Wstęp do informatyki

Wykład 5

Uniwersytet Wrocławski

Plan na dziś

1. Rekurencja oraz … kiedy NIE należy jej stosować:

   Rozważane problemy: potęgowanie,
   wyznaczanie liczb Fibonacciego,
   współczynników dwumianowych,…

2. Programowanie dynamiczne (jako alternatywa dla rekurencji).

REKURENCJA

… a recursive definition defines objects in terms of "**previously** defined" objects of the class to define …

wikipedia

Potęgowanie

Definicja rekurencyjna:

$$a^b = \begin{cases} 1 & b = 0 \\ a^{b-1} * a & b > 0 \end{cases}$$

- wartość a^b definiujemy w oparciu o a^{b-1}, tzn. o wartość **tej samej** funkcji dla **mniejszego** argumentu

$$f(a,b) = \begin{cases} 1 & b = 0 \\ f(a,b-1) * a & b > 0 \end{cases}$$

Potęgowanie cd.

Specyfikacja:

Wejście: a, b – liczby naturalne

Wyjście: a^b

Funkcja rekurencyjna: „wywołuje sama siebie”

NIEREKURENCYJNIE

```
int pot(int a, int b)
{ int i, rez;
   rez = 1;
   for(i=0; i<b; i++) rez = rez * a;
   return rez;

}
```

REKURENCYJNIE

```
int pot(int a, int b)
{ if (b==0) return 1;
   return a * pot(a, b-1);

}
```

$$a^b = \begin{cases} 1 & b = 0 \\ a^{b-1} * a & b > 0 \end{cases}$$

Potęgowanie cd.

Specyfikacja:

Wejście: a, b – liczby naturalne

Wyjście: a^b

Funkcja rekurencyjna: „wywołuje sama siebie”

NIEREKURENCYJNIE

```
def pot(a, b):
    rez = 1;
    for i in range(b):
        rez = rez * a
    return rez
```

REKURENCYJNIE

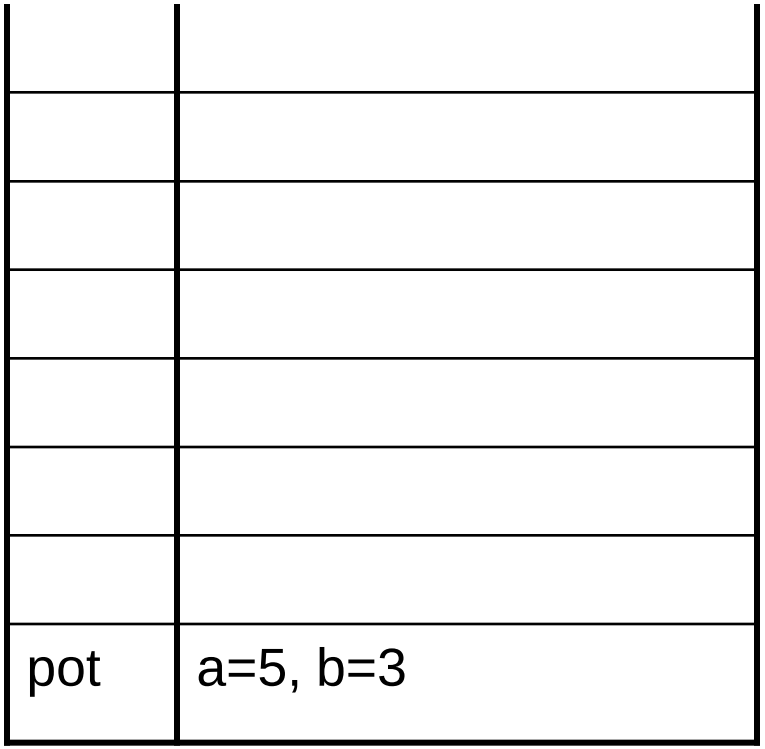
```
def pot(a, b):
    if b==0:
        return 1
    return a * pot(a, b-1)
```

$$a^b = \begin{cases} 1 & b = 0 \\ a^{b-1} * a & b > 0 \end{cases}$$

Potęgowanie cd

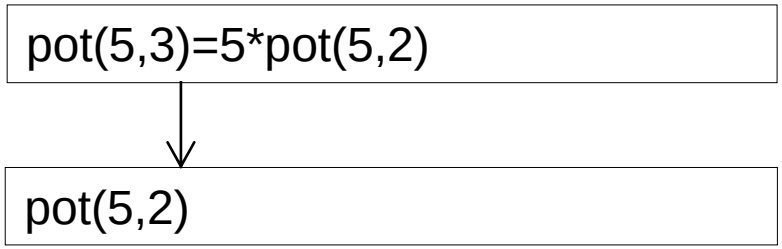
Co się dzieje na stosie wywołań?

pot(5,3)



Potęgowanie cd

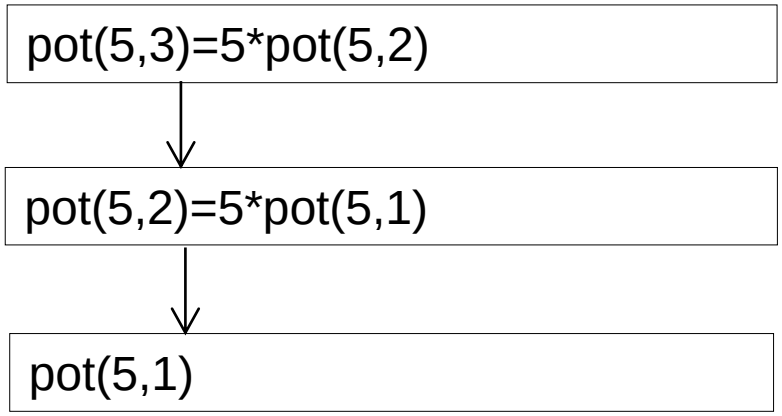
Co się dzieje na stosie wywołań?



pot	a=5, b=2
pot	a=5, b=3

Potęgowanie cd

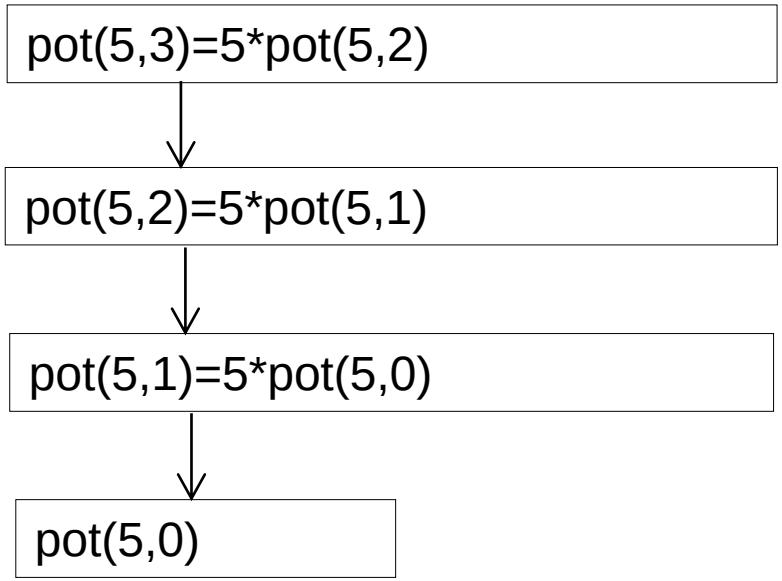
Co się dzieje na stosie wywołań?



pot	a=5, b=1
pot	a=5, b=2
pot	a=5, b=3

Potęgowanie cd

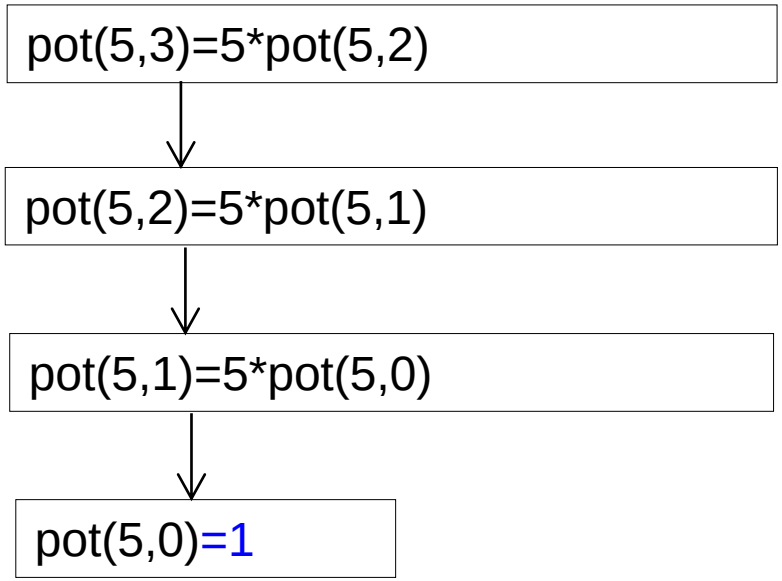
Co się dzieje na stosie wywołań?



pot	a=5, b=0
pot	a=5, b=1
pot	a=5, b=2
pot	a=5, b=3

Potęgowanie cd

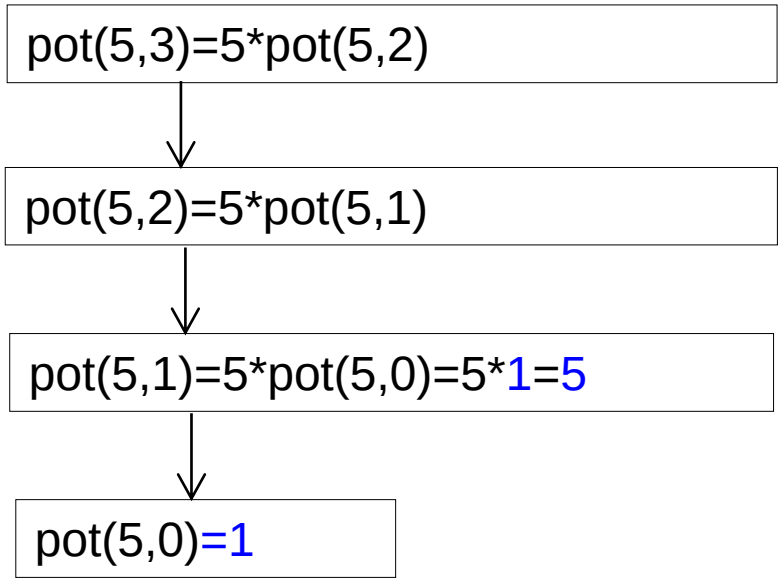
Co się dzieje na stosie wywołań?



pot	a=5, b=0
pot	a=5, b=1
pot	a=5, b=2
pot	a=5, b=3

Potęgowanie cd

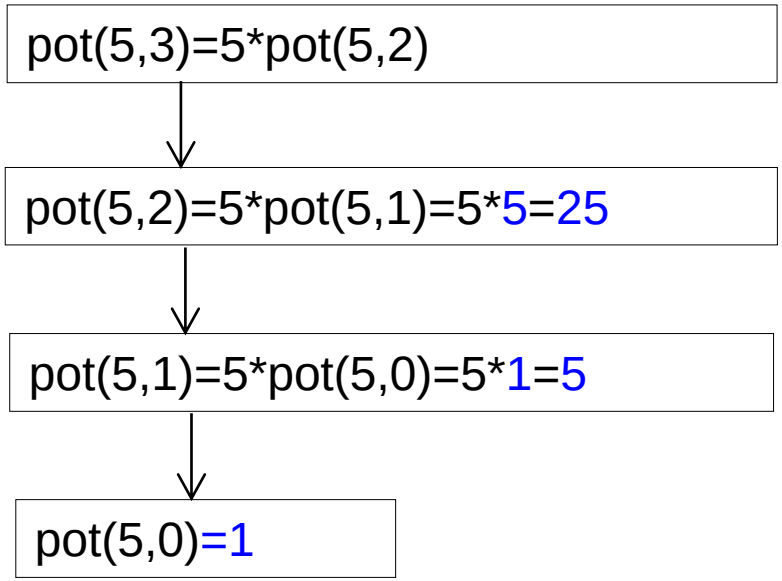
Co się dzieje na stosie wywołań?

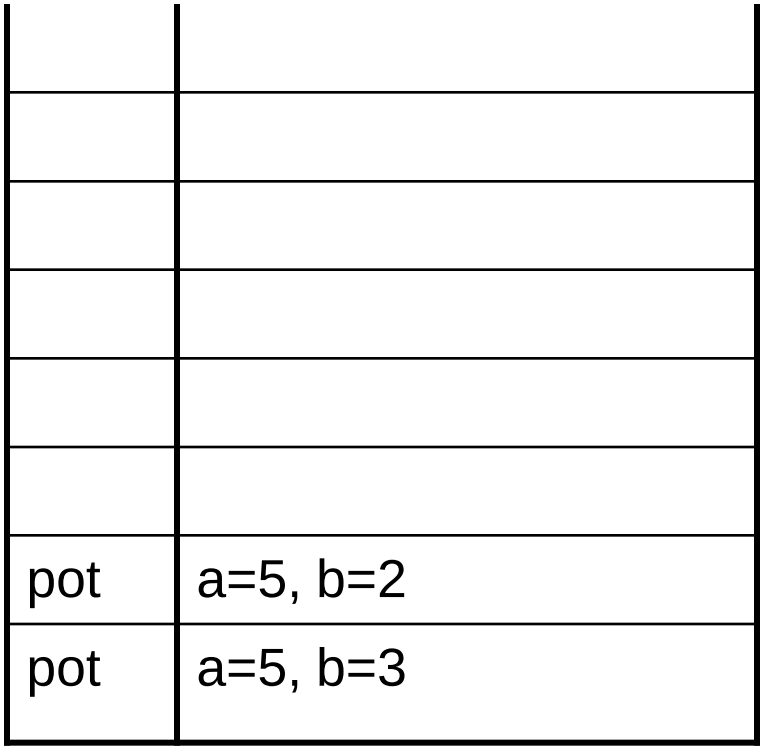


pot	a=5, b=1
pot	a=5, b=2
pot	a=5, b=3

Potęgowanie cd

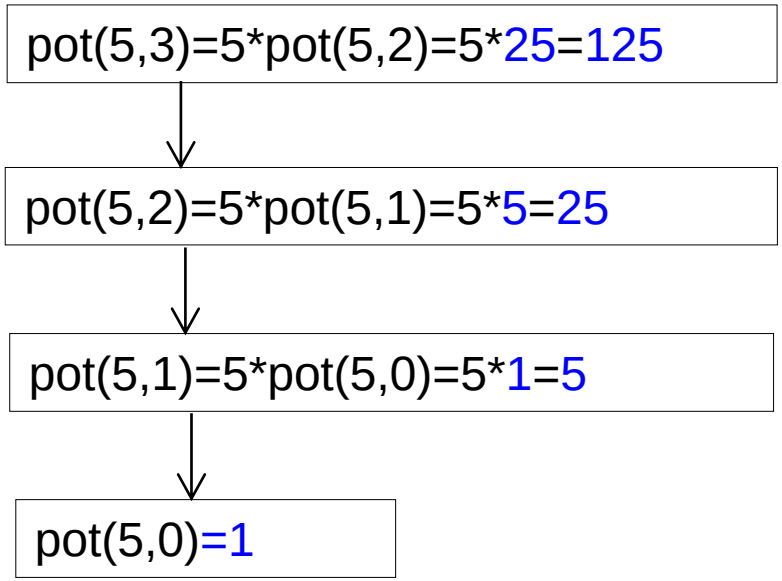
Co się dzieje na stosie wywołań?

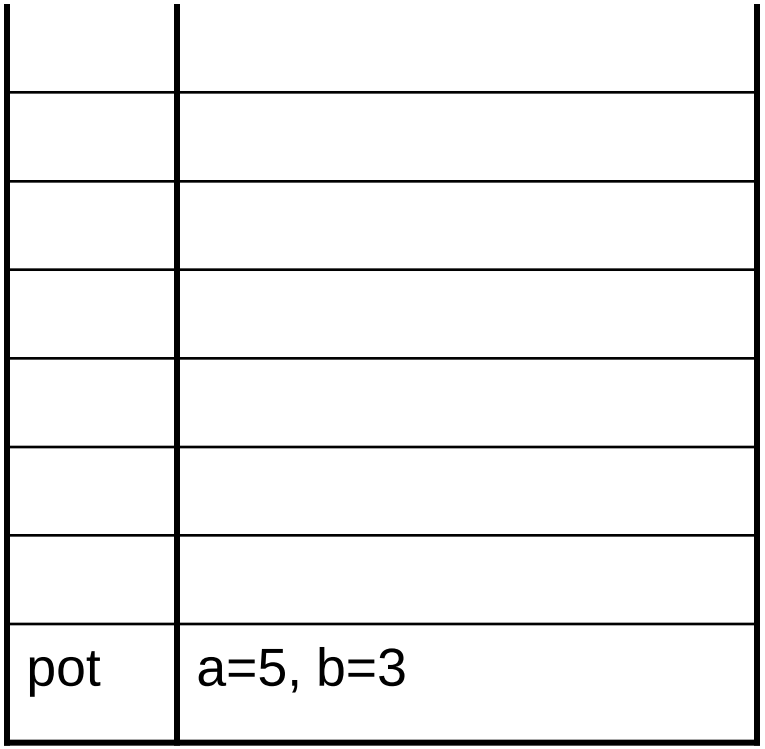




Potęgowanie cd

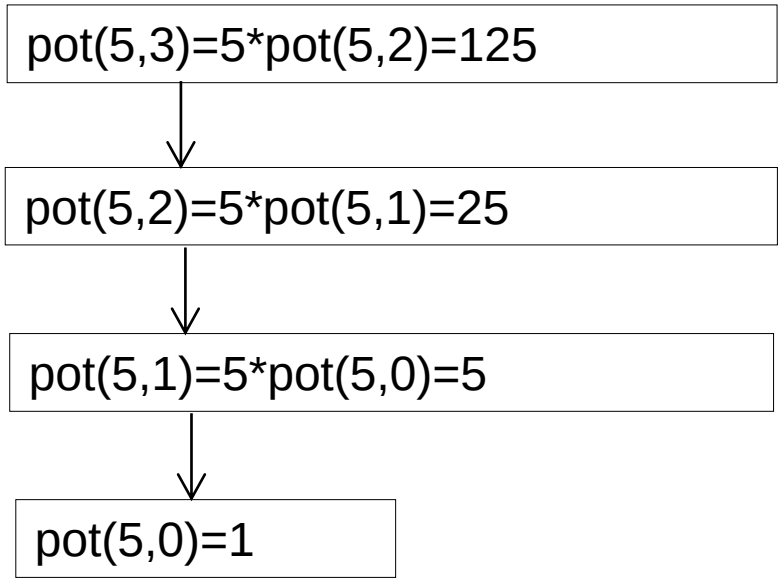
Co się dzieje na stosie wywołań?

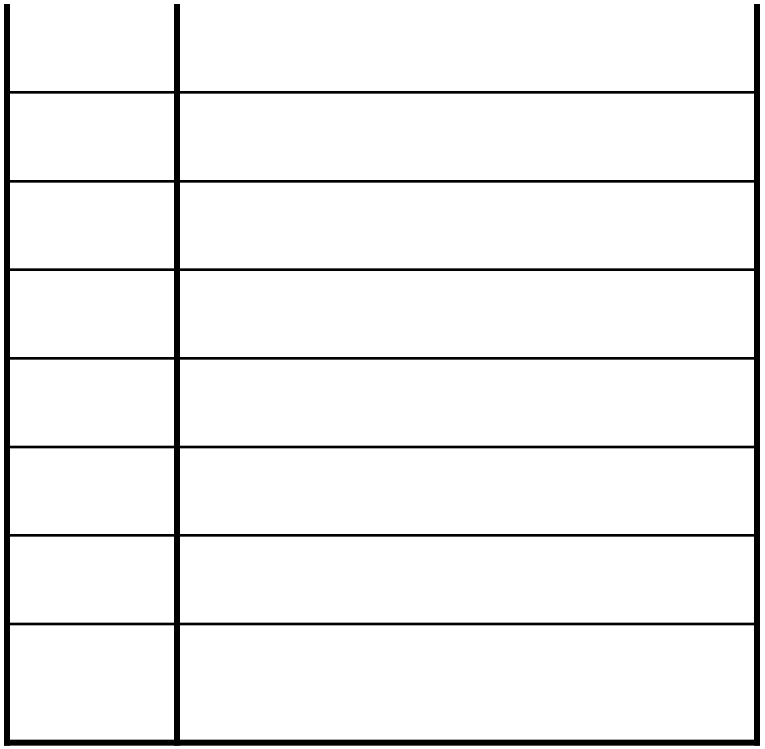




Potęgowanie cd

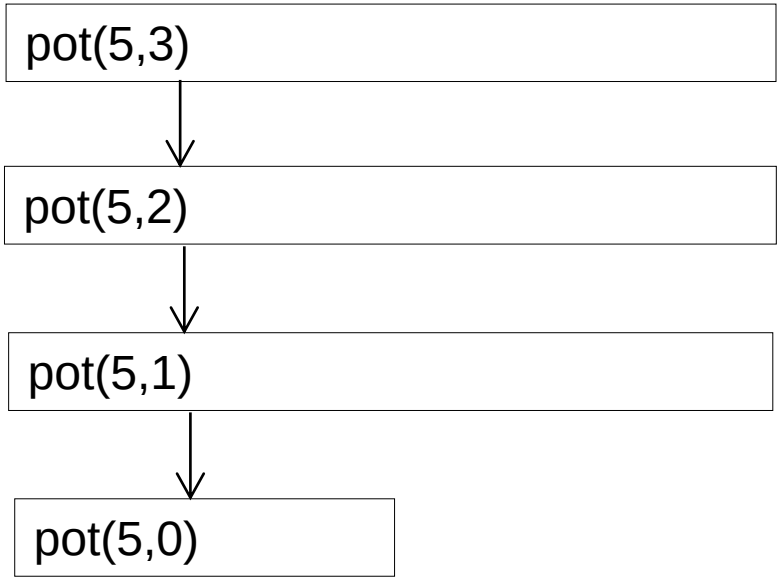
Co się dzieje na stosie wywołań?





Drzewo wywołań (rekurencyjnych)

Drzewo wywołań (rekurencyjnych) dla pot(5,3):



Krawędź („strzałka”) prowadzi od funkcji wywołującej do funkcji wywoływanej.

Skąd nazwa drzewo? O tym później…

Algorytm Euklidesa rekurencyjnie

Relacja rekurencyjna:

$$\mathrm{nwd}(n,m) = \begin{cases} n & m = 0 \\ \mathrm{nwd}(m, n \bmod m) & n \geq m \\ \mathrm{nwd}(m,n) & n < m \end{cases}$$

NIEREKURENCYJNIE

```
int nwd(int n, int m)
{ if (n<m) {
    k = m; m = n; n = k; }
  while (m!=0) {
    k = n % m;
    n = m;
    m = k;
  }
  return n;
}
```

REKURENCYJNIE

```
int nwd(int n, int m)
{ if (m==0) return n;
  if (m>n) return nwd(m, n);
  return nwd(m, n % m);
}
```

Algorytm Euklidesa rekurencyjnie

Relacja rekurencyjna:

$$\text{nwd}(n,m) = \begin{cases} n & m = 0 \\ \text{nwd}(m, n \bmod m) & n \geq m \\ \text{nwd}(m,n) & n < m \end{cases}$$

NIEREKURENCYJNIE

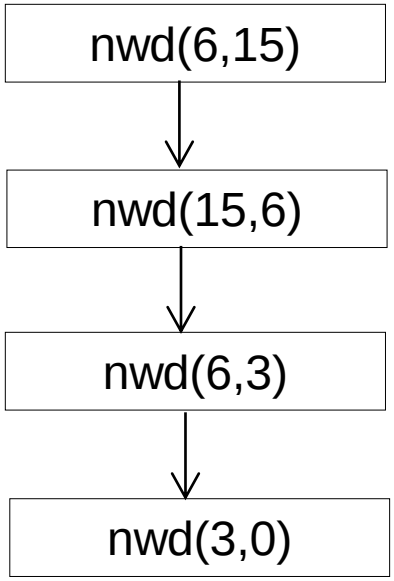
```
def nwd(n,m):
    if n<m:
        k = m
        m = n
        n = k
    while m!=0:
        k = n % m
        n =m
        m =k
    return n
```

REKURENCYJNIE

```
def nwd(n,m):
    if m==0:
        return n
    if m>n:
        return nwd(m, n)
    return nwd(m, n % m)
```

Algorytm Euklidesa rekurencyjnie

Drzewo wywołań:




```
int nwd(int n, int m)
{ if (m==0) return n;
  if (m>n) return nwd(m, n);
  return nwd(m, n % m);
}
```

```
def nwd(n,m):
    if m==0:
        return n
    if m>n:
        return nwd(m, n)
    return nwd(m, n % m)
```

Silnia

Definicja rekurencyjna:

$$0! = 1$$
$$n! = n \cdot (n-1)! \quad \text{for} \quad n > 0$$

$$\text{silnia}(0) = 1$$
$$\text{silnia}(n) = n \cdot \text{silnia}(n-1) \quad \text{for} \quad n > 0$$

Iteracyjnie:

```
int sil(int n)
{ int i, res=1;
  for(i=2; i<=n; i++) res*= i;
  return res;
}
```

```
def sil(n):
    res=1
    for i in range(2,n+1):
        res*=i
    return res
```

Rekurencyjnie:

```
int sil(int n)
{
  if (!n) return 1;
  return n * sil(n-1);
}
```

```
def sil(n):

    if n==0:
        return 1
    return n * sil (n-1)
```

Szybkie potęgowanie

Idea szybkiego potęgowania

Idea wyznaczania a^b:

Jeśli b jest naturalną potęgą dwójki:

- $a^2 = a * a$: jedno mnożenie,
- $a^4 = a^2 * a^2$: dwa mnożenia (a^2 liczymy tylko jeden raz),
- $a^8 = a^4 * a^4$: trzy mnożenia
- itd.

Przykład

a^{1024} : 10 mnożeń (zamiast 1024)!

Idea szybkiego potęgowania

Gdy b nie jest potęgą dwójki…

Obserwacja:

Każdą naturalną potęgę a można rozbić na „niewielką” liczbę potęg z wykładnikiem będącym potęgą dwójki.

Przykład

$a^{49}=a^{32}a^{16}a^{1}$

$a^{90}=a^{64}a^{16}a^{8}a^{2}$

Szybkie potęgowanie

Obserwacja

Niech $b_k\, b_{k-1} \ldots b_0$ to binarna reprezentacja liczby b. Wówczas $a^b = a^{b}{}_0$

$$a^{b} = a^{b_0 \cdot 2^0 + b_1 \cdot 2^1 + \quad + b_k \cdot 2^k} = a^{b_0 \cdot 2^0} \cdot a^{b_1 \cdot 2^1} \cdot \quad \cdot a^{b_k \cdot 2^k}$$

Obserwacja inaczej

Aby uzyskać a^b wystarczy wymnożyć potęgi a odpowiadające jedynkom w binarnej reprezentacji liczby b.

Szybkie potęgowanie

Obserwacja inaczej

Aby uzyskać a^b wystarczy wymnożyć potęgi a odpowiadające jedynkom w binarnej reprezentacji liczby b.

Algorytm

1. wczytaj a, b
2. rez ← 1
3. c ← a
4. dopóki b>0
   - Jeśli b % 2 =1:
     rez ← rez * c
   - b ← b / 2
   - c ← c * c
5. wypisz rez

Uwagi:

- zmienna **c** przechowuje w kolejnych iteracjach $a, a^2, a^4, a^8, \ldots$
- instrukcja „b ← b / 2” usuwa z b najmniej znaczący bit (**dzielenie całkowite!!!**)
- b % 2 to wartość najmniej znaczącego bitu b
- w rez przechowujemy „dotychczas domnożone potęgi liczby a

Szybkie potęgowanie

Algorytm:

1. wczytaj a, b
2. rez ← 1
3. dopóki b>0
   - Jeśli b % 2 =1:
     rez ← rez * a
   - b ← b / 2
   - a ← a * a
4. wypisz rez

Implementacja:

```
int pot(int a, int b)
{ int rez;
  rez = 1;
  while (b>0) {
    if (b%2) rez = rez * a;
    b = b / 2;
    a = a * a;
  }
  return rez;
}
```

Komentarze:

- Skomplikowane? …
- Jak wykazać poprawność (formalnie)?

Szybkie potęgowanie

Algorytm:

1. wczytaj a, b
2. rez ← 1
3. dopóki b>0
   - Jeśli b % 2 =1:
     rez ← rez * a
   - b ← b / 2
   - a ← a * a
4. wypisz rez

Implementacja:

```
def pot(a, b):
    rez = 1
    while b>0:
        if b%2:
            rez = rez * a
        b = b // 2
        a = a * a
    return rez
```

Komentarze:

- Skomplikowane? …
- Jak wykazać poprawność (formalnie)?

Szybkie potęgowanie… rekurencyjnie

Obserwacja (jeszcze) inaczej

Dla nieujemnych całkowitych *a* i *b* zachodzi:

$$a^b = \begin{cases} 1 & \text{gdy } b = 0 \\ (a^2)^{b/2} & \text{gdy } b > 0, \text{parzyste} \\ (a^2)^{(b-1)/2} * a & \text{gdy } b > 0, \text{nieparzyst e} \end{cases}$$

Szybkie potęgowanie… rekurencyjnie

Obserwacja (jeszcze) inaczej

Dla nieujemnych całkowitych *a* i *b* zachodzi:

$$a^b = \begin{cases} 1 & \text{gdy } b = 0 \\ (a^2)^{b/2} & \text{gdy } b > 0, \text{parzyste} \\ (a^2)^{(b-1)/2} * a & \text{gdy } b > 0, \text{nieparzyst e} \end{cases}$$

Dowód poprawności:

- b = 0: oczywiste
- b > 0, parzyste

  $a^b = a^{2*(b/2)} = (a^2)^{b/2}$
- b > 0, nieparzyste

  $a^b = a^{1+2*((b-1)/2)} = (a^2)^{(b-1)/2} * a$

Szybkie potęgowanie… rekurencyjnie

Obserwacja (jeszcze) inaczej

Dla nieujemnych całkowitych *a* i *b* zachodzi:

$$a^b = \begin{cases} 1 & \text{gdy } b = 0 \\ (a^2)^{b/2} & \text{gdy } b > 0, \text{parzyste} \\ (a^2)^{(b-1)/2} * a & \text{gdy } b > 0, \text{nieparzyst e} \end{cases}$$

Algorytm rekurencyjny (w oparciu o zależność rekurencyjną):

- jeśli b=0: zwróć 1
- jeśli b nieparzyste: zwróć a pomnożone przez wynik wywołania dla a równego a^2 oraz b równego (b-1) /2
- jeśli b parzyste: zwróć wynik wywołania dla a równego a^2 oraz b równego b / 2

Szybkie potęgowanie… rekurencyjnie

$$a^b = \begin{cases} 1 & \text{gdy } b = 0 \\ (a^2)^{b/2} & \text{gdy } b > 0, \text{parzyste} \\ (a^2)^{(b-1)/2} * a & \text{gdy } b > 0, \text{nieparzyst e} \end{cases}$$

Algorytm rekurencyjny:

- jeśli b=0: zwróć 1
- jeśli b nieparzyste: zwróć a pomnożone przez wynik wywołania dla a równego a^2 oraz b równego (b-1) /2
- jeśli b parzyste: zwróć wynik wywołania dla a równego a^2 oraz b równego b / 2

ALE: w C stosujemy dzielenie całkowite, czyli (b-1)/2 jest równe b/2 dla nieparzystego b.

Implementacja:

```
int pot(int a, int b)
{
  if (!b) return 1;
  if (b%2) return a* pot(a*a,b/2);
  return pot(a*a, b/2);
}
```

Szybkie potęgowanie… rekurencyjnie

$$a^b = \begin{cases} 1 & \text{gdy } b = 0 \\ (a^2)^{b/2} & \text{gdy } b > 0, \text{parzyste} \\ (a^2)^{(b-1)/2} * a & \text{gdy } b > 0, \text{nieparzyst e} \end{cases}$$

Algorytm rekurencyjny:

- jeśli b=0: zwróć 1
- jeśli b nieparzyste: zwróć a pomnożone przez wynik wywołania dla a równego a^2 oraz b równego (b-1) /2
- jeśli b parzyste: zwróć wynik wywołania dla a równego a^2 oraz b równego b / 2

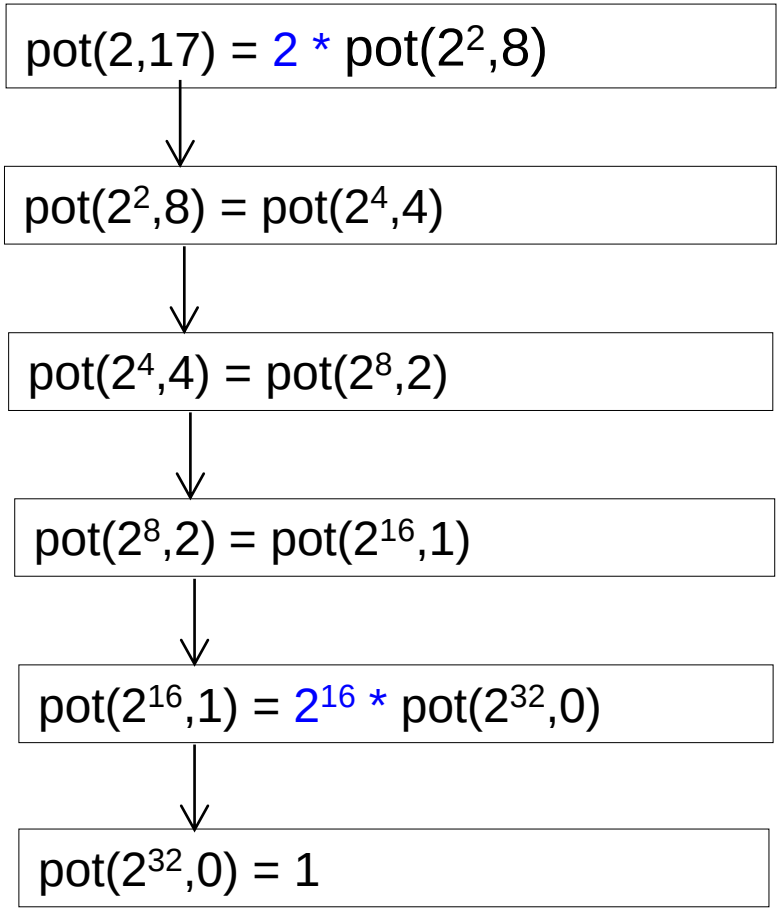
ALE: dla b nieparzystego (b – 1)/2 jest równe wynikowi dzielenia całkowitego b przez 2

Implementacja:

```
def pot(a, b):
    if not b:
        return 1
    if b%2:
        return a*pot(a*a,b//2)
    return pot(a*a,b/2)
```

Przykład: drzewo wywołań (rekurencyjnych)

Drzewo wywołań (rekurencyjnych) dla pot(2,17):



Implementacja:

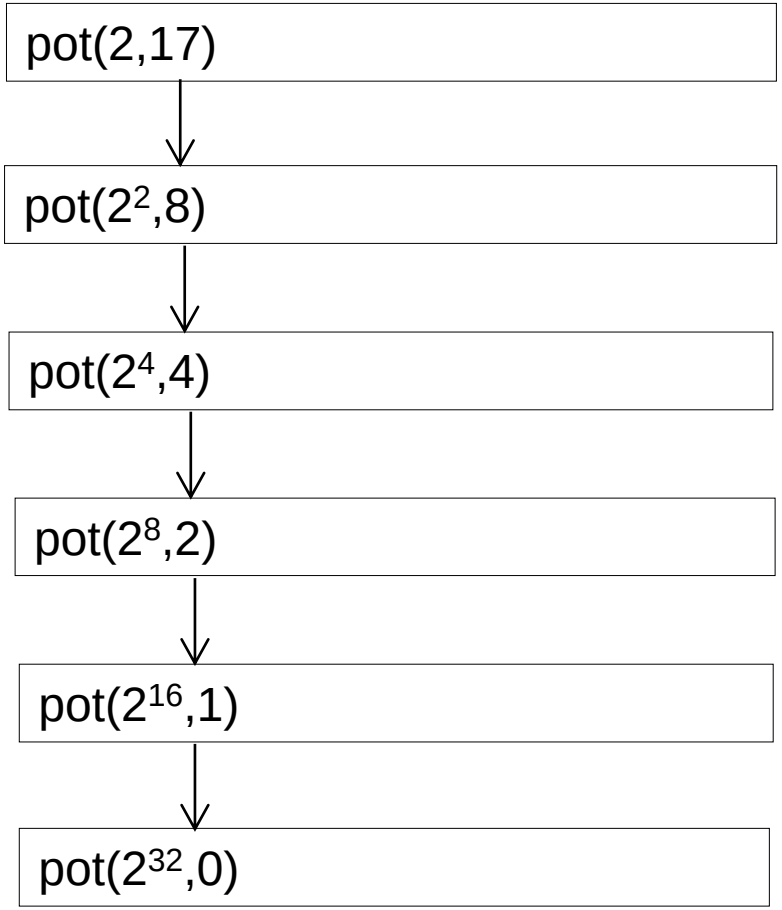
```
int pot(int a, int b)
{
  if (!b) return 1;
  if (b%2) return a* pot(a*a,b/2);
  return pot(a*a, b/2);
}
```

Implementacja:

```
def pot(a, b):
    if not b:
        return 1
    if b%2:
        return a*pot(a*a,b//2)
    return pot(a*a,b/2)
```

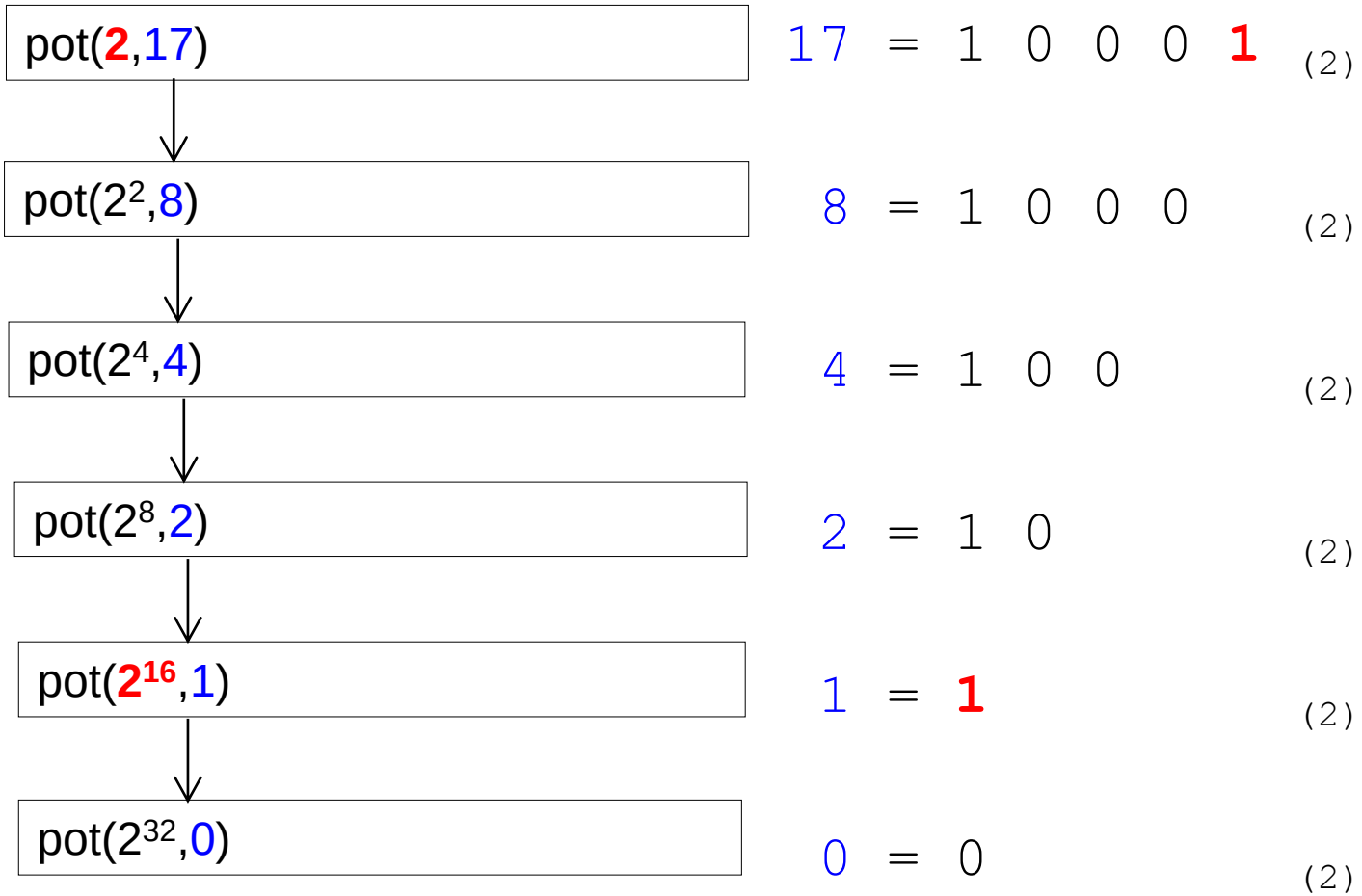
Przykład: drzewo wywołań (rekurencyjnych)

Drzewo wywołań (rekurencyjnych) dla pot(2,17):



Przykład: drzewo wywołań vs binarna repr. potęgi

Drzewo wywołań (rekurencyjnych) dla pot(2,17):



Szybkie potęgowanie… porównanie

Nierekurencyjnie:

```
int pot(int a, int b)
{ int rez;
  rez = 1;
  while (b>0) {
    if (b%2) rez = rez * a;
    b = b / 2;
    a = a * a;
  }
  return rez;
}
```

Rekurencyjnie:

```
int pot(int a, int b)
{
  if (!b) return 1;
  if (b%2) return a* pot(a*a,b/2);
  return pot(a*a, b/2);
}
```

Szybkie potęgowanie… porównanie

Nierekurencyjnie:

```
def pot(a, b):
    rez = 1
    while b>0:
        if b%2:
            rez = rez * a
        b = b // 2
        a = a * a
    return rez
```

Rekurencyjnie:

```
def pot(a, b):
    if not b:
        return 1
    if b%2:
        return a*pot(a*a,b//2)
    return pot(a*a,b/2)
```

Szybkie potęgowanie: złożoność

Implementacja nierekurencyjna

Pamięć: O(1)

Czas: O(log b)

– liczba powtórzeń (obrotów) pętli proporcjonalna do długości reprezentacji binarnej liczby b.

„Naiwne” a szybkie potęgowanie

Czas naiwnego potęgowania: O(b)

b	1	2	4	256	1024	65536
log b	0	1	2	8	10	16

Szybkie potęgowanie: złożoność

Implementacja rekurencyjna

Czas:

- Intuicyjnie: każde wywołanie „skraca” binarną reprezentację b o jeden; więc O(log b)
- Formalnie możemy czas wyrazić **zależnością rekurencyjną**:

  T(1) = 1

  T(b) = 1 + T(b / 2) dla parzystego b

  T(b) = 1 + T((b – 1) / 2) dla nieparzystego b

  Można pokazać, że T(b) = **O(log b)**

Pamięć:

- Musimy uwzględnić zajętość stosu wywołań (ile wywołań jednocześnie?)
- Liczba wywołań ograniczona przez czas (p. wyżej)
- Złożoność: **O(log b)**

Liczby Fibonacciego

Liczby Fibonacciego (czyli: kiedy nie używać rekurencji!)

Definicja:

$$F_0 = 1$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2} \text{ dla } n > 1$$

Rekurencyjnie:

```
long fib(int i)
{ if (i<=1) return 1;
  return fib(i-1)+fib(i-2);
}
```

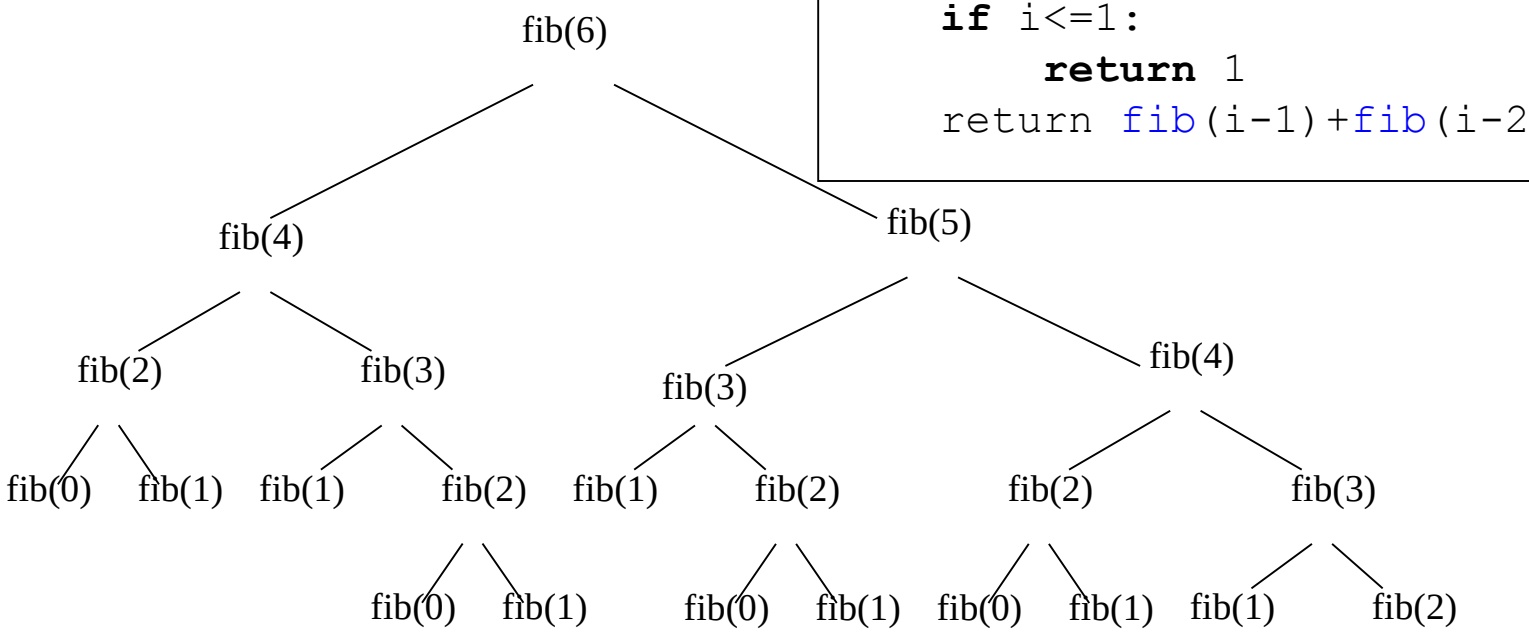
```
def fib(i):
    if i<=1:
        return 1
    return fib(i-1)+fib(i-2)
```

n	Fib_n
2	2
3	3
4	5
5	8
6	13
7	21
8	34
9	55
10	89
11	144

Liczby Fibonacciego

Drzewo wywołań rekurencyjnych:

```
def fib(i):
    if i<=1:
        return 1
    return fib(i-1)+fib(i-2)
```




```
long fib(int i)
{ if (i<=1) return 1;
  return fib(i-1)+fib(i-2);
}
```

Liczby Fibonacciego

Liczba „liści” drzewa wywołań dla fib(n):

- fib(n)

Złożoność czasowa co najmniej fib(n)

- $\geq$ liczba węzłów drzewa wywołań
- czyli, $\geq$ fib(n)

Wniosek:

- funkcja **rekurencyjna** nieefektywna!

Liczby Fibonacciego

Rozwiązanie nierekurencyjne:

- Utwórz tablicę int f[];
- „zapamiętuj” wartości F_i w f[i];
- użyj f[i-1] i f[i-2] obliczając F_i.

```
#define MaxN 1000

long fib1(int n)
{ long f[MaxN];
  f[0]=f[1]=1;
  for (i=2; i<=n; i++)
   f[i] = f[i-1]+f[i-2];
  return f[n];
}
```

```
MaxN=1000

def fib1(n):
    f=Array(MaxN)
    f[0]=1
    f[1]=1
    for i in range(2,n+1):
      f[i] = f[i-1]+f[i-2]
    return f[n]
```

Liczby Fibonacciego

Rozwiązanie nierekurencyjne:

- Czas: O(n)
- Pamięć: O(n)

```
#define MaxN 1000

long fib1(int n)
{ long f[MaxN];
  f[0]=f[1]=1;
  for (i=2; i<=n; i++)
   f[i] = f[i-1]+f[i-2];
  return f[n];
}
```

```
MaxN=1000

def fib1(n):
    f=Array(MaxN)
    f[0]=1
    f[1]=1
    for i in range(2,n+1):
      f[i] = f[i-1]+f[i-2]
    return f[n]
```

Pyt.: jak zmniejszyć wymagania pamięciowe?

ćwiczenia…

Liczby Fibonacciego - podsumowanie

- Rekurencja w oparciu o definicję ciągu: czas obliczenia F_n proporcjonalny do (co najmniej) wartości F_n.
- Iteracja ze spamiętywaniem wartości:
  - czas O(n)
  - czyli dużo(!) szybciej niż rekurencyjnie
- Pyt.: Czy można jeszcze szybciej?

  Odp.: Tak, wykorzystując:

  szybkie potęgowanie… dla iloczynu macierzy

Współczynnik dwumianowy

Współczynnik dwumianowy

Definicja:

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

Algorytm „naiwny”:

```
int newton(int n, int m)
{
  return sil(n)/(sil(m)*sil(n-m));
}
```

```
def newton(n, m):
  return sil(n)/(sil(m)*sil(n-m))
```

Wady:

- Duże liczby w trakcie obliczeń (zakres?).
- Powtarzanie obliczeń (np. (n-m)! jest „pośrednim wynikiem” przy wyliczaniu n!).

Współczynnik dwumianowy

Fakt (trójkąt Pascala)

$$\binom{n}{0} = \binom{n}{n} = 1$$

$$\binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m} \text{ dla } n, m > 0, m < n$$

Funkcja rekurencyjna

```
long newton(int n, int m)
{ if ((m==0) || (m==n)) return 1;
  else return newton(n-1,m-1)+newton(n-1,m);
}
```

```
def newton(n, m):
    if m==0 or m==n:
        return 1
    else:
        return newton(n-1,m-1)+newton(n-1,m)
```

Współczynnik dwumianowy

Rekurencyjnie:

```
long newton(int n, int m)
{ if ((m==0) || (m==n)) return 1;
  else return newton(n-1,m-1)+newton(n-1,m);
}
```

```
def newton(n, m):
    if m==0 or m==n:
        return 1
    else:
        return newton(n-1,m-1)+newton(n-1,m)
```

Złożoność czasowa: proporcjonalna do wartości funkcji

Wniosek: rozwiązanie **rekurencyjne** jest bardzo **niewydajne**!

Dlaczego? Wielokrotne powtarzanie „tych samych” obliczeń!

Współczynnik dwumianowy bez rekurencji

Wykorzystamy zależność:

$$\binom{n}{0} = \binom{n}{n} = 1$$

$$\binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m} \text{ dla } n, m > 0$$

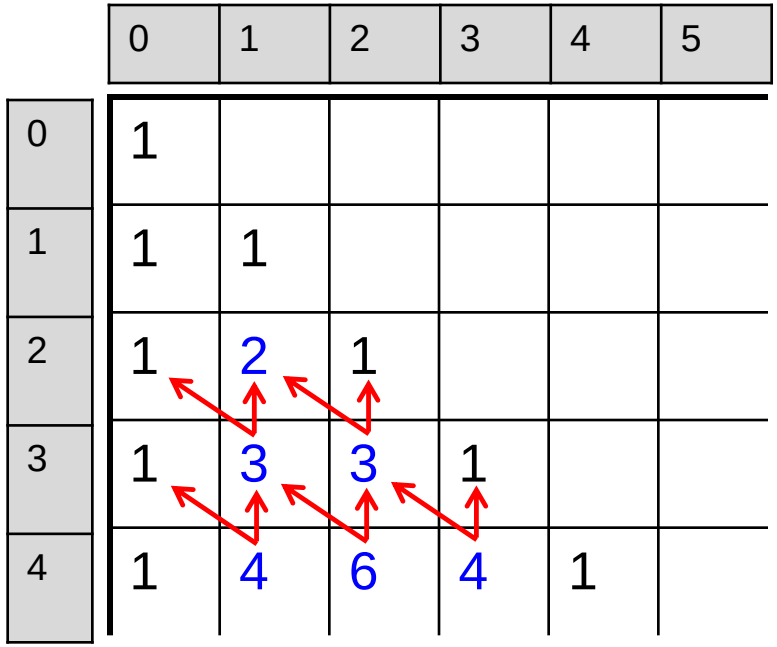
Wartości przechowujemy w tablicy dwuwymiarowej:

$$\binom{i}{j} = a[i][j]$$

Współczynnik dwumianowy bez rekurencji

Algorytm:

- umieść jedynki w pierwszej kolumnie i na głównej przekątnej
- Wypełniaj pozostałe komórki od góry w dół, od lewej do prawej (jak w trójkącie Pascala).



Wsp. dwumian. nierekur. – v. 1

```
long newton(int n, int m)
{ int a[n+1][n+1];
  int i, j;

  a[0][0]=1;
  for (i=1; i<=n; i++)
  { a[i][0]=1;
     for (j=1; j<i; j++)
       a[i][j]=a[i-1][j-1]+a[i-1][j];
     a[i][i]=1;
  }
  return a[n][m];
}
```

1					
1	1				
1	2	1			
1	3	3	1		
1	4	6	4	1	

Złożoność:

- Czas: O($n \cdot n$)
- Pamięć: O($n \cdot n$)

Wsp. dwumian. nierekur. – v. 1

```
def newton3(n, m):
    a=Array(n+1,n+1)
    a[0][0]=1
    for i in range(1,n+1):
        a[i][0]=1
        for j in range(1,i):
            a[i][j]=a[i-1][j-1]+a[i-1][j]
        a[i][i]=1
    return a[n][m]
```

1					
1	1				
1	2	1			
1	3	3	1		
1	4	6	4	1	

Złożoność:

- Czas: O($n \cdot n$)
- Pamięć: O($n \cdot n$)

Wsp. dwumian. nierekur. – v. 2

Jak zmniejszyć pamięć?

Obserwacja

Wartości w *i*-tym wierszu zależą jedynie od wartości w wierszu *i* – 1:

$$a[i][j] = a[i-1][j-1]+a[i-1][j]$$

Rozwiązanie

Przechowuj tylko dwa wiersze:

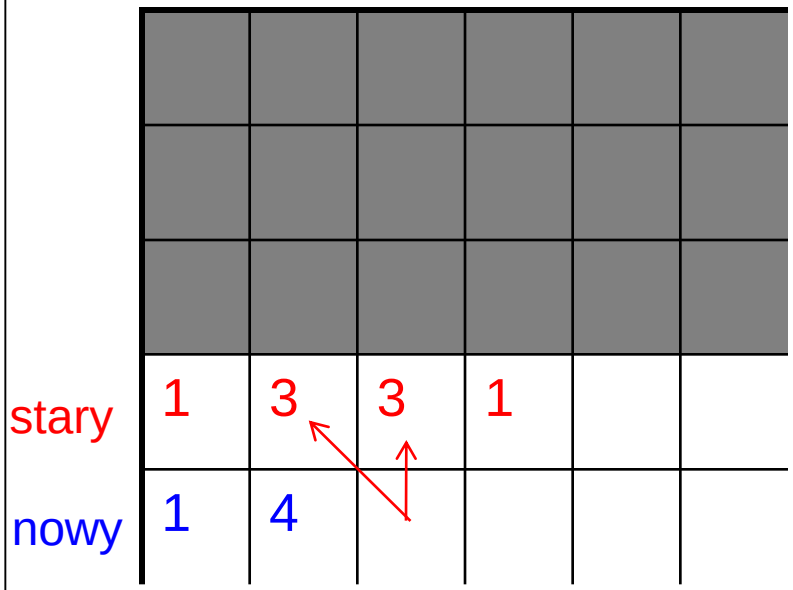
- bieżący;
- poprzedni.

1					
1	1				
1	2	1			
1	3	3	1		
1	4	6	4	1	

Wsp. dwumian. nierekur. – v. 2

```
long newton(int n, int m)
{    int i, j;
     long stary[n+1],nowy[n+1];

     stary[0]=1;
     for (i=0;i<n;i++)
     {  nowy[0]=1;
        for (j=1; j<=i; j++)
           nowy[j]=stary[j-1]+stary[j];
        nowy[i+1]=1;
        for (j=0;j<=i+1;j++)
         { stary[j]=nowy[j];
           nowy[j]=0;
         }
     }
     return(stary[m]);
}
```



Wsp. dwumian. nierekur. – v. 2

```
long newton(int n, int m)
{    int i, j;
     long stary[n+1],nowy[n+1];

     stary[0]=1;
     for (i=0;i<n;i++)
     {  nowy[0]=1;
        for (j=1; j<=i; j++)
          nowy[j]=stary[j-1]+stary[j];
        nowy[i+1]=1;
        for (j=0;j<=i+1;j++)
         { stary[j]=nowy[j];
           nowy[j]=0;
         }
     }
     return(stary[m]);
}
```

Uwagi:

- nowy[0]=1 wystarczy raz
- wypełnianie nowy zerami nie jest konieczne

Wsp. dwumian. nierekur. – v. 2

```
def newton4(n, m):
    stary=Array(n+1)
    nowy=Array(n+1)

    stary[0]=1
    for i in range(0,n):
        nowy[0]=1
        for j in range(1,i+1):
            nowy[j]=stary[j-1]+stary[j]
        nowy[i+1]=1
        for j in range(i+2):
            stary[j]=nowy[j]
            nowy[j]=0
    return stary[m]
```

Uwagi:

- nowy[0]=1 wystarczy raz
- wypełnianie nowy zerami nie jest konieczne

Wsp. dwumian. nierekur. – v. 2

```
long newton(int n, int m)
{    int i, j;
     long stary[n+1],nowy[n+1];
     nowy[0]=1;
     stary[0]=1;
     for (i=0;i<n;i++)
     {  nowy[0]=1;
        for (j=1; j<=i; j++)
          nowy[j]=stary[j-1]+stary[j];
        nowy[i+1]=1;
        for (j=0;j<=i+1;j++)
         { stary[j]=nowy[j];
           nowy[j]=0;
         }
     }
     return(stary[m]);
}
```

Uwagi:

- nowy[0]=1 wystarczy raz
- wypełnianie nowy zerami nie jest konieczne

Wsp. dwumian. nierekur. – v. 2.1

```
long newton(int n, int m)
{ int i, j;
  long stary[n+1],nowy[n+1];
  stary[0]=1;
  nowy[0]=1;
  for (i=0;i<n;i++)
  {
   for (j=1; j<=i; j++)  nowy[j]=stary[j-1]+stary[j];
   nowy[i+1]=1;
   for (j=1;j<=i+1;j++)  stary[j]=nowy[j];
  }
  return(nowy[m]);
}
```

Uwaga

Zamiana wartości tablic stary i nowy jest czasochłonna.

Obserwacja

Do policzenia nowy[j] potrzebujemy tylko stary[j] i stary[j-1].

Wsp. dwumian. nierekur. – v. 2.1

Uwaga

Zamiana wartości tablic stary i nowy jest czasochłonna.

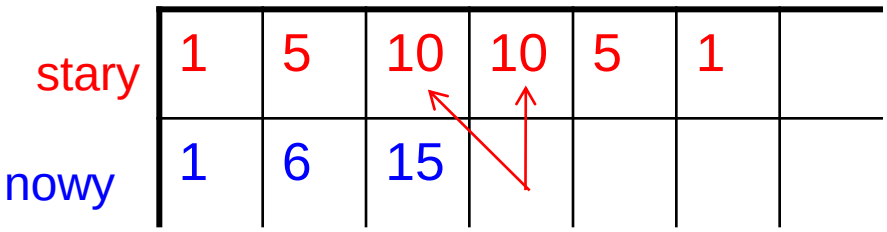
Obserwacja

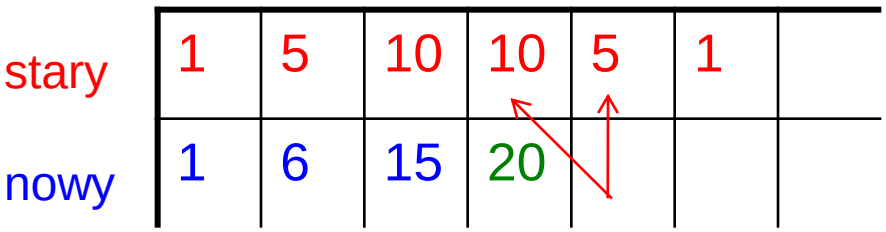
Do policzenia nowy[j] potrzebujemy tylko stary[j] i stary[j-1].

Rozwiązanie

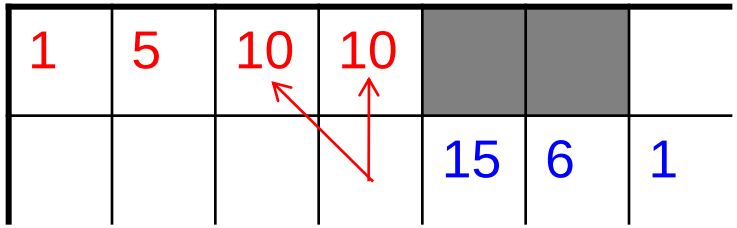
Używaj jednej tablicy, obliczaj wartości „od prawej do lewej”.

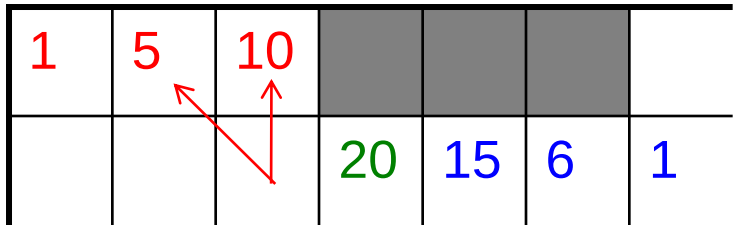
Dwie tablice





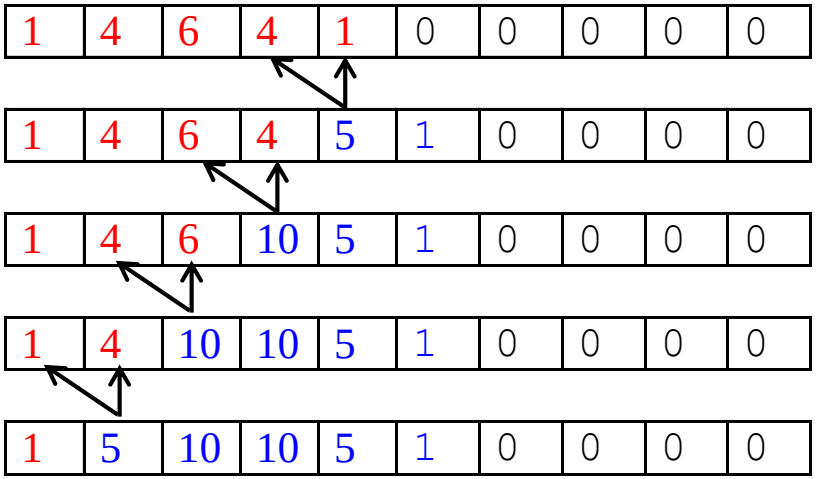
Jedna tablica





Wsp. dwumian. nierekur. – v. 3

Nowe rozwiązanie – przykład



Wsp. dwumian. nierekur. – v. 3

```
long newton(int n, int m)
{  int i, j;
    long nowy[m+1];

    nowy[0]=1;
    for (i=0;i<n;i++)
    {
        nowy[i+1]=1;
        for (j=i; j>0; j--)
          nowy[j]=nowy[j-1]+nowy[j];
    }
  return(nowy[m]);
}
```

Złożoność:

- Czas: O(*n*·*n*) [łatwo poprawić do O(*n*·*m*); jak???]
- Pamięć: O(*m*) [dynamicznie deklarowany rozmiar tablicy…]

Wsp. dwumian. nierekur. – v. 3

Obserwacja:

Wyznaczanie wartości w kolumnach m+1, m+2, … niekonieczne

```
long newton(int n, int m)
{  int i, j, maxc;
    long nowy[m+1];

    nowy[0]=1;
    for (i=0;i<n;i++)
    {  if (i<m) nowy[i+1]=1;
       maxc = i>m ? m : i;
       for (j=maxc; j>0; j--)
          nowy[j]=nowy[j-1]+nowy[j];
    }
  return(nowy[m]);
}
```

- Czas: O($n \cdot m$)
- Pamięć: O(m)

Wsp. dwumian. nierekur. – v. 3

Obserwacja:

Wyznaczanie wartości w kolumnach m+1, m+2, … niekonieczne

```
def newton(n, m):
    nowy=Array(m+1)
    nowy[0]=1
    for i in range(0,n):
        if i<m:
            nowy[i+1]=1
        if  i>m:
            maxc = m
        else:
            maxc = i
        j=maxc
        while j>0:
            nowy[j]=nowy[j-1]+nowy[j]
            j=j-1
    return nowy[m]
```

- Czas: O($n \cdot m$)
- Pamięć: O(m)

Podsumowanie

1. Rekurencja:
   - wygodne narzędzie do zapisywania algorytmów wykorzystujących zależności rekurencyjne
   - unikać, gdy wymaga wielokrotnego powtarzania obliczeń
2. Programowanie dynamiczne jako alternatywa dla nieefektywnej rekurencji:
   - spamiętywanie wyników dla mniejszych wartości argumentów
   - Wykorzystanie „spamiętanych” wartości zamiast wywołań rekurencyjnych