją T – jej rozmiar to $n \times n$). Potrzebne jest nam również zapamiętanie aktualnie wykonanego ruchu – a dokładnie informacji, który to był ruch od początku drogi i gdzie nas on zaprowadził.

Bardzo ważną rzeczą w tym algorytmie jest przekazywanie między rekurencyjnymi wywołaniami funkcji **szukajRuchu** wartości określającej, czy wywołana procedura dla danego numeru kroku znalazła prawidłowy ruch. Wywołania funkcji **szukajRuch** występują na różnych poziomach zagnieżdżenia – całkowite rozwiązanie problemu zostanie znalezione wtedy, gdy znajdziemy się na (n^2-1) zagnieżdżeniu, w którym będziemy szukali (n^2-1) ruchu skoczka. Jeśli okaże się, że istnieje taki krok (może istnieć nie więcej niż jeden poprawny krok), to będzie oznaczać, że znaleźliśmy trasę skoczka obiegającą całą szachownicę.

Odwołując się do naszych danych, musimy znaleźć 24 połączone ze sobą kroki. Jeśli nie jesteśmy w stanie znaleźć ani jednego poprawnego $(k+1)$ kroku (czyli wykonaliśmy do tej pory (k) kroków), o oznacza że musimy się wrócić do położenia po $(k-1)$ krokach w poszukiwaniu „lepszego” k -tego ruchu. Na tym właśnie polega zastosowanie algorytmu z powrotami – zagłębiamy się w ścieżce w sposób możliwie najdalszy, natomiast w razie potrzeby będziemy zmuszeni wykonać powrót (jeden albo więcej).

Skoro już wymieniliśmy tyle ważnych parametrów potrzebnych do rozwiązania tego zadania, to może w tym momencie spróbować przedstawić bardziej szczegółowy algorytm znajdowania drogi skoczka po szachownicy:

Procedura **szukajRuchu** (całkowite: **nrPolaNaTrasie**, współrzędna **x** położenia, współrzędna **y** położenia, poprawnośćKroku)

```
1. // nowy_x, nowy_y, poprawnośćKrokuLokalna - całkowite
2. poprawnośćKrokuLokalna = 0;
3. repeat (kolejno dla każdego z 8 teoretycznych ruchów skoczka z aktualnego położenia) {
4.     nowy_x = x + przemieszczenie w poziomie
5.     nowy_y = y + przemieszczenie w pionie
6.
7.     if (pole (nowy_x, nowy_y) należy do szachownicy oraz pole na szachownicy
8.     określone przez wartość (0 lub 1) T[nowy_x, nowy_y] było do tej
9.     pory nieodwiedzone {
10.         T[nowy_x, nowy_y] = nrPolaNaTrasie
11.         // zaznaczamy, w którym kroku skoczka odwiedziliśmy aktualne pole
12.         if (nrPolaNaTrasie < łączna liczba pól szachownicy) {
13.             szukajRuchu(nrPolaNaTrasie+1, nowy_x, nowy_y, poprawnośćKrokuLokalna) /
14.             / rekurencja
15.             if (poprawnośćKrokuLokalna jest równa 0)
16.                 T[nowy_x, nowy_y] = 0 // skasuj pole z aktualnej trasy
17.             }
18.         else
19.             poprawnośćKrokuLokalna = 1
20.     }
21. } until (poprawnośćKrokuLokalna jest równa 1 lub przejrzano wszystkie 8 ruchów
22.        z poprzedniego położenia)
23. poprawnośćKroku = poprawnośćKrokuLokalna
```

Problemem poruszania się konika po szachownicy zajmowało się w przeciągu ostatnich dwustu lat bardzo wielu wybitnych matematyków i naukowców. Dzisiaj wiemy na przykład, że w przypadku szachownicy o rozmiarze $m \times n$, gdzie $\min(m,n) \geq 5$, zawsze istnieje ścieżka skoczka przechodząca przez wszystkie pola dokładnie raz. Ponieważ w naszym przykładzie użyliśmy szachownicy 5×5 , więc implementacja powyższego pseudokodu dla takiego pola gra gwarantuje znalezienie poprawnej trasy dla konika. Poniżej prezentujemy jedno z rozwiązań problemu, gdy skoczek rozpoczyna swoją ścieżkę w punkcie środkowym szachownicy:

trasa skoczka przez wszystkie pola szachownicy



Istnieje wiele „ulepszeń” algorytmu znajdowania ścieżki skoczka – często stosowaną techniką jest wybieranie w danej chwili takich kroków prowadzących do kolejnych pól szachownicy, z których pozostało w aktualnym momencie najmniej ruchów wyjścia z nich (by uniknąć sytuacji, że dotrzemy do punktu, z którego nie będziemy mieli jak się dalej ruszyć naprzód – pozostanie nam tylko zastosowanie „powrotu”. A jest rzeczą oczywistą, że im więcej wykonanych powrotów, tym dłużej algorytm poszukuje „upragnionej” ścieżki.

W celu zapoznania się z animacjami przedstawiającymi poruszanie się skoczka po ścieżkach przechodzących przez całą szachownicę, odsyłamy na stronę Wikipedii:

http://en.wikipedia.org/wiki/Knight_tour

Problem plecakowy

Problem plecakowy (ang. *knapsack problem*) jest przykładem zagadnienia optymalizacyjnego, które może być rozwiązywane za pomocą wielu technik algorytmicznych (oczywiście z różną efektywnością). Niemniej jednak jest to doskonałe „modelowe” zadanie, dzięki któremu jesteśmy w stanie w przybliżony sposób zaprezentować sposoby działania kilku grup algorytmów. O trudności tego problemu niech świadczy fakt, że nie istnieje (przynajmniej nie został jeszcze odkryty) algorytm rozwiązujący **dyskretny** problem plecakowy, którego pesymistyczna złożoność obliczeniowa byłaby lepsza niż wykładnicza $O(2^n)$. Nie powinno nas to dziwić, gdyż problem plecakowy jest głównym przykładem algorytmu tzw. NP-trudnego – szczególnie zainteresowanych tematyką klasyfikacji problemów odsyłamy do naukowej literatury.

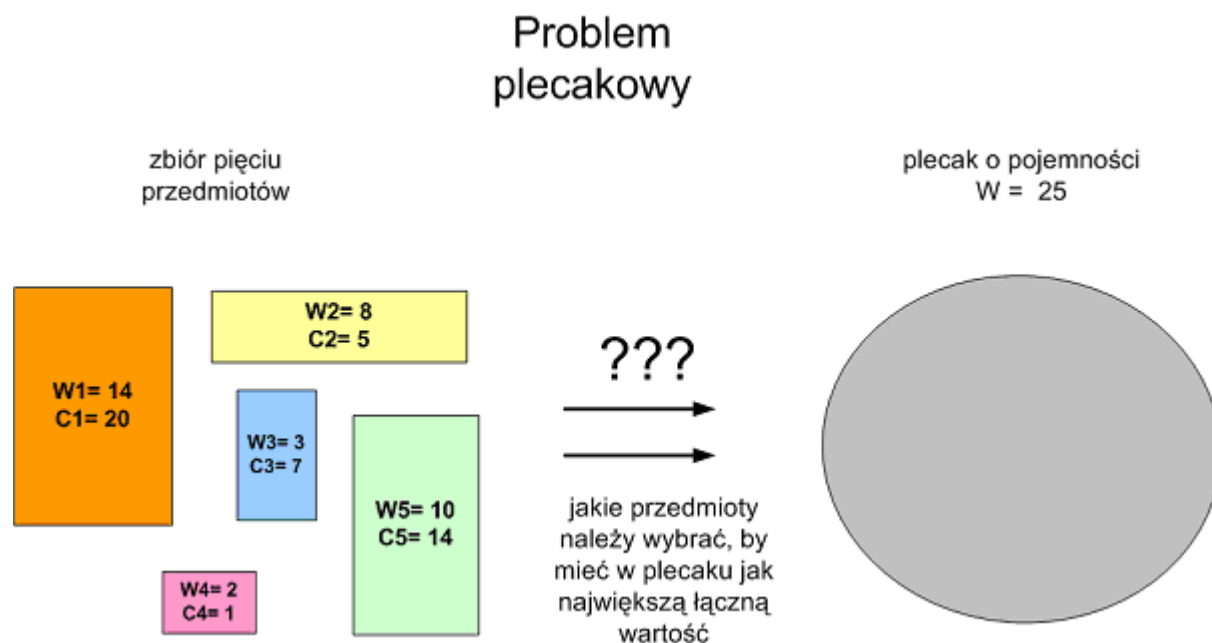
Główna zasada tego problemu brzmi: mamy pewien zbiór mniej lub bardziej cennych przedmiotów scharakteryzowanych za pomocą dwóch parametrów:

1. Wartość (cena) przedmiotu i-tego C_i
2. Masa przedmiotu i-tego W_i

Oprócz tych obiektów mamy pewien zbiorczy obiekt, zwyczajowo nazywany plecakiem, który z kolei posiada jeden parametr – całkowitą pojemność W (a ściślej rzecz biorąc powinna być to wytrzymałość na obciążenie). Zadanie jakie stoi przed osobą, która dokonuje załadunku przedmiotów do plecaka (może to być złodziej w muzeum sztuki albo turysta wyruszający na długą wyprawę), jest następujące:

- Należy dokonać takiego wyboru przedmiotów do plecaka, by ich sumaryczna wartość była możliwie maksymalna, zachowując przy tym ograniczenia dotyczące pojemności (W) plecaka.

Obrazowo nasz problem można przestawić w sposób następujący:



Innymi słowy, ze zbioru przedmiotów znajdujących się poza plecakiem należy wyłonić taki podzbiór, którego suma wag absolutnie nie może przekroczyć wartości W , a suma cen c_i będzie jednocześnie maksymalizowana (mamy do czynienia z zadaniem optymalizacji funkcji celu z ograniczeniami)

Klasycznym wariantem tego zagadnienia jest dyskretny problem plecakowy 0-1, w którym decyzja w rozpatrywaniu każdego przedmiotu jest binarna: albo pakujemy go do plecaka, albo nie. Inny wariant to, jak łatwo można się domyślić, ciągły problem plecakowy, który jest w prostszy w rozwiązywaniu i dlatego teraz przede wszystkim zajmijmy się problemem 0-1.

Zatem model problemu plecakowego wygląda następująco:

$$\max \sum_{i=1}^N c_i x_i$$

przy ograniczeniach :

$$1. \sum_{i=1}^N w_i x_i \leq W$$

$$2. x_i = \{0, 1\}$$

Ponieważ w lekcji 1 dokonaliśmy już przeglądu podziału algorytmów na grupy związane z metodami ich konstruowania, to teraz przyjrzymy się, jak dane metody „radzą sobie” z problemem *knapsack 0-1*. Dokonamy porównania efektywności:

- algorytmu „siłowego”
- algorytmu zachłannego
- programowania dynamicznego
- algorytmu z powrotami

1. Algorytm siłowy (bruteforce)

Jest to metodyka opierająca się na siłowym przeglądzie wszystkich możliwych rozwiązań – w tym wypadku algorytm:

- sprawdza wszystkie istniejące podzbiory zbioru n przedmiotów
- następnie odrzuca te spośród nich, które przekraczają sumaryczne ograniczenie wagowe
- finalnie algorytm porównuje ze sobą wszystkie pozostałe podzbiory i wybiera ten o największej całkowitej wartości

Jest matematycznie udowodnione, że liczba podzbiorów zbioru n -elementowego jest równa 2^n (każdy z n przedmiotów możemy określić na 2 sposoby – pakujemy lub nie, dlatego łącznie podzbiorów jest właśnie 2^n), więc tyle podzbiorów trzeba ze sobą porównać. A ponieważ dokonujemy wyszukania największego elementu metodą brutalną, złożoność obliczeniowa algorytmu siłowego jest niestety wykładnicza, co dyskwalifikuje go jako dobrą metodę rozwiązywania problemu przy dużych wartościach n . Jest to zdecydowanie najmniej optymalna metoda.

Rozwiązując przykład podany na obrazku algorytmem siłowym, musimy sprawdzić 2^5 możliwości, co przy tak małej wartości n nie jest żadnym problemem – ponieważ algorytm sprawdza wszystkie kombinacje ułożenia przedmiotów, więc nie powinno nas dziwić, że znajduje rozwiązanie optymalne. Tym rozwiązaniem jest włożenie do plecaka przedmiotów nr 1 i nr 5 – łączna wartość obu przedmiotów wynosi 34 przy ich sumarycznej wadze 24kg.

2. Programowanie dynamiczne

Jak zapewne dobrze pamiętacie, sposób formułowania metod opartych na programowaniu dynamicznym został omówiony we poprzedniej lekcji. W skrócie polegał on na znalezieniu rekurencyjnych zależności między rozwiązaniami podproblemów różnego poziomu, a następnie na rozwiązywaniu coraz większych zadań i wykorzystaniu ich wyników do otrzymania głównego rozwiązania. Ten rodzaj algorytmów bardzo dobrze się zachowuje przy rozwiązywaniu wielu problemów optymalizacyjnych. Nie inaczej jest w przypadku problemu plecakowego.

W przypadku rozwiązywania zagadnienia upakowania przedmiotów potrzebna nam będzie dodatkowa tablica na przechowywanie częściowych wyników podproblemów – określimy ją jako $P(i,j)$. W tablicy tej zapisywane będą mianowicie wartości optymalnych rozwiązań problemu upakowania, który będzie ograniczony do wyboru jedynie spośród pierwszych i przedmiotów, a pojemność plecaka będzie wynosić j . W trakcie rozwijania algorytmów zajmujących się upakowaniem plecaka opracowano wzór rekurencyjny na wartość $P(i,j)$, który w szandarowy sposób prezentuje możliwości programowania dynamicznego.

Otóż wartości tablicy P są obliczane zgodnie z poniższą zasadą:

$$P(i,j) = \begin{cases} \max(P(i-1,j), c_i + P(i-1, j-w_i)) & \text{dla } w_i \leq j \\ P(i-1,j) & \text{dla } w_i > j \end{cases}$$

Wy tłumaczenie tego wzoru rekurencyjnego jest stosunkowo przejrzyste. Jeśli chcemy dodać do rozwiązania i -ty element, który jest cięższy od dopuszczalnej pojemności plecaka, to ten ruch jest automatycznie kwalifikowany jako nieoptymalny i nadal rozwiązanie optymalne składa się z przedmiotów o numerach $1..i-1$. Natomiast jeżeli przedmiot i -ty, który zaczyna być w tej chwili brany pod uwagę, ma masę nie większą niż dopuszczalna „ładowność” plecaka, to porównujemy dwa przypadki:

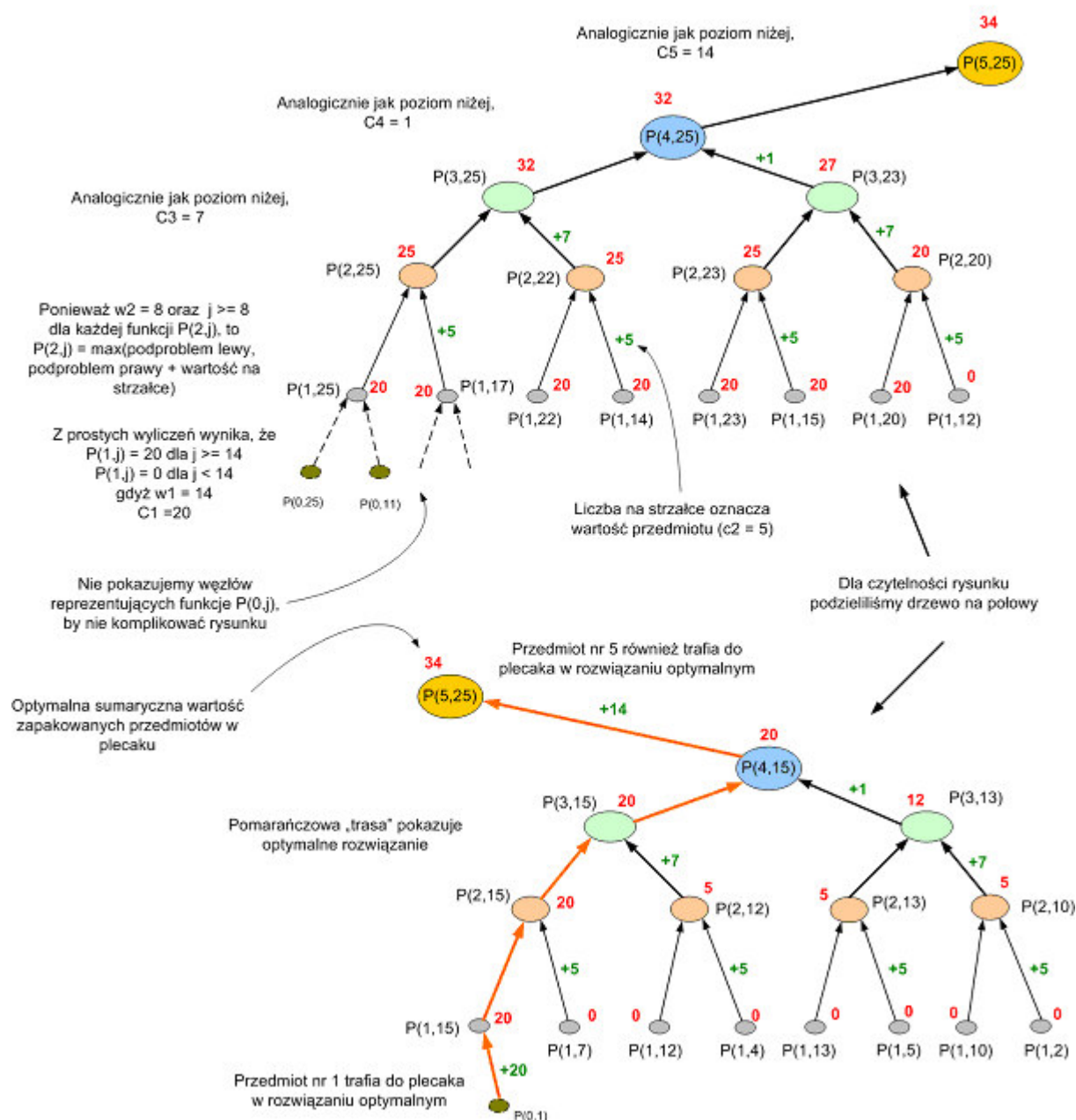
1. optymalny podzbiór elementów zawiera element i -ty
2. optymalny podzbiór elementów nie zawiera elementu i -tego

Pierwszy przypadek jest opisany przez drugi parametr funkcji \max w powyższym wzorze – do rozwiązania podproblemu z poprzedniego kroku dodajemy wartość ceny przedmiotu i -tego, a dostępną pojemność plecaka dla $(i-1)$ przedmiotów zmniejszamy o tyle, ile miejsca w plecaku zajmuje przedmiot i -ty (czyli w_i).

Z wiadomych względów $P(i,0)$ oraz $P(0,j)$ są równe 0. Aby znaleźć całościowe rozwiązanie musimy wyznaczyć wartość $P(n,W)$ – korzystając z rozwiązań najmniejszych podproblemów (właśnie tych o postaci $P(0,x)$ lub $P(y,0)$), przechodząc do coraz większych i ostatecznie do $P(n,W)$.

Skoro już omówiliśmy aspekt teoretyczny rozwiązywania problemu 0-1 *knapsack* przy zastosowaniu programowania dynamicznego, to możemy spróbować rozwiązać problem z rysunku znajdującego się na początku podrozdziału. Jak wiemy, należy odpowiednio przeliczyć wartości funkcji P – poniższe „drzewo rekurencji” pokazuje zależności między problemami i ich podproblemami – najpierw obliczamy wartości znajdujące się na samym dole drzewa, a następnie kroczymy z obliczeniami w górę, ostatnim kroku obliczając $P(5,25)$.

Problem plecakowy – programowanie dynamiczne



Jak widzimy, metoda programowania dynamicznego znalazła optymalne rozwiązanie naszego problemu – choć dla takiego rozmiaru na pewno szybszy jest algorytm zachłanny (który nie musi znajdować rozwiązania optymalnego – o czym dowiedzieć się już niebawem). Przebieg strzałek zaznaczonych na pomarańczowo pokazuje rozwiązanie optymalne – strzałki skierowane w lewo oznaczają wybór elementu do plecaka, natomiast strzałki skierowane w prawo określają rezygnację z wyboru.

Ponieważ liczba wpisów w tablicy P wynosi $n \cdot W$, więc złożoność obliczeniowa algorytmu jest $O(nW)$. Niestety (choć byłoby to bardzo zaskakujące, wręcz niemożliwe) nie jest to złożoność liniowa. Wartość W dla rzeczywistych problemów potrafi być niewspółmiernie kilka rzędów wielkości większa niż liczba plecaków n , co prowadzi do bardzo dużej złożoności omawianego algorytmu – w skrajnych przypadkach może dojść do sytuacji, że nawet algorytm „siłowy” jest bardziej wydajny niż zastosowanie w tym miejscu programowania dynamicznego – obliczanie współczynników gigantycznej tablicy jest bardzo czasochłonne. Sposób, w jaki rozwiązaliśmy nasze przykładowe zadanie, pokazuje technologię przyspieszenia działania algorytmu programowania dynamicznego. Otóż nie jest konieczne obliczanie całej tablicy P , lecz zaczynając od elementu $P(n, W)$ potrzebujemy obliczyć 1 wartość w ostatnim wierszu, 2 wartości w przedostatnim, 4 wartości w trzecim od końca itd. Możemy postawić zatem słuszny wniosek, że w najgorszym wypadku wyliczamy $2^n - 1$ wartości w tabeli P .

Zagadnienia dotyczące złożoności obliczeniowej tego algorytmu zostały w przystępny sposób omówione w książce R. Neapolitana, K. Naimpoura pt. „Podstawy algorytmów z przykładami w C++” – dla zachęty wspomnimy tylko, że jest tam zawarte wytłumaczenie,

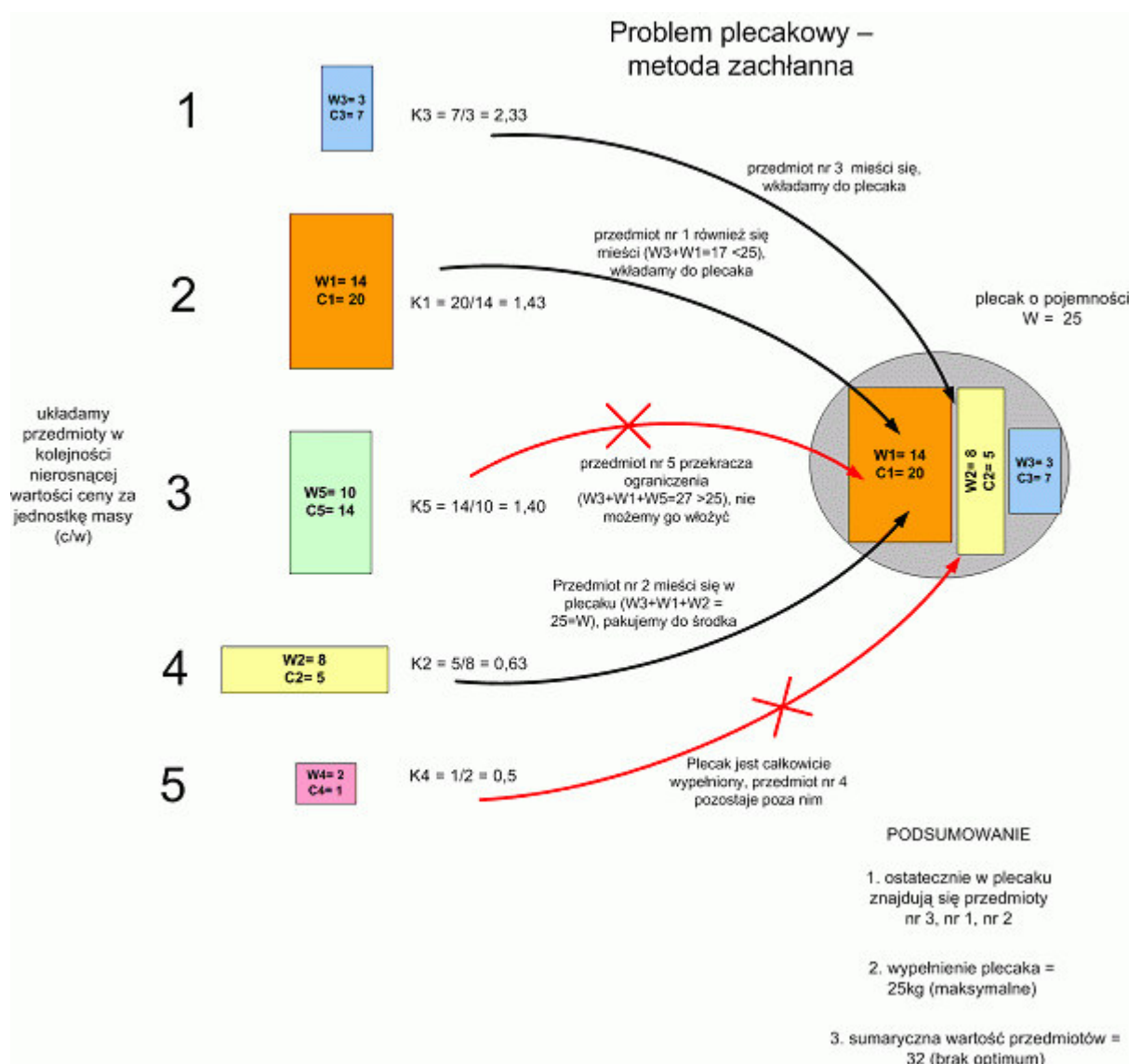
dlaczego algorytm programowania dynamicznego dla dyskretnego problemu plecakowego ma w najgorszym przypadku złożoność $O(\min(2^n, nW))$.

3. Algorytm zachłanny

Zachłanny algorytm, którego użycie w tym problemie wydaje się interesującą strategią, działa w kilku krokach:

- Najpierw szereguje przedmioty w kolejności nierosnącej, gdzie elementem porównawczym jest stosunek wartości przedmiotu do jego masy (sortowanie nierosnące względem wartości $k = \frac{c_i}{w_i}$ - cena za jednostkę masy)
- Następnie metoda zachłanna próbuje umieścić w plecaku wpierw przedmiot o największej wartości k , potem kolejne rzeczy stojące za nim na posortowanej liście przedmiotów
- Gdy okaże się, że pewien przedmiot nie może zostać zapakowany do plecaka ze względu na przekroczenie pojemności, to jest on pomijany i następuje próba umieszczenia kolejnych przedmiotów aż do całkowitego wyczerpania miejsca w plecaku lub do momentu, kiedy nie będzie możliwe jakiegokolwiek dodanie do plecaka przedmiotu jeszcze niezapakowanego – ze względu na przekroczenie ograniczenia.

Na poniższym schemacie przedstawimy rozwiązanie naszego przykładu metodą zachłanną:



Niestety metoda zachłanna nie znalazła w tym przypadku dla dyskretnego problemu plecakowego optymalnego rozwiązania. Nieoptymalne było w tym przypadku „zachłanne” umieszczenie teoretycznie najwartościowszego przedmiotu niebieskiego, co poskutkowało brakiem możliwości upakowania zielonego przedmiotu – który jest elementem składowym rozwiązania optymalnego.

Formalnie rozwiązanie problemu metodą zachłanną wymaga porównania rozwiązania otrzymanego w wyżej opisany sposób z rozwiązaniem, w którym w plecaku znajduje się element o największej wartości – czasem może zaistnieć taki układ danych, że to drugie rozwiązanie jest lepsze – choć również bez gwarancji bycia optymalnym.

Złożoność czasowa metody zachłannej zastosowanej do problemu *knapsack* zależy przede wszystkim od złożoności metody sortującej – najczęściej używane są w tym przypadku algorytmy porządkowania o złożoności $O(n \lg n)$ i zarazem złożoność całej metody zachłannej jest również tego rzędu.

4. Algorytm z powrotami

Problem plecakowy jest tak nurtującym zagadnieniem, że informatycy badali efektywność wielu grup algorytmów właśnie na problemie upakowania przedmiotów w plecaku. Było rzeczą oczywistą, że grupę metod, jakimi są algorytmy z powrotami, należało sprawdzić pod względem tego, jak sobie radzi z problemem plecakowym. Konstrukcja algorytmów z powrotami, jako przede wszystkim testowania przestrzeni stanów, dawała nadzieję, że efektywność ich wykorzystania w problemach *knapsack* będzie być może lepsza niż w przypadku innych metod.

W dużym skrócie, algorytm z powrotami użyty do znalezienia rozwiązania problemu plecakowego polega na systematycznym, rzetelnym podchodzeniu do prób dobierania kolejnych przedmiotów do plecaka aż do momentu, gdy stwierdzimy, że wszystkie możliwe wartościowe (i dopuszczalne) kombinacje dobierania plecaków już sprawdziliśmy i zarazem wyczerpaliśmy. Zasadniczą różnicą takiej wersji algorytmu z powrotami w porównaniu z podobnymi algorytmami, ale rozwiązującymi zadania nieoptymalizacyjne, jest to, że w żadnym momencie nie mamy pewności, że znalezione rozwiązanie jest tym optymalnym. Dopiero w momencie definitywnego zakończenia poszukiwań będziemy w stanie powiedzieć, który z węzłów w przestrzeni stanów (czyli jaka kombinacja plecaków) była najlepsza. Dlatego też jeżeli jesteśmy w punkcie, z którego musimy wykonać powrót, ale wartość zebranych do tej pory przedmiotów jest większa niż zapamiętane najbardziej optymalne rozwiązanie, to w tym momencie musimy uaktualnić tymczasową wartość optimum.

Algorytmy programowania dynamicznego, jak i metody z powrotami rozwiązują problem plecakowy w podobnej wielkości - oba typy należą do algorytmów rozwiązujących problem „0-1 *knapsack*” przy najbardziej pechowym układzie danych w czasie wykładniczym $O(2^n)$. Jednak z wielu przeprowadzonych testów efektywności algorytmów wynika, że przeważnie algorytm z powrotami jest nieco „sprawniejszy”.

Kalkulator

W rozdziale tym wprowadziliśmy dwa nowe pojęcia, które zostaną wykorzystane przy rozbudowie programu kalkulatora: podział programu na moduły oraz rekurencja.

Podział na moduły

Wydzielimy z dotychczasowego programu dwa moduły:

- Funkcje obliczeniowe. W module tym zamieścimy wszystkie funkcje obliczeniowe. Na początku zamieszczone zostaną definicje wszystkich typów zmiennych wykorzystywanych przez program. Następnie w części eksportowanej (w nagłówku pliku) zamieszczona zostanie tylko jedna funkcja - policz. Czyli interfejs modułu będzie wyglądał następująco:

```
1.  #ifndef kalk_6_biblH
2.  #define kalk_6_biblH
3.
4.  const int dl_wekt = 3;
5.
6.  // zdefiniujemy typ liczb zespolonych
7.  struct SZespolona
8.  {
9.      double re, im;
10. };
11.
12. /* Ponieważ każdy argument może być jednego z czterech typów, zdefiniujemy
13. rekord pamiętający argument. Będzie on zawierał tylko dwa pola: wektor
14. liczb zespolonych oraz typ argumentu. Następnie, w zależności od typu,
15. będziemy dany argument umieszczali w:
16.   liczba rzeczywista - części re pierwszego elementu wektora
17.   liczba zespolona - w pierwszym elemencie wektora
18.   wektor rzeczywisty - w częściach re wektora
19.   wektor zespolony - po prostu jako wektor.
```

```

20.
21. Nie jest to rozwiązanie optymalne. W profesjonalnym programie wykorzystalibyśmy
22. do tego celu konstrukcję którą nazywa się unią. Nie wprowadziliśmy jej tutaj
23. aby nie rozszerzać zakresu materiału przedstawianego Wam w trakcie tego kursu */
24. struct SArgument
25. {
26.     // Nowy dla was element - konstruktor.
27.     // Jest to cecha zapożyczona z programowania obiektowego,
28.     // my ją tylko wykorzystamy po to, by nasze argumenty można było
29.     // łatwo kopiować. Napiszemy dwie wersje: do inicjacji i kopiowania
30.     // struktury
31.
32.     // inicjacja
33.     SArgument()
34.     {
35.         // W tym przypadku wyzerujemy wszystkie wektory
36.         for (int i = 0; i < dl_wekt; i++)
37.             v[i].re = v[i].im = 0;
38.         // i przypiszemy domyślny typ
39.         typ = 0;
40.     }
41.     // kopiowanie
42.     SArgument(const SArgument& src)
43.     {
44.         // zrobimy to - co kompilator robi domyślnie dla typów
45.         // nietablicowych, lub co domyślnie robią inne języki programowania
46.         // skopiujemy zawartość tablicy
47.         for (int i = 0; i < dl_wekt; i++)
48.             v[i] = src.v[i];
49.         // no i skopiujemy jeszcze drugi element naszej struktury
50.         typ = src.typ;
51.     };
52.     // wektor liczb zespolonych
53.     SZespolona v[dl_wekt];
54.     // znaczniki typu argumentów: umówmy się, że:
55.     // 0 - oznacza liczbę,
56.     // 1 - liczbę zespoloną,
57.     // 2 - wektor rzeczywistych
58.     // 3 - wektor zespolonych
59.     int typ;
60. };
61.
62. bool policz(SArgument& w, SArgument x, SArgument y, char d);
63.
64. #endif

```

Całość modułu znajdziecie na końcu tego segmentu

- Interfejs użytkownika. Zawrzemy w nim wszystkie procedury służące do wprowadzania przez użytkownika danych do programu i wyświetlania wyników. Zauważcie, że tak na prawdę z poziomu programu głównego wywołujemy tylko i wyłącznie funkcję `czytaj_argument` i `pisz_wynik`. I tylko je musimy zamieścić w nagłówku modułu, cała reszta może pozostać ukryta.

```

1. #ifndef kalk_6_iuH
2. #define kalk_6_iuH
3.
4. // musimy powiadomić kompilator o typach które są wykorzystywane w
5. // funkcjach czytania argumentów i podawania wyniku
6. #include "kalk_6_bibl.h"
7.
8. bool czytaj_argument(SArgument& a);
9. void pisz_wynik(SArgument w, SArgument x, SArgument y, char d);
10.
11. #endif

```

Parser wyrażeń matematycznych

Aby wykorzystać rekurencję, postanowiliśmy wyposażyć nasz kalkulator w dodatkową możliwość - obliczanie wartości wyrażeń z uwzględnieniem wartości priorytetów. Aby to było możliwe, musimy napisać kod fachowo nazywany **parserem**.

Parserem będziemy nazywali jeden bądź kilka podprogramów które interpretują (przetwarzają) wprowadzony tekst

Do naszego parsera wykorzystamy w tym metodę fachowo nazywaną się **zejściem rekursywnym**. Inaczej mówiąc, będą to trzy funkcje wywołujące się rekursywnie (czyli będzie to rekursja pośrednia), z których każda będzie odpowiedzialna za wykonanie działań o tym samym priorytecie. W wyrażeniach arytmetycznych mamy trzy poziomy priorytetów operatorów: dodawanie i odejmowanie, mnożenie i dzielenie oraz nawiasy.

Spróbujemy w ten sposób oddać tok operacji, jakie wykonuje człowiek obliczając wartość wyrażenia arytmetycznego. Przy czym, aby nie komplikować kodu parsera (i bez tego jest wystarczająco skomplikowany), człowiek, którego naśladujemy, będzie liczył wyjątkowo "mechanicznie", tak, jakby nauczył się schematu zupełnie go nie rozumiejąc. Przykładowo, jeśli nakażemy mu obliczyć wartość wyrażenia $2*(3+4)$, będzie postępował następująco:

1. Wiemy, że na końcu należy wykonać dodawanie i odejmowanie. Aby to zrobić, potrzebujemy dwu składników. Więc - musimy je policzyć. Najpierw więc liczymy pierwszy składnik.
2. Aby policzyć pierwszy składnik, wykonujemy mnożenie i dzielenie na argumentach aż do napotkania znaku + lub -. Lecz znowu - aby wykonać mnożenie - dzielenie potrzebujemy wartości czynników, które mogą być liczbami, lecz mogą być również wyrażeniami w nawiasach. Więc znowu - obliczamy wartość pierwszego czynnika.
3. Teraz to już mamy tylko dwie możliwości - albo napotykamy liczbę - i wtedy ją sobie "zapamiętujemy", albo nawias - wtedy przechodzimy do punktu 1. To przejście do punktu 1 jest właśnie ową rekurencją pośrednią lub rekursją zstępującą. W przypadku naszego wyrażenia jest to liczba - 2. Zapamiętujemy więc ją jako czynnik i skreślamy z wyrażenia.
4. Wracamy zatem do punktu drugiego, znając już pierwszy czynnik. Skoro znamy czynnik, odczytujemy operator. Tutaj mamy trzy możliwości:
 1. Jest to mnożenie - więc musimy policzyć drugi czynnik
 2. Jest to dzielenie - także musimy policzyć drugi czynnik
 3. Jest to coś innego - w tym wypadku nic nie robimy i zapamiętujemy wartość pierwszego czynnika jako właśnie obliczony składnik

W naszym przypadku wystąpił operator mnożenia * (bo 2 już skreśliliśmy). Skreślamy więc gwiazdkę i wywołujemy procedurę jak w punkcie 3

5. Tym razem napotkaliśmy nawias otwierający. Skreślamy więc go i zaczynamy jakby "od początku" - liczymy wartość wyrażenia rozpoczynając od punktu 1.
Aby tutaj zbyt nie komplikować opisu, założmy, że wartość ta została obliczona - wynosi ona 7. Nasze wyrażenie przyjmuje po tym obliczeniu postać ... samego prawego nawiasu - reszta została skreślona w toku obliczeń podwyrażenia. Skoro jest to nawias zamykający - skreślamy go, a obliczoną wartość 7 przekazujemy jako drugi czynnik.
6. Wykonujemy mnożenie - $2*7 = 14$. Czternastkę przekazujemy jako pierwszy składnik.
7. Sprawdzamy jaki mamy teraz operator. Analogicznie jak przy mnożeniu i dzieleniu, mamy trzy możliwości:
 1. Jest to dodawanie - więc musimy policzyć drugi składnik
 2. Jest to odejmowanie - także musimy policzyć drugi składnik
 3. Jest to coś innego - w tym wypadku nic nie robimy. Wartość pierwszego składnika jest naszym wynikiem.

Nasze wyrażenie nie zawiera już żadnych znaków - nie ma operatora. Wniosek - składnik ten jest wynikiem obliczeń. Kończymy więc ...

Wygląda to na czarną magię ... Postarajcie się to jednak zrozumieć. Niżej znajdziecie animację powtarzającą dokładnie ten tok myślenia - może ona Wam pomoże. Lecz najpierw zapiszmy funkcje parsera:

Nasz parser będzie składał się z trzech funkcji, które będą odpowiadały tym trzem poziomom:

- `wyrazenie` będzie wykonywała dodawanie i odejmowanie na składnikach zwracanych poprzez funkcję `skladnik`.
- `skladnik` będzie wykonywała mnożenie i dzielenie na czynnikach zwracanych poprzez funkcję `czynnik`
- `czynnik` będzie zwracała wartość liczby bądź, jeśli napotka nawias - wywoła funkcję `wyrazenie` i koło się zamyka - mamy rekurencję pośrednią.

Nasz parser będzie wykorzystywał też swoje prywatne zmienne globalne:

- `blad_opis` - zmienna zawierająca opisy błędów występujących w wyrażeniu
- `blad_l` - liczba błędów które wystąpiły podczas interpretacji;
- `biezacy_symbol` - aktualny symbol
- `akt_wyrazenie` - przetwarzane wyrażenie
- `wartosc_liczby` - wartość liczbową symbolu (o ile jest on liczbą)

Dodatkowo potrzebna będzie jeszcze funkcja odpowiadająca za dekodowanie poszczególnych znaków składających się na nasze wyrażenie, które będzie przekazywane jako napis. My ją nazwalimy `daj_symbol`.

Nasz kalkulator będzie rozpoznawał następujące symbole:

- `sPLUS`, `sMINUS`, `sMNOZENIE`, `sDZIELENIE` odpowiadające operatorom matematycznym.
- `sLN`, `sPN` lewy i prawy nawias.
- `sLICZBA` liczba, jej wartość umieszczana jest w zmiennej `wartosc_liczby`.
- `sKONIEC` koniec wyrażenia

W każdej funkcji parsera zakłada się, że wywołano `daj_symbol`, czyli że w zmiennej `biezacy_symbol` zamieszczony jest typ następnego symbolu. Pozwala to parserowi widzieć kolejny symbol, a od każdej funkcji parsera wymaga, aby wczytała jeden symbol więcej niż to potrzebne aby wykonać swoje zadanie (ten jeden dodatkowy symbol zostaje na użytek następnej funkcji).

Funkcja wyrażenie w zasadzie nie wykonuje żadnego skomplikowanego algorytmu. Aby dodać do siebie dwie liczby, musimy jedynie je "poznać" (obliczyć), a następnie wykonać dodawanie / odejmowanie. Czyli algorytm może wyglądać następująco:

```
// Dodawanie i odejmowanie
double wyrażenie()
{
    // przydatne zmienne tymczasowe
    double lewa;
    // dodajemy / odejmujemy dwa składniki. Policzmy więc pierwszy z nich
    lewa = skladnik();
    // i wchodzimy w pętlę wykonującą wszystkie dodawania i odejmowania na
    // danym poziomie
    while (true)
    {
        // w zależności od bieżącego symbolu
        switch (biezacy_symbol)
        {
            // jeśli jest to dodawanie
            case sPLUS :
                // odczytujemy następny symbol
                daj_symbol();
                // wykonujemy dodawanie, obliczając "w locie" drugi składnik
                lewa += skladnik();
                break;

            // jeśli to odejmowanie
            case sMINUS :
                // odczytujemy następny symbol
                daj_symbol();
                // i wykonajmy odejmowanie
                lewa -= skladnik();
                break;

            // jeśli natomiast nie było to ani dodawanie ani odejmowanie, to nie mamy
            // już tu nic do roboty. Więc opuszczamy funkcję przypisując wynik
            default:
                return lewa;
        }
    }
};
```

Wartości dodawanych bądź odejmowanych liczb są zwracane poprzez funkcję **skladnik**, co odpowiada matematycznym priorytetom operatorów. Przecież przed dodawaniem zawsze musimy policzyć iloczyny/ilorazy, co właśnie robi funkcja `skladnik`.

Funkcja `skladnik` jest bardzo podobna do `wyrażenie`. Też najpierw wywołuje funkcję `czynnik` w celu wyznaczenia wartości argumentów, a następnie oblicza iloczyn bądź iloraz.

```
// Funkcja wykonuje mnożenie i dzielenie
double skladnik()
{
    //przydatne zmienne tymczasowe
    double lewa, dzielnik;
```

```

bool koniec;
// mnożymy przez siebie dwa czynniki. Więc odczytajmy najpierw pierwszy z
// nich
lewa = czynnik();

// następnie wchodzimy w pętlę, którą opuścimy dopiero po wykonaniu
// wszystkich mnożeń i dzielen na tym poziomie
do
{
    // w zależności od tego jaki jest bieżący symbol
    switch (biezacy_symbol)
    {

        // jeśli jest to mnożenie
        case sMNOZENIE :
            // odczytujemy następny symbol
            daj_symbol();
            // wykonujemy mnożenie
            lewa *= czynnik();
            break;

        // jeśli to dzielenie
        case sDZIELENIE :
            // odczytujemy następny symbol
            daj_symbol();
            // najpierw obliczamy dzielnik
            dzielnik = czynnik();
            // jeśli dzielnik = 0
            if (dzielnik == 0)
            {
                // no to mamy błąd. Powiadommy o tym użytkownika
                blad("Dzielenie przez 0");
                // i przypiszmy dzielnikowi wartość neutralną
                dzielnik = 1.0;
            }
            // wykonujemy dzielenie
            lewa /= dzielnik;
            break;

        // jeśli natomiast nie było to ani dzielenie ani mnożenie, to nie mamy
        // już tu nic do roboty. Więc opuszczamy funkcję przypisując wynik
        default:
            return lewa;
    };
}
while (true); // przykład pętli bez końca
};

```

Ostatnią istotną funkcją jest czynnik. Jej zadaniem jest zwrócić wartość liczby, jeśli bieżącym symbolem jest liczba, lub wywołać funkcję wyrażenie i zwrócić jego wartość, jeśli symbolem jest nawias otwierający:

```

// Obliczenie wartości czynnika
double czynnik()
{
    // zmienna pomocnicza
    double tmp;
    // na początek przypisujemy bezpieczną wartość czynnika
    double wynik = 1.0;
    // następnie w zależności od bieżącego symbolu
    switch (biezacy_symbol)
    {
        // jeśli jest to liczba
        case sLICZBA :
            // odczytujemy następny czynnik
            daj_symbol();
            // i zwracamy jej wartość
            wynik = wartosc_liczby;

```



```
break;

// jeśli jest to minus jednoargumentowy
case sMINUS :
    // odczytujemy następny czynnik
    daj_symbol();
    // i obliczamy wartość
    wynik = -czynnik();
break;

// jeśli jest to lewy nawias (otwierający)
case sLN :
    // odczytujemy następny czynnik (w ten sposób pozbyliśmy się nawiasu
    // otwierającego)
    daj_symbol();
    // obliczamy wartość wyrażenia w nawiasie
    tmp = wyrażenie();
    // jeśli po tym obliczeniu nie napotkamy nawiasu zamykającego
    if (biezacy_symbol != sPN)
    {
        // to musimy zgłosić błąd
        blad("Spodziewany prawy nawias");
    } else { // w przeciwnym wypadku
        // zwracamy wartość wyrażenia w nawiasie
        wynik = tmp;
        // i odczytujemy następny czynnik
        daj_symbol();
    };
break;

// jeśli to koniec wyrażenia, to zgłaszamy błąd !
case sKONIEC :
    blad("Nieoczekiwany koniec wyrażenia !");
break;

// jeśli nie napotkaliśmy żadnego z wymienionych symboli
// wyrażenie zawiera błąd składniowy
default:
    blad("Spodziewany czynnik");
};

return wynik;
};
```

Poniżej znajdziecie pełen kod procedur składających się na parser. Teraz natomiast przyjrzyjcie się jak on działa na omówionym wcześniej przykładzie $2*(3+4)$. Umieszczając kursor myszy nad instrukcją otrzymacie jej uproszczony kod z zaznaczonymi na czerwono wykonywanymi liniami. Po zakończeniu animacji możecie prześledzić całą historię obliczeń (wraz z kodem!) wykorzystując suwak, który pojawia się po prawej stronie.

→ *daj_symbol*

tekst = $2*(3 + 4)$
symbol =
wartosc =



Niewątpliwie nie będzie dla Was prostym zadaniem zrozumieć, jak działa taki parser. Lecz może potraktujecie to jako przygodę intelektualną? W podobny (choć znacznie bardziej skomplikowany) sposób działają prawdziwe interpretery i kompilatory w tym ... sam kompilator Pascala. Więc zahaczyliśmy o prawdziwą Informatykę ...

Nie obawiajcie się, nikt od Was nie będzie wymagał napisania podobnego kodu na egzaminie. Dla ambitnych mamy zadanie - spróbujcie zmodyfikować kod parsera tak, by nie korzystał ze zmiennych lokalnych *biezacy_symbol*, *akt_wyrazenie* i *wartosc_liczby* (zobaczcie kod modułu *kalk_parser* poniżej). Ten, kto potrafi to zrobić samodzielnie, może być prawie pewien 5 na egzaminie.

Program

Tak więc aktualnie program kalkulatora składa się z 7 plików:

kalk_6.cpp zawiera główny program. Zobaczcie, jaki czytelny się zrobił:

```

1.  #include <iostream>
2.  #include <cstdlib>
3.
4.  #include "kalk_6_bibl.h"
5.  #include "kalk_6_iu.h"
6.  #include "kalk_parser.h"
7.
8.
9.  //*****
10. //
11. //  Program główny
12. //
13. //*****
14.
15. int main(int argc, char* argv[])
16. {
17.     // argumenty
18.     SArgument x, y;
19.     // Zmienna przechowująca wynik obliczeń
20.     SArgument w;
21.     // Zmienna przechowująca wybrane działanie
22.     char dzialanie;
```

```
23. // Zmienne sterujące pracą programu
24. char wybor;
25. // Zmienna zawierająca wprowadzone wyrażenie
26. string wyr;
27.
28. cout << "Program Kalkulator v. 6\n";
29.
30. // Główna pętla programu. Będzie się wykonywała dopóki użytkownik nie wybierze
31. // opcji "Koniec"
32. do
33. {
34.     cout << endl;
35.
36.     do
37.     {
38.         cout << "Wybierz typ interfejsu [k-klasyczny, w-wyrażenie]: ";
39.         cin >> wybor;
40.         wybor = toupper(wybor);
41.     }
42.     while (wybor != 'K' && wybor != 'W');
43.
44.     if (wybor == 'W')
45.     {
46.
47.         // nowa wersja
48.
49.         // Zakładamy, że pętla ma zostać powtórzona w przypadku niepowodzenia w
50.         // obliczeniach
51.         wybor = 'n';
52.
53.         // Wczytajmy wyrażenie do obliczenia
54.         cout << "\nWprowadz wyrażenie zawierające liczby, operatory i nawiasy:\n";
55.         // opróżniamy bufor przed wczytywaniem wyrażenia
56.         cin.ignore();
57.         // wczytujemy całą linię ze spacjami
58.         getline(cin, wyr);
59.
60.         // policzmy jego wartość od razu dokonując kontroli parametrów
61.         if (policz_wyrazenie(wyr, w.v[1].re) != 0)
62.         {
63.             // wystąpiły błędy w wyrażeniu
64.             cout << "Błady w wyrażeniu:\n";
65.             cout << blad_opis << endl;
66.         }
67.         else
68.         {
69.             // nie było błędów
70.             cout << wyr << " = " << w.v[1].re << endl;
71.         };
72.
73.     }
74.     else
75.     {
76.
77.         // stara wersja
78.
79.         // Zakładamy, że pętla ma zostać powtórzona w przypadku niepowodzenia w
80.         // obliczeniach
81.         wybor = 'n';
82.
83.         // Do wprowadzenia argumentu wykorzystamy funkcję czytaj_argument
84.         if (!czytaj_argument(x))
85.             // jak wczytanie się nie powiodło - wracamy na początek pętli
86.             continue;
87.
88.         // Wczytanie wybranego działania. Kropka oznacza iloczyn skalarny
89.         cout << "Podaj działanie (+ - * / .): ";
90.         cin >> dzialanie;
```

```

91.
92.     // Wczytanie drugiego argumentu
93.     if (!czytaj_argument(y))
94.         // jak wczytanie się nie powiodło - wracamy na początek pętli
95.         continue;
96.
97.     cout << endl;
98.     // Wykonanie żądanej operacji - także wykorzystamy funkcję
99.     if (!policz(w, x, y, dzialanie))
100.        // jak obliczenia się nie powiodły - wracamy na początek pętli
101.        continue;
102.
103.        // wyświetlenie wyniku
104.        pisz_wynik(w,x,y,dzialanie);
105.    };
106.
107.    // zadajmy użytkownikowi pytanie, czy chce już zakończyć pracę z programem
108.    cout << "\nZakonczyz prace programu (n - nie, inny klawisz - tak) ? ";
109.
110.    // wczytajmy jego odpowiedź. Wykorzystamy ją do sprawdzenia warunku wyjścia
111.    // z pętli
112.    cin >> wybor;
113.
114.    // koniec pętli. Jeśli użytkownik wybrał 'n' - wracamy na jej początek, w
115.    // przeciwnym wypadku - opuszczamy pętlę i kończymy program
116.    }
117.    while (wybor == 'n' || wybor == 'N');
118.
119.    return 0;
120. }

```

Następny plik, kalk_6_bibl.h jest nagłówkiem modułu implementującego stare procedury obliczeniowe:

```

1.  #ifndef kalk_6_biblH
2.  #define kalk_6_biblH
3.
4.  const int dl_wekt = 3;
5.
6.  // zdefiniujemy typ liczb zespolonych
7.  struct SZespolona
8.  {
9.      double re, im;
10. };
11.
12. /* Ponieważ każdy argument może być jednego z czterech typów, zdefiniujemy
13. rekord pamiętający argument. Będzie on zawierał tylko dwa pola: wektor
14. liczb zespolonych oraz typ argumentu. Następnie, w zależności od typu,
15. będziemy dany argument umieszczali w:
16.   liczba rzeczywista - części re pierwszego elementu wektora
17.   liczba zespolona - w pierwszym elemencie wektora
18.   wektor rzeczywisty - w częściach re wektora
19.   wektor zespolony - po prostu jako wektor.
20.
21. Nie jest to rozwiązanie optymalne. W profesjonalnym programie wykorzystalibyśmy
22. do tego celu konstrukcję którą nazywa się unia. Nie wprowadziliśmy jej tutaj
23. aby nie rozszerzać zakresu materiału przedstawianego Wam w trakcie tego kursu */
24. struct SArgument
25. {
26.     // Nowy dla was element - konstruktor.
27.     // Jest to cecha zapożyczona z programowania obiektowego,
28.     // my ją tylko wykorzystamy po to, by nasze argumenty można było
29.     // łatwo kopiować. Napiszemy dwie wersje: do inicjacji i kopiowania
30.     // struktury
31.
32.     // inicjacja
33.     SArgument()
34.     {

```

```

35.     // W tym przypadku wyzerujemy wszystkie wektory
36.     for (int i = 0; i < dl_wekt; i++)
37.         v[i].re = v[i].im = 0;
38.     // i przypiszemy domyślny typ
39.     typ = 0;
40. }
41. // kopiowanie
42. SArgument(const SArgument& src)
43. {
44.     // zrobimy to - co kompilator robi domyślnie dla typów
45.     // nietablicowych, lub co domyślnie robią inne języki programowania
46.     // skopiujemy zawartość tablicy
47.     for (int i = 0; i < dl_wekt; i++)
48.         v[i] = src.v[i];
49.     // no i skopiujemy jeszcze drugi element naszej struktury
50.     typ = src.typ;
51. };
52. // wektor liczb zespolonych
53. SZespolona v[dl_wekt];
54. // znaczniki typu argumentów: umówmy się, że:
55.     // 0 - oznacza liczbę,
56.     // 1 - liczbę zespoloną,
57.     // 2 - wektor rzeczywistych
58.     // 3 - wektor zespolonych
59.     int typ;
60. };
61.
62. bool policz(SArgument& w, SArgument x, SArgument y, char d);
63.
64.
65. #endif

```

Plik z implementacją tego modułu znajdziecie poniżej:

```

1.  #include <iostream>
2.  #include <cstdlib>
3.
4.  #include "kalk_6_bibl.h"
5.
6.  using namespace std;
7.
8.  //*****
9.  //
10. //  Obliczenia
11. //
12. //*****
13.
14. // Pomocnicza funkcja zerująca argument
15. void zeruj_argument(SArgument &a)
16. {
17.     for (int i=0; i < dl_wekt; i++)
18.     {
19.         a.v[i].re = 0;
20.         a.v[i].im = 0;
21.     }
22. };
23.
24. bool dodaj_argumenty(SArgument &w, SArgument x, SArgument y)
25. {
26.     // możemy dodać do siebie dwie liczby (rzeczywiste lub urojone)
27.     if ((x.typ < 2) && (y.typ < 2))
28.     {
29.         w.v[0].re = x.v[0].re + y.v[0].re;
30.         w.v[0].im = x.v[0].im + y.v[0].im;
31.         // jeśli część urojona wyniku jest równa 0 - wynik jest rzeczywisty
32.         if (w.v[0].im == 0)
33.             w.typ = 0;

```

```

34.     // w przeciwnym wypadku jest urojony
35.     else
36.         w.typ = 1;
37. }
38. // Możemy też dodać dwa wektory
39. else
40. if ((x.typ >= 2) && (y.typ >= 2))
41. {
42.     for (int i=0; i<dl_wekt; i++)
43.     {
44.         w.v[i].re = x.v[i].re + y.v[i].re;
45.         w.v[i].im = x.v[i].im + y.v[i].im;
46.     }; // for
47.     // zakładamy, że wynik jest wektorem rzeczywistym
48.     w.typ = 2;
49.     // jeśli jednak występuje część urojona w którymkolwiek z pól - wynik
50.     // jest wektorem zespolonym
51.     for (int i=0; i<dl_wekt; i++)
52.         if (w.v[i].im != 0)
53.             w.typ = 3;
54. }
55. // Natomiast nie możemy dodać wektora do liczby
56. else
57. {
58.     cout << "Niewłaściwe argumenty !!!" << endl;
59.     // przerwanie funkcji i zwrócenie wartości wskazującej na błąd
60.     return false;
61. };
62.
63. return true;
64. };
65.
66. bool pomnoz_argumenty(SArgument &w, SArgument x, SArgument y)
67. {
68.     // Tutaj będzie trochę bardziej skomplikowanie:
69.     // Musimy rozpatrzyć wszystkie przypadki:
70.     switch (2 * (x.typ / 2) + y.typ / 2)
71.     {
72.     case 0 : // dwie liczby
73.         w.v[0].re = x.v[0].re * y.v[0].re - x.v[0].im * y.v[0].im;
74.         w.v[0].im = x.v[0].re * y.v[0].im + x.v[0].im * y.v[0].re;
75.         if (w.v[0].im == 0)
76.             w.typ = 0;
77.         else
78.             w.typ = 1;
79.         break;
80.     case 1 : // liczba wektor
81.         for (int i=0; i<dl_wekt; i++)
82.         {
83.             w.v[i].re = x.v[0].re * y.v[i].re - x.v[0].im * y.v[i].im;
84.             w.v[i].im = x.v[0].re * y.v[i].im + x.v[0].im * y.v[i].re;
85.         };
86.         // zakładamy, że wynik jest wektorem rzeczywistym
87.         w.typ = 2;
88.         // jeśli jednak występuje część urojona w którymkolwiek z pól - wynik
89.         // jest wektorem zespolonym
90.         for (int i=0; i<dl_wekt; i++)
91.             if (w.v[i].im != 0)
92.                 w.typ = 3;
93.         break;
94.     case 2 : // wektor liczba
95.         // skorzystajmy z przemienności mnożenia
96.         return pomnoz_argumenty(w, y, x);
97.         // zauważcie, że w tym przypadku break nie jest potrzebny
98.     case 3 : // dwa wektory
99.         w.v[0].re = 0;
100.        w.v[0].im = 0;

```



```

101.     for (int i=0; i < dl_wekt; i++)
102.     {
103.         w.v[0].re += x.v[i].re * y.v[i].re - x.v[i].im * y.v[i].im;
104.         w.v[0].im += x.v[i].re * y.v[i].im + x.v[i].im * y.v[i].re;
105.     };
106.     if (w.v[0].im == 0)
107.         w.typ = 0;
108.     else
109.         w.typ = 1;
110.     break;
111.     default : // nie może się zdarzyć
112.         cout << " ??? " << endl;
113.         return false;
114. };
115.
116.     return true;
117. }
118.
119. bool odejmij_argumenty(SArgument& w, SArgument x, SArgument y)
120. {
121.     SArgument t1, t2;
122.
123.     // odwrócimy znak drugiego argumentu, mnożąc go przez -1
124.     // najpierw skopiujemy drugi argument
125.     t1 = y;
126.     // następnie utworzymy nowy argument równy -1
127.     zeruj_argument(t2);
128.     t2.v[0].re = -1;
129.     t2.typ = 0;
130.     if (!pomnoz_argumenty(t1, t1, t2))
131.     {
132.         cout << "Problem z odejmowaniem!\n";
133.         return false;
134.     }
135.     // i wykonajmy dodawanie
136.     if (!dodaj_argumenty(w, x, t1))
137.     {
138.         cout << "Problem z odejmowaniem!\n";
139.         return false;
140.     }
141.
142.     // Jeśli dotarliśmy do tego punktu - to znaczy, że odnieśliśmy sukces
143.     return true;
144. }
145.
146. bool podziel_argumenty(SArgument &w, SArgument x, SArgument y)
147. {
148.     // Możemy podzielić liczbę przez liczbę lub wektor przez liczbę
149.     if (y.typ < 2)
150.     {
151.         // najpierw musimy sprawdzić, czy nie dzielimy przez 0
152.         if (y.v[0].re*y.v[0].re + y.v[0].im*y.v[0].im == 0)
153.         {
154.             // jeśli tak, to działania nie da się wykonać
155.             cout << "Nie można dzielić przez 0 ... \n";
156.             return false;
157.         }
158.         // jeśli nie, to dzielimy
159.         else
160.         {
161.             if (x.typ < 2)
162.             {
163.                 w.v[0].re = (x.v[0].re * y.v[0].re + x.v[0].im * y.v[0].im) /
164.                     (y.v[0].re * y.v[0].re + y.v[0].im * y.v[0].im);
165.                 w.v[0].im = (x.v[0].im * y.v[0].re - x.v[0].re * y.v[0].im) /
166.                     (y.v[0].re * y.v[0].re + y.v[0].im * y.v[0].im);
167.             }
168.             else

```

```

169.     {
170.         for (int i = 0; i < dl_wekt; i++)
171.         {
172.             w.v[i].re = (x.v[i].re * y.v[1].re + x.v[i].im * y.v[1].im) /
173.                         (y.v[1].re * y.v[1].re + y.v[1].im * y.v[1].im);
174.             w.v[i].im = (x.v[i].im * y.v[1].re - x.v[i].re * y.v[1].im) /
175.                         (y.v[1].re * y.v[1].re + y.v[1].im * y.v[1].im);
176.         }
177.     }
178. }
179. }
180. // natomiast nie możemy dzielić przez wektor
181. else
182. {
183.     cout << "Nie dzielimy przez wektor\n";
184.     return false;
185. };
186. w.typ = x.typ;
187.
188. return true;
189. }
190.
191.
192. bool policz(SArgument& w, SArgument x, SArgument y, char d)
193. {
194.     // Sprawdzenie działania, które wprowadził użytkownik
195.     switch (d)
196.     {
197.         // dodawanie:
198.         case '+':
199.             if (!dodaj_argumenty(w, x, y))
200.                 return false;
201.             break;
202.         // odejmowanie:
203.         case '-':
204.             if (!odejmuj_argumenty(w, x, y))
205.                 return false;
206.             break;
207.         // mnożenie (skalarne):
208.         case '*':
209.             if (!pomnoz_argumenty(w, x, y))
210.                 return false;
211.             break;
212.         // dzielenie:
213.         case '/':
214.             if (!podziel_argumenty(w, x, y))
215.                 return false;
216.             break;
217.         // nieznane działanie:
218.         default:
219.             // wyświetlenie informacji dla użytkownika
220.             cout << "Nieprawidłowy symbol operacji " << d << endl;
221.             return false;
222.     }
223.     // Jeśli tutaj dotarliśmy - to już sukces
224.     return true;
225. };

```

Następny moduł zawiera funkcje i procedury kontaktujące się z użytkownikiem. Na początek nagłówek tego modułu:

```

1.  #ifndef kalk_6_iuH
2.  #define kalk_6_iuH
3.
4.  // musimy powiadomić kompilator o typach które są wykorzystywane w
5.  // funkcjach czytania argumentów i podawania wyniku
6.  #include "kalk_6_bibl.h"

```

```

7.
8. bool czytaj_argument(SArgument& a);
9. void pisz_wynik(SArgument w, SArgument x, SArgument y, char d);
10.
11. #endif

```

implementacja tego modułu:

```

1. #include <iostream>
2. #include <cstdlib>
3.
4. #include "kalk_6_iu.h"
5.
6. using namespace std;
7.
8. //*****
9. //
10. //  Odczyt danych z klawiatury
11. //
12. //*****
13.
14. // Funkcja czytająca liczbę rzeczywistą z klawiatury
15. SZespolona czytaj_liczba()
16. {
17.     SZespolona l;
18.     // zerujemy część urojoną
19.     l.im = 0;
20.     // i wczytujemy rzeczywistą
21.     cout << "Podaj liczbę rzeczywistą: ";
22.     cin >> l.re;
23.     return l;
24. };
25.
26. // Funkcja czytająca liczbę zespoloną z klawiatury
27. SZespolona czytaj_zespolona()
28. {
29.     SZespolona l;
30.     cout << "Podaj liczbę zespoloną (re im): ";
31.     cin >> l.re >> l.im;
32.     return l;
33. };
34.
35. // Funkcja czytająca wektor liczb rzeczywistych z klawiatury
36. // ponieważ pośrednio mamy do czynienia z tablicą (jako częścią struktury)
37. // aby było możliwe bezpośrednie zwrócenie wartości przez funkcję konieczna
38. // była opisana wyżej modyfikacja struktury (dodanie konstruktora
39. // kopiującego
40. SArgument czytaj_wektor_rzeczywistych()
41. {
42.     SArgument a;
43.     cout << "Podaj trzy współrzędne wektora:\n";
44.     for (int i = 0; i < dl_wekt; i++)
45.         a.v[i] = czytaj_liczba();
46.     return a;
47. };
48.
49. // Procedura czytająca wektor liczb zespolonych z klawiatury
50. SArgument czytaj_wektor_zespolonych()
51. {
52.     SArgument a;
53.     cout << "Podaj trzy współrzędne wektora:\n";
54.     for (int i = 0; i < dl_wekt; i++)
55.         a.v[i] = czytaj_zespolona();
56.     return a;
57. };
58.
59. // Następnym krokiem będzie zdefiniowanie funkcji wczytującej dowolny dozwolony

```

```

60. // argument. Będzie ona zwracała wartość true w przypadku sukcesu, false -
61. // jeśli użytkownik poda niewłaściwą opcję
62. // Jeden parametr przekazywany jako zmienna:
63. // a - typu SArgument
64. // tu będzie umieszczany wczytany argument
65. bool czytaj_argument(SArgument& a)
66. {
67.     // zmienna pomocnicza typ
68.     char typ;
69.     cout << "Podaj typ argumentu:\n";
70.     cout << "(l - liczba, z - liczba zespol., v - wektor, w - wektor zespol.) : ";
71.     cin >> typ;
72.     switch (typ)
73.     {
74.         case 'l' : case 'L' : // wczytanie liczby
75.             a.v[0] = czytaj_liczba();
76.             a.typ = 0;
77.             break;
78.         case 'z' : case 'Z' : // wczytanie liczby zespolonej
79.             a.v[0] = czytaj_zespolona();
80.             a.typ = 1;
81.             break;
82.         case 'v' : case 'V' : // wczytanie wektora
83.             a = czytaj_wektor_rzeczywistych();
84.             a.typ = 2;
85.             break;
86.         case 'w' : case 'W' : // wczytanie wektora zespolonych
87.             a = czytaj_wektor_zespolonych();
88.             a.typ = 3;
89.             break;
90.         default:
91.             cout << "Wybrales niewlasciwa opcje. Powrot na poczatek petli\n";
92.             // zaznaczymy niepowodzenie operacji wczytywania
93.             return false;
94.     }
95.
96.     // skoro do tej pory nie opuściliśmy funkcji, to oznacza że wszystko jest ok
97.     return true;
98. }
99.
100. //*****
101. //
102. // Wyprowadzanie danych na ekran
103. //
104. //*****
105.
106. void pisz_liczba(double l)
107. {
108.     cout << l;
109. };
110.
111. void pisz_zespolona(SZespolona l)
112. {
113.     cout << l.re << "+" << l.im << "*i";
114. };
115.
116. void pisz_wektor_rzeczywistych(SArgument a)
117. {
118.     cout << "[";
119.     for (int i = 0; i < dl_wekt-1; i++)
120.         cout << a.v[i].re << ", ";
121.     cout << a.v[dl_wekt-1].re << "];";
122. };
123.
124. void pisz_wektor_zespolonych(SArgument a)
125. {
126.     cout << "[";
127.     for (int i = 0; i < dl_wekt-1; i++)

```

```

128.     cout << a.v[i].re << "+" << a.v[i].im << "*i, ";
129.     cout << a.v[dl_wekt-1].re << "+" << a.v[dl_wekt-1].im << "*i]";
130. };
131.
132. // Funkcja wypisująca argument na ekranie
133. void pisz_argument(SArgument a)
134. {
135.     switch (a.typ)
136.     {
137.         case 0 : pisz_liczba(a.v[0].re); break;
138.         case 1 : pisz_zespolona(a.v[0]); break;
139.         case 2 : pisz_wektor_rzeczywistych(a); break;
140.         case 3 : pisz_wektor_zespolonych(a); break;
141.         default: cout << " ??? "; // to się nigdy nie powinno zdarzyć ...
142.     }
143. };
144.
145. // Funkcja wypisująca całe wyrażenie
146. void pisz_wynik(SArgument w, SArgument x, SArgument y, char d)
147. {
148.     pisz_argument(x);
149.     cout << d;
150.     pisz_argument(y);
151.     cout << "=";
152.     pisz_argument(w);
153. }

```

Na koniec implementacja naszego rekurencyjnego parsera wyrażeń matematycznych. Umieściliśmy ją w plikach kalk_parser.h i kalk_parser.cpp

Plik nagłówkowy:

```

1.  #ifndef kalk_parserH
2.  #define kalk_parserH
3.
4.  #include <string>
5.
6.  using namespace std;
7.
8.  // deklaracja zmiennej zawierającej opisy błędów występujących w wyrażeniu
9.  extern string blad_opis;
10. // deklaracja zmiennej zawierającej liczbę błędów które wystąpiły podczas
11. // interpretacji
12. extern int blad_l;
13.
14. /* jedna jedyna funkcja eksportowana będzie obliczać wartość wyrażenia
15.    przekazywanego jej jako parametr w. Wynik będzie zamieszczony w parametrze
16.    v, natomiast funkcja będzie zwracać liczbę błędów napotkanych w wyrażeniu */
17. int policz_wyrazenie(string w, double& v);
18.
19. #endif

```

Plik implementacji:

```

1.  #pragma hdrstop
2.
3.  #include "kalk_parser.h"
4.
5.  #pragma package(smart_init)
6.
7.
8.  // zmienna zawierająca opisy błędów występujących w wyrażeniu
9.  string blad_opis;
10. // liczba błędów które wystąpiły podczas interpretacji
11. int blad_l;
12.

```

```
13.
14. // Zdefiniujemy sobie typ wyliczeniowy który będzie opisywał rozpoznawane
15. // przez nasz kalkulator symbole
16. enum TSymbol { sPLUS, sMINUS, sMNOZENIE, sDZIELENIE, sLN, sPN,
17.               sLICZBA, sKONIEC } ;
18.
19. // definicje zmiennych lokalnych, tzn. widocznych dla wszystkich porcedur w
20. // module natomiast niewidoczne dla reszty programu
21.
22. // Aktualny symbol
23. TSymbol biezacy_symbol;
24. // przetwarzane wyrażenie
25. string akt_wyrazenie;
26. // wartość liczbowa symbolu (o ile jest on liczbą)
27. double wartosc_liczby;
28.
29. /* Nasz parser będzie się składał z trzech funkcji dokonujących analizy
30.    składniowej, jednej dokonującej rozpoznania symboli oraz głównej funkcji
31.    będącej "oknem na świat" modułu. Dodatkowo zamieścimy procedurę
32.    zapamiętującą napotkane błędy */
33.
34. // funkcja zapewniająca obsługę błędów
35. void blad(string s)
36. {
37.     blad_l = blad_l + 1;
38.     if (blad_opis == "")
39.         blad_opis = s;
40.     else
41.         blad_opis += string(" | ") + s;
42. }
43.
44. /* funkcja dekodująca ciąg znaków na symbole. Pobiera ona symbol
45.    znajdujący się na początku łańcucha akt_wyrazenie, rozpoznaje go,
46.    a następnie umieszcza jego typ w zmiennej biezacy_symbol, wartość
47.    liczbową (o ile posiada taką) w zmiennej wartosc_liczby oraz
48.    __usuwa symbol z łańcucha__ */
49. void daj_symbol()
50. {
51.     // długość symbolu
52.     int usunac_znakow = 0;
53.     // zmienna pomocnicza
54.     int tmp;
55.
56.     /* najpierw usuwamy z poszátku wszystkie odstępy zwane niekiedy białymi
57.        spacjami.*/
58.
59.     while (isspace(akt_wyrazenie[usunac_znakow]) && usunac_znakow < akt_wyrazenie.size())
60.         usunac_znakow++;
61.
62.     akt_wyrazenie.erase(0, usunac_znakow);
63.
64.     // zakładamy że do usunięcia będzie jedynie jeden znak
65.     usunac_znakow = 1;
66.
67.     // jeśli wyrażenie się nam skończyło, bieżącym symbolem jest koniec
68.     if (akt_wyrazenie.empty())
69.     {
70.         biezacy_symbol = sKONIEC;
71.         // w przeciwnym wypadku
72.     } else {
73.
74.         // rozpoznanie na podstawie pierwszego znaku
75.         switch (akt_wyrazenie[0])
76.         {
77.             case '+': biezacy_symbol = sPLUS; break;
78.             case '-': biezacy_symbol = sMINUS; break;
79.             case '*': biezacy_symbol = sMNOZENIE; break;
```



```
80.         case '/' : biezacy_symbol = sDZIELENIE; break;
81.         case '(' : biezacy_symbol = sLN; break;
82.         case ')' : biezacy_symbol = sPN; break;
83.         // jeśli jest to cyfra
84.         case '0' : case '1' : case '2' : case '3' : case '4' :
85.         case '5' : case '6' : case '7' : case '8' : case '9' :
86.             /* konwertujemy napis na liczbę korzystając z funkcji bibliotecznej
87.             strtod. W przypadku jej wykorzystania, drugi argument funkcji będzie
88.             zawierał wskaźnik do znaku na którym funkcja skończyła przetwarzanie.
89.             dzięki temu dowiemy się jaka jest długość liczby */
90.             char *koniec;
91.             wartosc_liczby = strtod(akt_wyrazenie.c_str(), &koniec);
92.             biezacy_symbol = sLICZBA;
93.             /* w C i C++ wartości wskaźników można dodawać i odejmować, więc
94.             długość odczytanej liczby obliczamy następująco: */
95.             usunac_znakow = koniec - akt_wyrazenie.c_str();
96.             break;
97.             // nasz kalkulator nie rozpoznaje już żadnych innych symboli
98.         default :
99.             blad("Nierozpoznany symbol");
100.            biezacy_symbol = sKONIEC;
101.        }
102.
103.        // na koniec usunięcie rozpoznanego symbolu z wyrażenia
104.        akt_wyrazenie.erase(0, usunac_znakow);
105.    }; // koniec else
106.}; // koniec funkcji
107.
108./* ponieważ mamy do czynienia z rekursją pośrednią (funkcja wyrażenie wywołuje
109.pośrednio funkcję czynnik, która znowuż może wywołać funkcję wyrażenie)
110.musimy poinformować kompilator, że gdzieś dalej będzie zdefiniowana funkcja
111.wyrażenie */
112.double wyrażenie();
113.
114.// Obliczenie wartości czynnika
115.double czynnik()
116.{
117.    // zmienna pomocnicza
118.    double tmp;
119.    // na początek przypisujemy bezpieczną wartość czynnika
120.    double wynik = 1.0;
121.    // następnie w zależności od bieżącego symbolu
122.    switch (biezacy_symbol)
123.    {
124.        // jeśli jest to liczba
125.        case sLICZBA :
126.            // odczytujemy następny czynnik
127.            daj_symbol();
128.            // i zwracamy jej wartość
129.            wynik = wartosc_liczby;
130.            break;
131.
132.        // jeśli jest to minus jednoargumentowy
133.        case sMINUS :
134.            // odczytujemy następny czynnik
135.            daj_symbol();
136.            // i obliczamy wartość
137.            wynik = -czynnik();
138.            break;
139.
140.        // jeśli jest to lewy nawias (otwierający)
141.        case sLN :
142.            // odczytujemy następny czynnik (w ten sposób pozbyliśmy się nawiasu
143.            // otwierającego)
144.            daj_symbol();
145.            // obliczamy wartość wyrażenia w nawiasie
146.            tmp = wyrażenie();
147.            // jeśli po tym obliczeniu nie napotkamy nawiasu zamykającego
```

```
148.         if (biezacy_symbol != sPN)
149.         {
150.             // to musimy zgłosić błąd
151.             blad("Spodziewany prawy nawias");
152.         } else { // w przeciwnym wypadku
153.             // zwracamy wartość wyrażenia w nawiasie
154.             wynik = tmp;
155.             // i odczytujemy następny czynnik
156.             daj_symbol();
157.         };
158.         break;
159.
160.         // jeśli to koniec wyrażenia, to zgłaszamy błąd !
161.         case sKONIEC :
162.             blad("Nieoczekiwany koniec wyrażenia !");
163.         break;
164.
165.         // jeśli nie napotkaliśmy żadnego z wymienionych symboli
166.         // wyrażenie zawiera błąd składniowy
167.         default:
168.             blad("Spodziewany czynnik");
169.     };
170.
171.     return wynik;
172. };
173.
174. // Funkcja wykonuje mnożenie i dzielenie
175. double skladnik()
176. {
177.     //przydatne zmienne tymczasowe
178.     double lewa, dzielnik;
179.     bool koniec;
180.     // mnożymy przez siebie dwa czynniki. Więc odczytajmy najpierw pierwszy z
181.     // nich
182.     lewa = czynnik();
183.
184.     // następnie wchodzimy w pętlę, którą opuścimy dopiero po wykonaniu
185.     // wszystkich mnożeń i dzielen na tym poziomie
186.     do
187.     {
188.         // w zależności od tego jaki jest bieżący symbol
189.         switch (biezacy_symbol)
190.         {
191.
192.             // jesli jest to mnożenie
193.             case sMNOZENIE :
194.                 // odczytujemy następny symbol
195.                 daj_symbol();
196.                 // wykonujemy mnożenie
197.                 lewa *= czynnik();
198.             break;
199.
200.             // jeśli to dzielenie
201.             case sDZIELENIE :
202.                 // odczytujemy następny symbol
203.                 daj_symbol();
204.                 // najpierw obliczamy dzielnik
205.                 dzielnik = czynnik();
206.                 // jeśli dzielnik = 0
207.                 if (dzielnik == 0)
208.                 {
209.                     // no to mamy błąd. Powiadommy o tym użytkownika
210.                     blad("Dzielenie przez 0");
211.                     // i przypiszmy dzielnikowi wartość neutralną
212.                     dzielnik = 1.0;
213.                 }
214.                 // wykonujemy dzielenie
215.                 lewa /= dzielnik;
```

```
216.         break;
217.
218.         // jeśli natomiast nie było to ani dzielenie ani mnożenie, to nie mamy
219.         // już tu nic do roboty. Więc opuszczamy funkcję przypisując wynik
220.         default:
221.             return lewa;
222.     };
223. }
224. while (true); // przykład pętli bez końca
225.
226. };
227.
228. // Dodawanie i odejmowanie
229. double wyrażenie()
230. {
231.     // przydatne zmienne tymczasowe
232.     double lewa;
233.     // dodajemy / odejmujemy dwa składniki. Policzmy więc pierwszy z nich
234.     lewa = skladnik();
235.     // i wchodzimy w pętlę wykonującą wszystkie dodawania i odejmowania na
236.     // danym poziomie
237.     while (true)
238.     {
239.         // w zależności od bieżącego symbolu
240.         switch (biezacy_symbol)
241.         {
242.             // jeśli jest to dodawanie
243.             case sPLUS :
244.                 // odczytujemy następny symbol
245.                 daj_symbol();
246.                 // wykonujemy dodawanie, obliczając "w locie" drugi składnik
247.                 lewa += skladnik();
248.                 break;
249.
250.             // jeśli to odejmowanie
251.             case sMINUS :
252.                 // odczytujemy następny symbol
253.                 daj_symbol();
254.                 // i wykonujemy odejmowanie
255.                 lewa -= skladnik();
256.                 break;
257.             // jeśli natomiast nie było to ani dodawanie ani odejmowanie, to nie mamy
258.             // już tu nic do roboty. Więc opuszczamy funkcję przypisując wynik
259.             default:
260.                 return lewa;
261.         }
262.     };
263. }
264.
265.
266. // główna funkcja modułu
267. int policz_wyrażenie(string w, double &v)
268. {
269.     // zainicjujemy najpierw obsługę błędów w naszym module
270.     blad_l = 0;
271.     blad_opis = "";
272.
273.     // następnie przepisujemy do zmiennej lokalnej obliczane wyrażenie
274.     akt_wyrażenie = w;
275.     // zainicjujemy parser (pobierając pierwszy symbol)
276.     daj_symbol();
277.     // wykonanie obliczeń
278.     v = wyrażenie();
279.
280.     if (biezacy_symbol != sKONIEC)
281.         blad("Nierozpoznane znaki na koncu wyrażenia !");
282.
283.     // zwracamy liczbę błędów
```

```
284.     return blad_l;  
285. };
```

Zadania

Zadania do lekcji 2:

Napisać programy, które realizują następujące zadania dla komputera:

1. Napisać funkcję rekurencyjną, która w jakiejś tablicy całkowitej [n] (n-stała) odwraca kolejność elementów zawartych pomiędzy jakimś indeksem lewym a jakimś indeksem prawym w taki sposób, że najpierw zamienia miejscami dwa skrajne elementy (znajdujące się pod indeksem lewym i prawym), a następnie odwraca kolejność elementów pozostałych, czyli wywołuje samą siebie dla obszaru pomniejszonego o te dwa skrajne elementy. W treści funkcji należy odpowiednio sformułować warunek końca. Funkcję wykorzystać w programie, który wczytuje tablice całkowite A[n] i B[n], po czym odwraca kolejność wszystkich elementów w tablicy A (czyli zamienia 1 -szy z n-tym, drugi z n-1-szym itd.), zaś w tablicy B - odwraca kolejność elementów od 3 do n.

```
1.  #include <iostream>  
2.  #include <cstdlib>  
3.  #include <cmath>  
4.  
5.  using namespace std;  
6.  
7.  const int n = 10;  
8.  
9.  void zamien(int *t, int p, int k)  
10. {  
11.     // koniec rekurencji  
12.     if (p >= k)  
13.         return;  
14.  
15.     // zamiana  
16.     int tmp = t[p];  
17.     t[p] = t[k];  
18.     t[k] = tmp;  
19.  
20.     // wywołanie rekurencyjne  
21.     zamien(t, p+1, k-1);  
22. }  
23.  
24. int main(int argc, char* argv[])  
25. {  
26.     cout << "Zadanie 4: wprowadź tablice A i B" << endl;  
27.     int A[n], B[n];  
28.  
29.     for (int i = 0; i < n; i++)  
30.         cin >> A[i];  
31.     for (int i = 0; i < n; i++)  
32.         cin >> B[i];  
33.  
34.     cout << "Przed:\n";  
35.     for (int i = 0; i < n; i++)  
36.         cout << A[i] << " ";  
37.     cout << endl;  
38.     for (int i = 0; i < n; i++)  
39.         cout << B[i] << " ";  
40.     cout << endl;  
41.  
42.     zamien(A, 0, n-1);  
43.     zamien(B, 2, n-1);  
44.  
45.     cout << "Po:\n";  
46.     for (int i = 0; i < n; i++)  
47.         cout << A[i] << " ";
```

```
48.     cout << endl;
49.     for (int i = 0; i < n; i++)
50.         cout << B[i] << " ";
51.     cout << endl;
52.
53.     system("pause");
54.     return 0;
55. }
```