

# TEORETYCZNE PODSTAWY INFORMATYKI

24/10/2016

WFAiS UJ, Informatyka Stosowana  
I rok studiów, I stopień

# Wykład 4 – część I

2

Iteracja

Rekurencja

Indukcja

- Iteracja**
- Rekurencja**
- Indukcja**
- Algorytmy sortujące**
- Rozwiązywanie rekurencji**

# Iteracja

3

- Źródłem „potęgi” komputerów jest zdolność do **wielokrotnego wykonywania (powtarzania) tego samego zadania** lub jego różnych wersji.

**Iteracja = „powtarzanie”**

- W informatyce z pojęciem iteracji (ang. iteration) można się spotkać przy różnych okazjach. Wiele zagadnień związanych z modelami danych, np. listami, opiera się na powtórzeniach typu:
  - ▣ ***lista jest albo pusta, albo składa się z jednego elementu poprzedzającego inny, kolejny element itd....***

# Iteracja

4

- Programy i algorytmy wykorzystują iteracje do **wielokrotnego wykonywania określonych zadań** bez konieczności definiowania ogromnej liczby pojedynczych kroków, np. w przypadku zadania: **wykonaj dany krok 1000 razy**.
- Najprostszym sposobem wielokrotnego wykonania sekwencji operacji jest wykorzystanie **konstrukcji iteracyjnej**, jaką jest instrukcja **for** lub **while** w języku C.

# Prosty przykład iteracji

5

- Mamy tablicę  $n$  liczb całkowitych, sprawdzamy czy jakaś liczba całkowita „ $x$ ” jest elementem tej tablicy. Algorytm przegląda całą tablicę, po napotkaniu  $A[i] = x$  kończy działanie.
  - Jeżeli  $A[0] = x$  to algorytm  $O(1)$
  - Jeżeli  $A[n-1] = x$  to algorytm  $O(n)$
- Średni czas działania algorytmu:
  - $\langle T(n) \rangle = \sum_{i=0}^{n-1} (c i + d) \cdot (1/n)$   
 $= c \cdot (n-1) / 2 + d$
  - $\langle T(n) \rangle \sim c \cdot n/2$  dla dużego  $n$

1	$A[0]$
8	
7	
5	
3	
4	$A[i]$
8	
9	
7	$A[n-1]$

# Iteratory

6

- Iteracje, poza **organizacją pętli** wykonujących obliczenia, używane są powszechnie do przetwarzania tablic. Możliwe jest przetwarzanie w kolejności indeksów malejących oraz w kolejności indeksów rosnących.
- Potrzebujemy też bardziej poręcznego mechanizmu, separującego logikę związaną z wyborem elementów od reszty kodu. Mechanizm taki zwany jest **iteratorem** lub **enumeratorem**, dostarcza możliwości iterowania po dowolnym zbiorze danych określonym przez jakąś strukturę danych lub inny bardziej ogólny schemat.
  - ▣ **Iterator** ma umożliwić wykonywanie operacji: następny, poprzedni, ostatni, pierwszy, bieżący, itp...
  - ▣ **Iterator to jest pewna koncepcja, implementacja zależy od języka programowania.**

# Predykatory

7

- **Iteratorem filtrującym** nazywamy iterator działający na bazie innego **iteratora i procedury klasyfikującej** (akceptującej lub odrzucającej) elementy zwracane przez ten ostatni.
- Procedura taka nosi nazwę **predykatora**.
- **Iterator filtrujący** ignoruje wszystkie te elementy które nie spełniają warunków określonych przez predykator.

# Rekurencja

8

- Zagadnieniem blisko związanym z **powtórzeniami** (iteracją) jest **rekurencja** (ang. recursion) – technika, w której **definiuje się pewne pojęcie bezpośrednio lub pośrednio na podstawie tego samego pojęcia.**
- Np. można zdefiniować pojęcie lista stwierdzeniem:
  - ▣ **lista jest albo pusta, albo jest sklejeniem elementu i listy**
- **Definicje rekurencyjne** są szeroko stosowane do specyfikacji **gramatyk języków programowania** (patrz następne wykłady).



# Rekurencja (przykład)

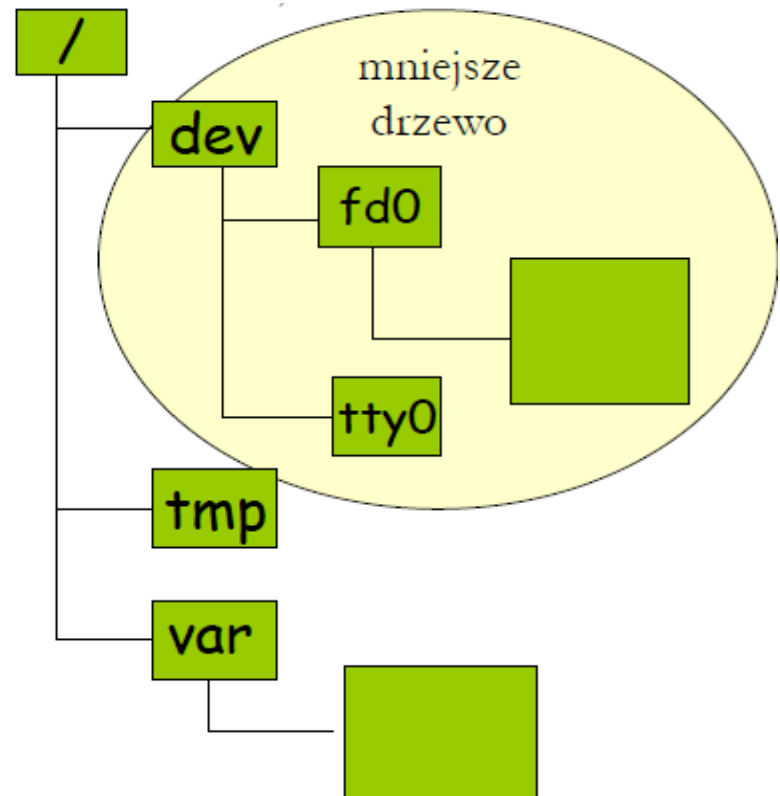
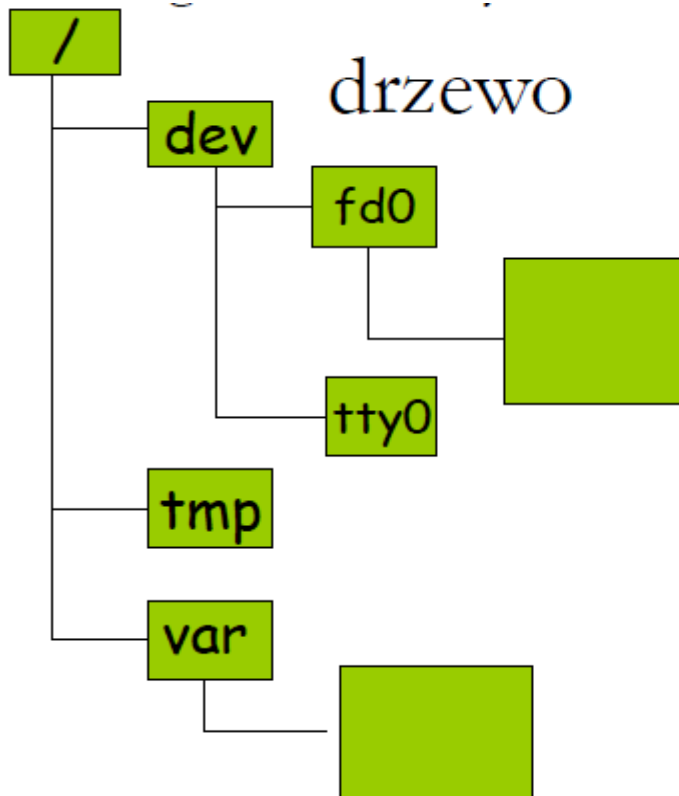
9

- Rozpatrzmy system plików, jak na dysku komputera. W systemie tym istnieje **katalog najwyższego poziomu** (root), w którym znajdują się **pliki i podkatalogi**.
- Owa zagnieżdżona struktura bywa nazywana powszechnie **drzewem katalogów** (directory tree) – drzewo to zakorzenione jest w katalogu najwyższego poziomu, zaś pliki mogą być uważane za liście tego drzewa.

# Rekurencja

10

- Każda z gałęzi może być traktowana jak inne mniejsze drzewo.



# Rekurencja

11

- **Podobieństwo dwóch obiektów różniących się między sobą skalą lub granulacją** jest interesującą koncepcją niezwykle użyteczną w rozwiązywaniu problemów.
- Strategia podziału oryginalnego problemu na „mniejsze” podproblemy tej samej natury - zwana strategią **„dziel i zwyciężaj”** (divide and conquer) – jest **jednym z przykładów rekurencji**.
- Rekurencja jest przykładem wielokrotnego wykorzystywania tych samych rozwiązań: **metoda wywołuje samą siebie**.

# Definicja rekurencyjna

12

- **Definicja rekurencyjna** składa się z dwóch części.
  - ▣ W pierwszej, zwanej **podstawową** lub **warunkiem początkowym**, są wyliczone elementy podstawowe, stanowiące części składowe wszystkich pozostałych elementów zbioru.
  - ▣ W drugiej części, zwanej **krokiem indukcyjnym**, są podane **reguły umożliwiające konstruowanie nowych obiektów z elementów podstawowych lub obiektów zbudowanych wcześniej**. Reguły te można stosować wielokrotnie, tworząc nowe obiekty.

# Definicja rekurencyjna

13

**Rekurencyjna definicja funkcji silnia !**

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

**Rekurencyjna definicja ciągu Fibonacciego?**

$$F(n) = n \quad \text{jeśli } n < 2$$

$$F(n) = F(n-2) + F(n-1) \quad \text{jeśli } n \geq 2$$

# Rekurencja w jęz. programowania

14

- Rekurencja jest **zaimplementowana w wielu językach programowania**.
  - ▣ Np. w języku C, **funkcja f** może wywołać samą siebie albo bezpośrednio z poziomu funkcji f, albo pośrednio wywołując inną funkcję (która wywołuje inną funkcję, która wywołuje inną funkcję, ... , która wywołuje funkcję f).
- Często można opracowywać **algorytmy rekurencyjne**, naśladując definicje rekurencyjne zawarte w specyfikacji programu, który jest implementowany.

# Rekurencja w jęz. programowania

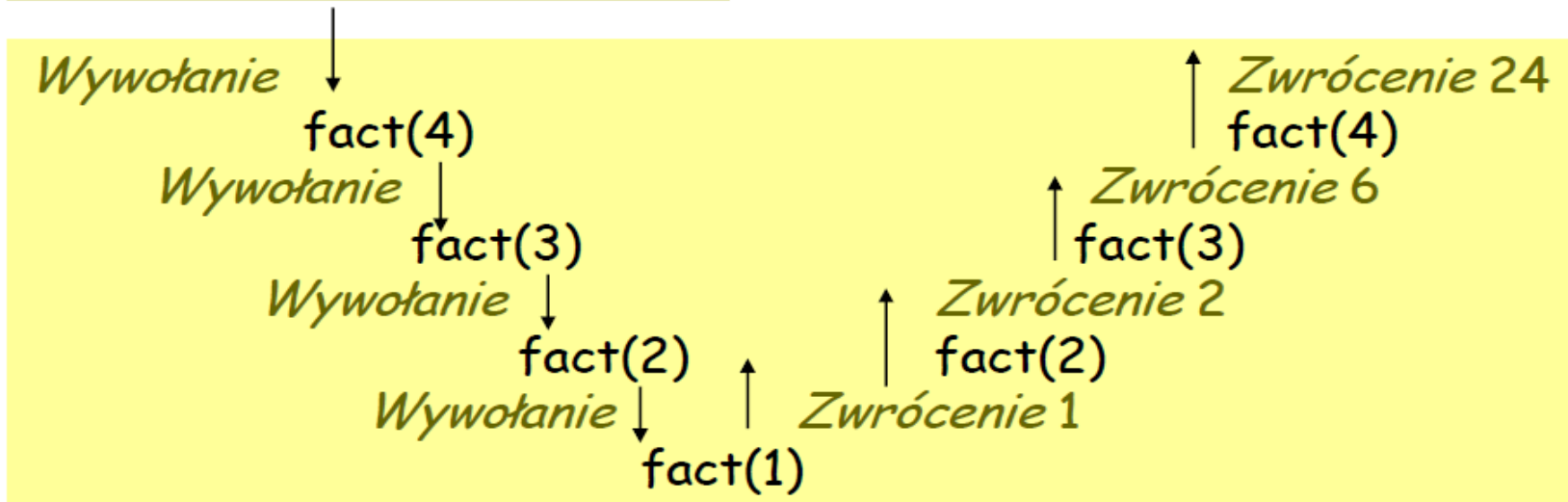
15

- **Implementacja w języku C jest prosta... ale kod źródłowy nie sugeruje jak to się odbywa.**
  - ▣ **Jak funkcja wywołująca sama siebie może w ogóle działać i jeszcze dawać prawidłowy wynik?**
- **Realizacja (pomysł od E. W. Dijkstry) przy pomocy stosu i systemu operacyjnego (patrz następne wykłady).**

# Funkcja rekurencyjna obliczająca $n!$ dla $n \geq 1$

16

```
int fact(int n)
{
  if (n ≤ 1)
    return 1; /*podstawa*/
  else
    return n · fact(n-1) /* indukcja*/
}
```





# Rekurencja czy iteracja?

17

*Jak obliczać ciąg Fibonacciego?*

$F(n) = n$                       jeśli  $n < 2$

$F(n) = F(n-2) + F(n-1)$       jeśli  $n \geq 2$

**Iteracja:**

```
IterativeFib (int n) {
    if (n < 2) return n;
    else {
        int tmp, current = 1, last = 0;
        for (i = 2, i <= n, ++i) {
            tmp = current;
            current += last;
            last = tmp;
        }
        return current;
    }
}
```

**Rekurencja:**

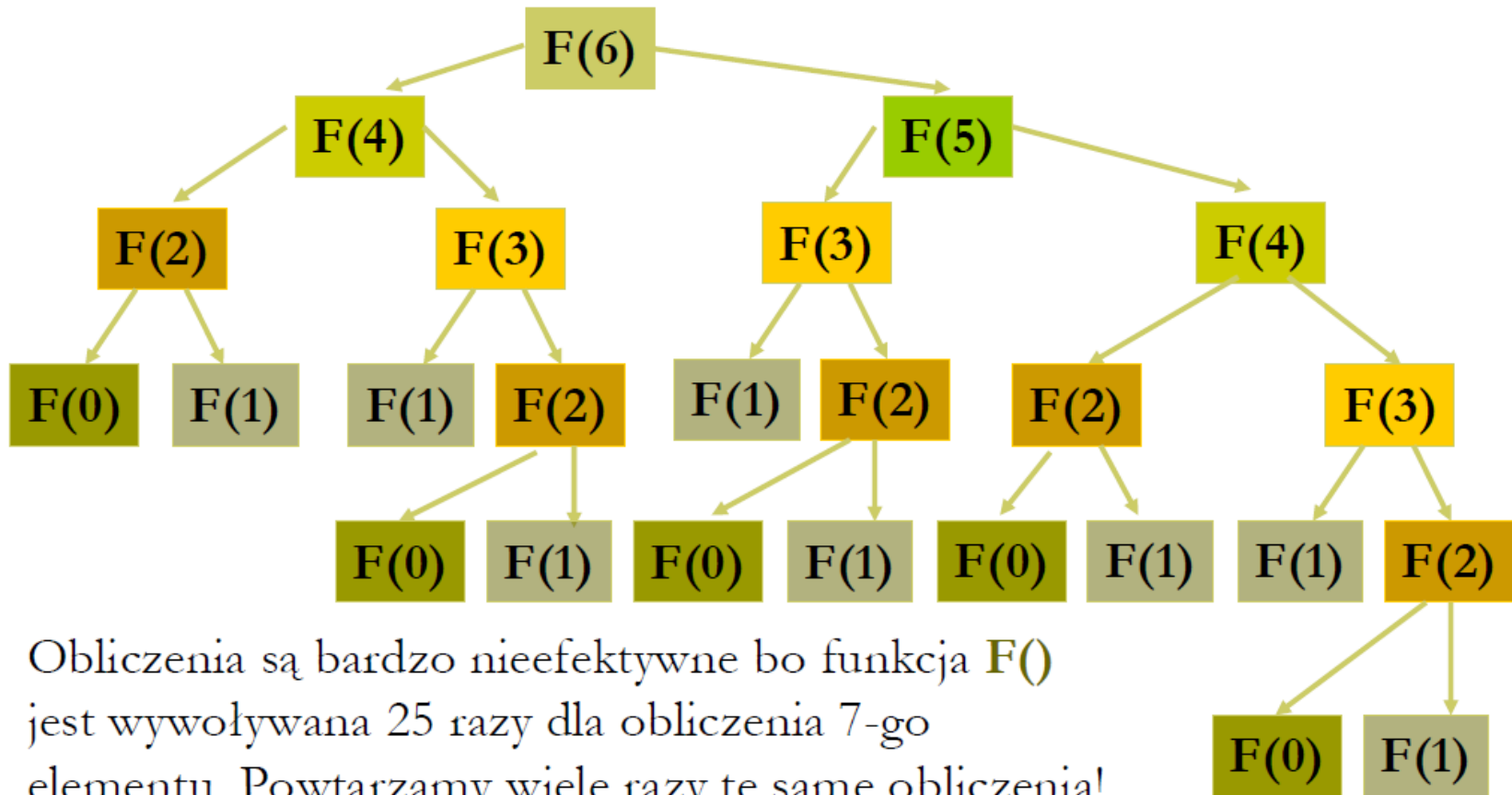
```
Fib (int n)
{
    if (n < 2)
        return n;
    else
        return Fib(n-2) + Fib(n-1);
}
```

Można pokazać że liczba wywołań funkcji **Fib()** jest  $2 \cdot \text{Fib}(n+1) - 1$  w celu obliczenia wartości **Fib(n)**.

Trzeba mieć dobry pomysł na prosty algorytm iteracyjny.

# Rekurencja dla ciągu Fibonacciego

18



Obliczenia są bardzo nieefektywne bo funkcja  $F()$  jest wywoływana 25 razy dla obliczenia 7-go elementu. Powtarzamy wiele razy te same obliczenia!

# Rekurencja czy iteracja?

19

## □ Jak obliczać ciąg Fibonacciego?

- $F(n) = n$                       jeśli  $n < 2$
- $F(n) = F(n-2) + F(n-1)$  jeśli  $n \geq 2$

n	liczba dodawań	Przypisania	
		Algorytm iteracyjny	Algorytm rekurencyjny
6	5	15	25
10	9	27	177
15	14	42	1973
20	19	57	21891
25	24	72	242785
30	29	87	2692537

- **Algorytm rekurencyjny ma złożoność  $O(2^n)$ , to zbyt wysoka cena za prostotę. ( $3 \cdot 10^6$  wywołań dla  $F(30)$ )**
- **Algorytm iteracyjny ma złożoność  $O(n)$**

# Rekurencja czy iteracja?

20

- **Każdy problem mający rozwiązanie rekurencyjne daje się także rozwiązać w sposób iteracyjny, choć jego rozwiązanie iteracyjne może być mniej czytelne w porównaniu z rekurencyjnym, a niekiedy wręcz sztuczne.**
- **Rekurencja może być ponadto symulowana w sposób iteracyjny, przy użyciu struktur danych zwanych stosami.**
- **Programy rekurencyjne są często mniejsze i łatwiejsze do zrozumienia od ich iteracyjnych odpowiedników.**
- **Co ważniejsze, niektóre problemy (szczególnie niektóre problemy wyszukiwania) są znacznie łatwiejsze do rozwiązania za pomocą programów rekurencyjnych.**

# Indukcja

21

- **Zagadnieniem również związanym z iteracją i rekurencją jest indukcja (ang. induction):**
  - ▣ **technika stosowana w matematyce do dowodzenia, że twierdzenie  $S(n)$  jest prawdziwe dla wszystkich nieujemnych liczb całkowitych  $n$  lub, uogólniając, dla wszystkich liczb całkowitych  $\geq$  od pewnego ograniczenia dolnego.**

# Indukcja

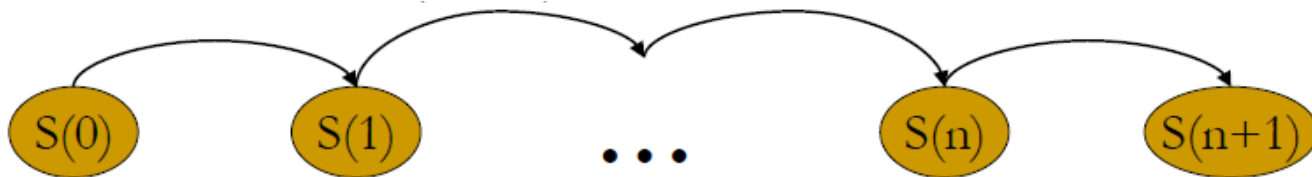
22

- Niech  $S(n)$  będzie dowolnym twierdzeniem dotyczącym liczby całkowitej  $n$ . W najprostszej formie dowodu indukcyjnego (**indukcja częściowa**) twierdzenia  $S(n)$  dowodzi się dwóch faktów:
  - **Przypadku podstawowego:** za który często przyjmuje się twierdzenie  $S(0)$ . Przypadkiem podstawowym może jednak być równie dobrze  $S(k)$  dla dowolnej liczby całkowitej  $k$ . Dowodzi się wówczas prawdziwości twierdzenia  $S(n)$  dla  $n \geq k$ .
  - **Kroku indukcyjnego:** gdzie dowodzi się, że dla wszystkich  $n \geq 0$  (lub wszystkich  $n \geq k$ ), prawdziwość  $S(n)$  implikuje prawdziwość  $S(n+1)$ .

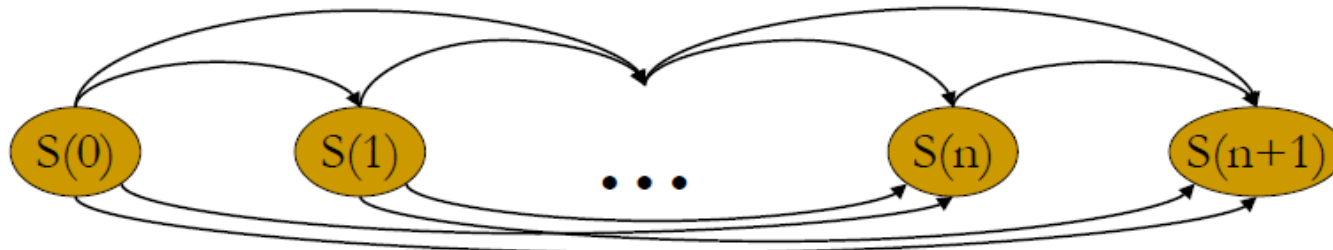
# Indukcja zupełna i częściowa

23

- **Indukcja częściowa (słaba):** wykorzystujemy wyłącznie hipotezę indukcyjną  $S(n)$  do wykazania prawdziwości  $S(n+1)$ .



- **Indukcja zupełna (silna):** Możemy wykorzystać każdą z wartości  $S(i)$ , od podstawy aż do  $n$  do wykazania prawdziwości  $S(n+1)$ .



# Indukcja zupełna i częściowa

24

- Dla indukcji zupełnej dowodzimy, że twierdzenie  $S(n)$ , dla wszystkich  $n \geq 0$  jest prawdziwe na podstawie dwóch faktów:
  - **Przypadku podstawowego:** dowodzi się prawdziwości  $S(0)$  ( lub  $S(k)$  jeżeli to jest przypadek podstawowy)
  - **Kroku indukcyjnego:** gdzie dowodzi się, że dla wszystkich  $n \geq 0$  (lub wszystkich  $n \geq k$ ), że prawdziwość twierdzeń  $S(0), S(1), S(2), \dots, S(n)$  implikuje prawdziwość  $S(n+1)$ .



# Indukcja zupełna i częściowa

25

- **Indukcje z większą liczbą przypadków podstawowych:**

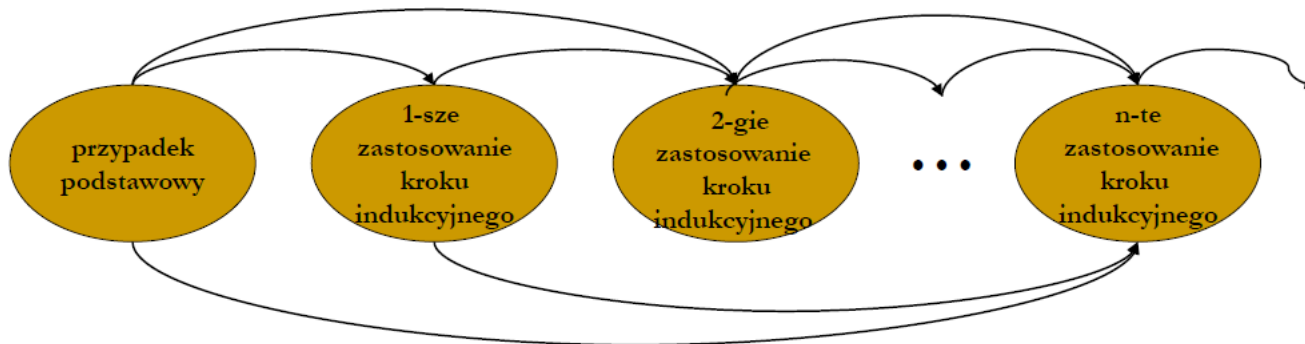
**Niekiedy przydatne jest wykorzystanie więcej niż jednego przypadku podstawowego:**

- **Przypadek podstawowy:** dowodzi się poprawności wszystkich przypadków podstawowych, czyli  $S(i_0), S(i_1), S(i_2), \dots, S(i_m)$ .
- **Krok indukcyjny:** gdzie dowodzi się, że dla wszystkich  $n \geq i_m$  (lub wszystkich  $n > k$ ), z prawdziwość twierdzeń  $S(i_0), S(i_1), S(i_2), \dots, S(n)$  dla  $n \geq i_m$ , implikuje prawdziwość  $S(n+1)$ .

# Definicje indukcyjne

26

- W definicji indukcyjnej definiuje się jedną lub więcej klas reprezentujących ściśle powiązane ze sobą obiekty (lub fakty) na bazie tych samych obiektów.
- Definicja indukcyjna powinna zawierać:
  - ▣ jedną lub więcej reguł podstawowych, z których niektóre definiują pewne obiekty proste,
  - ▣ jedną lub więcej reguł indukcyjnych, za pomocą których definiuje się większe obiekty na bazie mniejszych z tego samego zbioru.



# Definicje indukcyjne

27

- Istnieje ścisłe powiązanie pojęć dowodów indukcyjnych, definicji indukcyjnych oraz programów rekurencyjnych.
- Każde z tych pojęć opiera się na „kroku podstawowym” i „kroku indukcyjnym”.
- W „zwykłych” („częściowych”) indukcjach kolejne kroki zależą wyłącznie od kroków poprzednich.
- Często zachodzi konieczność przeprowadzania dowodów za pomocą indukcji zupełnej, w której każdy krok może zależeć od wszystkich kroków wcześniejszych.
- **Indukcja ma zasadnicze znaczenie w dowodzeniu poprawności programów lub ich fragmentów.**

# Wykład 4 – część II

28

Elementy  
technik  
sortowania

- **Algorytmy iteracyjne: sortowanie przez wybieranie**
- **Algorytmy rekurencyjne: sortowanie przez dzielenie i scalanie**

# Elementy technik sortowania

29

- **Najprostszym sposobem wielokrotnego wykonania sekwencji operacji jest wykorzystanie konstrukcji iteracyjnej (instrukcje `for`, `while` w języku C).**
- **Przykład:**
  - ▣ **Przypuśćmy że mamy listę liczb całkowitych (7, 4, 2, 8, 9, 7, 7, 2, 1).**
  - ▣ **Sortujemy tę listę (w porządku niemalejącym) permutując ją do postaci (1, 2, 2, 4, 7, 7, 7, 8, 9).**
  - ▣ **Należy zauważyć, że sortowanie nie tylko porządkuje wartości, tak że każda jest równa lub mniejsza kolejnej liczbie z listy, ale także zachowuje liczbę wystąpień każdej wartości.**
- **Algorytm sortujący pobiera na wejściu dowolną listę i zwraca jako wynik listę posortowaną. Każdy element występujący w liście pierwotnej występuje również w liście posortowanej.**

# Elementy technik sortowania

30

- Listę elementów dowolnego typu można sortować wówczas, gdy istnieje możliwość zdefiniowania między nimi **relacji mniejszości** oznaczanej typowo **" < "**
- Jeżeli elementami do posortowania są liczby całkowite lub rzeczywiste, symbol **" < "** oznacza znaną wszystkim relację mniejszości
- Jeżeli elementami są ciągi znaków, można np. stosować **„porządek leksykograficzny”**.
- Jeżeli elementy są skomplikowane (struktury) to możemy **do posortowania wykorzystać część każdego elementu** (jedno konkretne pole).

# Sortowanie przez wybieranie – iteracyjny algorytm sortujący

31

- Mamy tablicę **A** zawierającą **n** liczb całkowitych które chcemy posortować w porządku niemalejącym. Można to zrobić wielokrotnie powtarzając krok:
  - wyszukaj najmniejszy element nieposortowanej części tablicy
  - wymień go z elementem znajdującym się na pierwszej pozycji nieposortowanej części tablicy
- **Pierwsza iteracja:** wybiera najmniejszy element w  $A[0, n-1]$ , zamienia z elementem na pozycji  $A[0]$ ;
- **Druga iteracja:** wybiera najmniejszy element w  $A[1, n-1]$ , zamienia z elementem na pozycji  $A[1]$ ;
- **Trzecia iteracja:** ...
- **I-ta iteracja** wymaga przejrzania  $(n-i)$  elementów.

po  $i+1$   
iteracjach



1	$A[0]$
2	
2	
4	
7	
7	$A[i]$
8	
9	
7	$A[n-1]$

# Sortowanie przez wybieranie – rekurencyjny algorytm sortujący

32

- Mamy tablicę  $A$  zawierającą  $n$  liczb całkowitych które chcemy posortować w porządku niemalejącym.
- Można to robić rekurencyjnie
  - wybieramy najmniejszy element z reszty tablicy  $A$  (czyli z  $A[i, \dots, n-1]$ ),
  - wymieniamy wybrany w poprzednim kroku element z elementem  $A[i]$ ,
  - sortujemy resztę tablicy czyli  $A[i+1, \dots, n-1]$ .
- Podstawa:
  - Jeśli  $i = n-1$ , to pozostaje do posortowania jedynie ostatni element tablicy. Ponieważ pojedynczy element jest zawsze posortowany nie trzeba podejmować żadnych działań.
- Indukcja:
  - Jeśli  $i < n-1$ , to należy znaleźć najmniejszy element w tablicy  $A[i, \dots, n-1]$ , wymienić go z elementem  $A[i]$  i rekurencyjnie posortować tablice  $A[i+1, \dots, n-1]$ .
- Kompletny algorytm realizujący powyższą rekurencję rozpoczyna się od  $i=0$ .

1	$A[0]$
2	
2	
4	
7	
7	$A[i]$
8	
9	
7	$A[n-1]$



# Sortowanie przez „dzielenie i scalanie” – rekurencyjny algorytm sortujący

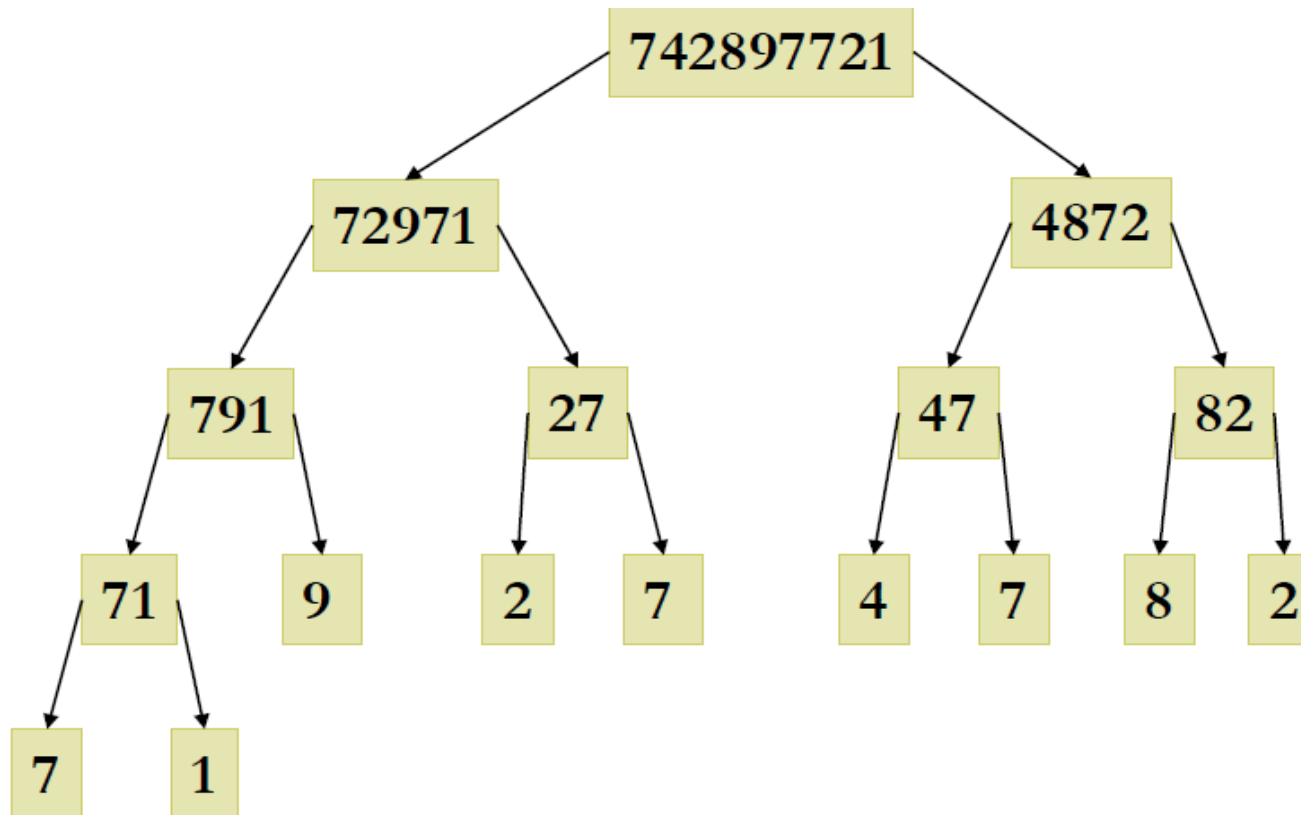
33

- Najlepszy opis **sortowania przez scalanie** opiera się na rekurencji i ilustruje równocześnie bardzo korzystne zastosowanie techniki „dziel i zwyciężaj”.
- Listę  $(a_1, a_2, a_3, \dots, a_n)$  sortuje się **dzieląc** na **dwie listy** o dwukrotnie mniejszych rozmiarach. Następnie obie listy są sortowane osobno. Aby zakończyć proces sortowania oryginalnej listy  $n$ -elementów, **obie listy zostają scalone** przy pomocy specjalnego algorytmu.
- **Scalanie:**
  - Prostym sposobem scalania dwóch list jest analiza od ich początków. W każdym kroku należy znaleźć mniejszy z dwóch elementów będących aktualnie na czele list, wybrać go jako kolejny element łączonej listy i usunąć go z „pierwotnej listy”, wskazując na kolejny pierwszy element. W przypadku równych pierwszych elementów można dodawać je do łączonej listy w dowolnej kolejności.

# Rekurencyjne dzielenie i scalanie

34

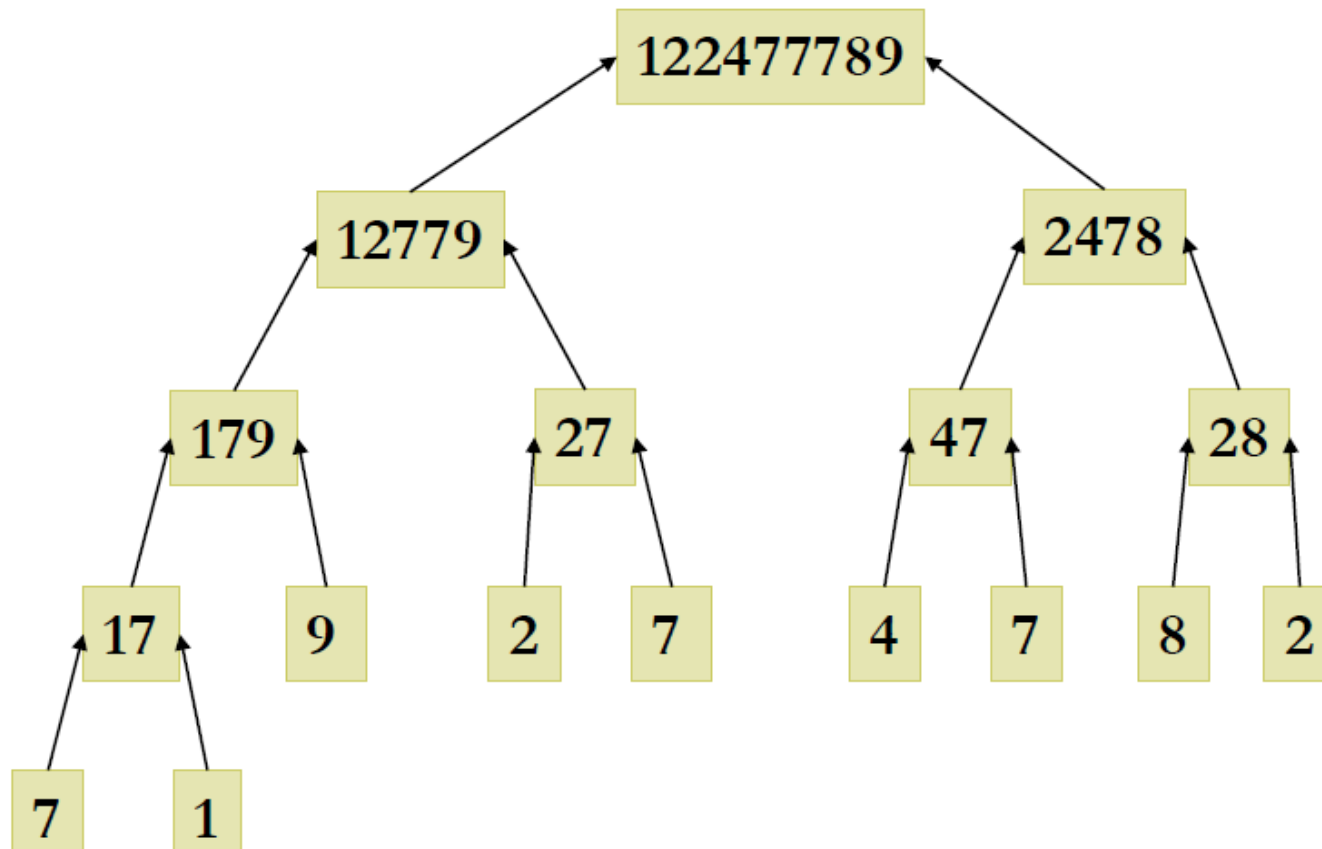
## □ Dzielenie:



# Rekurencyjne dzielenie i scalanie

35

## □ Scalanie:



# Sortowanie przez „dzielenie i scalanie” – rekurencyjny algorytm sortujący

36

## □ Podstawa:

- **Jeśli lista do posortowania jest pusta lub jednoelementowa, zostaje zwrócona ta sama lista – jest ona już posortowana.**

## □ Krok indukcyjny:

- **Jeżeli lista ma nie mniej niż 2 elementy to podziel listę na dwie (np. elementy o parzystym indeksie i elementy o nieparzystym indeksie). Posortuj każdą z dwóch list osobno i scal.**

# Techniki sortowania

37

- **Istnieje wiele różnych sposobów sortowania:**
  - ▣ **sortowanie przez wybieranie** jest łatwym ale wolnym sposobem sortowania,
  - ▣ **sortowanie przez scalanie** jest szybszym ale też bardziej skomplikowanym algorytmem,
  - ▣ **inne algorytmy sortowania:** patrz następne wykłady.
- **Można pokazać że algorytm sortowania przez wybieranie zachowuje się jak  $O(n^2)$ , natomiast algorytm sortowania przez scalanie  $O(n \log n)$ .**
- **To są tylko ograniczenia asymptotyczne, w praktyce już dla kilkudziesięciu elementów sortowanie przez scalanie jest szybsze.**

# Techniki sortowania

38

- Dla małych  $n$  algorytm sortowania przez wybieranie jest szybszy niż sortowania przez scalanie. Wobec tego optymalne będzie używanie go jako elementu algorytmu sortowania przez scalanie.
- Często rozważamy modyfikację algorytmu sortowania przez scalanie, w której doprowadzamy do podziału na  $n/k$  list, a każdą z nich sortujemy przez wybieranie (a więc nie doprowadzamy aż do list 2-elementowych!).

# Podsumowanie

39

- **Jak wskazuje praktyka programistyczna, większość algorytmów daje się zaliczyć do jednej z dwóch kategorii: pierwszą z nich tworzą algorytmy o charakterze iteracyjnym, drugą – zdecydowanie mniejszą – o charakterze rekurencyjnym.**
- **Dla wielu problemów znane jest tylko rozwiązanie rekurencyjne.**
- **Istnieje ściśle powiązanie dowodów indukcyjnych, definicji rekurencyjnych oraz programów rekurencyjnych. Każde opiera się na podstawie i kroku indukcyjnym.**
- **W indukcjach częściowych kolejne kroki zależą wyłącznie od kroków poprzednich.**
- **Często zachodzi konieczność przeprowadzania dowodów za pomocą indukcji zupełnej, w której każdy krok może zależeć od wszystkich wcześniejszych.**
- **Indukcja ma zasadnicze znaczenie w dowodzeniu poprawności działania programów lub ich fragmentów.**

# Wykład 4 – część III

40

Rozwiązywa  
nie rekurencji

- **Rekurencja dla algorytmu „dziel i zwyciężaj”**
- **Rozwiązywanie rekurencji**
  - **Metoda podstawiania**
  - **Metoda iteracyjna**
    - **Drzewa rekursji**
  - **Metoda uniwersalna**



# Czas działania programu

41

- Dla konkretnych danych wejściowych jest wyrażony liczba wykonanych prostych (elementarnych) operacji lub “kroków”. Jest dogodne założenie że operacja elementarna jest maszynowo niezależna.
- Każde wykonanie  $i$ -tego wiersza programu jest równe  $c_i$ , przy czym  $c_i$  jest stałą.
- Kiedy algorytm zawiera rekurencyjne wywołanie samego siebie, jego czas działania można często opisać zależnością rekurencyjną (rekurencja) wyrażającą czas dla problemu rozmiaru  $n$  za pomocą czasu dla podproblemów mniejszych rozmiarów.
- Możemy więc użyć narzędzi matematycznych aby rozwiązać rekurencje i w ten sposób otrzymać oszacowania czasu działania algorytmu.

# Rekurencja dla algorytmu typu „dziel i zwyciężaj”

42

- Rekurencja odpowiadającą czasowi działania algorytmu typu “dziel i zwyciężaj” opiera się na **podziale jednego poziomu rekursji na trzy etapy**.
  - ▣ Niech  $T(n)$  będzie czasem działania dla jednego problemu rozmiaru  $n$ .
  - ▣ Jeśli rozmiar problemu jest odpowiednio mały, powiedzmy  $n \leq c$  dla pewnej stałej  $c$ , to jego rozwiązanie zajmuje stały czas, co zapiszemy jako  $\Theta(1)$ .
  - ▣ Załóżmy że dzielimy problem na  $a$  podproblemów, każdy rozmiaru  $n/b$ . Jeśli  $D(n)$  jest czasem dzielenia problemu na podproblemy, a  $C(n)$  czasem scalania rozwiązań podproblemów w pełne rozwiązanie dla oryginalnego problemu, to otrzymujemy rekurencje
    - ▣  $T(n) = \Theta(1)$                       jeśli  $n \leq c$
    - ▣  $T(n) = a T(n/b) + D(n) + C(n)$     w przeciwnym przypadku

# Rekurencja dla algorytmu typu „dziel i zwyciężaj”

43

- **Przykład: algorytm sortowania przez scalanie**
  - **dziel:** znajdujemy środek przedziału, zajmuje to czas stały  $D(n) = \theta(1)$ ,
  - **zwyciężaj:** rozwiązujemy rekurencyjnie dwa podproblemy, każdy rozmiaru  $n/2$ , co daje czas działania  $2 T(n/2)$ ,
  - **połącz:** działa w czasie  $\theta(n)$ , a więc  $C(n) = \theta(n)$ .
- **Ostatecznie:**
  - $T(n) = \theta(1)$                       jeśli  $n=1$
  - $T(n) = 2 T(n/2) + \theta(1) + \theta(n)$       jeśli  $n>1$
- **Rozwiązaniem tej rekurencji jest  $T(n) = \theta(n \log n)$ .**

# Metody rozwiązywania rekurencji

44

## □ Metoda podstawiania:

- zgadujemy oszacowanie, a następnie dowodzimy przez indukcję jego poprawność.

## □ Metoda iteracyjna:

- przekształcamy rekurencję na sumę, korzystamy z technik ograniczania sum.

## □ Metoda uniwersalna:

- stosujemy oszacowanie na rekurencję mające postać  $T(n) = a T(n/b) + f(n)$ , gdzie  $a \geq 1$ ,  $b > 1$ , a  $f(n)$  jest daną funkcją.

# Metoda podstawiania

45

- Polega na **zgadnięciu postaci rozwiązania**, a następnie wykazaniu przez indukcję, że jest ono poprawne.
- Trzeba też znaleźć odpowiednie stałe.
- Bardzo skuteczna, stosowana tylko w przypadkach kiedy łatwo jest przewidzieć postać rozwiązania.

# Metoda podstawiania

46

## □ Przykład:

### □ Postać rekurencji:

$$T(n) = 2T(n/2) + n$$

### □ Zgadnięte rozwiązanie:

$$T(n) = \theta(n \log n)$$

### □ Podstawa:

$$n=2; T(1)=1; T(2)=4;$$

### □ Indukcja:

$$T(n) \leq 2 (c(n/2)\log(n/2)) + n \leq c n \log(n/2) + n$$

$$T(n) \leq c n \log(n/2) + n = c n \log(n) - c n \log(2) + n$$

$$T(n) \leq c n \log(n) - c n \log(2) + n = c n \log(n) - c n + n$$

$$T(n) \leq c n \log(n) - c n + n \leq c n \log(n)$$

spełnione dla  $c \geq 1$ ;

# Metoda iteracyjna

47

- Polega na **rozwijaniu (iterowaniu) rekurencji i wyrażanie jej jako sumy składników zależnych tylko od  $n$  warunków brzegowych. Następnie mogą być użyte techniki sumowania do oszacowania rozwiązania.**

# Metoda iteracyjna

48

## □ Przykład:

### □ **Postać rekurencji:**

$$T(n) = 3T(n/4) + n$$

### □ **Iterujemy:**

$$T(n) = n + 3T(n/4) = n + 3((n/4) + 3T(n/16)) = n + 3(n/4) + 9T(n/16)$$

$$T(n) = n + 3(n/4) + 9T(n/16) = n + 3n/4 + 9n/16 + 27T(n/64)$$

## □ **Iterujemy tak długo aż osiągniemy warunki brzegowe.**

**Składnik  $i$ -ty w ciągu wynosi  $3^i n/4^i$ .**

**Iterowanie kończymy, gdy  $n=1$  lub  $n/4^i = 1$  (czyli  $i > \log_4(n)$ ).**

$$T(n) \leq n + 3n/4 + 9n/16 + 27n/64 + \dots + 3 \log_4 n \theta(1)$$

$$T(n) \leq 4n + 3 \log_4 n \theta(1) = \theta(n)$$



# Metoda iteracyjna

49

- **Metoda iteracyjna jest zazwyczaj związana z dużą ilością przekształceń algebraicznych, więc zachowanie prostoty nie jest łatwe.**
- **Punkt kluczowy to skoncentrowanie się na dwóch parametrach:**
  - ▣ **liczbie iteracji koniecznych do osiągnięcia warunku brzegowego**
  - ▣ **oraz sumie składników pojawiających się w każdej iteracji.**

# Drzewa rekursji

50

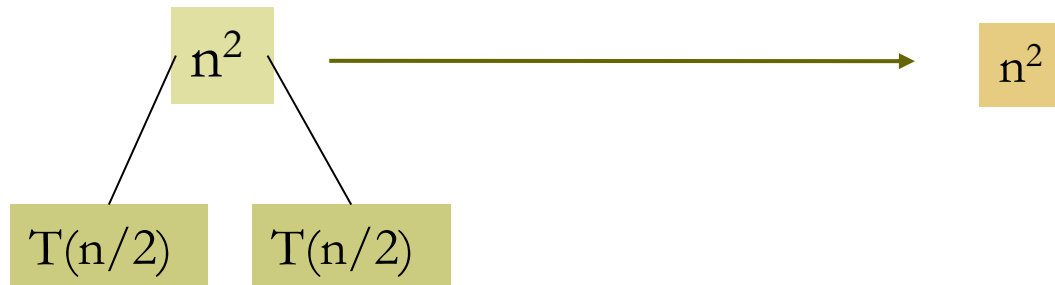
- **Pozwalają w dogodny sposób zilustrować rozwijanie rekurencji, jak również ułatwiają stosowanie aparatu algebraicznego służącego do rozwiązywania tej rekurencji.**
- **Szczególnie użyteczne gdy rekurencja opisuje algorytm typu “dziel i zwyciężaj”.**

# Drzewa rekursji dla algorytmu i zwyciężaj”

„dziel

51

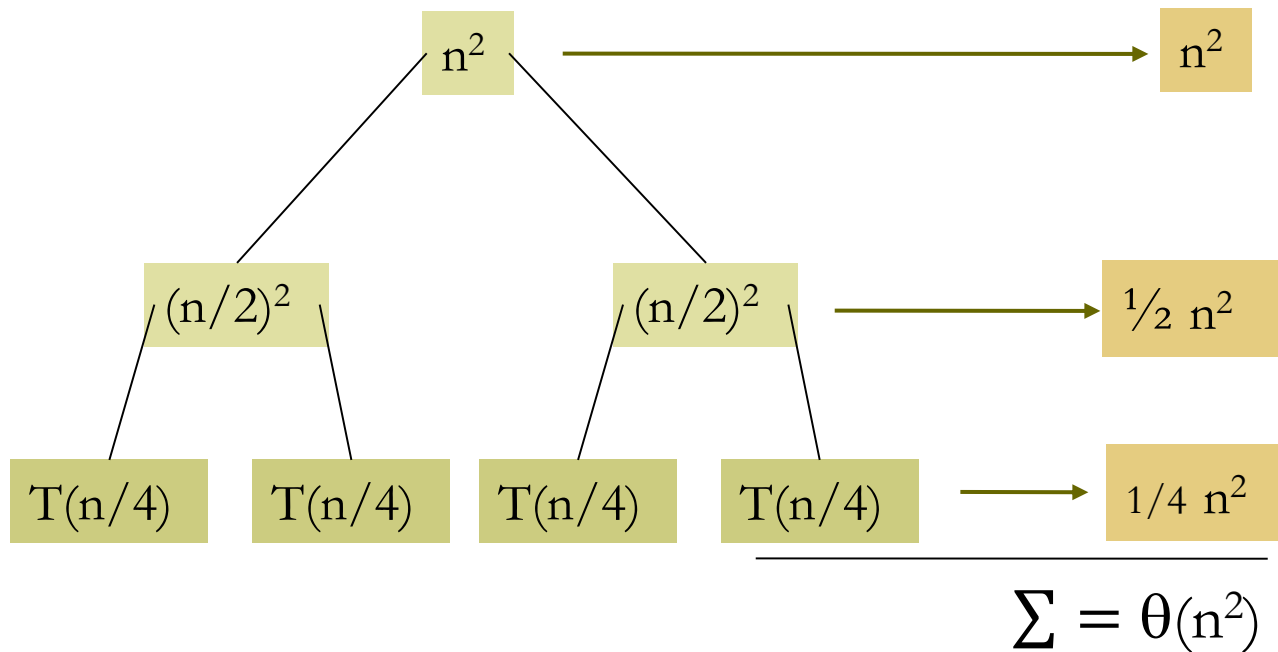
□  $T(n) = 2 T(n/2) + n^2$



# Drzewa rekursji dla algorytmu „dziel i zwyciężaj”

52

□  $T(n) = 2 T(n/2) + n^2$

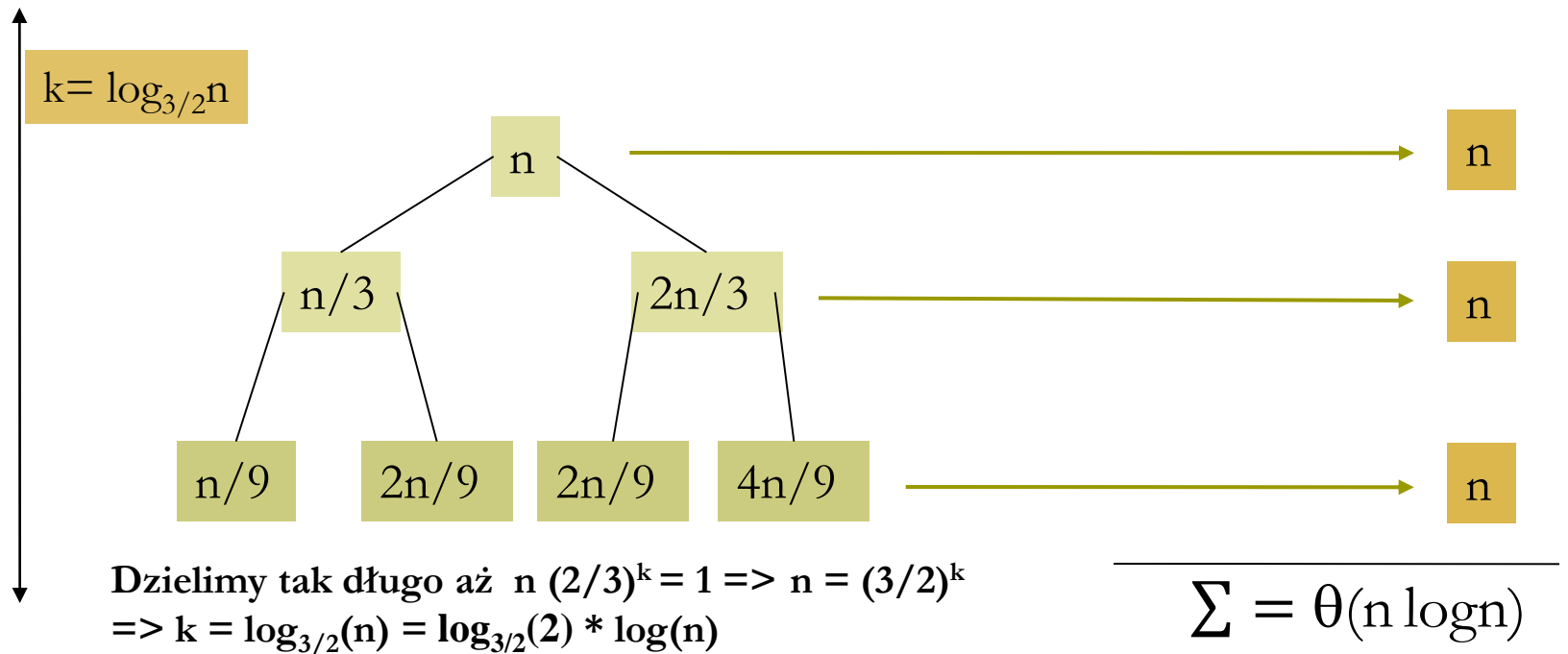


□ Rozwiązaniem tego równania jest  $T(n) = \theta(n^2)$

# Drzewa rekursji

53

□  $T(n) = T(n/3) + T(2n/3) + n$



□ Rozwiązaniem tego równania jest  $T(n) = \theta(n \log n)$

# Funkcje tworzące

54

- **Rekurencje były badane już od 1202 roku przez L. Fibonacciego**
- **A. de Moivre w 1730 roku wprowadził pojęcie funkcji tworzących do rozwiązywania rekurencji.**

# Metoda rekurencji uniwersalnej

55

- **Metoda rekurencji uniwersalnej podaje “uniwersalny przepis” rozwiązywania równania rekurencyjnego postaci:**
  - $T(n) = a T(n/b) + f(n)$   
gdzie  $a \geq 1$  i  $b > 1$  są stałymi, a  $f(n)$  jest funkcja asymptotycznie dodatnia.
- **Za wartość  $(n/b)$  przyjmujemy najbliższą liczbę całkowitą (mniejsza lub większą od wartości dokładnej).**

# Metoda rekurencji universalnej

56

- Rekurencja opisuje czas działania algorytmu, który dzieli **problem rozmiaru  $n$**  na  **$a$  problemów, każdy rozmiaru  $n/b$** , gdzie  $a$  i  $b$  są dodatnimi stałymi.
- Każdy z  $a$  problemów jest rozwiązywany rekurencyjnie w czasie  $T(n/b)$ .
- Koszt dzielenia problemu oraz łączenia rezultatów częściowych jest opisany funkcją  $f(n)$ .



# Twierdzenie o rekurencji uniwersalnej

57

- Niech  $a \geq 1$  i  $b > 1$  będą stałymi, niech  $f(n)$  będzie pewną funkcją i niech  $T(n)$  będzie zdefiniowane dla nieujemnych liczb całkowitych przez rekurencje

$$T(n) = a T(n/b) + f(n)$$

gdzie  $(n/b)$  oznacza najbliższą liczbę całkowitą do wartości dokładnej  $n/b$ .

- Wtedy funkcja  $T(n)$  może być ograniczona asymptotycznie w następujący sposób:
  - Jeśli  $f(n) = O(n^{\log_b a - \epsilon})$  dla pewnej stałej  $\epsilon > 0$ , to  $T(n) = \Theta(n^{\log_b a})$ .
  - Jeśli  $f(n) = \Theta(n^{\log_b a})$  to  $T(n) = \Theta(n^{\log_b a} \log n)$ .
  - Jeśli  $f(n) = n^{\log_b a + \epsilon}$  dla pewnej stałej  $\epsilon > 0$  i jeśli  $af(n/b) \leq cf(n)$  dla pewnej stałej  $c < 1$  i wszystkich dostatecznie dużych  $n$ , to  $T(n) = \Theta(f(n))$ .

# Twierdzenie o rekurencji uniwersalnej

58

## □ “Intuicyjnie...”:

- W każdym z trzech przypadków porównujemy funkcje  $f(n)$  z funkcją  $n^{\log_b a}$ . Rozwiązanie rekurencji zależy od większej z dwóch funkcji.
- Jeśli funkcja  $n^{\log_b a}$  jest większa, to rozwiązaniem rekurencji jest:
  - $T(n) = \Theta(n^{\log_b a})$ .
- Jeśli funkcje są tego samego rzędu, to mnożymy przez  $\log n$  i rozwiązaniem jest:
  - $T(n) = \Theta(n^{\log_b a} \log n) = T(n) = \Theta(f(n) \log n)$ .
- Jeśli  $f(n)$  jest większa, to rozwiązaniem jest:
  - $T(n) = \Theta(f(n))$ .

# Przykład

59

$$\square T(n) = 9 T(n/3) + n$$

$$a=9,$$

$$b=3,$$

$$f(n)=n,$$

a zatem  $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$ .

- $\square$  Ponieważ  $f(n) = O(n^{\log_3 9 - \epsilon})$ , gdzie  $\epsilon = 1$ , możemy zastosować przypadek 1 z twierdzenia i wnioskować że rozwiązaniem jest  $T(n) = \Theta(n^2)$ .

# Przykład

60

$$\square T(n) = T(2n/3) + 1$$

$$a=1,$$

$$b=3/2,$$

$$f(n)=1,$$

$$\text{a zatem } n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1.$$

- $\square$  Stosujemy przypadek 2, gdyż  $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$ ,  
a zatem rozwiązaniem rekurencji jest  $T(n) = \Theta(\log n)$ .

# Przykład

61

□  $T(n) = 3T(n/4) + n \log n$

$$a=3,$$

$$b=4,$$

$$f(n) = n \log n,$$

a zatem  $n^{\log_b a} = n^{\log_4 3} = O(n^{0,793})$ .

- Ponieważ  $f(n) = \Omega(n^{\log_4 3 + \epsilon})$ , gdzie  $\epsilon \approx 0.2$ , więc stosuje się tutaj przypadek 3, jeśli możemy pokazać że dla  $f(n)$  zachodzi warunek regularności.

Dla dostatecznie dużych  $n$ :

$$af(n/b) = 3(n/4)\log(n/4) \leq (3/4)n\log(n) = c f(n)$$

dla  $c=3/4$ .

- Warunek jest spełniony i możemy napisać że rozwiązaniem rekurencji jest  $T(n) = \Theta(n \log n)$ .

# Przykład

62

□  $T(n) = 2T(n/2) + n \log n$

$a=2,$

$b=2,$

$f(n)=n \log n,$

a zatem  $n^{\log_b a} = n.$

- Wydaje się że powinien to być przypadek 3, gdyż  $f(n)=n \log n$  jest asymptotycznie większe niż  $n^{\log_b a} = n$ , ale nie wielomianowo większy.
- Stosunek  $f(n)/ n^{\log_b a} = (n \log n)/n = \log n$  jest asymptotycznie mniejszy niż  $n^\epsilon$  dla każdej dodatniej stałej  $\epsilon$ .
- W konsekwencji rekurencja ta “wpada” w lukę między przypadkiem 2 i 3.

# Pytania do wykładu

63

- **Podaj przykład algorytmu iteracyjnego, wskaż gdzie w algorytmie występuje iteracja**
- **Podaj przykład algorytmu rekurencyjnego, wskaż gdzie w algorytmie występuje rekurencja.**
- **Na czym polega dowód indukcyjny?**
- **Na czym polega proces sortowania przez wybieranie? Jaką ma złożoność obliczeniową?**
- **Na czym polega proces sortowania przez „dzielenie i scalanie”? Jaką ma złożoność obliczeniową?**