Podstawy Programowania 2

Rekurencja, metoda dziel i zwyciężaj

Arkadiusz Chrobot

Zakład Informatyki

9 marca 2020

Plan

Rekurencja

Wstęp

Na poprzednim wykładzie podaliśmy definicję stosu i omawialiśmy jego implementację w postaci dynamicznej struktury danych. To pojęcie pojawiło się również kiedy zapoznawaliśmy się z funkcjami. Stos w ich kontekście jest fragmentem obszaru pamięci przydzielonego wykonującemu się programowi, gdzie umieszczane są informacje niezbędne do działania funkcji. Te informacje zorganizowane są w postaci *ramki stosu*, nazywanej także *rekordem aktywacyjnym*. W ramce stosu znajduje się miejsce między innymi na adres powrotu, zmienne lokalne i parametry. Zastosowanie stosu do obsługi funkcji ma ciekawe konsekwencje. Jedną z nich jest możliwość tworzenia *podprogramów rekurencyjnych*, które w przypadku języka C są *funkcjami rekurencyjnymi*. Funkcja rekurencyjna, to taka, która na pewnym etapie wykonania wywołuje samą siebie. Tego typu podprogramy realizują *algorytmy rekurencyjne*, czyli takie, które dzielą rozwiązywany problem na problemy mniejszego rozmiaru, aż zredukują je do przypadków, które będą mogły rozwiązać. Następnie uzyskane wyniki łączą, aby rozwiązać problem wyjściowy.

Rekurencja

Wstęp

Przyjrzyjmy się działaniu rekurencji i jej związkom ze stosem, nazywanym dalej stosem sprzętowym, na przykładzie funkcji rekurencyjnej obliczającej silnię. Przypomnijmy, że dla n będącej liczbą naturalną silnia jest zdefiniowana następująco:

$$n! = \begin{cases} 1 & \text{jeśli } n = 0 \text{ lub } n = 1 \\ (n-1)! \cdot n & \text{dla } n > 1 \end{cases}$$

Jak łatwo zauważyć definicja ta jest rekurencyjna - dla $n > 1$ silnia jest wyliczana na podstawie znajomości wyniku tego działania dla argumentu mniejszego o 1. W zapisie definicji możemy również dostrzec przypadki brzegowe, czyli takie dla których możemy od razu ustalić wynik. To te, dla $n = 0$ lub $n = 1$. Następny slajd zawiera definicję funkcji, która jest zaimplementowana na podstawie podanej definicji silni.

Rekurencja

Wstęp - silnia

```
1  unsigned long int factorial(unsigned char n)
2  {
3      if(n==0||n==1)
4          return 1;
5      else
6          return factorial(n-1)*n;
7  }
```

Rekurencja

Wstęp - silnia

Funkcja ta oblicza prawidłowo wartość silni dla parametru $n < 65$. Powyżej tej wartości następuje przekroczenie górnego zakresu typu `unsigned long int`. Prześledźmy jak będzie działać taka funkcja dla `n=3`. W momencie jej wywołania na stosie sprzętowym powstanie dla jej instancji ramka. Funkcja sprawdzi warunek w trzecim wierszu. Okaże się on być fałszywy, zatem spróbuje ona ustalić wartość wyrażenia `factorial(2)*3`. W tym celu wywoła samą siebie, ale tym razem dla `n=2`. Tak jak poprzednio dla tego wywołania zostanie utworzona rama stosu i tak jak poprzednio funkcja sprawdzi warunek w trzecim wierszu, który i tym razem będzie fałszywy. To sprawi, że podejmie ona próbę określenia wartości wyrażenia `factorial(1)*2`. Aby to zrobić ponownie wywoła samą siebie dla `n=1`. Tak jak poprzednio ramka stosu zostanie utworzona dla tego wywołania, ale tym razem warunek w wierszu trzecim będzie prawdziwy i to wywołanie funkcji zakończy się zwracając wartość `1`.

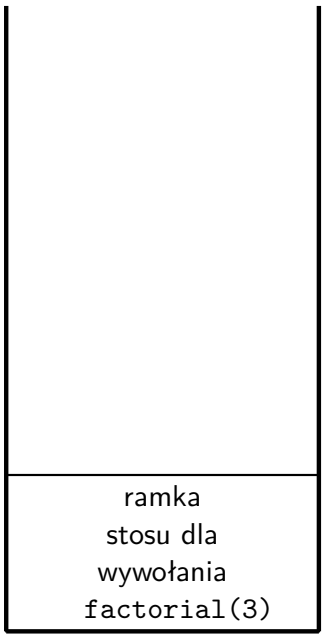
Rekurencja

Wstęp - silnia

Po tym jak funkcja zakończy działanie wyznaczając wartość silni dla `n=1` sterowanie wróci do jej wywołania dla `n=2`. Tym razem jednak funkcja będzie mogła ustalić wartość wyrażenia `factorial(1)*2`, podstawiając wartość 1 za pierwszy argument tego mnożenia. Po obliczeniu wyniku (2) ta instancja funkcji zwraca go i kończy się. Sterowanie wraca do wywołania funkcji, które próbowało obliczyć wynik silni dla `n=3`. Teraz ono potrafi ustalić wynik wyrażenia `factorial(2)*3`, podstawiając za pierwszy argument w tym działaniu wartość (2). Funkcja ostatecznie kończy swe działanie i zwraca wynik 6, czyli wartość silni dla `n=3`. Przy każdym zakończeniu pojedynczej instancji funkcji ze stosu sprzętowego zdejmowana jest jedna ramka. Zawartość stosu ilustrują poglądowo kolejne slajdy.

Rekurencja

Wstęp - silnia



Wywołanie funkcji dla `n`=3

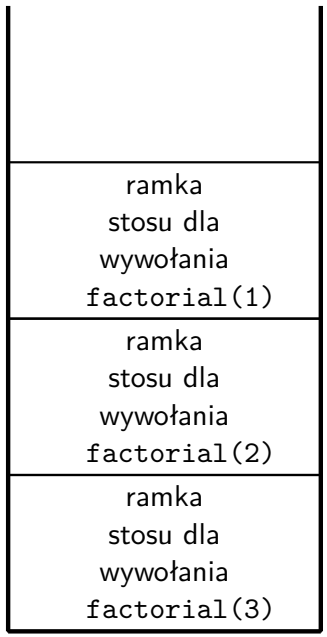
Rekurencja

Wstęp - silnia

Wywołanie rekurencyjne `factorial(2)*3`

Rekurencja

Wstęp - silnia



Wywołanie rekurencyjne `factorial(1)*2*3`

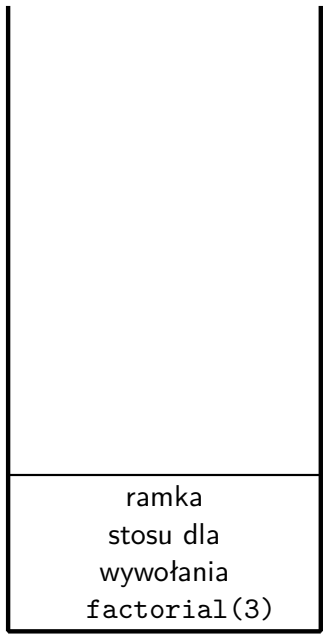
Rekurencja

Wstęp - silnia

Wywołanie `factorial(1)` zwraca wynik i kończy się

Rekurencja

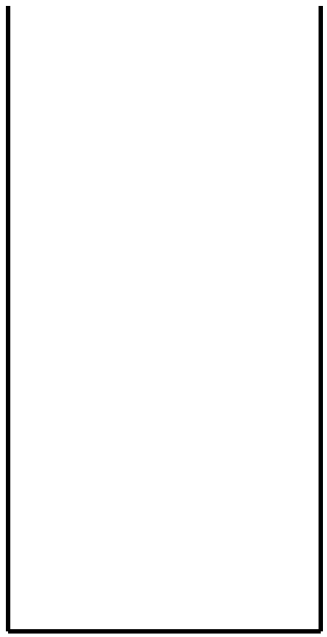
Wstęp - silnia



Po zakończeniu `factorial(2)`

Rekurencja

Wstęp - silnia



Po zakończeniu `factorial(3)`

Rekurencja

Wstęp - silnia

Zazwyczaj działanie funkcji rekurencyjnych ilustruje się nie poprzez rysowanie stanu stosu, lecz za pomocą tzw. *drzew wywołań* (ang. *call tree*) nazywanych też *drzewami rekurencji* lub *drzewami rekursji*. W przypadku funkcji `factorial()` to drzewo dla `n=3` będzie bardzo proste (matematycy by je określili jako zdegenerowane).

Rekurencja

Wstęp - silnia

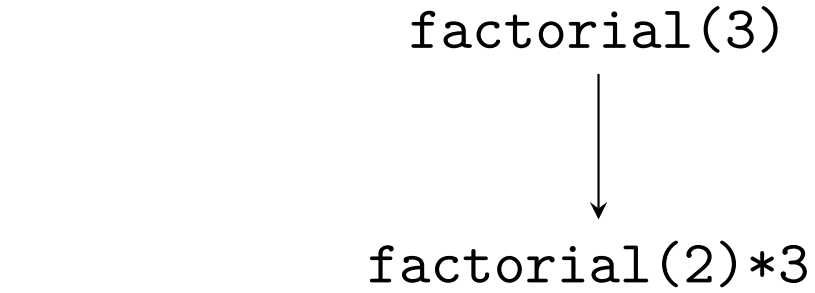
Zazwyczaj działanie funkcji rekurencyjnych ilustruje się nie poprzez rysowanie stanu stosu, lecz za pomocą tzw. *drzew wywołań* (ang. *call tree*) nazywanych też *drzewami rekurencji* lub *drzewami rekursji*. W przypadku funkcji `factorial()` to drzewo dla `n=3` będzie bardzo proste (matematycy by je określili jako zdegenerowane).

`factorial(3)`

Rekurencja

Wstęp - silnia

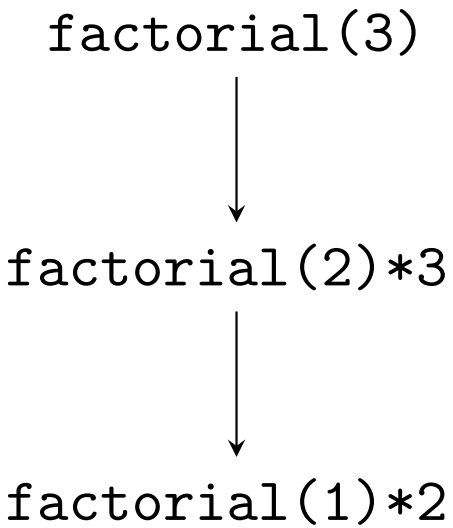
Zazwyczaj działanie funkcji rekurencyjnych ilustruje się nie poprzez rysowanie stanu stosu, lecz za pomocą tzw. *drzew wywołań* (ang. *call tree*) nazywanych też *drzewami rekurencji* lub *drzewami rekursji*. W przypadku funkcji `factorial()` to drzewo dla `n=3` będzie bardzo proste (matematycy by je określili jako zdegenerowane).



Rekurencja

Wstęp - silnia

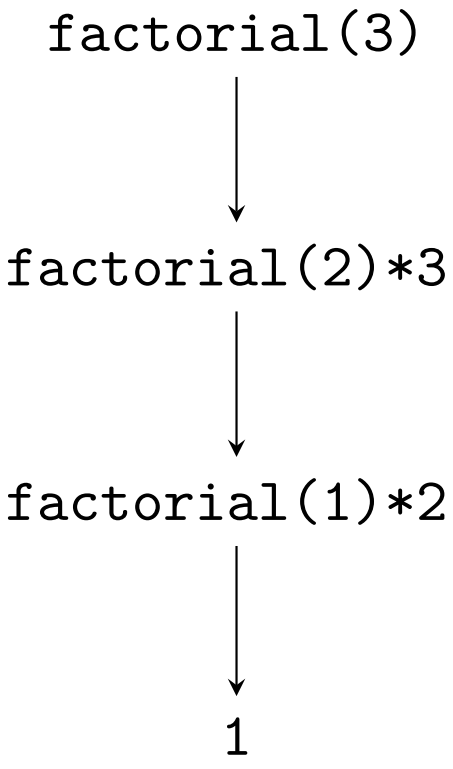
Zazwyczaj działanie funkcji rekurencyjnych ilustruje się nie poprzez rysowanie stanu stosu, lecz za pomocą tzw. *drzew wywołań* (ang. *call tree*) nazywanych też *drzewami rekurencji* lub *drzewami rekursji*. W przypadku funkcji `factorial()` to drzewo dla `n=3` będzie bardzo proste (matematycy by je określili jako zdegenerowane).



Rekurencja

Wstęp - silnia

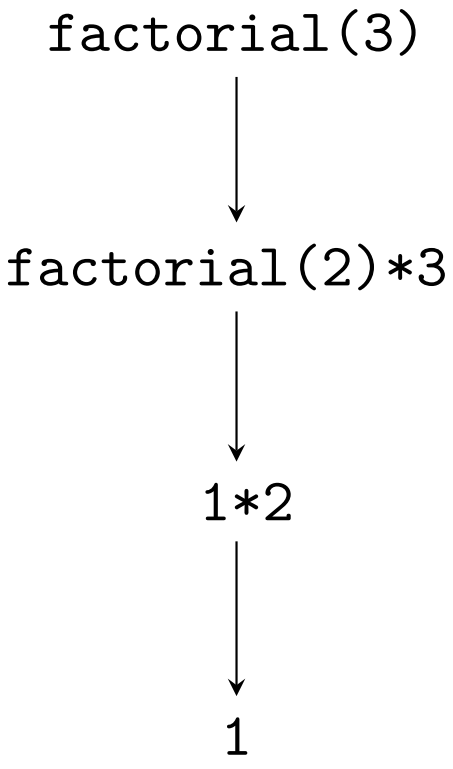
Zazwyczaj działanie funkcji rekurencyjnych ilustruje się nie poprzez rysowanie stanu stosu, lecz za pomocą tzw. *drzew wywołań* (ang. *call tree*) nazywanych też *drzewami rekurencji* lub *drzewami rekursji*. W przypadku funkcji `factorial()` to drzewo dla `n=3` będzie bardzo proste (matematycy by je określili jako zdegenerowane).



Rekurencja

Wstęp - silnia

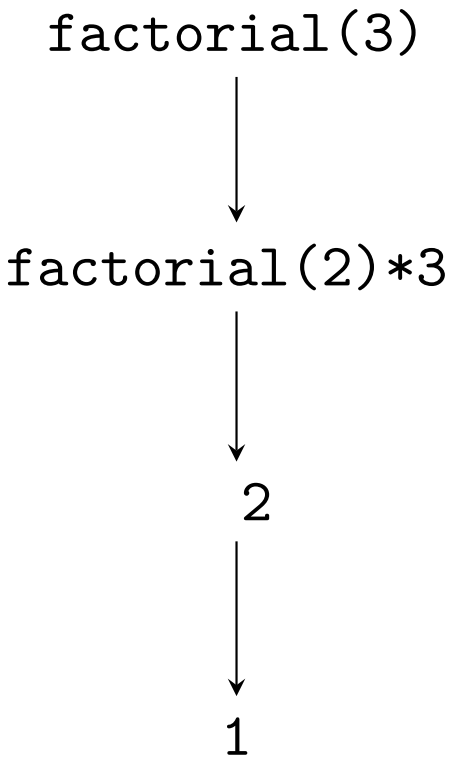
Zazwyczaj działanie funkcji rekurencyjnych ilustruje się nie poprzez rysowanie stanu stosu, lecz za pomocą tzw. *drzew wywołań* (ang. *call tree*) nazywanych też *drzewami rekurencji* lub *drzewami rekursji*. W przypadku funkcji `factorial()` to drzewo dla `n=3` będzie bardzo proste (matematycy by je określili jako zdegenerowane).



Rekurencja

Wstęp - silnia

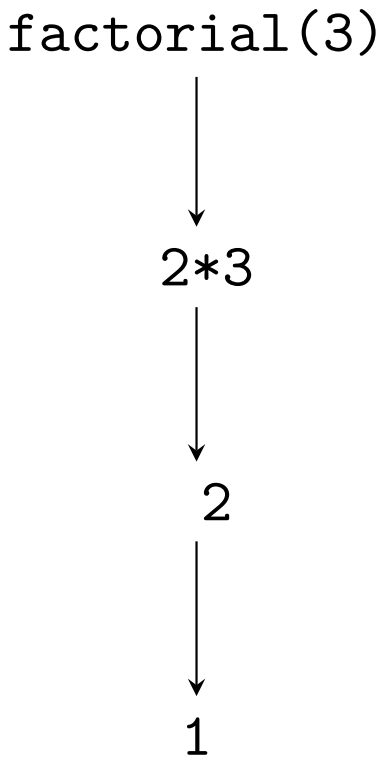
Zazwyczaj działanie funkcji rekurencyjnych ilustruje się nie poprzez rysowanie stanu stosu, lecz za pomocą tzw. *drzew wywołań* (ang. *call tree*) nazywanych też *drzewami rekurencji* lub *drzewami rekursji*. W przypadku funkcji `factorial()` to drzewo dla `n=3` będzie bardzo proste (matematycy by je określili jako zdegenerowane).



Rekurencja

Wstęp - silnia

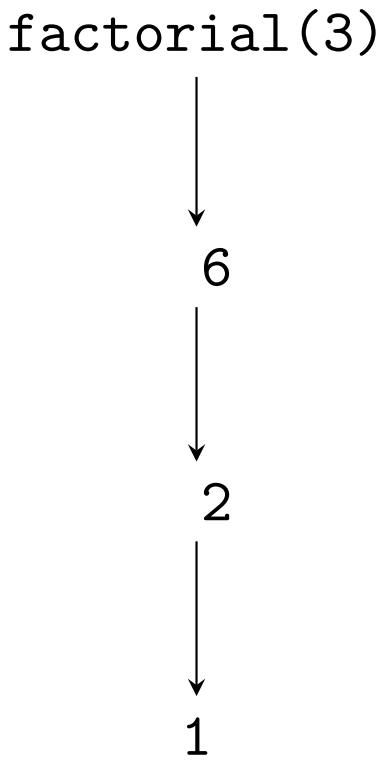
Zazwyczaj działanie funkcji rekurencyjnych ilustruje się nie poprzez rysowanie stanu stosu, lecz za pomocą tzw. *drzew wywołań* (ang. *call tree*) nazywanych też *drzewami rekurencji* lub *drzewami rekursji*. W przypadku funkcji `factorial()` to drzewo dla `n=3` będzie bardzo proste (matematycy by je określili jako zdegenerowane).



Rekurencja

Wstęp - silnia

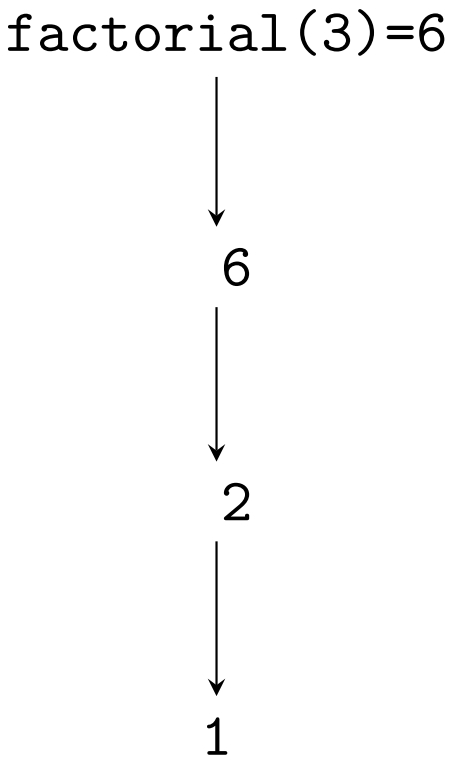
Zazwyczaj działanie funkcji rekurencyjnych ilustruje się nie poprzez rysowanie stanu stosu, lecz za pomocą tzw. *drzew wywołań* (ang. *call tree*) nazywanych też *drzewami rekurencji* lub *drzewami rekursji*. W przypadku funkcji `factorial()` to drzewo dla `n=3` będzie bardzo proste (matematycy by je określili jako zdegenerowane).



Rekurencja

Wstęp - silnia

Zazwyczaj działanie funkcji rekurencyjnych ilustruje się nie poprzez rysowanie stanu stosu, lecz za pomocą tzw. *drzew wywołań* (ang. *call tree*) nazywanych też *drzewami rekurencji* lub *drzewami rekursji*. W przypadku funkcji `factorial()` to drzewo dla `n=3` będzie bardzo proste (matematycy by je określili jako zdegenerowane).



Metoda dziel i zwyciężaj

Metoda dziel i zwyciężaj (ang. *divide and conquer*) jest techniką konstrukcji algorytmów rekurencyjnych. Składa się ona z trzech kroków:

Metoda dziel i zwyciężaj

1. **Dziel**
   Podziel problem na problemy tego samego typu, ale o mniejszym rozmiarze.
2. **Zwyciężaj**
   Rozwiąż problemy o mniejszym rozmiarze rekurencyjnie, chyba że są one tak małe, że nie wymagają zastosowania rekursji[a] - wtedy użyj metod bezpośrednich.
3. **Połącz**
   Połącz rozwiązania problemów mniejszych rozmiarów, aby otrzymać rozwiązanie problemu wyjściowego.

[a] Rekurencja jest nazywana inaczej rekursją.

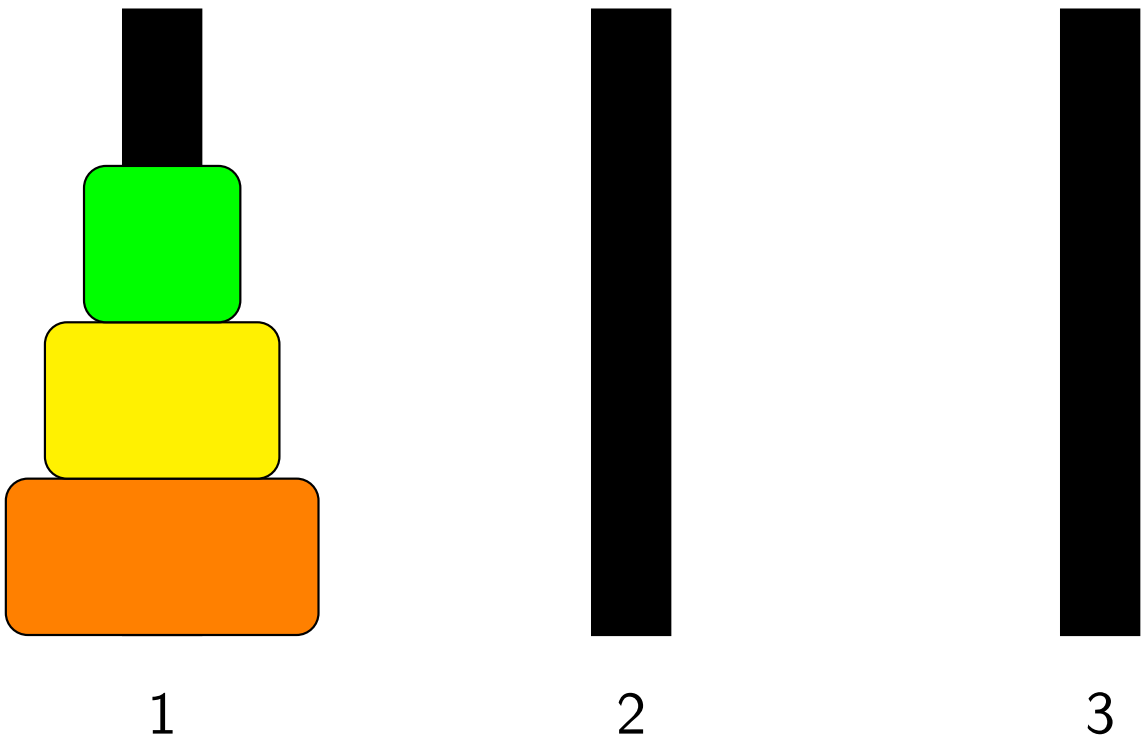
Metoda dziel i zwyciężaj

Wieże Hanoi

Z metodą dziel i zwyciężaj zapoznamy się na przykładzie problemu wież Hanoi, który został sformułowany przez francuskiego matematyka Edouarda Lucasa w 1883 roku. W tym problemie dana jest wieża składająca się z krążków o różnych średnicach znajdujących się na słupku i ułożonych tak, że krążki mniejsze leżą na krążkach większych. Oprócz tego dane są jeszcze dwa puste słupki. Należy przenieść wieżę na trzeci słupek, ale wykonując tę pracę można w jednym kroku przenieść tylko jeden krążek i nie wolno położyć krążka o większej średnicy na krążek o mniejszej średnicy. Animacja znajdująca się na następnym slajdzie pokazuje rozwiązanie tego problemu dla wieży składającej się z trzech krążków.

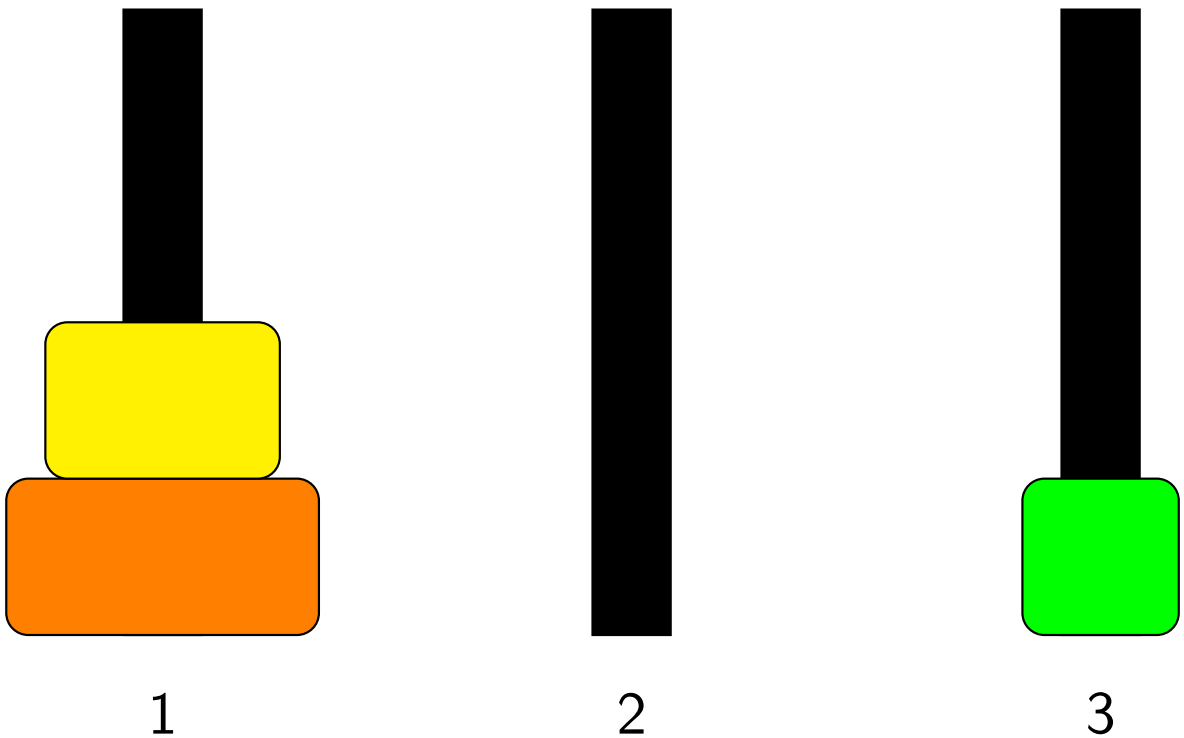
Metoda dziel i zwyciężaj

Wieże Hanoi



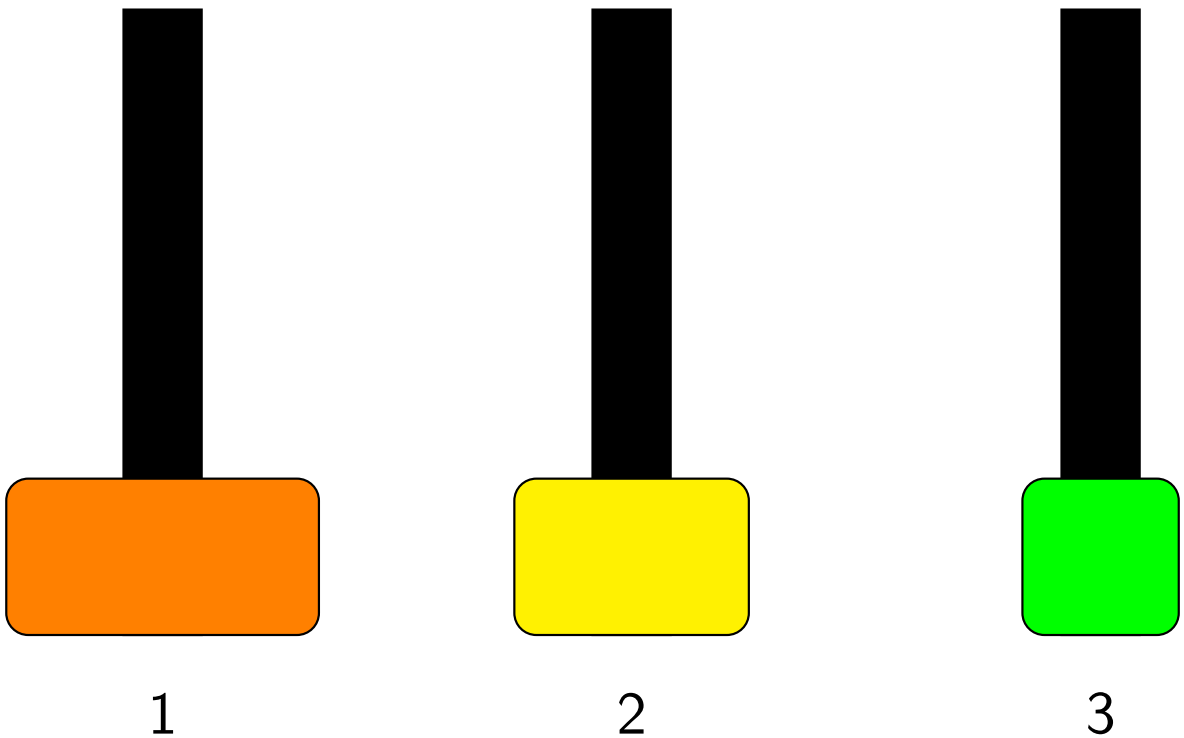
Metoda dziel i zwyciężaj

Wieże Hanoi



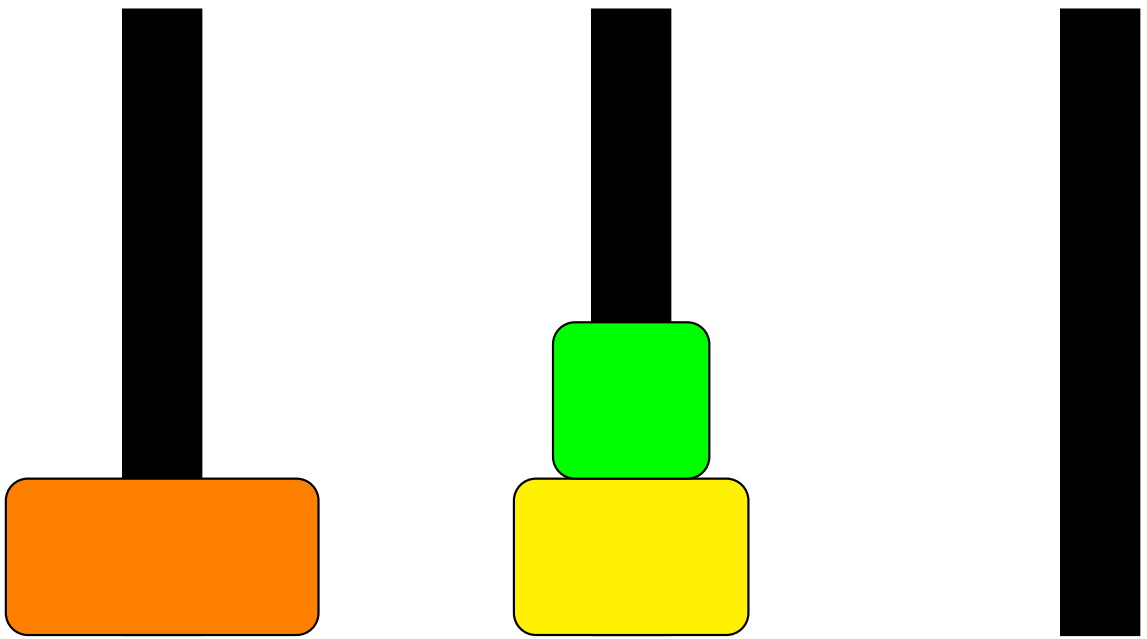
Metoda dziel i zwyciężaj

Wieże Hanoi



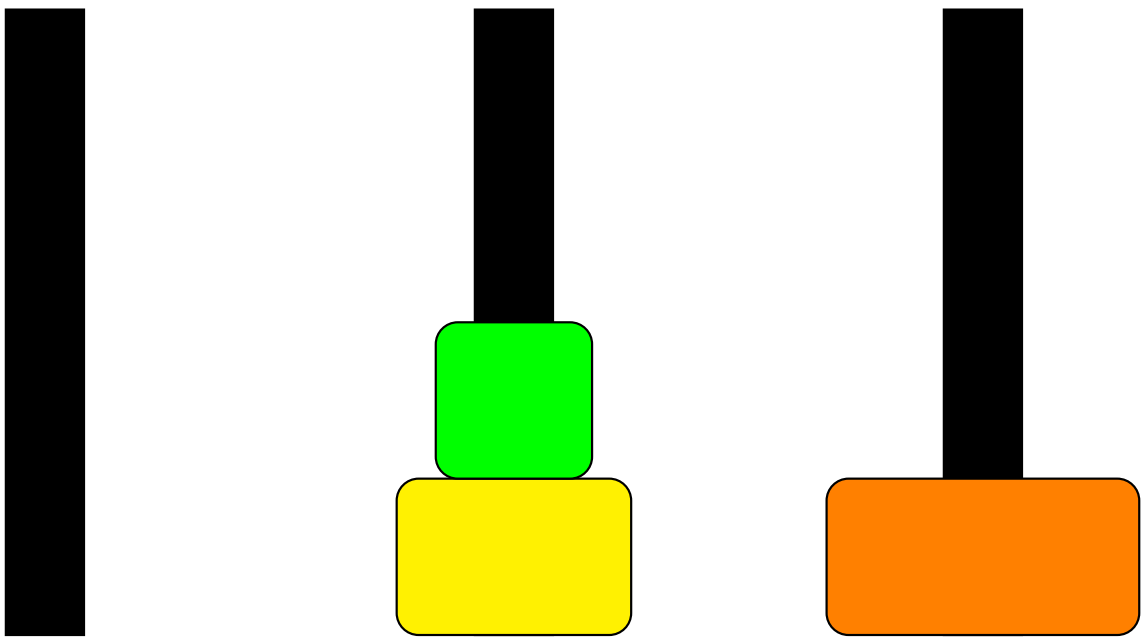
Metoda dziel i zwyciężaj

Wieże Hanoi



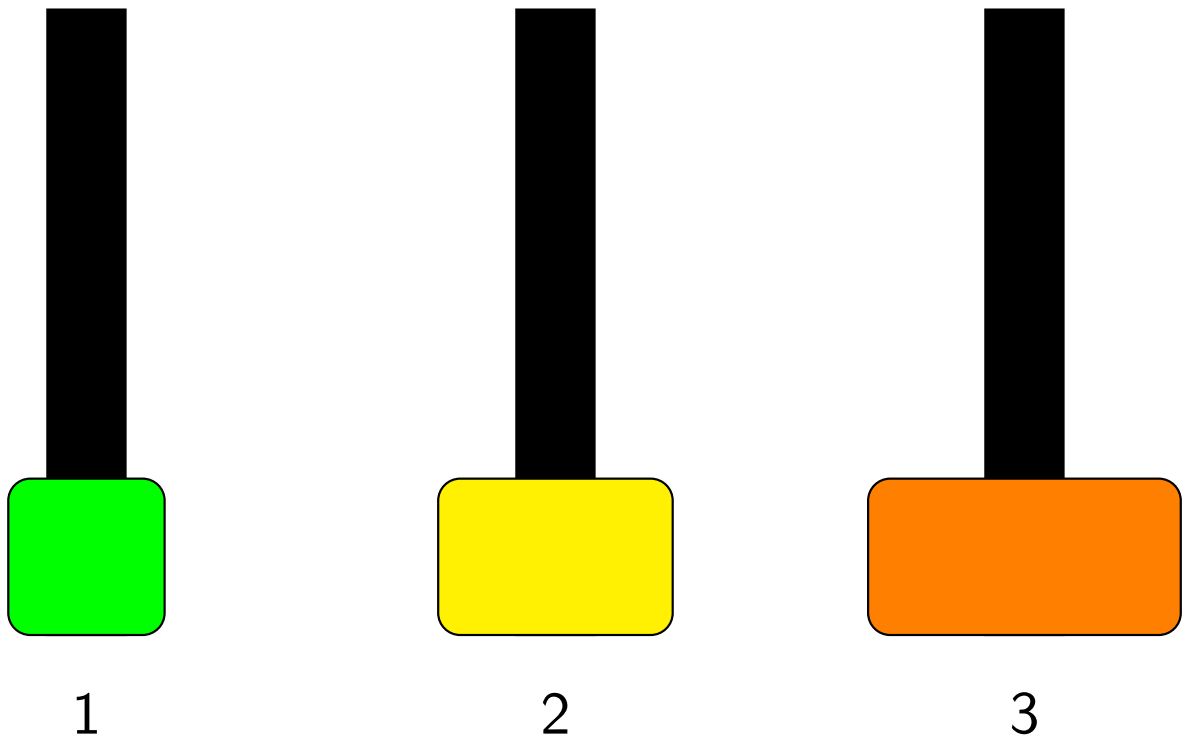
Metoda dziel i zwyciężaj

Wieże Hanoi



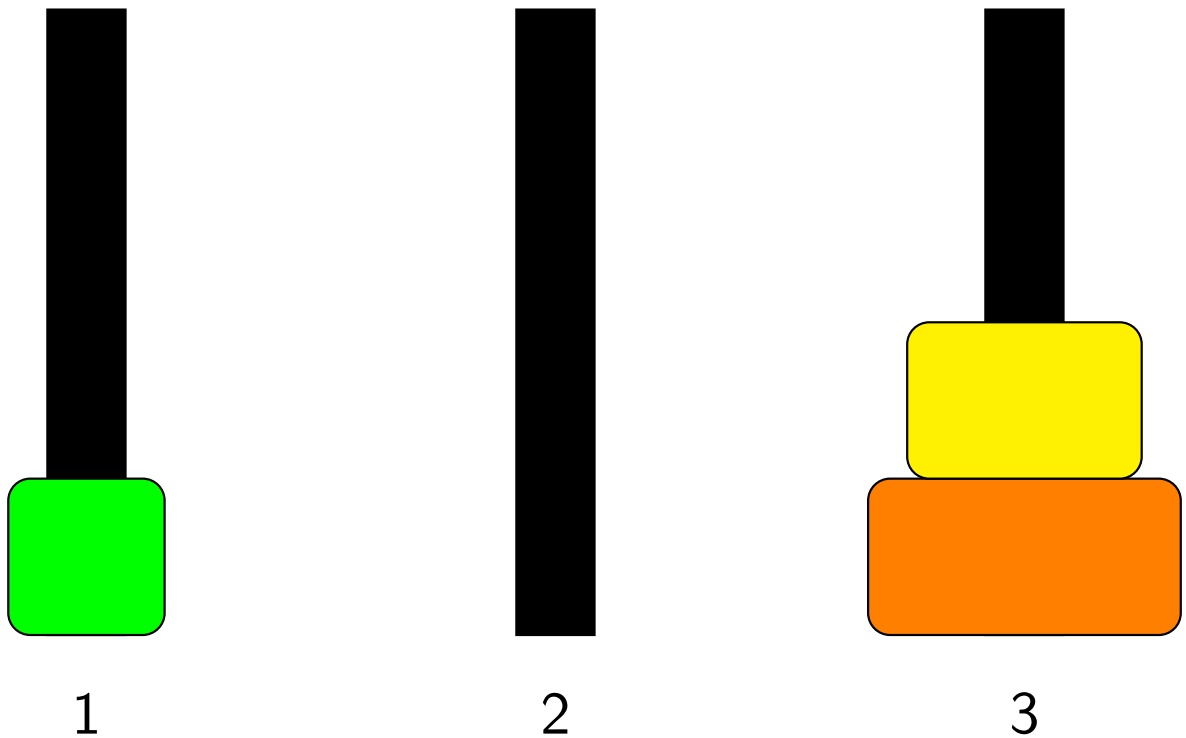
Metoda dziel i zwyciężaj

Wieże Hanoi



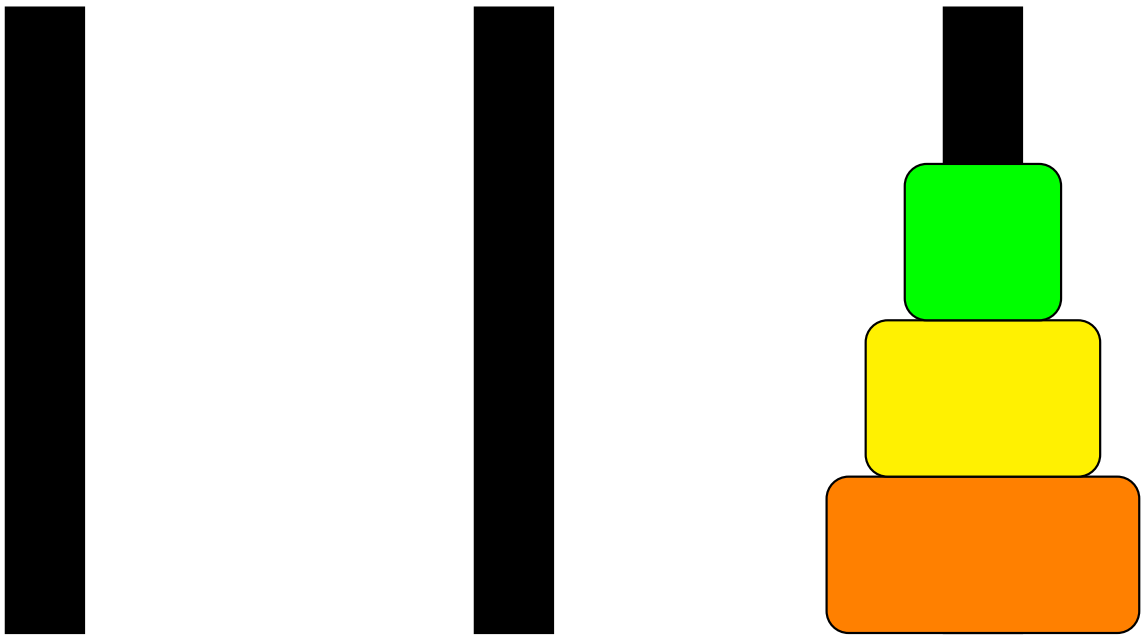
Metoda dziel i zwyciężaj

Wieże Hanoi



Metoda dziel i zwyciężaj

Wieże Hanoi



Metoda dziel i zwyciężaj

Wieże Hanoi

Na wykładzie rozwiążemy dwa problemy dotyczące wież Hanoi. Zaczniemy od prostszego, który został przedstawiony w książce autorstwa D.E.Knuth'a, R.L.Grahama i O.Patashnika pt. „Matematyka konkretna" (PWN, Warszawa 1996). Brzmi on następująco:

Pierwszy problem wież Hanoi

Wiedząc ile wieża ma krążków oblicz minimalną liczbę kroków, które należy wykonać aby ją przenieść.

Metoda dziel i zwyciężaj

Wieże Hanoi

Do rozwiązania tego zadania użyjemy metody dziel i zwyciężaj. Zgodnie z jej opisem, który został przedstawiony wcześniej należy najpierw znaleźć sposób podziału problemu na problemy o mniejszym rozmiarze. W przypadku wież Hanoi jest to zadanie stosunkowo proste - wieże składają się z dyskretnych krążków, więc zadaniem o bezpośrednio mniejszym rozmiarze od przeniesienia n krążków jest przeniesienie $n-1$ krążków. W drugim kroku musimy przyjąć strategię rozwiązywania rekurencyjnego mniejszych problemów. Gdybyśmy wiedzieli w jaki sposób przenieść te $n-1$ krążków, to przeniesienie n krążków byłoby proste. Przyjmijmy zatem, że wiemy jak to zrobić. Wtedy przełożenie n krążków przebiegałoby następująco:

1. Przenieś $n-1$ krążków ze słupka początkowego na słupek pomocniczy.
2. Przenieś n-ty krążek ze słupka początkowego na słupek docelowy.
3. Przenieś $n-1$ krążków ze słupka pomocniczego na słupek docelowy.

Metoda dziel i zwyciężaj

Wieże Hanoi

Oznaczmy minimalną liczbę kroków potrzebnych do przeniesienia wieży o n krążkach jako T_n, a jako T_{n-1} minimalną liczbę kroków potrzebnych do przeniesienia wieży o $n-1$ krążkach. Z opisu z poprzedniego slajdu wynika, że $T_n = 2 \cdot T_{n-1} + 1$. Ta sama relacja co dla wież o n i $n-1$ zachodzi także dla wież o $n-1$ i $n-2$ krążkach, tzn. wiedząc jak przenieść wieżę o $n-2$ krążkach poradzimy sobie z przeniesieniem wieży o $n-1$ krążkach. Należy jeszcze ustalić przypadek brzegowy, czyli określić problem o tak małym rozmiarze, że nie będzie potrzebny podział rekurencyjny aby go rozwiązać. Możemy przyjąć, że tym problemem będzie przeniesienie wieży pustej, tj. składającej się z $n = 0$ krążków. W takim przypadku nie trzeba wykonywać żadnych ruchów, co można zapisać jako $T_0 = 0$. Krok łączenia jest stosunkowo prosty: mając rozwiązanie dla 0 krążków znajdujemy rozwiązanie dla 1 krążka, potem 2 itd. Następny slajd zawiera kod źródłowy funkcji, który został napisany zgodnie z tymi ustaleniami.

Metoda dziel i zwyciężaj

Wieże Hanoi

```
unsigned long int find_hanoi_steps(unsigned char levels)
{
    if(levels==0)
        return 0;
    else
        return 2*find_hanoi_steps(levels-1)+1;
}
```

Metoda dziel i zwyciężaj

Wieże Hanoi

Przez parametr `levels` do funkcji przekazywana jest liczba krążków, dla której trzeba wyznaczyć minimalną liczbę kroków niezbędnych do przeniesienia wieży Hanoi, która się z nich składa. Zwróćmy uwagę, że posługując się metodą „dziel i zwyciężaj" możemy również napisać funkcję obliczającą silnię. Zatem jest to metoda o dosyć licznych zastosowaniach. Następny slajd przedstawia drzewo wywołań rekurencyjnych funkcji `find_hanoi_steps()` dla liczby krążków równej `4`.

Metoda dziel i zwyciężaj

Wieże Hanoi

```
find_hanoi_steps(4)
```

Metoda dziel i zwyciężaj

Wieże Hanoi

```
find_hanoi_steps(4)
```

↓

```
2*find_hanoi_steps(3)+1
```

Metoda dziel i zwyciężaj

Wieże Hanoi

```
find_hanoi_steps(4)
        ↓
2*find_hanoi_steps(3)+1
        ↓
2*find_hanoi_steps(2)+1
```

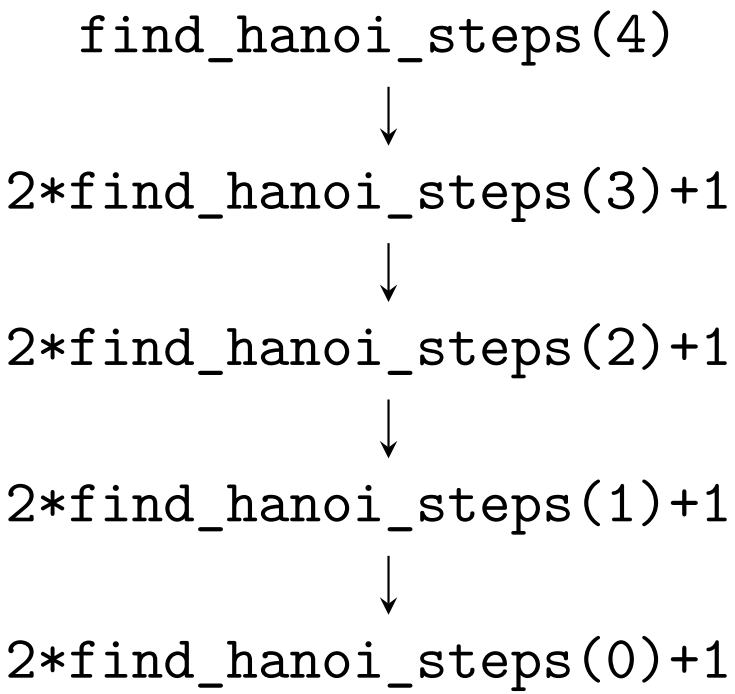
Metoda dziel i zwyciężaj

Wieże Hanoi

```
find_hanoi_steps(4)
        ↓
2*find_hanoi_steps(3)+1
        ↓
2*find_hanoi_steps(2)+1
        ↓
2*find_hanoi_steps(1)+1
```

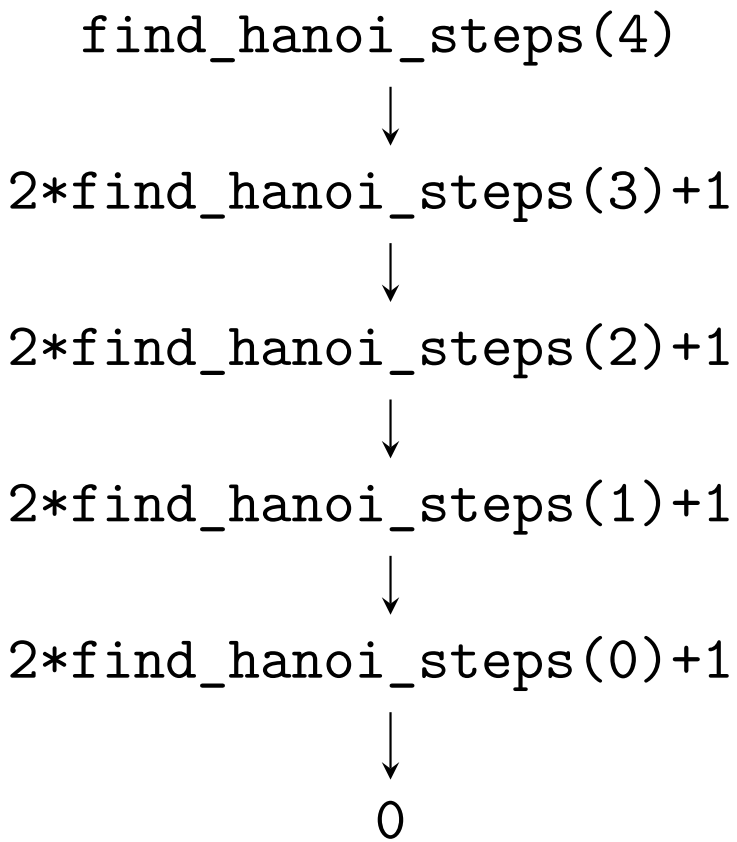
Metoda dziel i zwyciężaj

Wieże Hanoi



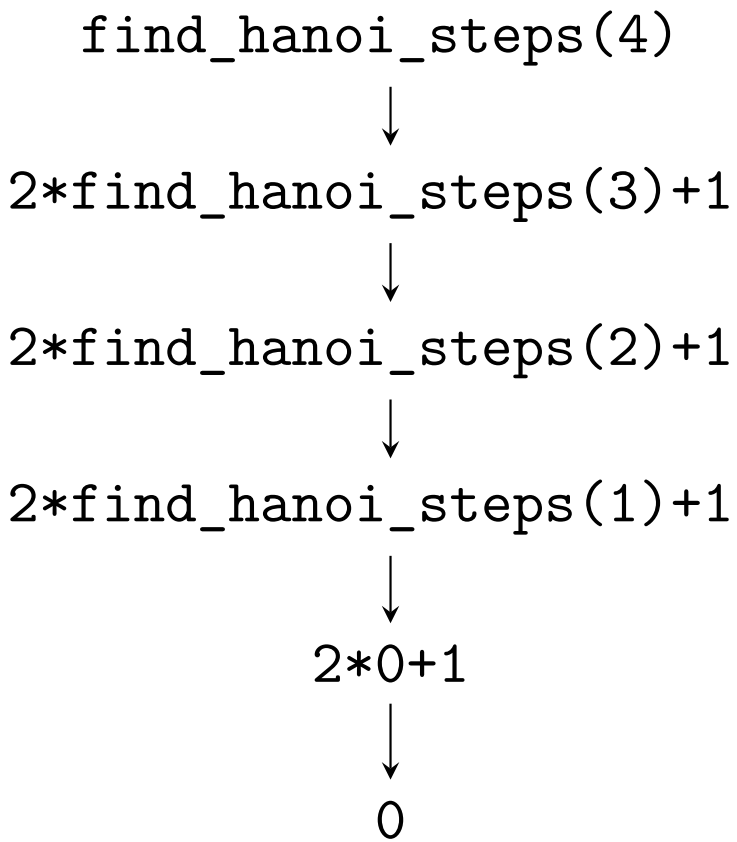
Metoda dziel i zwyciężaj

Wieże Hanoi



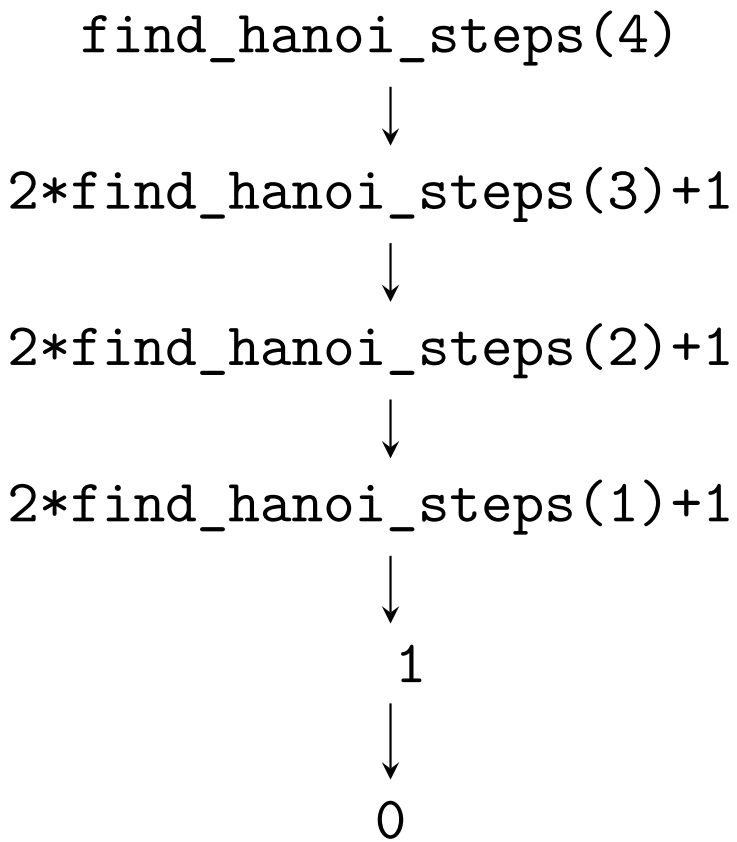
Metoda dziel i zwyciężaj

Wieże Hanoi



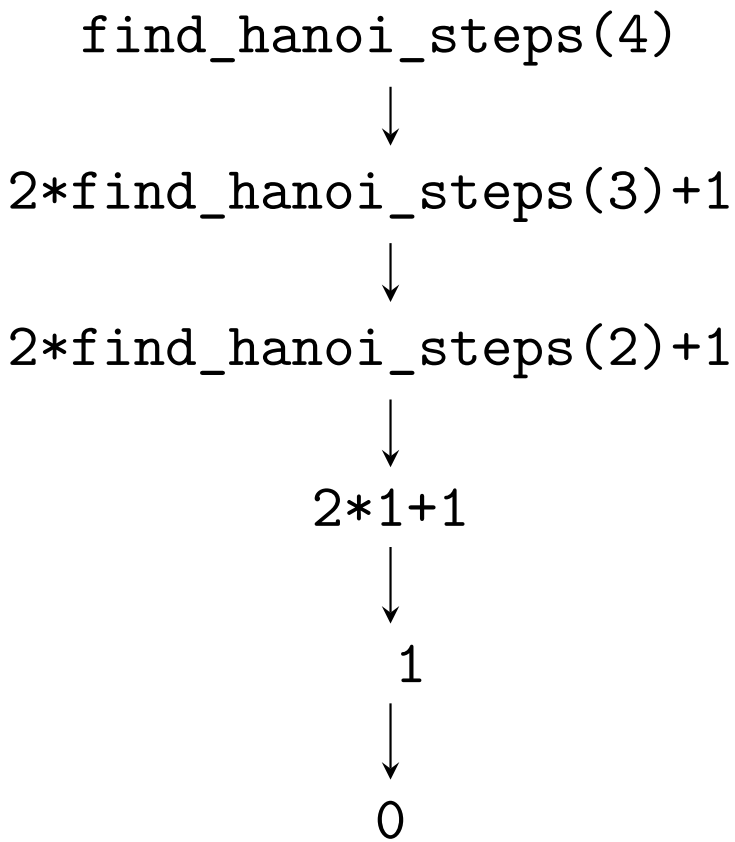
Metoda dziel i zwyciężaj

Wieże Hanoi



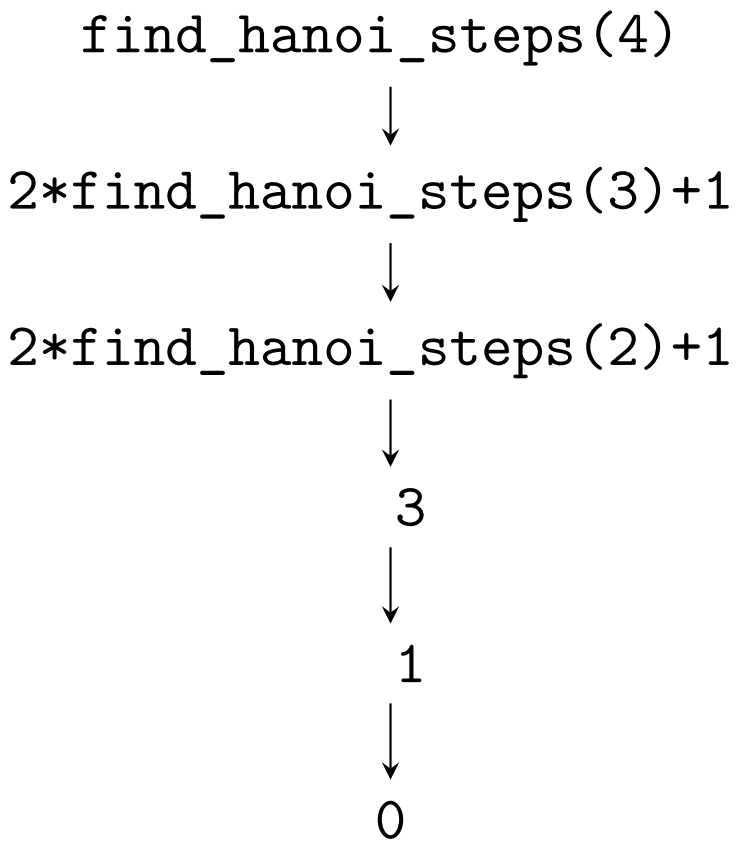
Metoda dziel i zwyciężaj

Wieże Hanoi



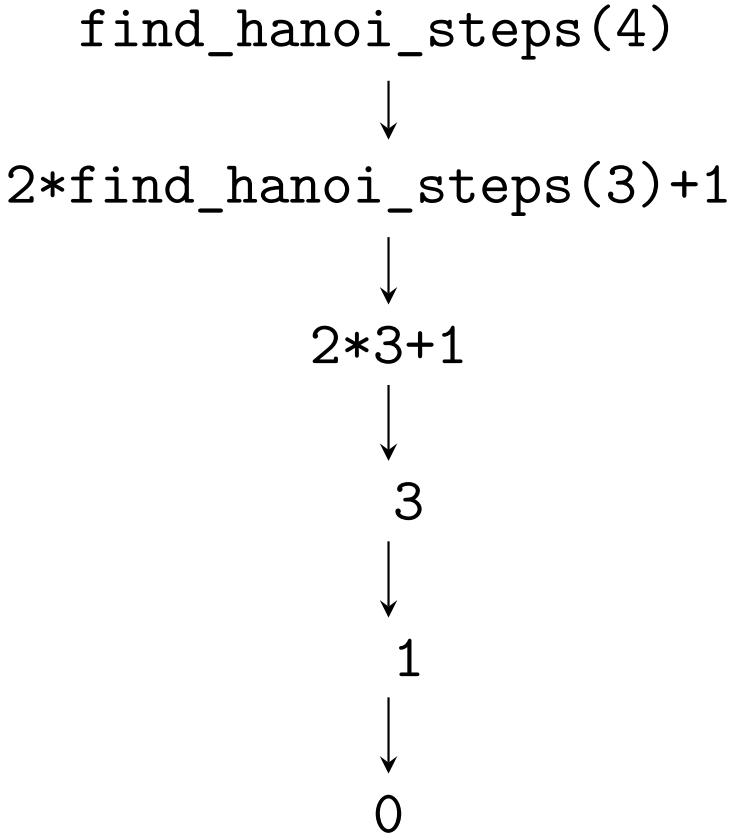
Metoda dziel i zwyciężaj

Wieże Hanoi



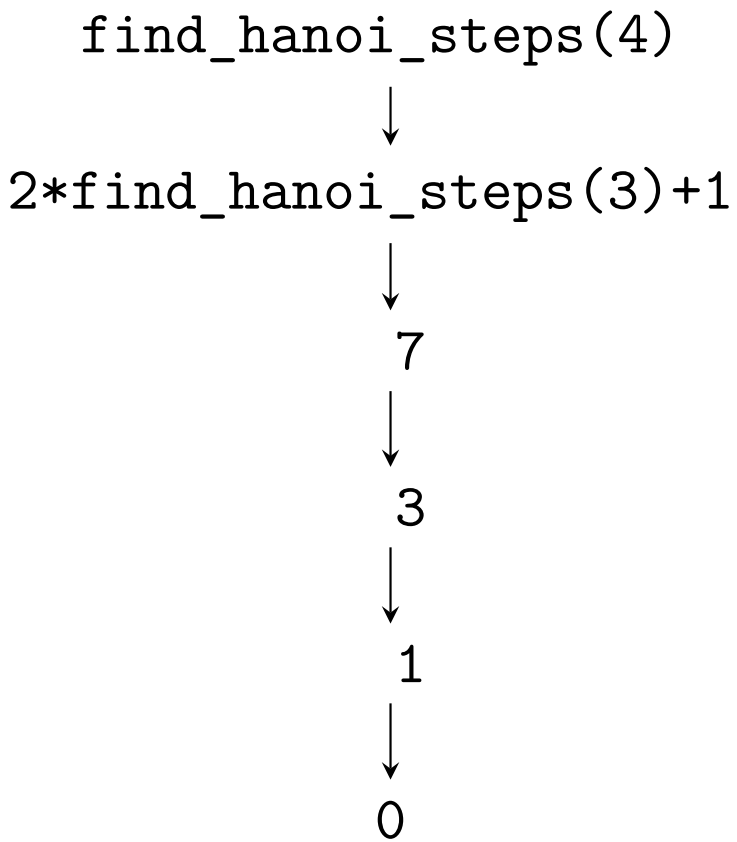
Metoda dziel i zwyciężaj

Wieże Hanoi



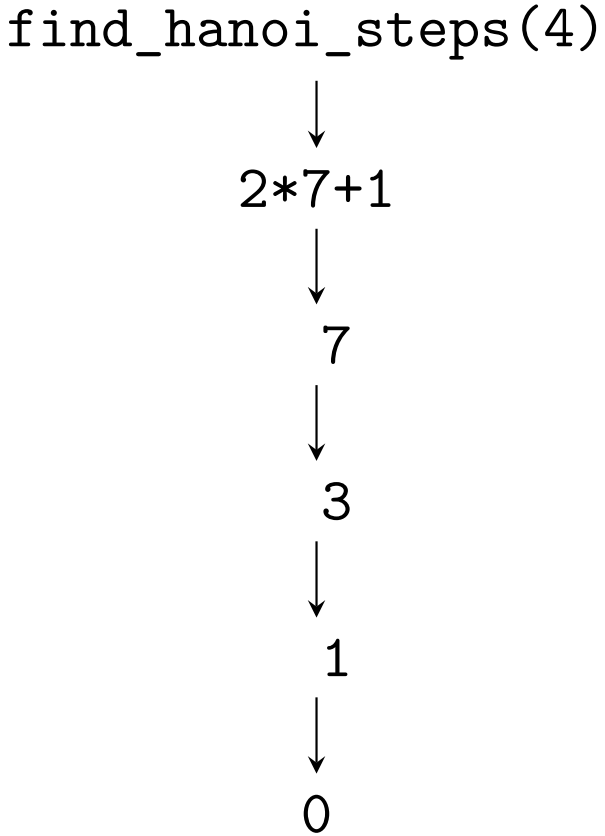
Metoda dziel i zwyciężaj

Wieże Hanoi



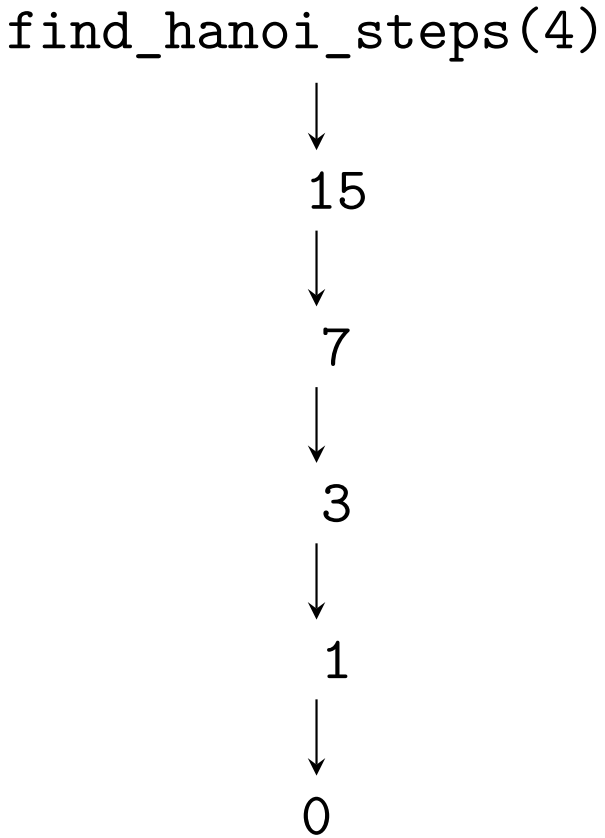
Metoda dziel i zwyciężaj

Wieże Hanoi



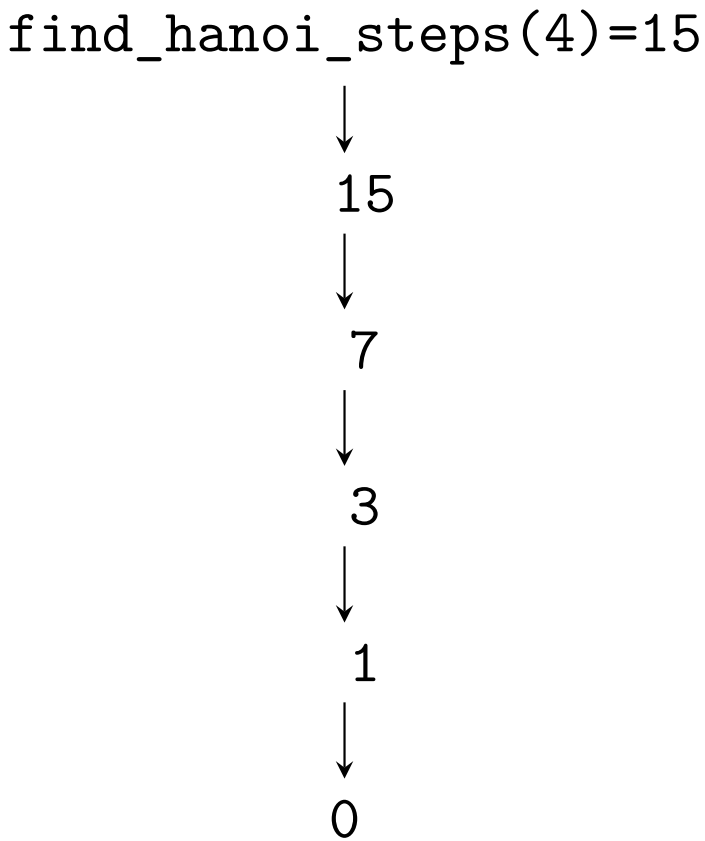
Metoda dziel i zwyciężaj

Wieże Hanoi



Metoda dziel i zwyciężaj

Wieże Hanoi



Metoda dziel i zwyciężaj

Wieże Hanoi

Spróbujmy teraz przedstawić i rozwiązać drugi problem dotyczący wież Hanoi. Brzmi on następująco:

Drugi problem wież Hanoi

Znajdź taki algorytm przekładania wieży Hanoi na słupek docelowy, aby przy tej pracy była wykonywana minimalna liczba kroków.

Okazuje się, że przy rozwiązywaniu tego problemu pomocne jest rozumowanie, które przeprowadzaliśmy dla pierwszego problemu wież Hanoi. Przypomnijmy, że aby przenieść wieżę pustą nie musimy nic robić, czyli wykonujemy 0 kroków. Aby przenieść wieżę składającą się z n krążków musimy najpierw przenieść $n-1$ krążków ze słupka źródłowego na słupek pomocniczy, potem przenieść n-ty krążek ze słupka źródłowego na słupek docelowy i w końcu przenieść $n-1$ krążków ze słupka pomocniczego na słupek docelowy. Kolejny slajd zawiera kod źródłowy funkcji napisanej w oparciu o te rozważania.

Metoda dziel i zwyciężaj

Wieże Hanoi

```
1  void hanoi_movements(unsigned int disks,
2                       unsigned char first,
3                       unsigned char middle,
4                      unsigned char last)
5  {
6      if(disks) {
7          hanoi_movements(disks-1,first,last,middle);
8          printf("Przenieś krążek nr %u ze słupka nr %u na\
9   słupek nr %u\n",disks,first,last);
10         hanoi_movements(disks-1,middle,first,last);
11     }
12 }
```

Metoda dziel i zwyciężaj

Wieże Hanoi

Przez pierwszy parametr tej funkcji przekazywana jest liczba krążków, z których składa się wieża do przeniesienia. Trzy kolejne parametry, to numery kolejno: słupka początkowego, pomocniczego i końcowego. Funkcja sprawdza w wierszu nr 6, czy liczba krążków nie jest zerowa. Jeśli ten warunek jest prawdziwy, to będzie ona informowała użytkownika, który krążek (o jakim numerze) ma przenieść z jakiego słupka na jaki (też numery), w danym kroku. Zgodnie z ustaleniami najpierw funkcja musi przenieść mniejszą wieżę ze słupka źródłowego na pomocniczy, który w tym przypadku będzie słupkiem docelowym. Stąd mamy pierwsze wywołanie rekurencyjne (wiersz nr 7). Następnie funkcja wypisuje komunikat z instrukcją przeniesienia n-tego krążka (wiersze 8 i 9), a następnie ponownie się wywołuje rekurencyjnie (wiersz 10), aby przenieść $n-1$ krążków ze słupka pomocniczego (tym razem będzie on słupkiem źródłowym) na słupek końcowy.

Zalety i wady rekurencji

Niewątpliwą zaletą rekurencji jest zwięzłość zapisu funkcji. Zbadamy to na dwóch przykładach. Pierwszy będzie dotyczył funkcji, która konwertuje zapis dziesiętny liczby naturalnej większej od 0 na binarny. Drugi, a właściwie dwa następne będą dotyczyły programów, które rysują na ekranie w trybie graficznym fraktal, tzw. trójkąt Sierpińskiego.

Zalety i wady rekurencji

Konwersja zapisu dziesiętnego na binarny

Algorytm konwersji zapisu liczby z kodu dziesiętnego na binarny nie wydaje się być rekurencyjny. Polega on na sukcesywnym wyznaczaniu reszty z dzielenia liczby przez dwa, a następnie wyniku całkowitego dzielenia tej liczby przez dwa do momentu, aż to ostatnie działanie da 0. Potem należy już tylko odczytać reszty w odwrotnej kolejności w stosunku do tej, w jakiej zostały uzyskane. Ostatnie stwierdzenie sugeruje użycie stosu, co oznacza, że zastosowanie rekurencji uprościło by rozwiązanie problemu - funkcje rekurencyjne korzystają ze stosu sprzętowego, który jest obsługiwany sprzętowo. Zauważmy także, że wynik dzielenia przez dwa staje się daną wejściową dla kolejnego kroku algorytmu, a więc może on być wyrażony w postaci rekursywnej. Przypadkiem brzegowym będzie osiągnięcie wyniku zerowego z dzielenia przez dwa. Kolejny slajd zawiera kod źródłowy funkcji, która dokonuje opisywanej konwersji, wypisując przy tym uzyskaną liczbę binarną na ekranie.

Zalety i wady rekurencji

Konwersja zapisu dziesiętnego na binarny

```
1  void convert_to_binary(unsigned long int number)
2  {
3      if(number) {
4          convert_to_binary(number/2);
5          printf("%lu",number%2);
6      }
7  }
```

Zalety i wady rekurencji

Konwersja zapisu dziesiętnego na binarny

Zaprezentowana funkcja sprawdza, czy przekazana jej przez parametr liczba jest większa od zera. Jeśli tak, to wywołuje się rekurencyjnie dla tej liczby pomniejszonej dwukrotnie. Ten ciąg wywołań trwa tak długo, aż przy którymś dzieleniu wartość liczby będzie równa zero. Wówczas następuje powrót z wywołań rekurencyjnych i w każdym z nich wykonywana jest funkcja `printf()`, która wypisuje resztę z dzielenia przez dwa przekazanej przez parametr liczby. Algorytm według którego działa ta funkcja można zatem otrzymać stosując metodę dziel i zwyciężaj. Przypadkiem brzegowym jest uzyskanie zera z dzielenia. Redukcja rozmiaru problemu, to tym razem podzielenie liczby przed dwa. Jest także krok łączenia - to wypisanie reszty z dzielenia. Ta funkcja nie wykona konwersji liczby `0`. Można to zmienić obejmując instrukcją warunkową tylko wywołanie rekurencyjne, ale wówczas dla każdej konwertowanej liczby otrzymamy na ekranie wiodące `0` (na najstarszym bicie).

Zalety i wady rekurencji

Trójkąt Sierpińskiego

Następny przykład wymaga trochę bardziej skomplikowanego kodu. Będzie on rysował na ekranie komputera trójkąt Sierpińskiego - fraktal, którego nazwa pochodzi od jego odkrywcy, polskiego matematyka Wacława Sierpińskiego. Trójkąt Sierpińskiego to fraktal, który otrzymywany jest przez podział trójkąta (najczęściej równobocznego lub równoramiennego, ale może być także dowolny inny, oprócz zdegenerowanego) na cztery mniejsze trójkąty i wyjęcie z niego trójkąta środkowego. Czynność należy powtórzyć dla pozostałych trzech trójkątów, następnie dla trójkątów pozostałych w tych trójkątach itd. Jak łatwo stwierdzić algorytm tworzenia takiego fraktala jest rekurencyjny. Jako warunek zakończenia rekurencji możemy przyjąć utworzenie trójkątów o zakładanej minimalnej długości boków lub osiągnięcie założonego poziomu „zagnieżdżeń" trójkątów. W prezentowanych programach przyjmiemy drugi, prostszy wariant. Najpierw jednak przedstawimy dwa możliwe algorytmy rekurencyjne generowania takiego fraktala.

Zalety i wady rekurencji

Trójkąt Sierpińskiego - pierwszy sposób

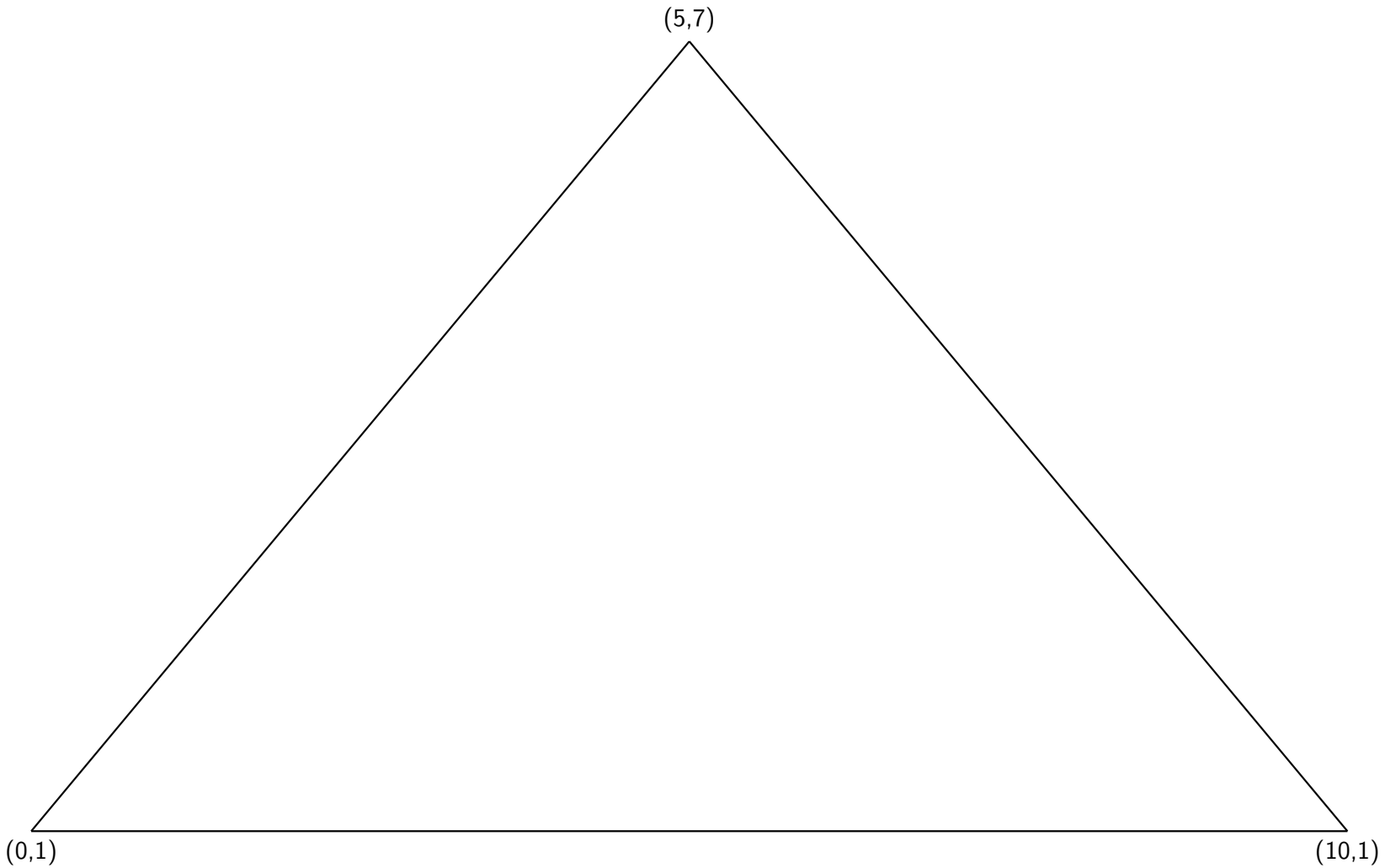
Pierwszy sposób polega na narysowaniu trójkąta, wyznaczeniu środków jego boków i narysowaniu trzech trójkątów, których jeden wierzchołek pokrywa się z wierzchołkiem wyjściowego trójkąta, a dwa pozostałe leżą w środkach odpowiednich jego boków. To działanie należy dalej powtórzyć, dla każdego z tych trzech trójkątów i następnych trzech, tak długo, aż osiągnięty zostanie zakładany poziom zagnieżdżenia. Animacja na następnym slajdzie pokazuje kilka pierwszych kroków tego algorytmu. Potem zamieszczony jest kod źródłowy programu, który rysuje ten fraktal opisaną metodą.

Zalety i wady rekurencji

Trójkąt Sierpińskiego - pierwszy sposób

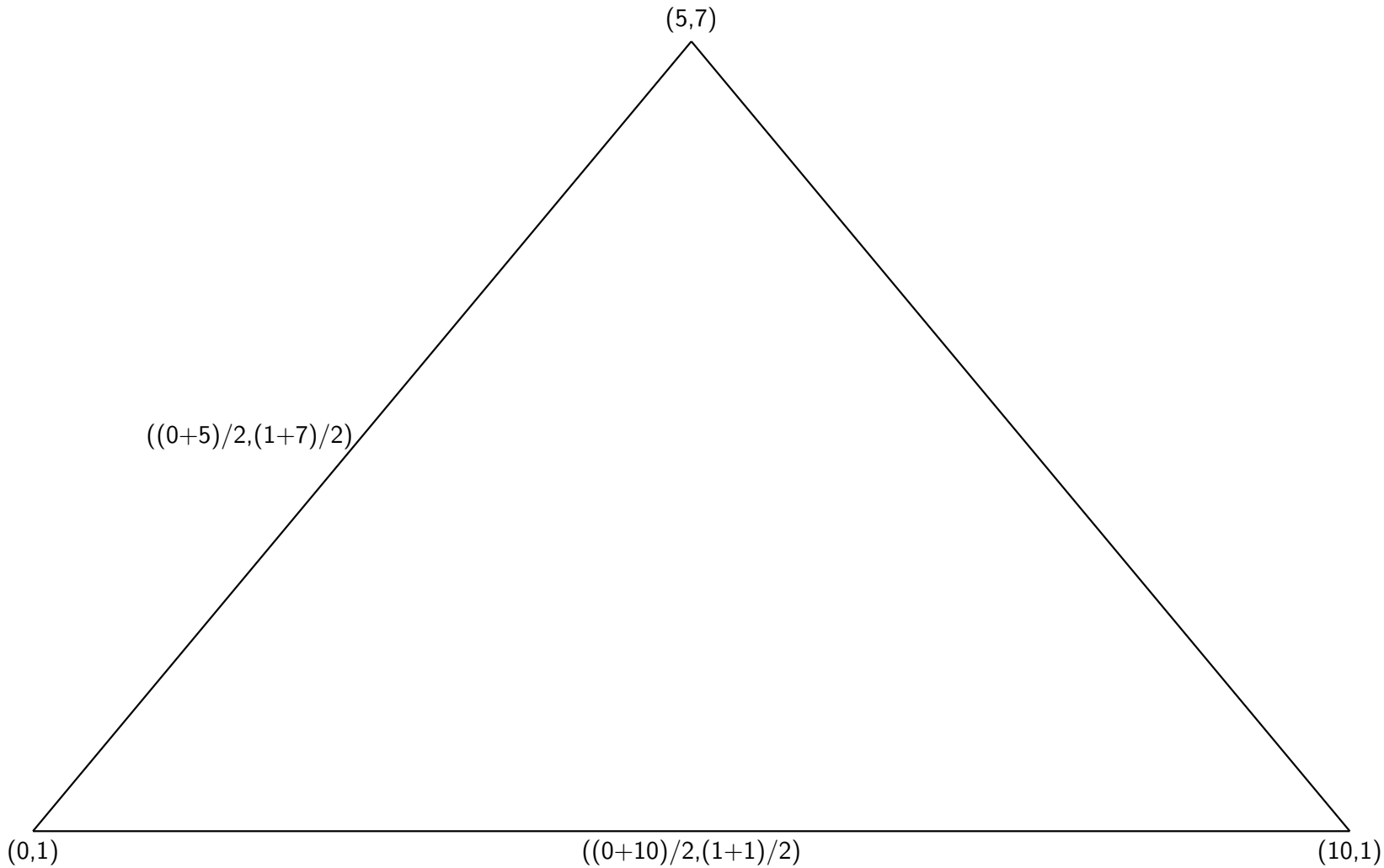
Zalety i wady rekurencji

Trójkąt Sierpińskiego - pierwszy sposób



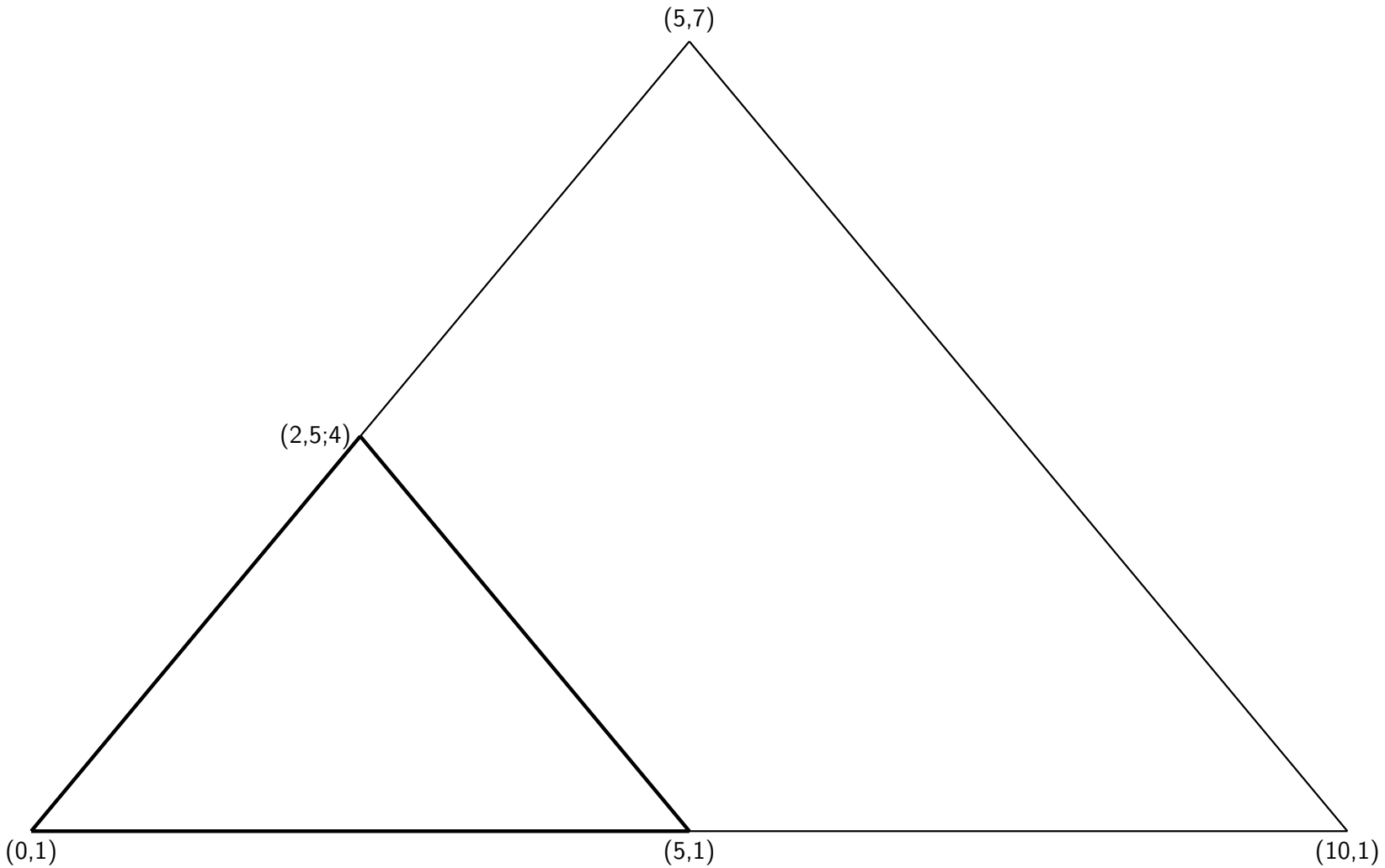
Zalety i wady rekurencji

Trójkąt Sierpińskiego - pierwszy sposób



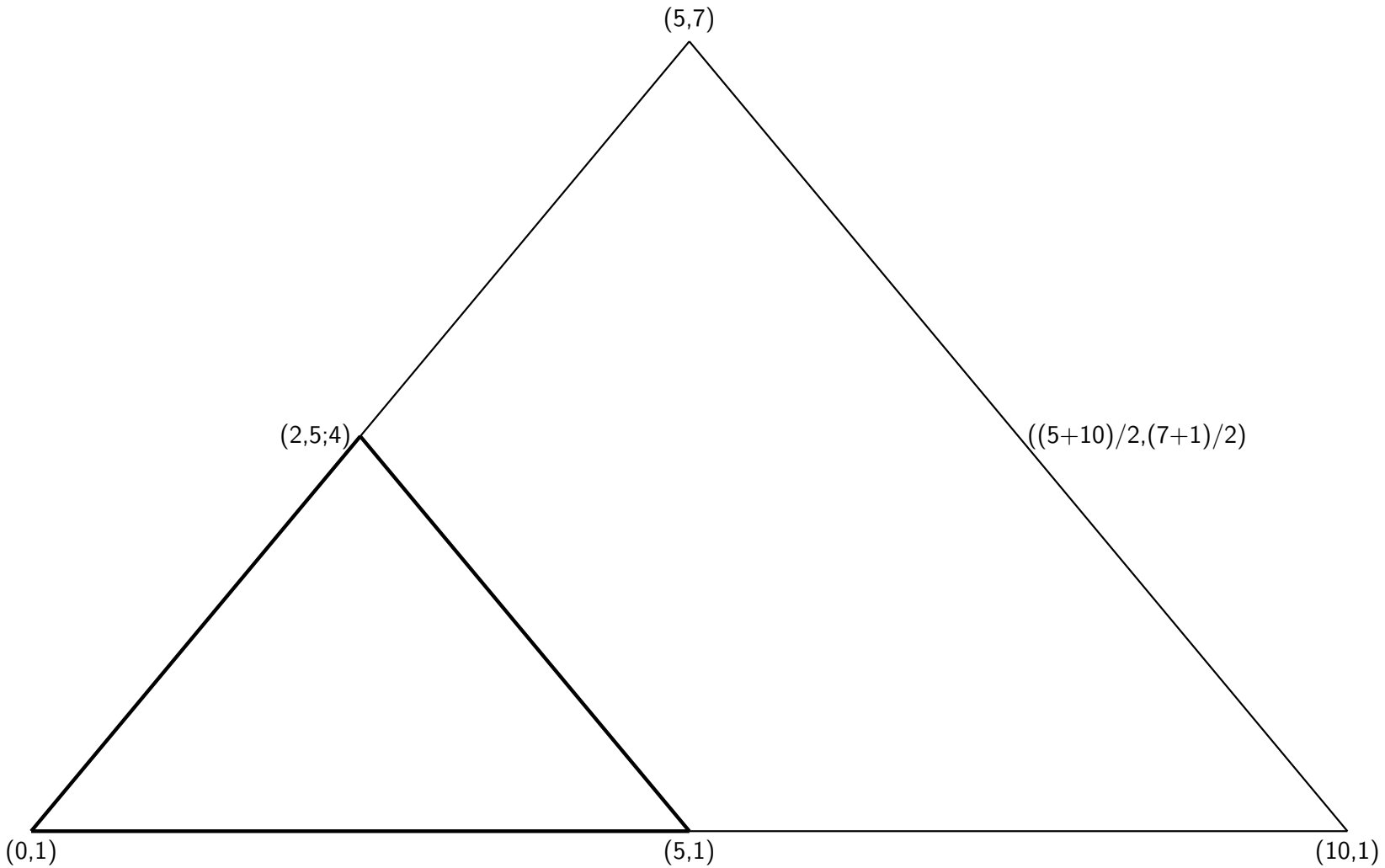
Zalety i wady rekurencji

Trójkąt Sierpińskiego - pierwszy sposób



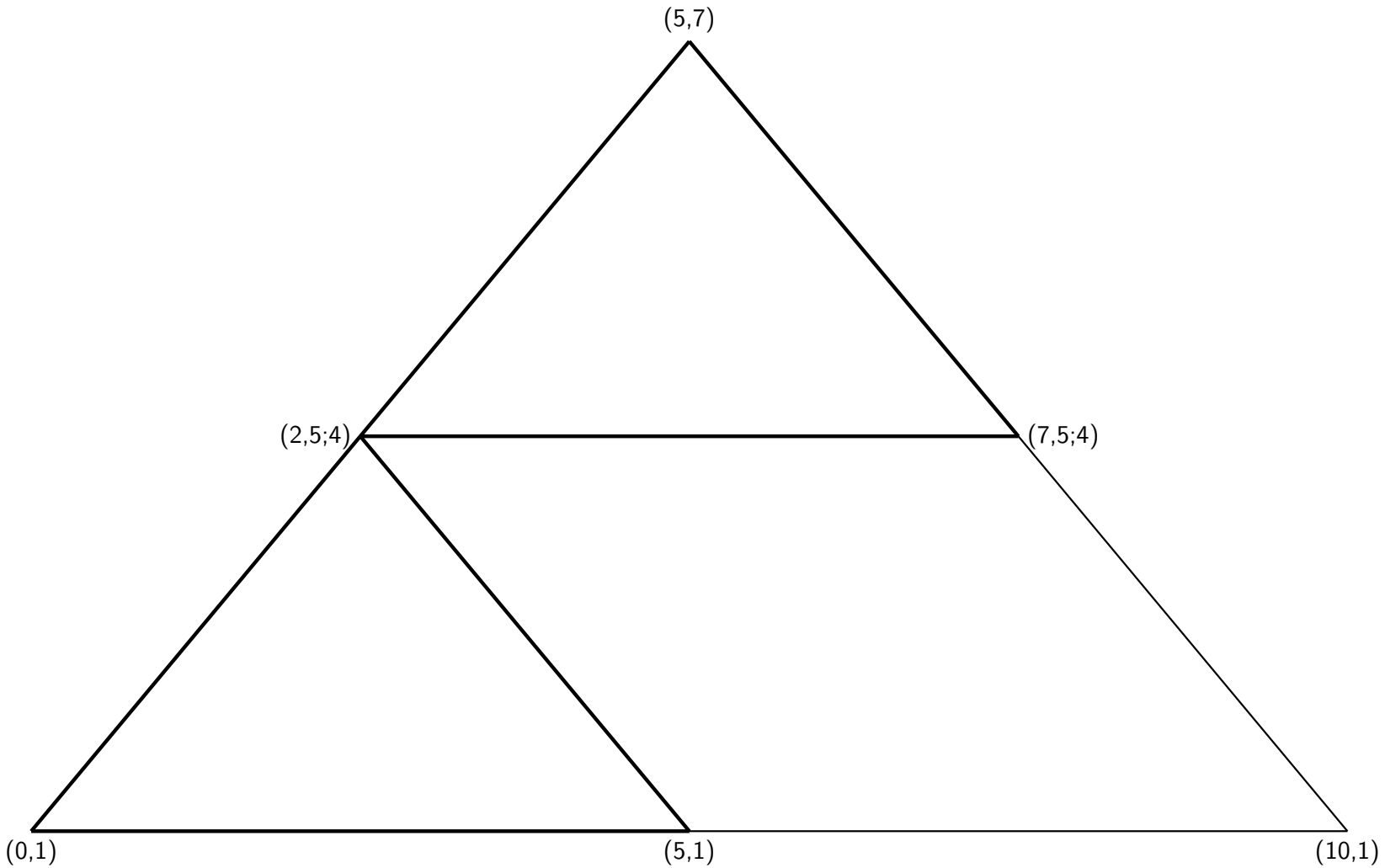
Zalety i wady rekurencji

Trójkąt Sierpińskiego - pierwszy sposób



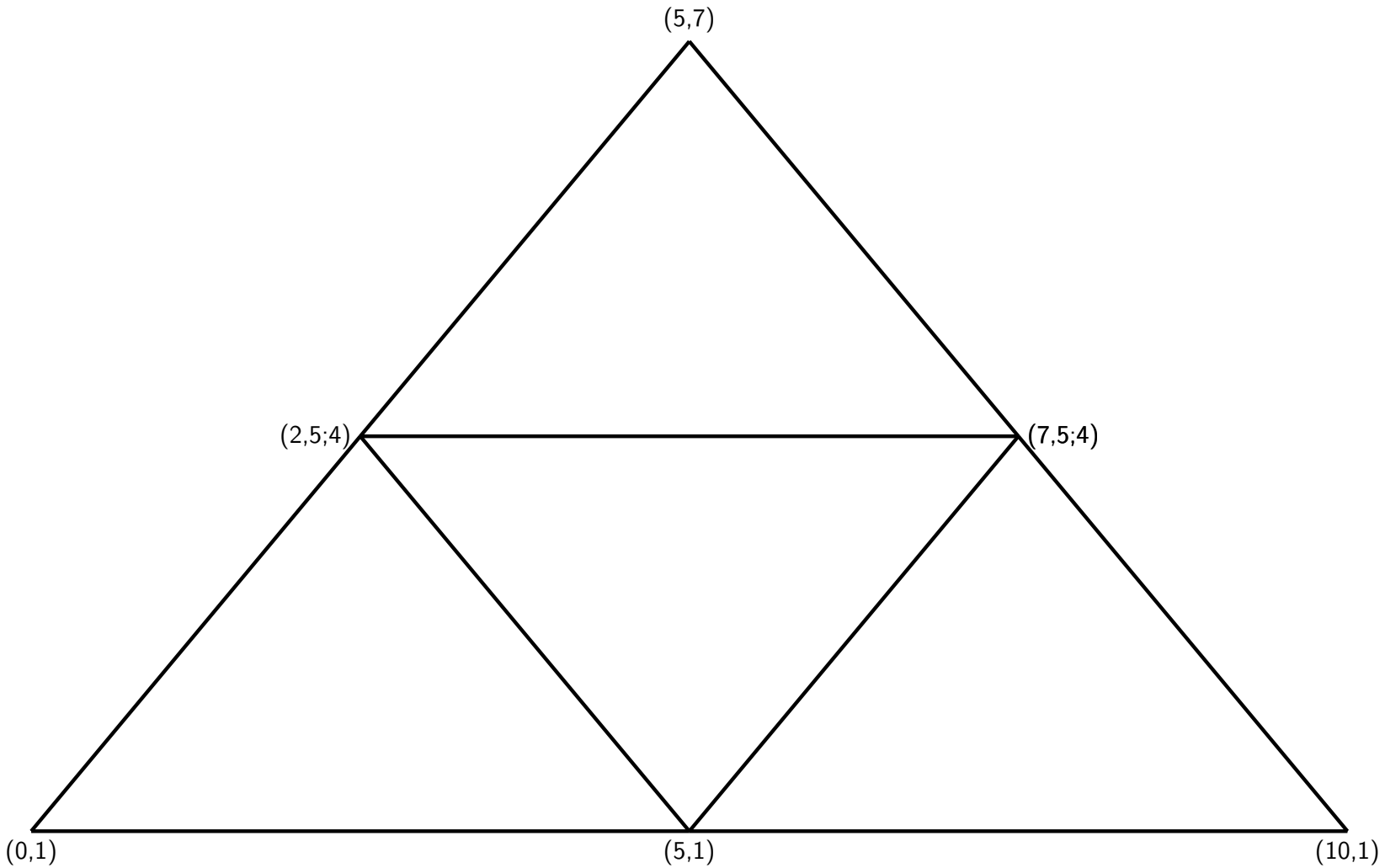
Zalety i wady rekurencji

Trójkąt Sierpińskiego - pierwszy sposób



Zalety i wady rekurencji

Trójkąt Sierpińskiego - pierwszy sposób



Zalety i wady rekurencji

Trójkąt Sierpińskiego - pierwszy sposób

```
#include<allegro.h>


void draw_triangle(int x, int y, int x1, int y1, int x2, int y2, unsigned char n)
{
        int colour = makecol(255,255,255);
        line(screen,x,y,x1,y1,colour);
        line(screen,x1,y1,x2,y2,colour);
        line(screen,x2,y2,x,y,colour);
        if(n--) {
                draw_triangle((x+x1)/2,(y+y1)/2,x1,y1,(x2+x1)/2,(y2+y1)/2,n);
                draw_triangle(x,y,(x1+x)/2,(y1+y)/2,(x2+x)/2,(y2+y)/2,n);
                draw_triangle((x+x2)/2,(y+y2)/2,(x1+x2)/2,(y1+y2)/2,x2,y2,n);
        }
}
```

Zalety i wady rekurencji

Trójkąt Sierpińskiego - pierwszy sposób

```
int main(void) {
        int width=1336,height=768;
        if(allegro_init())
                allegro_message("allegro_init: %s\n",allegro_error);
        if(install_keyboard())
                allegro_message("install_keyboard: %s\n",allegro_error);
        set_color_depth(32);
        if(set_gfx_mode(GFX_AUTODETECT_FULLSCREEN,width,height,0,0))
                allegro_message("%s\n",allegro_error);
        height-=1;
        draw_triangle(width/2,0,0,height,width,height,14);
        clear_keybuf();
        while(!keypressed());
        allegro_exit();
        return 0;
}
END_OF_MAIN()
```

Zalety i wady rekurencji

Trójkąt Sierpińskiego - pierwszy sposób

Zaprezentowany program używa biblioteki Allegro w wersji 4.4. Definicja funkcji `draw_triangle()` (pierwszy slajd) jest krótka, ale warto się jej przyjrzeć. Ma ona siedem parametrów. Przez pierwsze dwa przekazywane są współrzędne „górnego" wierzchołka trójkąta, przez kolejne dwa przekazywane są współrzędne „lewego wierzchołka", a przez następne dwa „prawego". Przez ostatni parametr przekazywana jest liczba zagnieżdżeń. Funkcja najpierw rysuje na ekranie trójkąt zgodnie z przekazanymi jej w bieżącym wywołaniu współrzędnymi. Następnie sprawdza, czy liczba zagnieżdżeń jest różna od zera i zmniejsza ją o jeden. Jeśli warunek jest prawdziwy, to funkcja wywołuje się rekurencyjnie trzykrotnie, aby narysować trzy nowe trójkąty. Proszę zwrócić uwagę na argumenty tych wywołań. Liczba zagnieżdżeń jest zmniejszana w instrukcji warunkowej, więc do wywołań rekurencyjnych jest przekazana już pomniejszona. Zgodnie z opisem metody tylko jeden zestaw współrzędnych jest przekazywany bez zmian dla każdego z wywołań. Dla pozostałych są wyliczane odpowiednie średnie arytmetyczne.

Zalety i wady rekurencji

Trójkąt Sierpińskiego - pierwszy sposób

W funkcji `main()`, oprócz funkcji związanych z biblioteką Allegro, jest wywoływana `draw_triangle()`.

Zalety i wady rekurencji

Trójkąt Sierpińskiego - drugi sposób

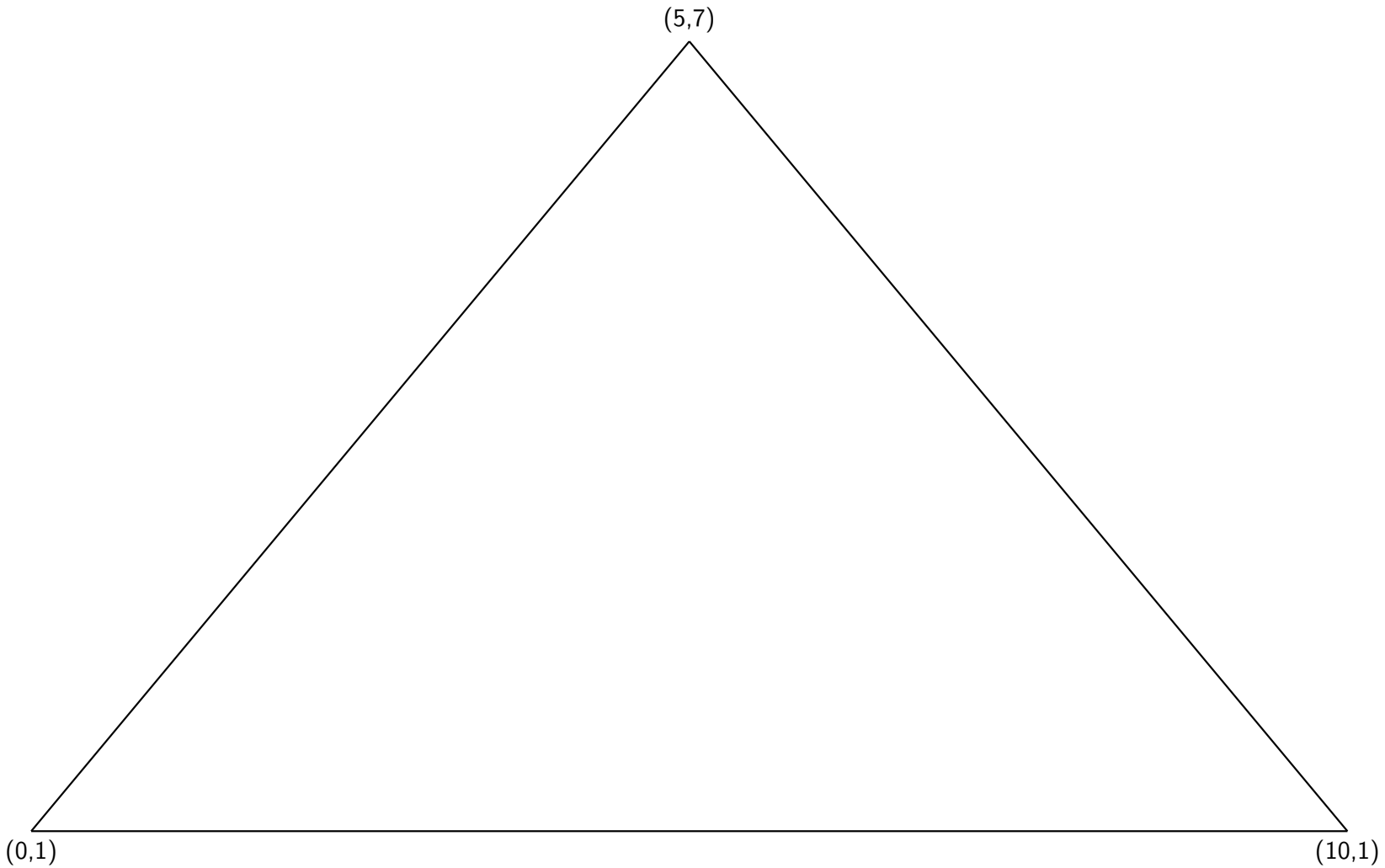
Drugi sposób rysowania trójkąta Sierpińskiego jest niewielką modyfikacją pierwszego. Początek jest taki sam - rysowany jest trójkąt i wyznaczane są środki jego boków. Następny krok jest jednak inny. Środki tych boków są łączone, aby uzyskać wewnętrzny, odwrócony trójkąt. To działanie powtarzane jest dla utworzonych w ten sposób trzech pozostałych trójkątów i dla następnych takich trójkątów, aż zostanie uzyskany pożądany poziom zagnieżdżeń. Kilka początkowych kroków tego algorytmu prezentuje animacja zamieszczona na następnym slajdzie. Kolejne slajdy zawierają kod źródłowy programu rysującego trójkąt Sierpińskiego tą metodą.

Zalety i wady rekurencji

Trójkąt Sierpińskiego - drugi sposób

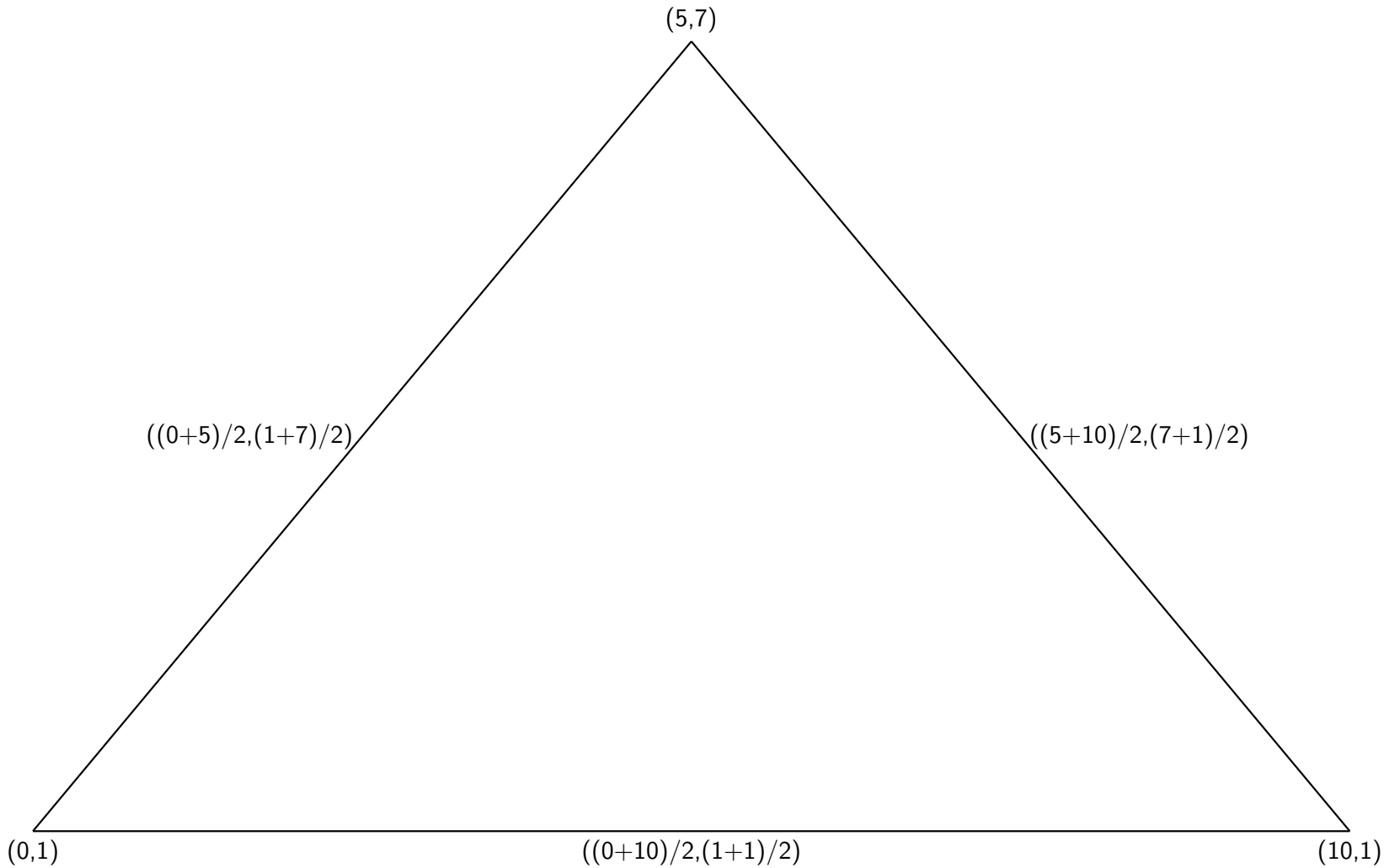
Zalety i wady rekurencji

Trójkąt Sierpińskiego - drugi sposób



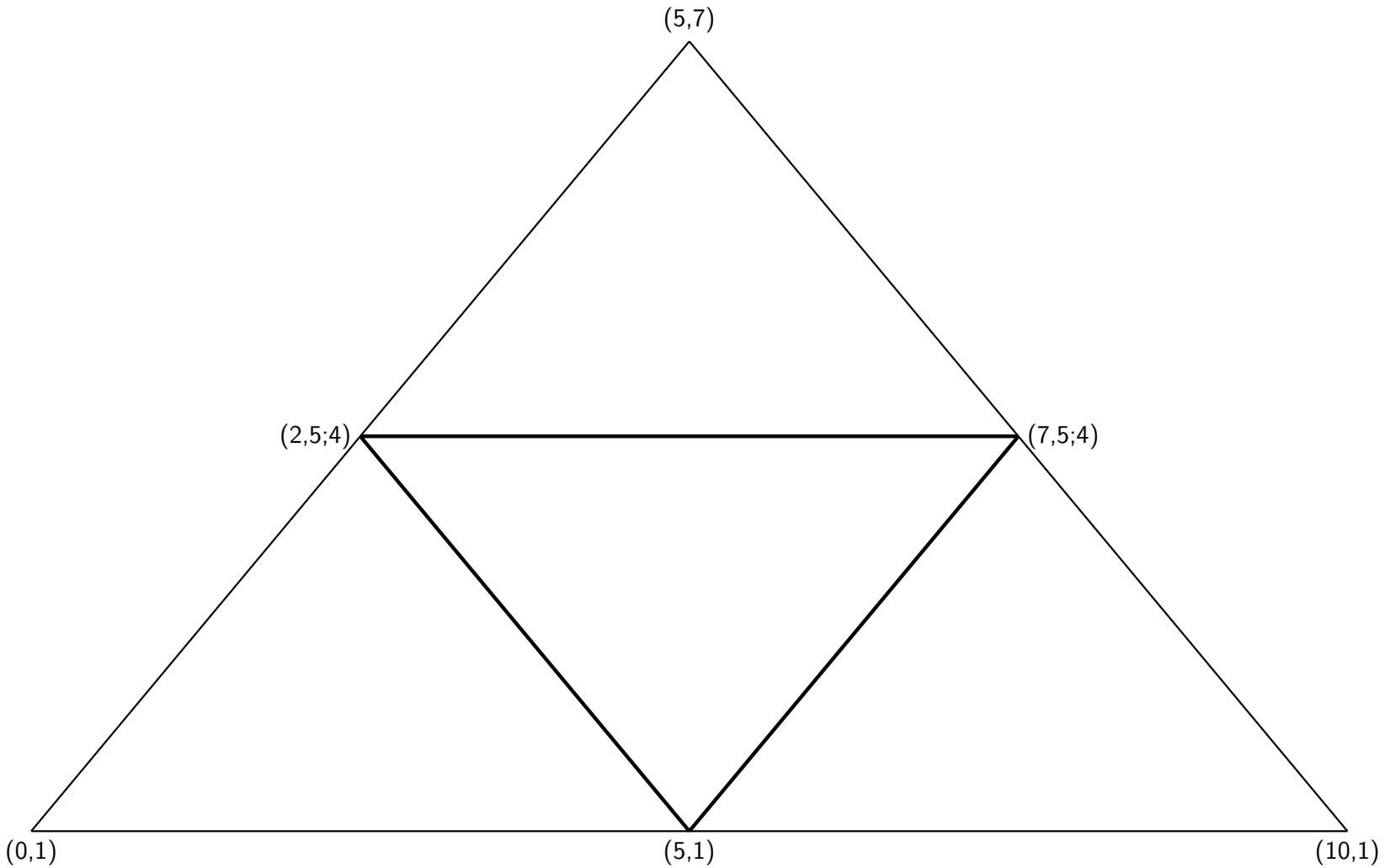
Zalety i wady rekurencji

Trójkąt Sierpińskiego - drugi sposób



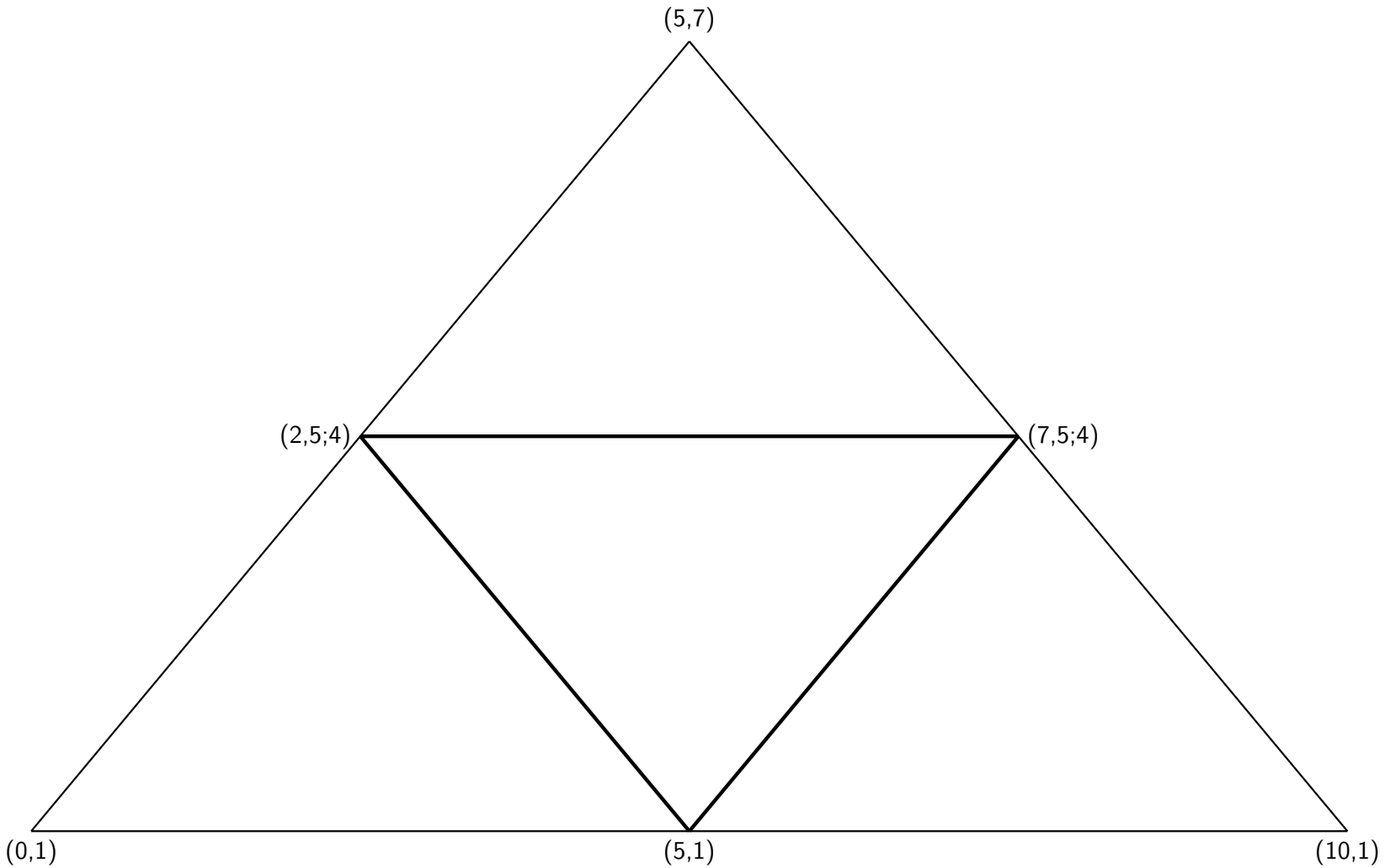
Zalety i wady rekurencji

Trójkąt Sierpińskiego - drugi sposób



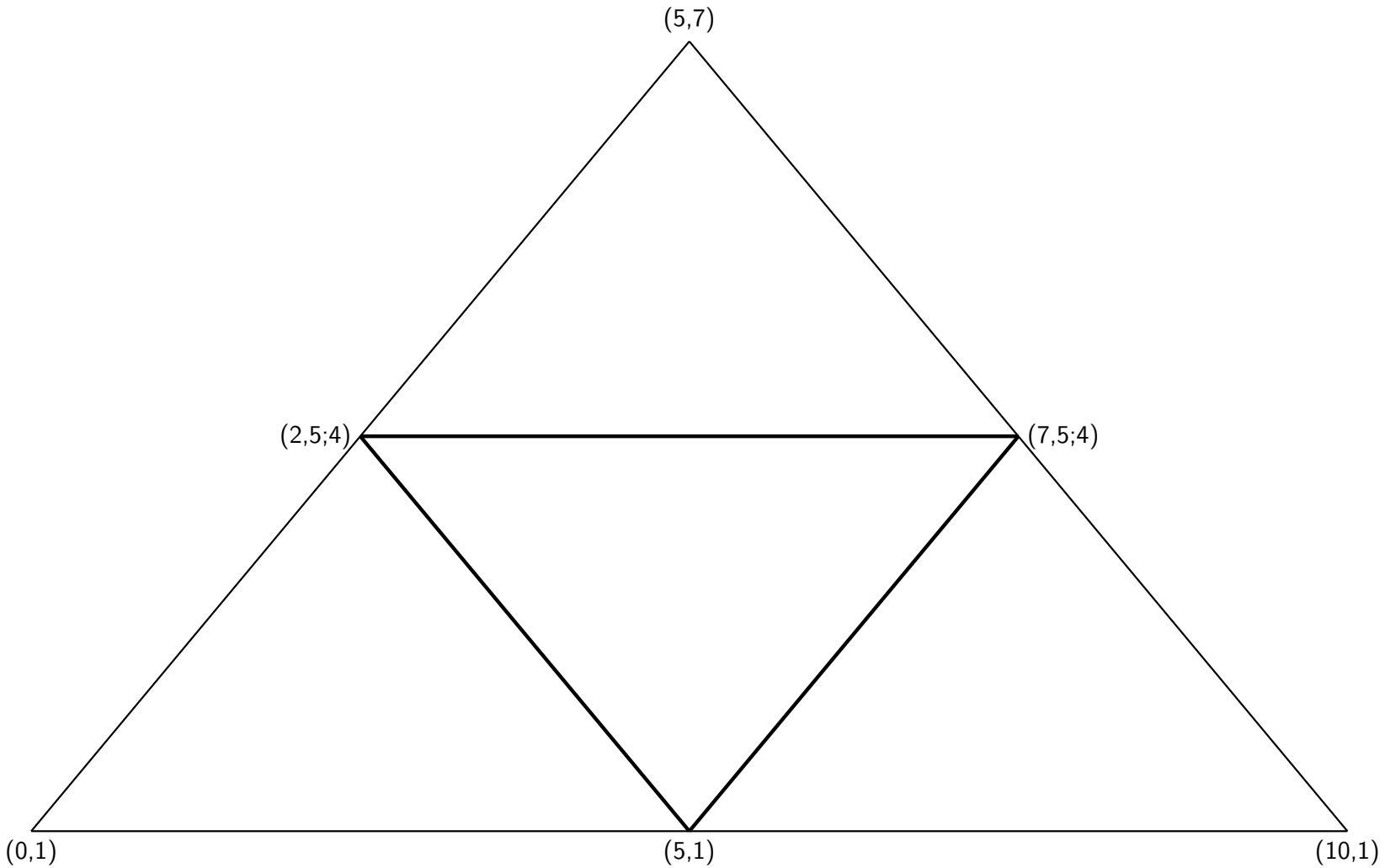
Zalety i wady rekurencji

Trójkąt Sierpińskiego - drugi sposób



Zalety i wady rekurencji

Trójkąt Sierpińskiego - drugi sposób



Zalety i wady rekurencji

Trójkąt Sierpińskiego - drugi sposób

```
#include<allegro.h>

void draw_triangle(int x, int y, int x1, int y1, int x2, int y2, int colour, unsigned char n)
{
        line(screen,(x+x1)/2,(y+y1)/2,(x1+x2)/2,(y1+y2)/2,colour);
        line(screen,(x1+x2)/2,(y1+y2)/2,(x+x2)/2,(y+y2)/2,colour);
        line(screen,(x+x1)/2,(y+y1)/2,(x+x2)/2,(y+y2)/2,colour);
        if(n--) {
                draw_triangle((x+x1)/2,(y+y1)/2,x1,y1,(x2+x1)/2,(y2+y1)/2,colour,n);
                draw_triangle(x,y,(x1+x)/2,(y1+y)/2,(x2+x)/2,(y2+y)/2,colour,n);
                draw_triangle((x+x2)/2,(y+y2)/2,(x1+x2)/2,(y1+y2)/2,x2,y2,colour,n);
        }
}
```

Zalety i wady rekurencji

Trójkąt Sierpińskiego - drugi sposób

```
int main(void) {
        int width=1336,height=768;
        if(allegro_init())
                allegro_message("allegro_init: %s\n",allegro_error);
        if(install_keyboard())
                allegro_message("install_keyboard: %s\n",allegro_error);
        set_color_depth(32);
        if(set_gfx_mode(GFX_AUTODETECT_FULLSCREEN,width,height,0,0))
                allegro_message("%s\n",allegro_error);
        height-=1;
        int colour = makecol(255,255,255);
        line(screen,width/2,0,0,height,colour);
        line(screen,0,height,width,height,colour);
        line(screen,width,height,width/2,0,colour);
        draw_triangle(width/2,0,0,height,width,height,colour,14);
        clear_keybuf();
        while(!keypressed());
        allegro_exit();
        return 0;
}
END_OF_MAIN()
```

Zalety i wady rekurencji

Trójkąt Sierpińskiego - drugi sposób

Zaprezentowany program różni się kilkoma szczegółami od swojego poprzednika. Oto najważniejsze z nich:

1. Funkcja `draw_triangle()` ma dodatkowy parametr, przez który przekazywany jest kolor rysowanego trójkąta. Dzięki temu jego wartość nie jest wyznaczana wewnątrz funkcji. To rozwiązanie można zastosować też w poprzednim programie.
2. W tej samej funkcji kilkukrotnie obliczane są środki boków trójkąta, najpierw przy rysowaniu trójkąta wewnętrznego, a następnie w wywołaniach rekurencyjnych. Można to zmienić zapamiętując wyniki obliczeń w zmiennych lokalnych, które będą następnie używane jako argumenty wywoływanych funkcji. Zwiększy to efektywność działania programu, ale wydłuży zapis funkcji `draw_triangle()`.
3. Pierwszy trójkąt rysowany jest na zewnątrz funkcji `draw_triangle()`, w funkcji `main()`.

Zalety i wady rekurencji

Efektywność rekurencji

Zastanówmy się nad efektywnością rekurencji. Każde wywołanie rekurencyjne wiąże się z utworzeniem na stosie ramki stosu. Jest to działanie, które jest czasochłonne, a ponadto wymaga poświęcenia pewnej ilości wolnej pamięci, a zatem wpływa negatywnie na efektywność funkcji rekurencyjnych. Im więcej takich wywołań, tym mniej efektywne są funkcje rekursywne w porównaniu z odpowiadającymi im funkcjami iteracyjnymi. Jest jeszcze jedna przyczyna, dla której stosowanie rekurencji nie zawsze jest wskazane. Zilustrujemy ją przy pomocy zagadnienia generowania kolejnych liczb ciągu Fibonacciego.

Zalety i wady rekurencji

Efektywność rekurencji

Ciąg Fibonacciego został odkryty przez średniowiecznego włoskiego matematyka Leonadra Fibonacciego, który za jego pomocą chciał opisać wzrost populacji królików. Choć do tego pierwotnego zagadnienia ten ciąg okazał się niewystarczający, to jednak wiele praktycznych zagadnień, również z dziedziny sztuki, można przy jego pomocy wyrazić w sposób formalny. Ciąg ten zdefiniowany jest następująco:

$$\mathit{fibonacci}(n) = \begin{cases} 0 & \text{jeśli } n = 0 \\ 1 & \text{jeśli } n = 1 \\ \mathit{fibonacci}(n-2) + \mathit{fibonacci}(n-1) \text{ dla } n \geqslant 2 \end{cases}$$

Liczba n w tym wzorze jest liczbą naturalną i określa położenie danego wyrazu ciągu Fibonacciego. Wszystkie liczby należące do tego ciągu są liczbami naturalnymi. Sama postać definicji ciągu zachęca do tego, aby zaimplementować funkcję wyznaczającą kolejne wyrazy takiego ciągu w postaci rekurencyjnej. Kod takiej funkcji jest przedstawiony na następnym slajdzie.

Zalety i wady rekurencji

Ciąg Fibonacciego

```
1  unsigned int get_fibonacci_number(unsigned char order)
2  {
3      if(order==0)
4          return 0;
5      if(order==1)
6          return 1;
7      return get_fibonacci_number(order-1)+get_fibonacci_number(order-2);
8  }
```

Zalety i wady rekurencji

Ciąg Fibonacciego

Zapis tej funkcji na pewno jest zwięzły i zrozumiały, gdyż wprost wynika z definicji ciągu. Analiza działania tej funkcji jest nieco trudniejsza z uwagi na dwa wywołania rekurencyjne występujące w wierszu nr 7. W takich przypadkach należy pamiętać, że wartości wyrażeń są wyznaczane „od lewej do prawej", a więc najpierw będzie wykonywane „lewe" wywołanie i dopiero kiedy ono się zakończy rozpocznie się wywołanie „prawe", a po jego zakończeniu będzie można wyznaczyć całość wyrażenia. Stwórzmy drzewo rekurencji dla tej funkcji wywołanej z parametrem `order` równym `4`. Dla uproszczenia na tym drzewie nazwa funkcji zostanie zastąpiona literą `f`.

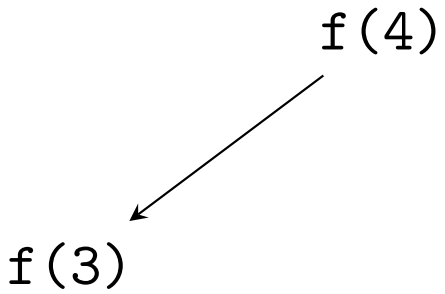
Zalety i wady rekurencji

Ciąg Fibonacciego

`f(4)`

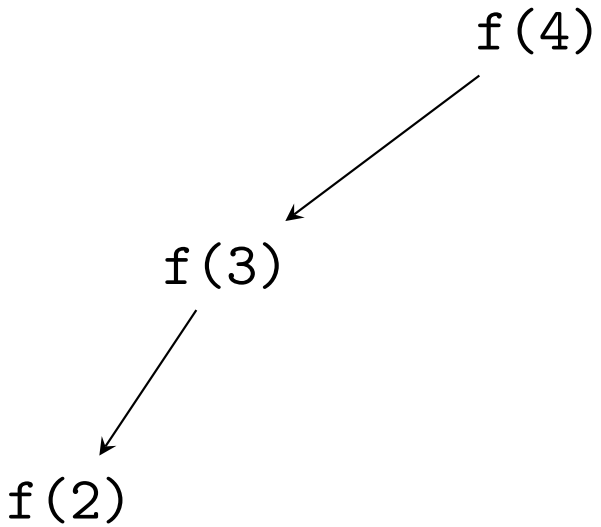
Zalety i wady rekurencji

Ciąg Fibonacciego



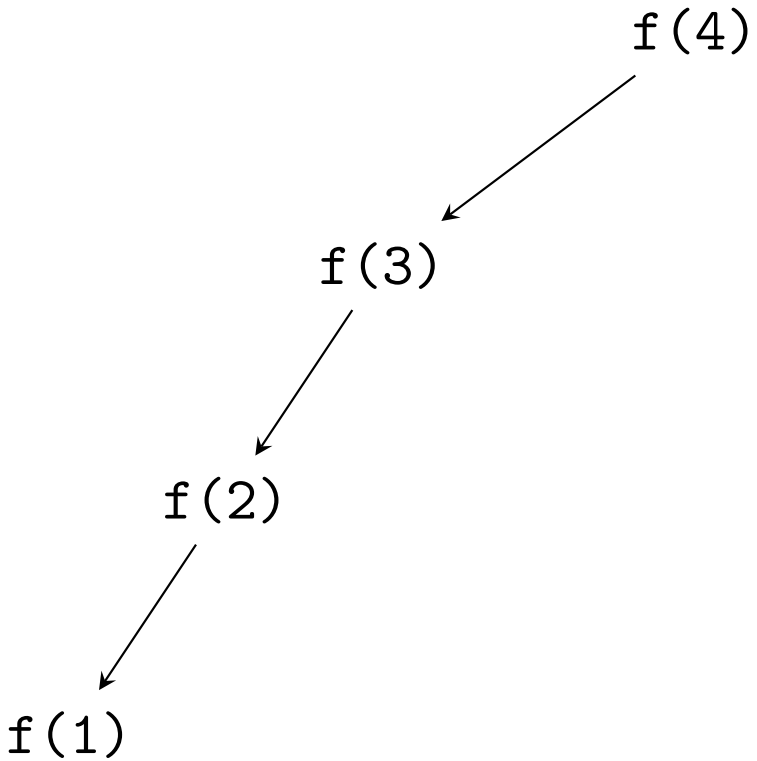
Zalety i wady rekurencji

Ciąg Fibonacciego



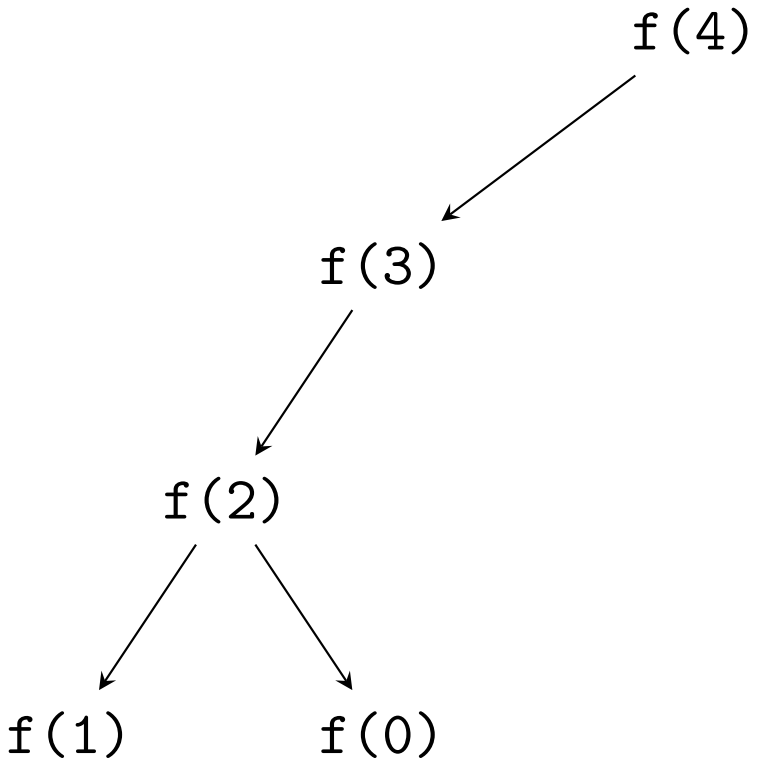
Zalety i wady rekurencji

Ciąg Fibonacciego



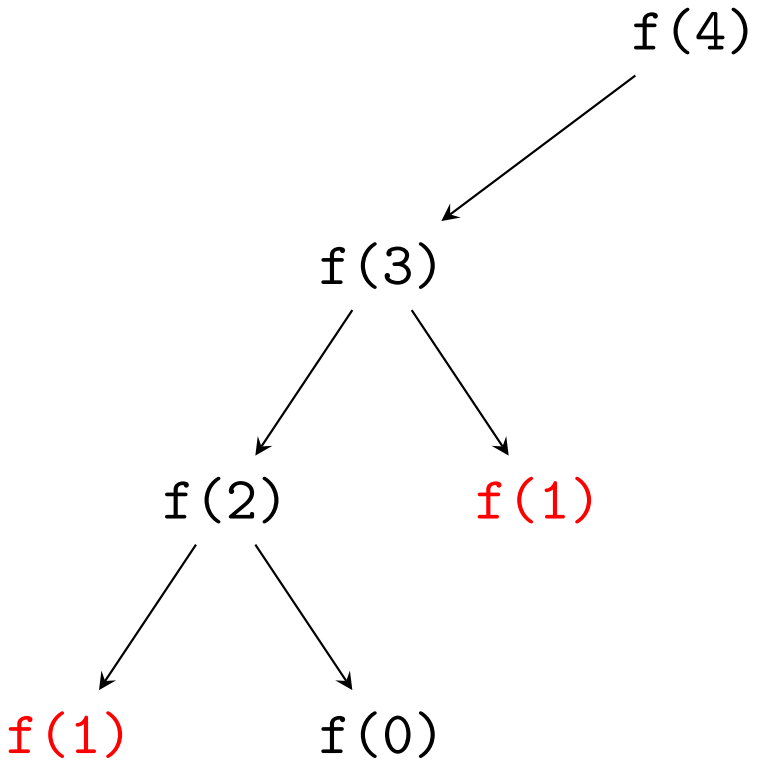
Zalety i wady rekurencji

Ciąg Fibonacciego



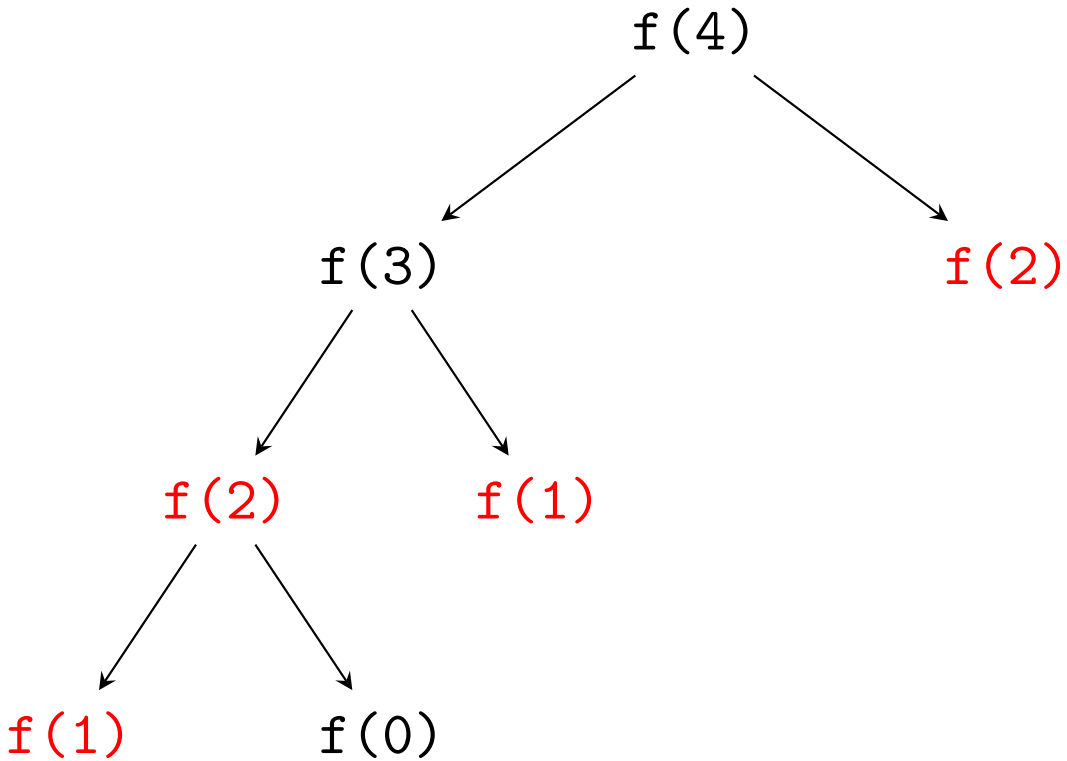
Zalety i wady rekurencji

Ciąg Fibonacciego



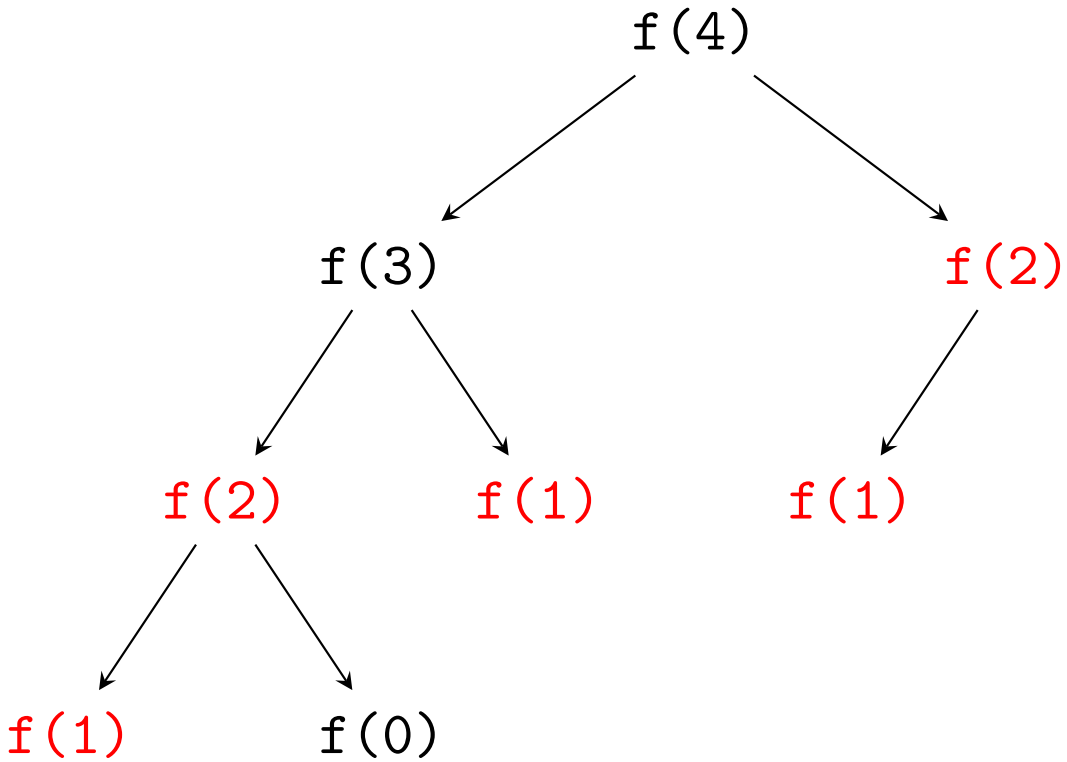
Zalety i wady rekurencji

Ciąg Fibonacciego



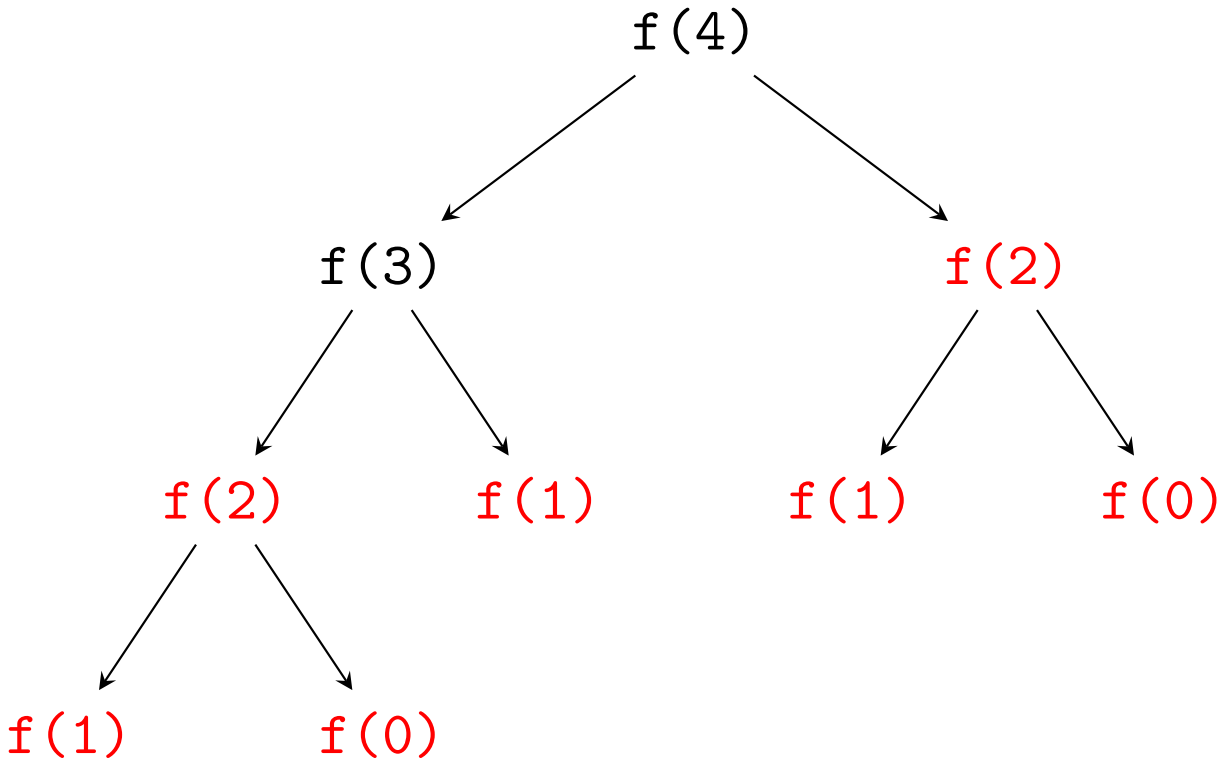
Zalety i wady rekurencji

Ciąg Fibonacciego



Zalety i wady rekurencji

Ciąg Fibonacciego



Zalety i wady rekurencji

Ciąg Fibonacciego

Analizując to drzewo możemy się przekonać, że wiele wywołań rekurencyjnych liczy tę samą wartość (zostały one zaznaczone w drzewie na czerwono). To wpływa na jej efektywność mierzoną jako czas działania i ilość zajętej pamięci. Wyznaczenie pierwszych 45 wyrazów ciągu Fibonacciego tą funkcją okazuje się trwać kilka sekund, nawet na komputerach o dużej mocy obliczeniowej.
Powstaje pytanie, czy można stworzyć wersję tej funkcji, która nie wykonuje takich nadmiarowych obliczeń. Odpowiedź na nie jest pozytywna. Wystarczy aby taka funkcja, zgodnie z definicją ciągu Fibonacciego, pamiętała tylko wartości dwóch ostatnich wyrazów ciągu. To wystarcza, aby wyznaczyć wartość następnego. Dodatkowo ta funkcja nie musi być rekurencyjna, wystarczy że będzie iteracyjna. Jej kod znajduje się na następnym slajdzie.

Zalety i wady rekurencji

Ciąg Fibonacciego - wersja iteracyjna

```
unsigned int get_fibonacci_number(unsigned char order)
{
    unsigned int current = 0, next = 1, result = 0;
    unsigned int i;
    for(i=0;i<order;i++) {
        result = current + next;
        current = next;
        next = result;
    }
    return current;
}
```

Zalety i wady rekurencji

Ciąg Fibonacciego

Kod tej funkcji nie jest aż tak czytelny, jak jej wersji rekurencyjnej, ale mimo to jest i tak stosunkowo prosty do zrozumienia. Zmienna `next` przechowuje wartość wyrazu następującego po tym obliczanym, a zmienna `current` przechowuje wartość wyrazu obliczanego. Aby obliczyć kolejny wyraz należy zsumować te dwie wartości i zapisać wynik w `next`, wcześniej zapamiętując wartość tej zmiennej w `current`. Obliczenia dokonywane są w pętli, której liczba wykonań zależy od pozycji w ciągu wyrazu, którego wartość jest wyznaczana. Ta funkcja również liczy nadmiarową wartość, ale tylko jedną. Jest to wartość wyrazu następnego po tym, który chcemy, aby był wyznaczony. Obliczenie przy jej pomocy 45 wyrazów ciągu na tym samym komputerze, na którym były one obliczane przy pomocy wersji rekurencyjnej tej funkcji trwa o wiele krócej. Dodatkowo nie jest zużywana wolna pamięć na stosie na wywołania rekurencyjne.

Zalety i wady rekurencji

Ciąg Fibonacciego

Na mocy teorii złożoności można wykazać, że algorytm według którego działa funkcja rekurencyjna licząca wyrazy ciągu Fibonacciego jest algorytmem wykładniczym. Dla czwartego wyrazu ciągu funkcja tworzy drzewo rekurencji o czterech poziomach. Dla trzeciego wyrazu były by to trzy poziomy. Ponieważ każdy węzeł tego drzewa ma co najwyżej dwóch potomków, to liczba jego elementów będzie maksymalnie wynosiła 2^{order}, gdzie `order` jest pozycją liczonego elementu w ciągu. Dodatkowo im większe będzie drzewo wywołań, tym więcej będzie w nim instancji[1] funkcji, które liczą tę samą wartość. Wersja iteracyjna funkcji jest z kolei implementacją algorytmu liniowego. Do jej realizacji jest użyta tylko jedna pętla, a więc czas jej działania jest dużo krótszy, a zajętość pamięci stała, w porównaniu z wersją rekurencyjną.

[1] Instancja funkcji, to inaczej jej wywołanie z konkretną wartością argumentu.

Zalety i wady rekurencji

Symbol Newtona

Wyznaczenie wartości używanego w kombinatoryce symbolu Newtona wymaga użycia dosyć skomplikowanego wzoru. Jednakże podane niżej własności tego działania wskazują, że można wyznaczyć jego wartość w sposób rekurencyjny:

$$\binom{n}{k} = \begin{cases} 1 & \text{dla } k = 0 \ \texttt{lub} \ k = n \\ n & \text{dla } k = 1 \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{dla pozostałych przypadków} \end{cases}$$

Na następnym slajdzie znajduje się kod źródłowy funkcji, która została zaimplementowana na bazie tej formuły matematycznej, podobnie jak opisywana wcześniej funkcja obliczająca wyrazy ciągu Fibonacciego.

Zalety i wady rekurencji

Symbol Newtona

```
unsigned int get_newton_symbol_value(unsigned char n, unsigned char k)
{
    if(k==0||k==n)
        return 1;
    if(k==1)
        return n;
    return get_newton_symbol_value(n-1,k-1)+get_newton_symbol_value(n-1,k);
}
```

Zalety i wady rekurencji

Symbol Newtona

Podobnie jak funkcja obliczająca wyrazy ciągu Fibonacciego, funkcja wyznaczająca symbol Newtona nie jest najbardziej efektywnym rozwiązaniem tego problemu. Oprócz nadmiarowych wywołań rekurencyjnych, czyli takich, które liczą te same wartości, w przypadku tej funkcji powstaje niebezpieczeństwo przekroczenia zakresu typu `unsigned int` i uzyskania nieprawidłowych wyników. Stworzenie iteracyjnego odpowiednika tej funkcji nie jest już takie proste. Wymaga ono użycia tablicy dwuwymiarowej i dwóch pętli, z których jedna będzie zagnieżdżona w drugiej.

Częste błędy

Używając rekurencji można popełnić wiele błędów. Najczęstszą ich kategorią są błędy polegające na złym określeniu warunków zakończenia rekursji. Takie błędy mogą mieć charakter matematyczny lub informatyczny. Pierwszy występuje wtedy, gdy błędnie określimy przypadki brzegowe i sposób podziału problemu, co spowoduje, że warunek zakończenia rekurencji nigdy nie zostanie osiągnięty. Drugi związany jest z niewłaściwym doborem typów zmiennych lub złym zapisem kodu funkcji rekurencyjnej. W przypadku typów danych może nastąpić przekroczenie ich zakresu i wtedy warunek zakończenia rekursji również nie będzie nigdy spełniony. Podobny efekt może wystąpić przy złym doborze instrukcji wykonywanych w funkcji. Szczególnie „zdradzieckie" są operatory inkrementacji i dekrementacji, w obu odmianach (przyrostkowej i przedrostkowej), dlatego należy unikać stosowania ich do argumentów wywołań rekurencyjnych funkcji. Czasem błędy mogą być spowodowane interpretacją zapisu kodu funkcji przez kompilator, dlatego należy zwracać uwagę na generowane przez niego ostrzeżenia.

Częste błędy

Rekurencja ma wiele wspólnego z indukcją matematyczną. Poprawności algorytmów rekurencyjnych dowodzi się stosując właśnie tę technikę. Z błędami o charakterze informatycznym można sobie poradzić stosując takie narzędzia jak debuggery i zachowując ostrożność pisząc funkcje rekurencyjne. W świecie idealnym błędy w funkcjach rekurencyjnych powodowałby, że nigdy one by się nie kończyły, ciągle realizowane byłyby kolejne ich wywołania. W świecie rzeczywistym funkcje rekurencyjne, które są błędnie napisane kończą się z powodu wyczerpania wolnego miejsca na stosie, potrzebnego do tworzenia kolejnych ramek stosu. Taki błąd objawia się zazwyczaj komunikatem o *przepełnieniu stosu* (ang. *stack overflow*). Taki komunikat można uzyskać także w przypadku poprawnie napisanych funkcji rekurencyjnych, ale uruchamianych w środowiskach, gdzie programy dysponują małym stosem sprzętowym. W takich przypadkach należ rozważyć zamianę przekazywania argumentów przez wartość na przekazywanie przez stałą lub przez wskaźnik.

Podsumowanie

Metoda dziel i zwyciężaj jest potężnym narzędziem do tworzenia algorytmów rekurencyjnych. Z kolei technika rekurencji pozwala je zaimplementować w zwięzłej postaci. Dowiedzieliśmy się, że nie zawsze rekurencyjna realizacja takich algorytmów jest efektywna. Czasem lepiej poświęcić trochę czasu na refleksję i znaleźć iteracyjną lub nawet prostszą wersję tego algorytmu. Wróćmy na chwilę do pierwszego z opisanych problemów wież Hanoi. Z lektury wspomnianej książki pt. „Matematyka konkretna" dowiemy się, że otrzymane równania rekurencyjne sprowadzają się do wzoru $T_n = 2^n - 1$, gdzie n jest liczbą krążków w wieży. To oznacza, że nie potrzebna jest nawet pętla do obliczenia tej wartości. Wystarczy funkcja obliczającą wartość tego wzoru, a wiec działająca w stałym czasie.

Podsumowanie

Okazuje się jednak, że nie powinniśmy rezygnować z poznania techniki rekurencji. Jej znajomość jest nieodzowna dla każdego dobrego programisty. Istnieją algorytmy, których nie da się wyrazić prosto w postaci iteracyjnej. Ewentualne próby pozbycia się rekurencji z ich implementacji kończyłby się koniecznością bezpośredniego stworzenia dla nich stosu, np. w formie dynamicznej struktury danych, co jest dosyć czasochłonnym zajęciem w porównaniu z zapisaniem funkcji w postaci rekurencyjnej. Niektóre proste funkcje rekurencyjne mogą zostać automatycznie zastąpione iteracyjnymi odpowiednikami przez kompilator, co także przemawia na korzyść stosowania rekurencji. Ostatnim argumentem za stosowaniem rekursji jest istnienie wydajnych algorytmów rekurencyjnych, które najłatwiej zaimplementować w postaci funkcji rekurencyjnych, a ich iteracyjne odpowiedniki są bardziej skomplikowane w zapisie i równie wydajne. Przykładem takiego algorytmu jest *algorytm szybkiego sortowania* (ang. *QuickSort*), który poznamy na innym wykładzie.

Pytania

?

KONIEC

Dziękuję Państwu za uwagę!