

Teoretyczne podstawy informatyki

Wykład 4

Iteracja, indukcja i rekurencja

Iteracja, indukcja i rekurencja to podstawowe zagadnienia pojawiające się przy wielu typach modeli danych, struktur danych czy algorytmów.

Iteracja

Źródłem potęgi komputerów jest zdolność do wielokrotnego wykonywania tego samego zadania lub jego różnych wersji.

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, i kolejny element itd....

Iteracja

Programy i algorytmy wykorzystują iteracje do wielokrotnego wykonywania określonych zadań bez konieczności definiowania ogromnej liczby pojedynczych kroków, np. w przypadku zadania *wykonaj kolejny 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

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 $O(n)$

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

$$\langle f \rangle = \sum (c i + d) \cdot 1/n = c (n-1) / 2 + d$$

$$\langle f \rangle \sim c n/2 \quad \text{dla dużego } n$$

Iteratory

- 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

- 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

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 elementem poprzedzającym listę

- Definicje rekurencyjne są szeroko stosowane do specyfikacji gramatyk języków programowania (patrz następne wykłady)

Rekurencja

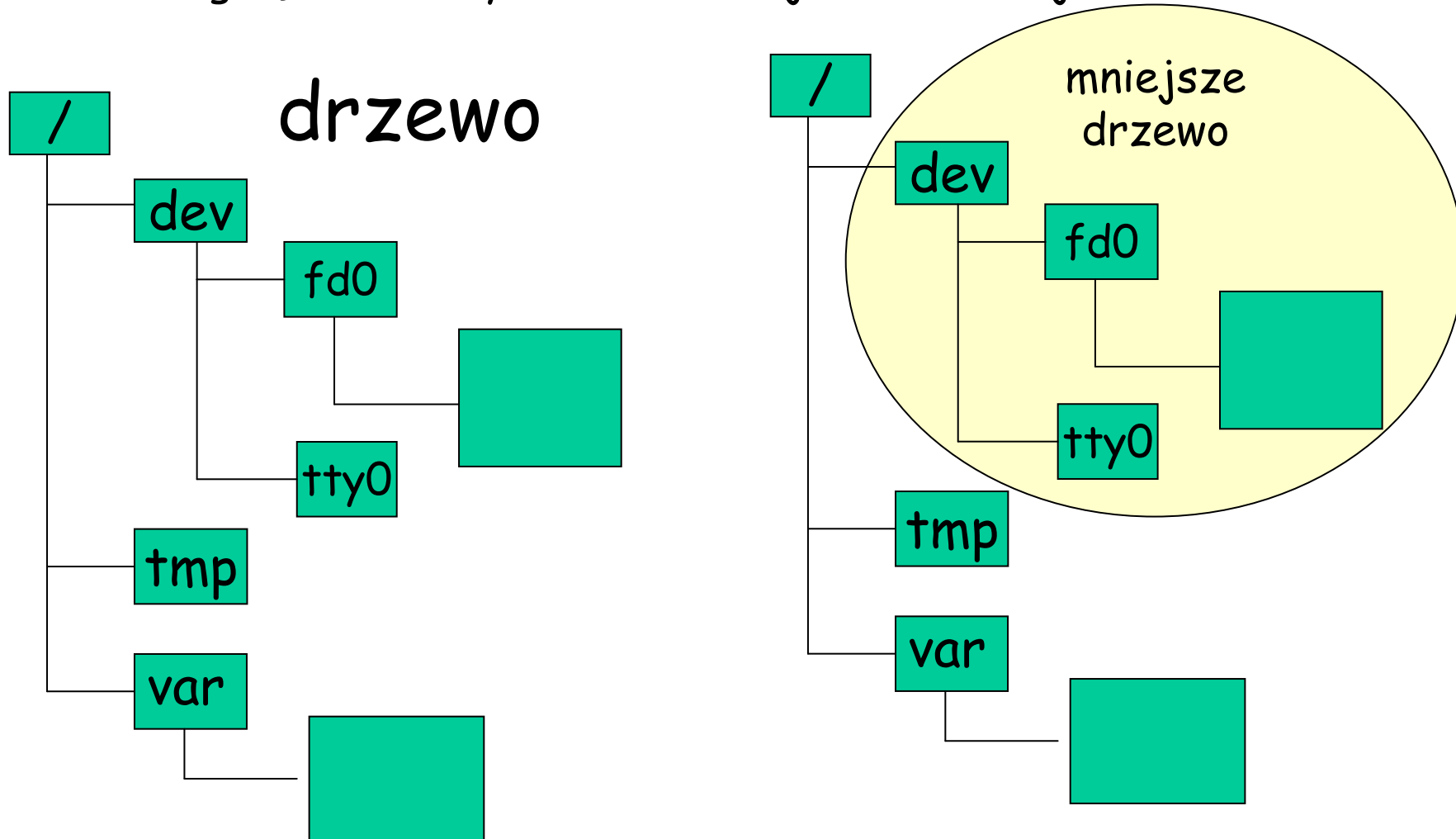
*„ Żeby zrozumieć rekurencję trzeba najpierw
zrozumieć rekurencję”*

(autor nieznany)

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 nazywana bywa powszechnie drzewem katalogów (directory tree) - drzewo to zakorzenione jest w katalogu najwyższego poziomu, zaś pliki mogą być uważane za liście tego drzewa.

Rekurencja

Każda z gałęzi może być traktowana jak inne mniejsze drzewo.



Rekurencja

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 w pewnym sensie przykładem wielokrotnego wykorzystywania tych samych rzeczy: metoda wywołuje samą siebie.

Definicja rekurencyjna

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

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

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 opracować algorytmy rekurencyjne, naśladując definicje rekurencyjne zawarte w specyfikacji programu, który jest implementowany.

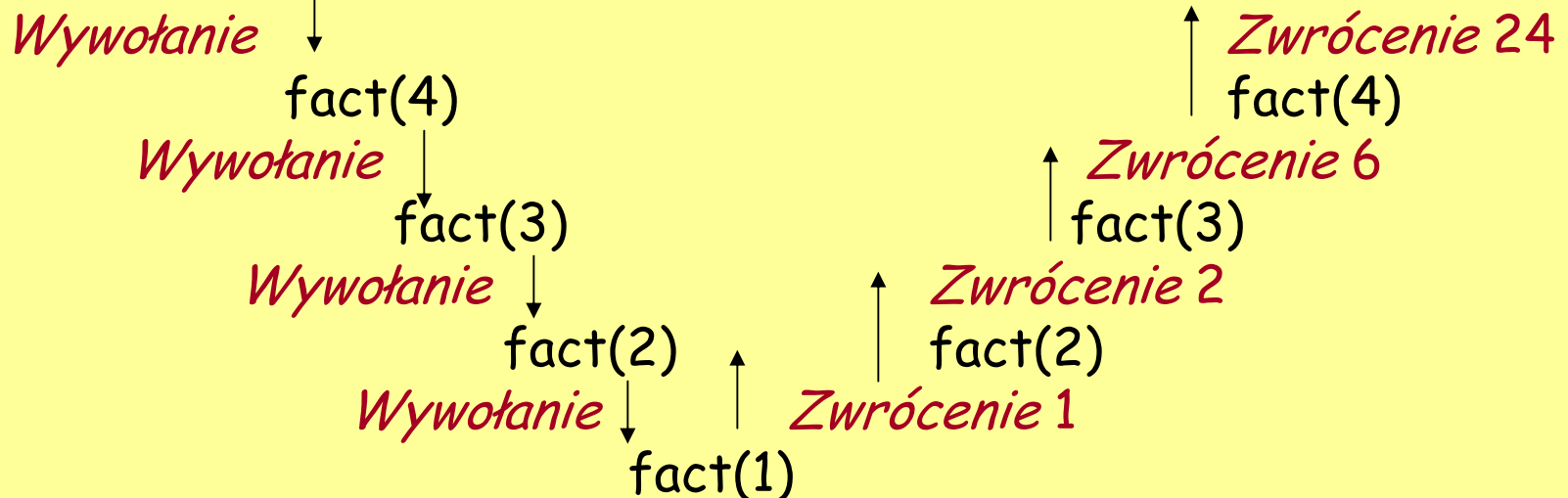
Definicja rekurencyjna

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 $n \geq 1$

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



Rekurencja czy iteracja.... czyli nie nadużywać rekurencji

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

```

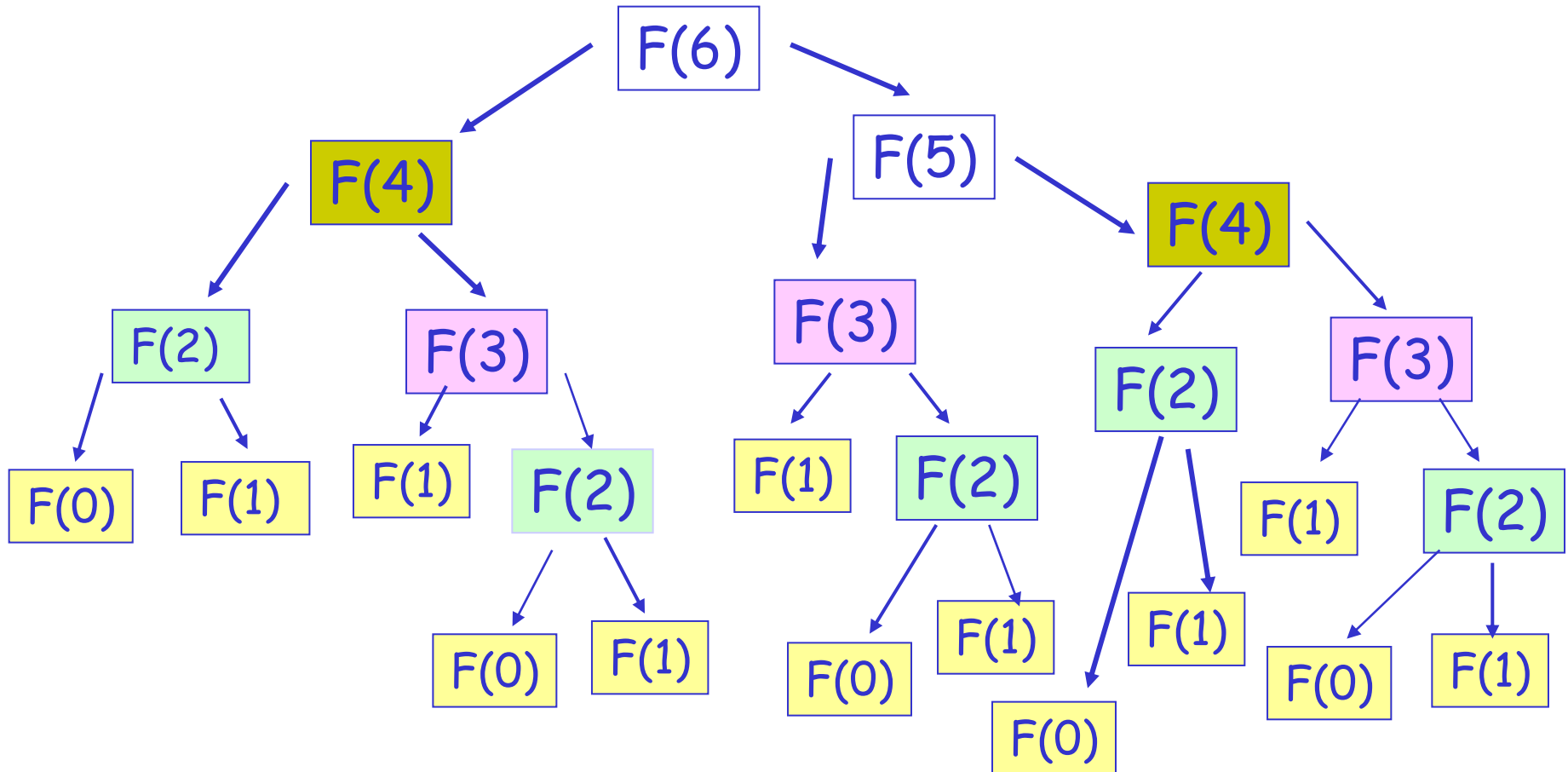
Fib (int n)
{ if (n < 2) return n;
  else
    return Fib(n-2) + Fib(n-1);
}

```

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

Trzeba było mieć dobry pomysł na prosty algorytm iteracyjny.

Jak rozwija się rekurencja dla obliczeń liczby Fibonacciego?



- 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.... czyli nie nadużywać rekurencji

Jak obliczać ciąg Fibonacciego?

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

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

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 jest $O(2^n)$, to zbyt wysoka cena za prostotę! ($3 \cdot 10^6$ wywołań dla $F(30)$). Algorytm iteracyjny jest $O(n)$.

Rekurencja czy iteracja....

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 (patrz dalsze wykłady)

Rekurencja czy iteracja....

Istnieje powszechne przekonanie że nauczenie się programowania iteracyjnego czy też stosowania nierekurencyjnych wywołań funkcji jest łatwiejsze niż nauczenie się programowania rekurencyjnego. Po zdobyciu odpowiedniego doświadczenia, często okazuje się że programowanie rekurencyjne jest równie łatwe.

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

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 ograniczenia dolnego.

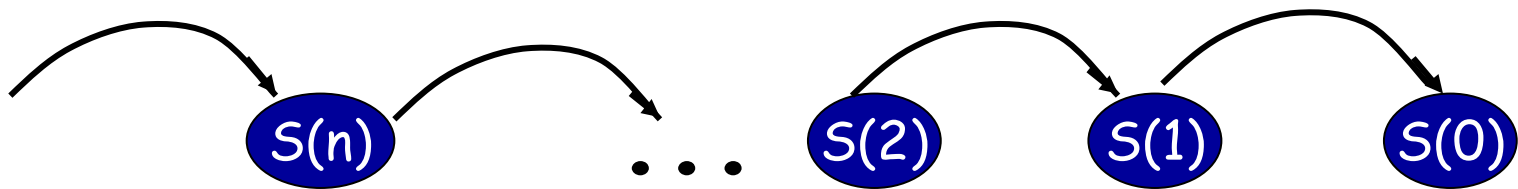
Indukcja

Niech $S(n)$ będzie dowolnym twierdzeniem dotyczącym liczby całkowitej n . W najprostszej formie dowodu indukcyjnego (**indukcja częściowa**) twierdzenia $S(n)$ dowodzi się dwóch faktów:

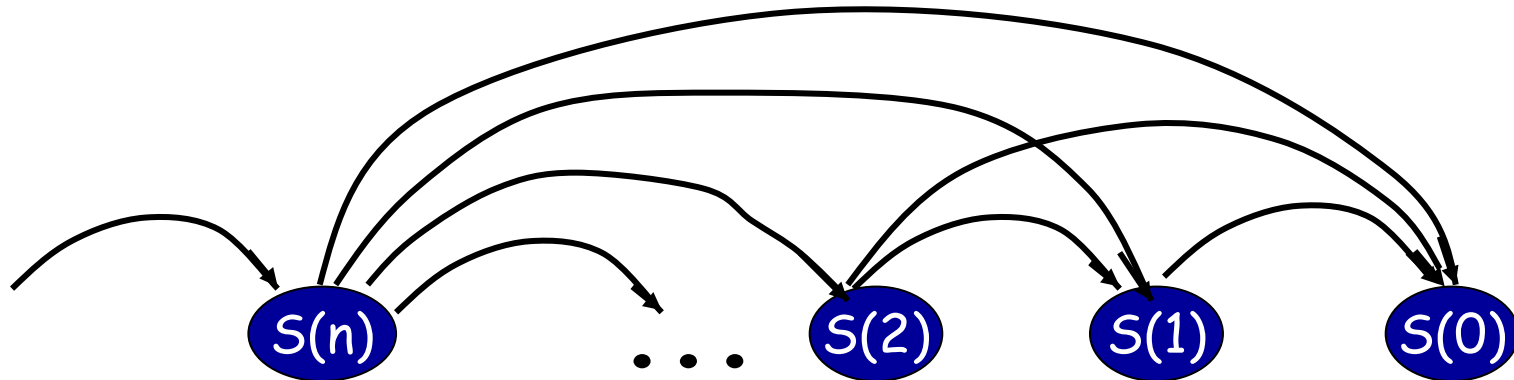
1. **Przypadku podstawowego:** za który często przyjmuje się twierdzenie $S(0)$. Przypadkiem podstawowym może jednak być również dobrze $S(k)$ dla dowolnej liczby całkowitej k . Dowodzi się wówczas prawdziwości twierdzenia $S(n)$ dla $n \geq k$.
2. **Kroku indukcyjnego:** gdzie dowodzi się, że dla wszystkich $n \geq 0$ (lub wszystkich $n \geq k$), prawdziwość $S(n)$ implikuje prawdziwość $S(n+1)$.

Indukcja zupełna i częściowa

- **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

Dla **indukcji zupełnej** dowodzimy, że twierdzenie $S(n)$, dla wszystkich $n \geq 0$ jest prawdziwe na podstawie dwóch faktów:

1. **Przypadku podstawowego:** dowodzi się prawdziwości $S(0)$ (lub $S(k)$ jeżeli to jest przypadek podstawowy)
2. **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

➤ *Indukcje z większą liczbą przypadków podstawowych*

Niekiedy przydatne jest wykorzystanie więcej niż jednego przypadku podstawowego:

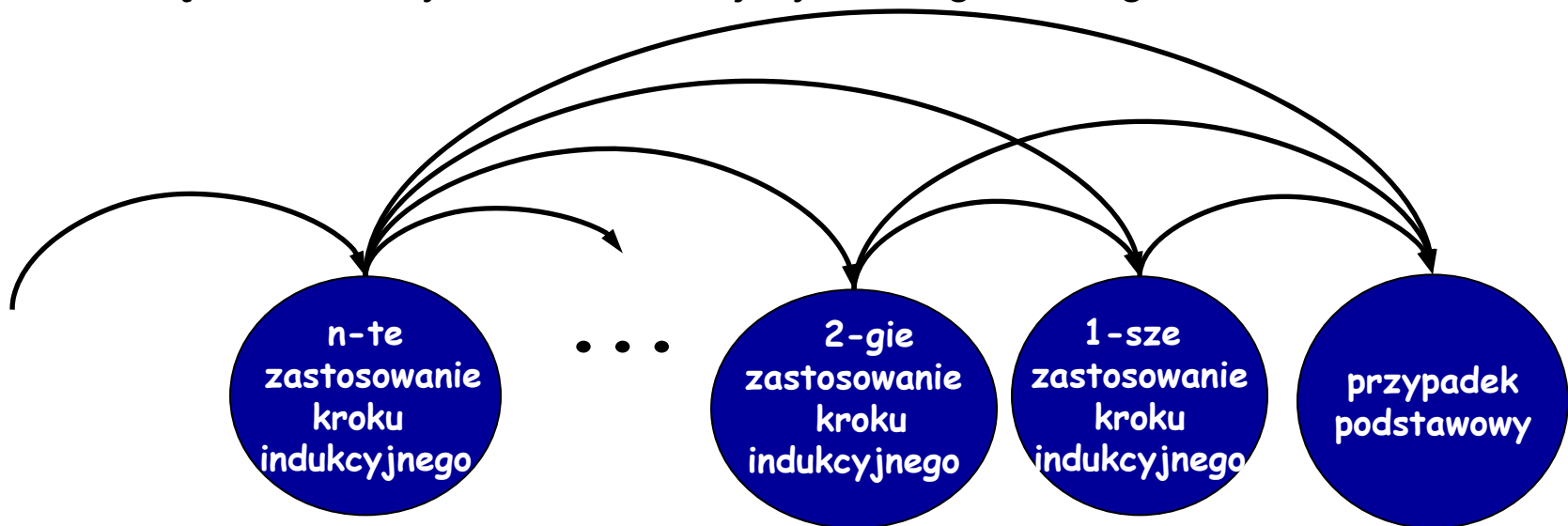
1. Przypadek podstawowy: dowodzi się poprawności wszystkich przypadków podstawowych, czyli $S(i_0)$, $S(i_1)$, $S(i_2), \dots, S(j_0)$
2. Krok indukcyjny: gdzie dowodzi się, że dla wszystkich $n \geq j_0$ (lub wszystkich $n > k$), że prawdziwość twierdzeń $S(i_0)$, $S(i_0+1), \dots, S(n)$ dla $n \geq j_0$, implikuje prawdziwość $S(n+1)$.

Definicje indukcyjne (raz jeszcze)

W definicji indukcyjnej (zwanej też rekursywną) definiuje się jedną lub więcej klas reprezentujących ściśle powiązane ze sobą obiekty (lub fakty) na bazie tych samych obiektów.

Definicja rekurencyjna powinna zawierać:

1. jedną lub więcej reguł podstawowych, z których niektóre definiują pewne obiekty proste
2. jedną lub więcej reguł indukcyjnych, za pomocą których definiuje się większe obiekty na bazie mniejszych z tego samego zbioru



Definicje indukcyjne

Istnieje ściśle powiązanie pojęć dowodów indukcyjnych, definicji rekurencyjnych oraz programów rekurencyjnych.

Każde 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 wcześniejszych.

➤ *Indukcja ma zasadnicze znaczenie w dowodzeniu poprawności programów lub ich fragmentów*

Elementy technik sortowania

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.

Listę elementów dowolnego typu można sortować wówczas, gdy istnieje możliwość zdefiniowania między nimi relacji mniejszości oznaczanej typowo " < "

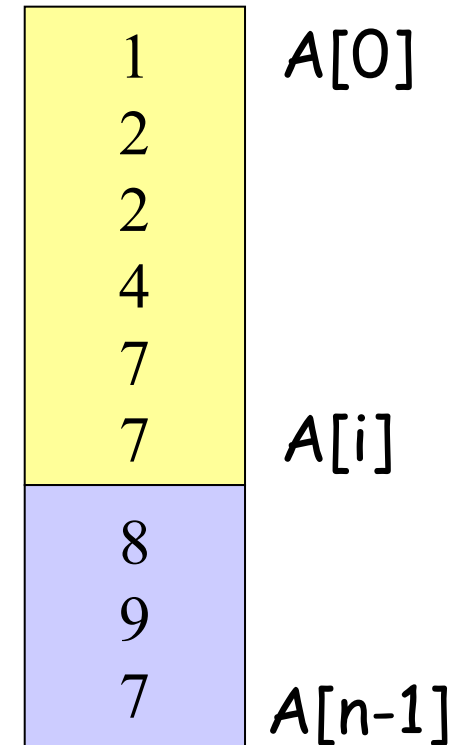
- Jeżeli elementami do posortowania są liczby całkowite lub rzeczywiste, symbol " < " oznacza znaną wszystkim relację mniejszości
- Jeżeli elementami są ciągi znaków, można np. stosować „porządek leksykograficzny”.
- Jeżeli elementy są skomplikowane (struktury) to możemy do posortowania wykorzystać część każdego elementu (jedno konkretne pole).

Sortowanie przez wybieranie - iteracyjny alg. sortujący

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 ostatnim elementem znajdującym się na pierwszej pozycji nieposortowanej części tablicy

po $i+1$
iteracjach
→



1. **Pierwsza iteracja:** wybiera najmniejszy element w $A[0, n-1]$, zamienia z elementem na pozycji $A[0]$;
2. **Druga iteracja:** wybiera najmniejszy element w $A[1, n-1]$, zamienia z elementem na pozycji $A[1]$;
3. **Trzecia iteracja:**

➤ i -ta iteracja wymaga przejrzania $(n-i)$ elementów

Sortowanie przez wybieranie - rekurencyjny alg. sortujący

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

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 posortowane osobno. Aby zakończyć proces sortowania oryginalnej listy n -elementów, **obie listy zostają scalone** przy pomocy specjalnego algorytmu.

Scalanie:

Prostym sposobem scalania dwóch liczb 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 postępować dowolnie.

Sortowanie przez „dzielenie i scalanie” - rekurencyjny algorytm sortujący

Podstawa:

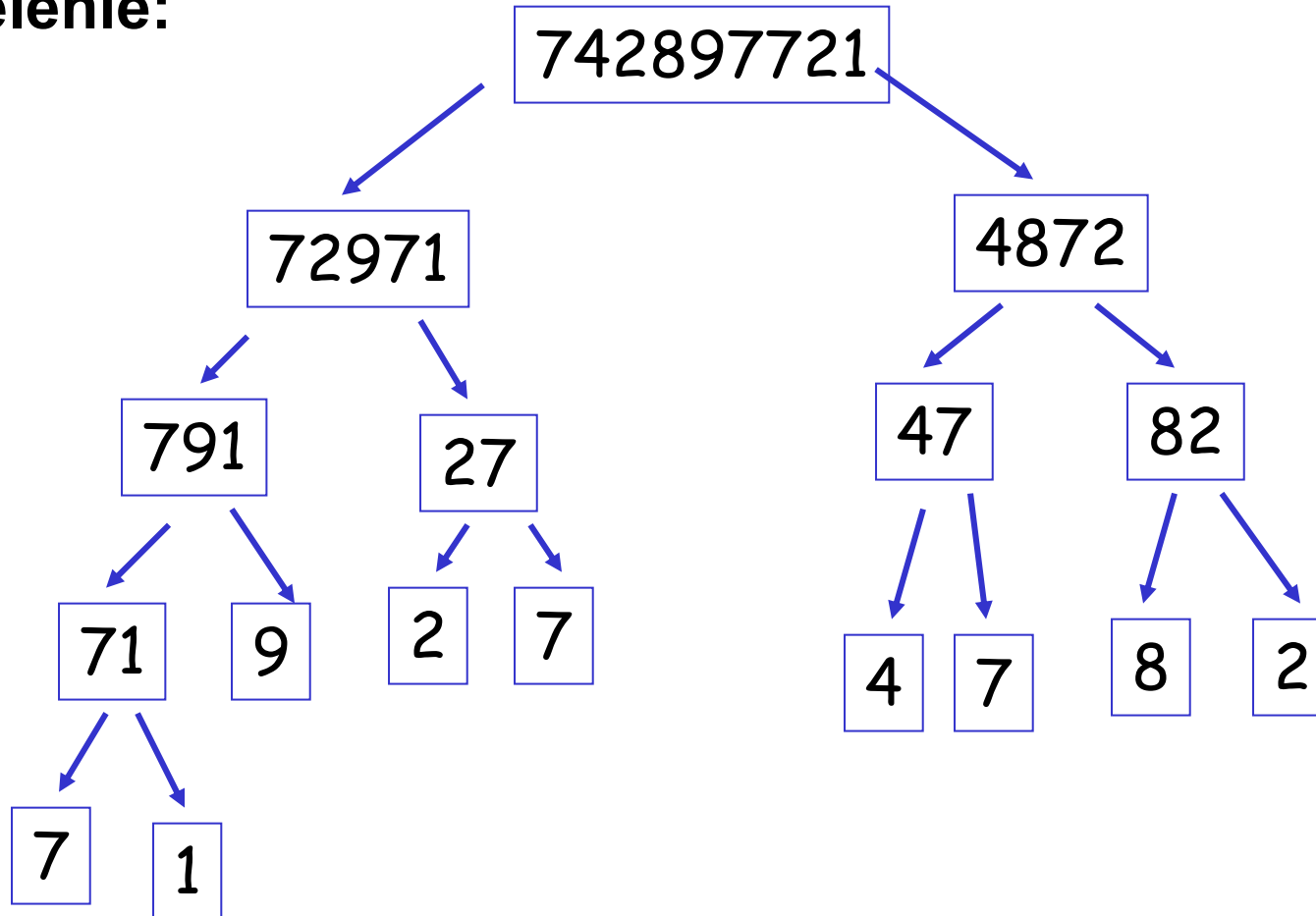
Jeśli lista do posortowania jest pusta lub jednoelementowa, zostaje zwrócona ta sama lista – jest ona już posortowana.

Krok indukcyjny:

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

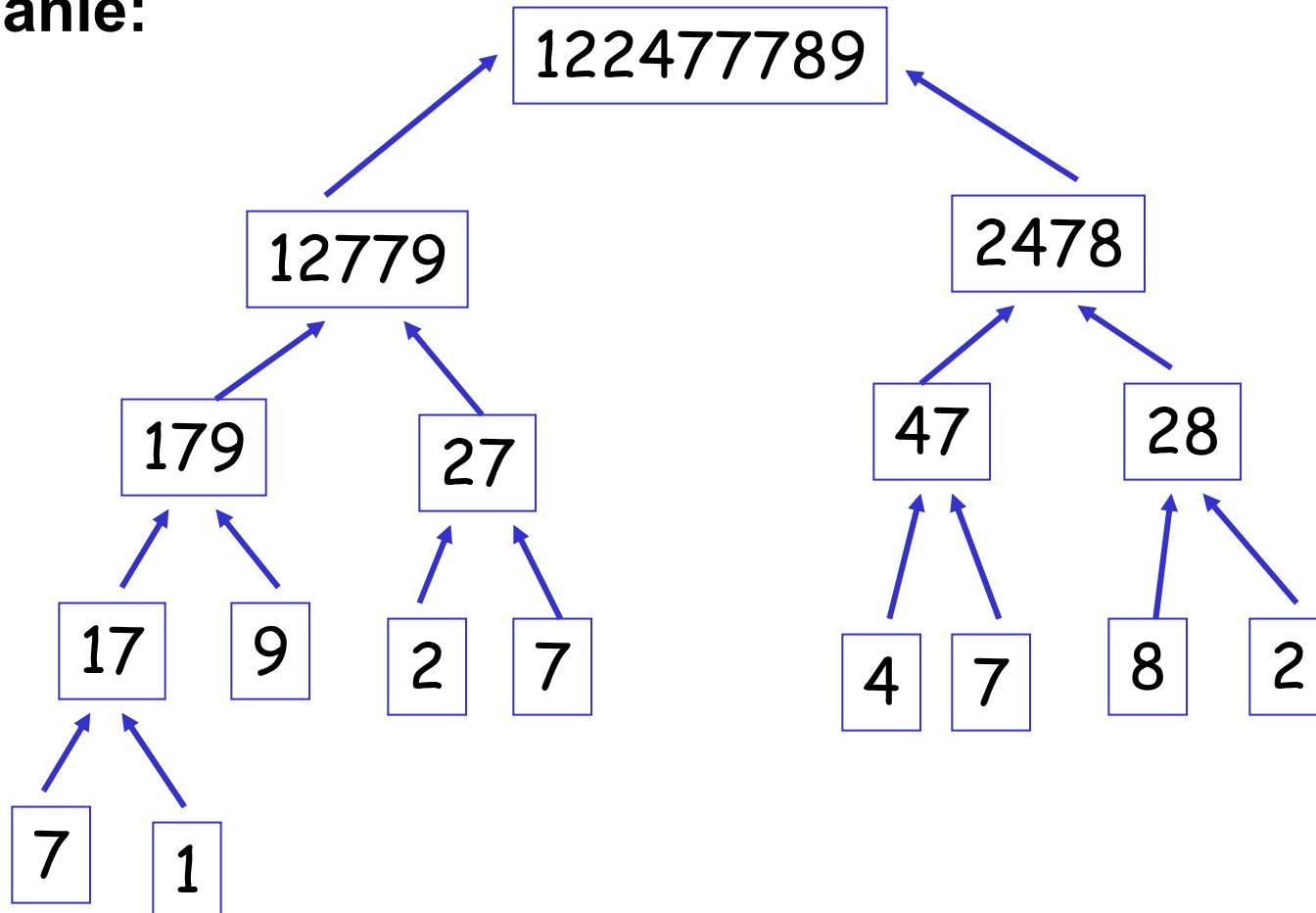
Rekurencyjne dzielenie i scalanie

Dzielenie:



Rekurencyjne dzielenie i scalanie

Scalanie:



Techniki sortowania

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 $O(n^2)$, **algorytm sortowania przez scalanie** $O(n \log n)$.

To są tylko ograniczenia górne, w praktyce już dla kilkudziesięciu elementów sortowanie przez scalanie jest szybsze.

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.

Rozważamy często 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!)

Posumowanie

- ⇒ 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 działaniu kroku indukcyjnego.
- ⇒ 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