Wykład 3:
Złożoność obliczeniowa algorytmów
Elementy kombinatoryki

Złożoność obliczeniowa algorytmów

Złożoność obliczeniowa i asymptotyczna

- Złożoność obliczeniowa:
  - Jest to miara służąca do porównywania efektywności algorytmów.
  - Mamy dwa kryteria efektywności:
    - czas,
    - pamięć.
- Do oceny efektywności stosujemy jednostki logiczne wyrażające związek miedzy **rozmiarem danych *N*** (wielkość pliku lub tablicy) a **ilością czasu *T*** potrzebną na ich przetworzenie.
- Funkcja wyrażająca zależność miedzy ***N*** a ***T*** jest zwykle bardzo skomplikowana, a jej obliczenie ma znaczenie jedynie w odniesieniu do dużych rozmiarów danych
- Przybliżona miara efektywności to tzw. złożoność asymptotyczna.

Szybkość wzrostu poszczególnych składników funkcji

Funkcja: $f(n) = n^2 + 100n + \log_{10} n + 1000$

n	f(n)	n^2	100•n	$\log_{10} n$	1000
1	1 101	0.1%	9%	0.0%	91%
10	2 101	4.8%	48%	0.05%	48%
100	21 002	48%	48%	0.001%	4.8%
10^3	1 101 003	91%	9%	0.0003%	0.09%
10^4		99%	1%	0.0%	0.001%
10^5		99.9%	0.1%	0.0%	0.0000%

- **n** – rozmiar danych, **f(n)** – ilość wykonywanych operacji
- Dla dużych wartości **n**, funkcja rośnie jak n^2, pozostałe składniki mogą być zaniedbane.

Notacja „wielkie *O*”

(wprowadzona przez P. Bachmanna w 1894r)

Definicja:

f(n) jest *O*(g(n)), jeśli istnieją liczby dodatnie c i N takie że:

$f(n) < c \cdot g(n)$ dla wszystkich $n \geq N$.

Przykład:

$f(n) = n^2 + 100n + \log_{10} n + 1000$

możemy przybliżyć jako:

$f(n) \approx n^2 + 100n + O(\log_{10} n)$

albo jako:

$f(n) \approx O(n^2)$

Notacja **„wielkie *O*”** ma kilka pozytywnych własności które możemy wykorzystać przy szacowaniu efektywności algorytmów.

Własności notacji „wielkie *O*"

Własność 1 (przechodniość):
Jeśli f(n) jest *O*(g(n)) i g(n) jest *O*(h(n)), to f(n) jest *O*(h(n))

Własność 2:
Jeśli f(n) jest *O*(h(n)) i g(n) jest *O*(h(n)), to f(n)+g(n) jest *O*(h(n))

Własność 3:
Funkcja an^k jest $O(n^k)$

Własność 4:
Funkcja n^k jest $O(n^{k+j})$ dla dowolnego dodatniego j

Z tych wszystkich własności wynika, że dowolny wielomian jest **„wielkie *O*"** dla **n** podniesionego do najwyższej w nim potęgi, czyli :
$f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$ jest $O(n^k)$

(jest też oczywiście $O(n^{k+j})$ dla dowolnego dodatniego **j**)

Własności notacji „wielkie *O*"

Własność 5:
Jeśli f(n) = c g(n), to f(n) jest *O*(g(n))

Własność 6:
Funkcja $\log_a n$ jest *O*($\log_b n$) dla dowolnych a i b większych niż 1

Własność 7:
$\log_a n$ jest *O*($\log_2 n$) dla dowolnego dodatniego a

- Jedną z najważniejszych funkcji przy ocenianiu efektywności algorytmów jest **funkcja logarytmiczna**.
- Jeżeli można wykazać że złożoność algorytmu jest rzędu logarytmicznego, algorytm można traktować jako bardzo dobry.
- Istnieje wiele funkcji lepszych w tym sensie niż logarytmiczna, jednak zaledwie kilka spośród nich, jak ***O*($\log_2 \log_2 n$)** czy ***O*(1)** ma praktyczne znaczenie.

Notacja Ω i Θ

- Notacja „wielkie *O*" odnosi się do górnych ograniczeń funkcji.
- Istnieje symetryczna definicja dotycząca dolnych ograniczeń:

Definicja
f(n) jest Ω(g(n)), jeśli istnieją liczby dodatnie c i N takie że, f(n) $\geq$ c g(n) dla wszystkich n $\geq$ N.

Równoważność
f(n) jest Ω(g(n)) wtedy i tylko wtedy, gdy g(n) jest $\mathcal{O}$(f(n))

Definicja
f(n) jest Θ(g(n)), jeśli istnieją takie liczby dodatnie c_1, c_2 i N takie że, $c_1 g(n) \leq f(n) \leq c_2 g(n)$ dla wszystkich n $\geq$ N.

Możliwe problemy

- Celem wprowadzonych wcześniej sposobów zapisu (notacji) jest porównanie efektywności rozmaitych algorytmów zaprojektowanych do rozwiązania tego samego problemu.
- Jeżeli będziemy stosować tylko notacje „wielkie O" do reprezentowania złożoności algorytmów, to niektóre z nich możemy zdyskwalifikować zbyt pochopnie.

- **Przykład:**
  - Załóżmy, że mamy dwa algorytmy rozwiązujące pewien problem, wykonywana przez nie liczba operacji to odpowiednio $\mathbf{10^8 n}$ i $\mathbf{10n^2}$. Pierwsza funkcja jest $\mathbf{O(n)}$, druga $\mathbf{O(n^2)}$.
  - Opierając się na informacji dostarczonej przez notacje „wielkie *O*" odrzucilibyśmy drugi algorytm ponieważ funkcja kosztu rośnie zbyt szybko.
  - To prawda … ale dopiero dla odpowiednio dużych $\mathbf{n}$, ponieważ dla $\mathbf{n<10^7}$ drugi algorytm wykonuje mniej operacji niż pierwszy.
  - Istotna jest więc też stała $\mathbf{(10^8)}$, która w tym przypadku jest zbyt duża aby notacja była znacząca.

Przykłady rzędów złożoności

- Algorytmy można klasyfikować ze względu na złożoność czasową lub pamięciową. W związku z tym wyróżniamy wiele klas algorytmów.
  - **Algorytm stały:** czas wykonania pozostaje taki sam niezależnie od ilości przetwarzanych elementów.
  - **Algorytm kwadratowy:** czas wykonania wynosi $O(n^2)$.
  - **Algorytm logarytmiczny:** czas wykonania wynosi $O(\log n)$.
  - itd …
- **Analiza złożoności algorytmów jest niezmiernie istotna** i nie można jej lekceważyć argumentując potencjalną szybkością obliczeń komputera. Nie sposób jej przecenić szczególnie zastanawiajac się nad doborem struktury danych.

Klasy algorytmów i ich czasy wykonania na komputerze działającym z szybkością 1 instrukcja/μs

klasa	złożoność	liczba operacji i czas wykonania			
	n	**10**		**10^3**	
stały	$O(1)$	1	1μs	1	1 μs
logarytmiczny	$O(\log n)$	3.32	3μs	9.97	10 μs
liniowy	$O(n)$	10	10μs	10^3	1ms
kwadratowy	$O(n^2)$	10^2	100μs	10^6	1s
wykładniczy	$O(2^n)$	1024	10ms	10^{301}	$>> 10^{16}$ lat

Funkcje niewspółmierne

- Bardzo wygodna jest możliwość porównywania dowolnych funkcji **f(n)** i **g(n)** za pomocą notacji „duże *O*”
  - albo **f(n) = *O*(g(n))**
  - albo **g(n) = *O*(f(n))**

  Albo jedno i drugie czyli **f(n) = Θ(g(n))**.
- Istnieją pary funkcji niewspółmiernych (ang. *incommensurate*), z których żadne nie jest „dużym *O*” dla drugiej.

  Przykład:

  Rozważmy funkcję **f(n)=n** dla nieparzystych **n** oraz **f(n)=n^2** dla parzystych **n**.
  Oznacza to, że f(1)=1, f(2)=4, f(3)=3, f(4)=16, f(5)=5 itd…
  Podobnie, niech **g(n)=n^2** dla nieparzystych **n** oraz **g(n)=n** dla parzystych **n**.
  W takim przypadku, funkcja **f(n)** nie może być ***O*(g(n))** ze względu na parzyste argumenty **n**, analogicznie **g(n)** nie może być ***O*(f(n))** ze względu na nieparzyste elementy **n**.
  Obie funkcje mogą być ograniczone jako ***O*(n^2)**.

Analiza czasu działania programu

- Mając do dyspozycji definicję „duże *O*" oraz własności (1)-(7) będziemy mogli, wg. kilku prostych zasad, skutecznie analizować czasy działania większości programów spotykanych w praktyce.
- Efektywność algorytmów ocenia się przez szacowanie ilości czasu i pamięci potrzebnych do wykonania zadania, dla którego algorytm został zaprojektowany.
- Najczęściej jesteśmy zainteresowani ***złożonością czasową***, mierzoną zazwyczaj liczbą przypisań i porównań realizowanych podczas wykonywania programu.
- Bardzo często interesuje nas tylko ***złożoność asymptotyczna***, czyli czas działania dla dużej ilości analizowanych zmiennych.

Czas działania instrukcji prostych

- Przyjmujemy zasadę że czas działania pewnych prosty operacji na danych wynosi O(1), czyli jest niezależny od rozmiaru danych wejściowych.
  - Operacje arytmetyczne, np. (+), (-)
  - Operacje logiczne (&&)
  - Operacje porównania (<=)
  - Operacje dostępu do struktur danych, np. indeksowanie tablic (A[i])
  - Proste przypisania, np. kopiowanie wartości do zmiennej.
  - Wywołania funkcji bibliotecznych, np. scanf lub printf
- Każdą z tych operacji można wykonać za pomocą pewnej (niewielkiej) liczby rozkazów maszynowych.

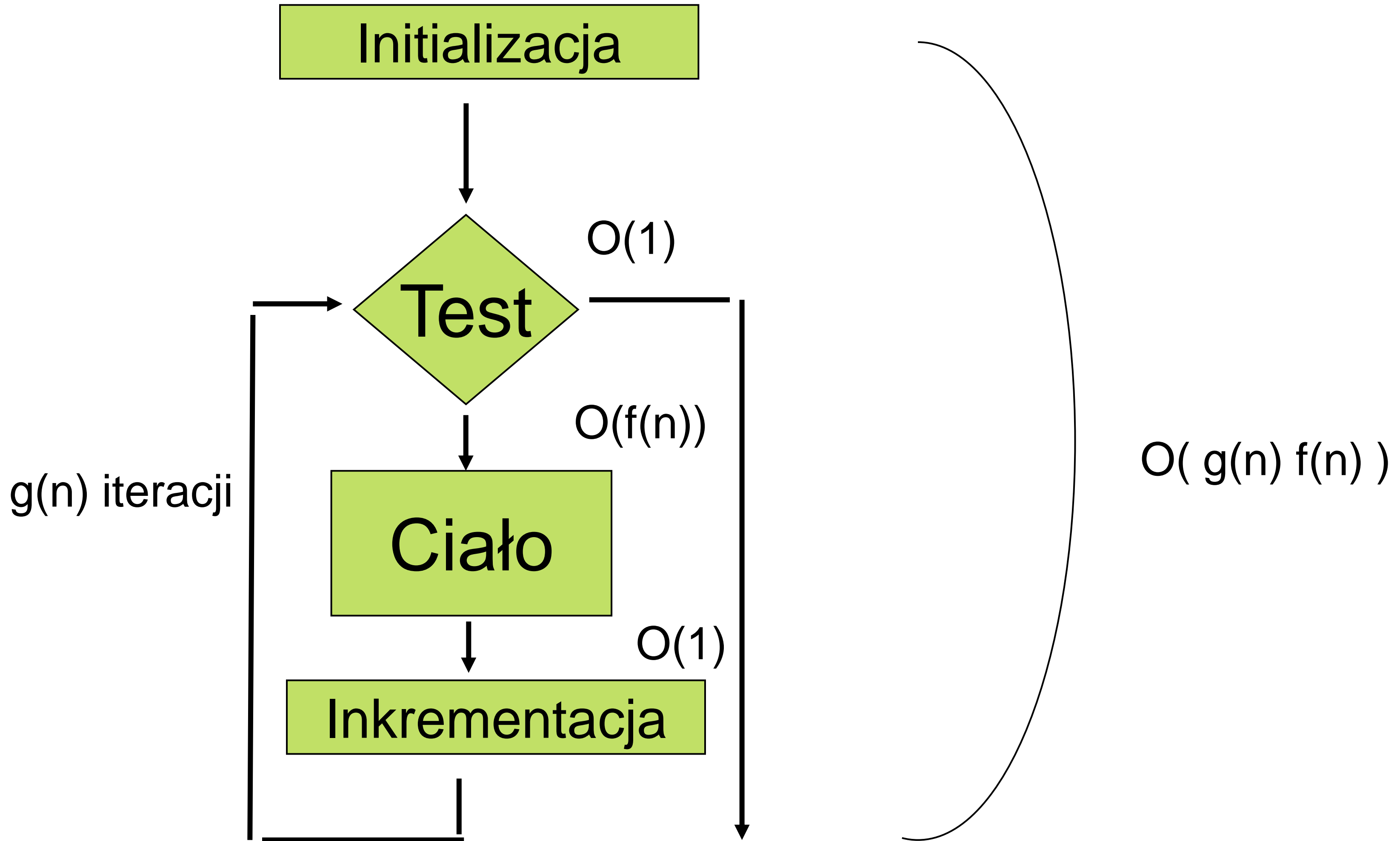
Czas działania pętli „for”

- Przykład 1: Prosta pętla

  ```
  for (i=sum=0; i<n; i++) sum+=a[i];
  ```

- Powyższa **pętla powtarza się n razy**, podczas każdego jej przebiegu realizuje **dwa przypisania**:
  - aktualizujące zmienną „sum”
  - zmianę wartości zmiennej „i”
- Mamy zatem **2n** przypisań podczas całego wykonania pętli.
- Złożoność asymptotyczna algorytmu **jest *O*(n)**.

Czas działania instrukcji „for"



Czas działania pętli „for"

- Przykład 2: Pętla zagnieżdżona

```
for  (i=0; i<n; i++)     {
          for (j=1, sum=a[0]; j<=i; j++)
                         sum+=a[j];  }
```

- Na samym początku zmiennej „i" nadawana jest wartość początkowa.
- **Pętla zewnętrzna powtarza się n razy**, a w każdej jej iteracji wykonuje się wewnętrzna pętla oraz instrukcja przypisania wartości zmiennym „i", „ j", „sum".
- **Pętla wewnętrzna wykonuje się „i" razy** dla każdego $i \in \{1, \ldots, n-1\}$, a na każdą iteracje przypadają dwa przypisania: jedno dla „sum", jedno dla „j".
- Mamy zatem: $1+3n+2(1+2+...+n-1) = 1+3n+n(n-1) = O(n)+O(n^2) = O(n^2)$ przypisań wykonywanych w całym programie.
- **Złożoność asymptotyczna algorytmu jest $O(n^2)$.** Pętle zagnieżdżone mają zwykle większą złożoność niż pojedyncze, jednak nie musi tak być zawsze.

Czas działania pętli „for"

- Przykład 3: Znajdź najdłuższą podtablice zawierającą liczby uporządkowane rosnąco.

```
for (i=0; len=1; i<n-1; i++) {
        for (i1=i2=k=i; k<n-1 && a[k]<a[k+1]; k++,i2++);
                if(len < i2-i1+1)  len=i2-i1+1; }
```

- Jeśli liczby w tablicy są uporządkowane malejąco, to pętla zewnętrzna wykonuje się n-1 razy, a w każdym jej przebiegu pętla wewnętrzna wykona się tylko raz.
  Złożoność asymptotyczna algorytmu jest więc $O(n)$.
- Jeśli liczby w tablicy są uporządkowane rosnąco, to pętla zewnętrzna wykonuje się n-1 razy, a w każdym jej przebiegu pętla wewnętrzna wykona się i razy dla $i \in \{1,...,n-1\}$.
  Złożoność asymptotyczna algorytmu jest więc $O(n^2)$.

Czas działania pętli „for”

- Z reguły dane nie są uporządkowane i ocena złożoności algorytmu jest rzeczą niełatwą ale bardzo istotną.
- Staramy się wyznaczy złożoność
  - w „przypadku optymistycznym”,
  - „przypadku pesymistycznym”
  - oraz „przypadku średnim”.
- Często posługujemy się przybliżeniami opartymi o notacje „wielkie O, Ω i Θ”.

Czas działania instrukcji warunkowych

Instrukcje warunkową **if-else** zapisuje się w postaci:

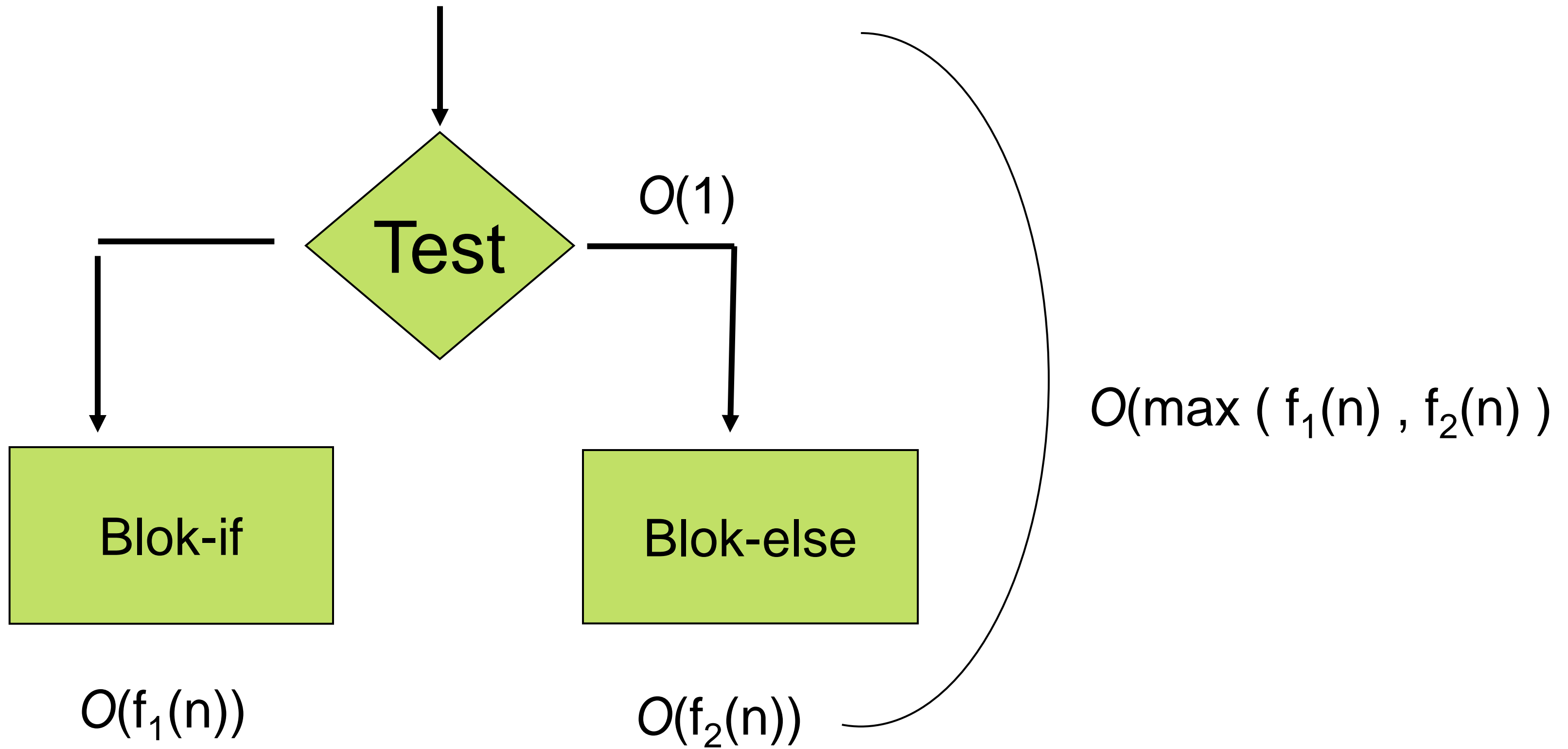
```
 if (<warunek>)
     <blok-if>
else
     <blok-else>
```

Gdzie

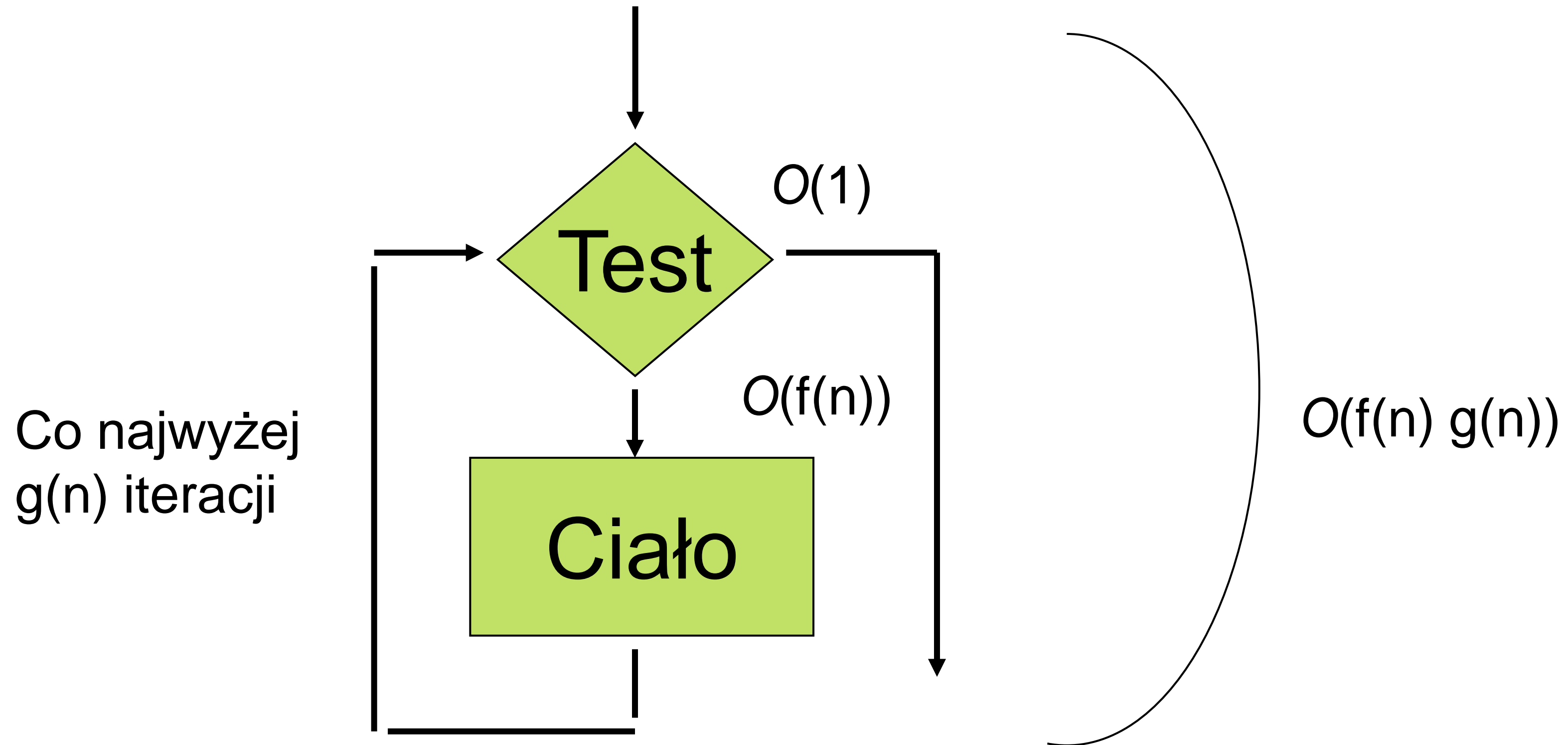
1. **<warunek>** jest wyrażeniem które trzeba obliczyć. Warunek niezależnie od tego jak skomplikowany wymaga wykonania stałej liczby operacji chyba ze zawiera wywołanie funkcji, więc czasu $O(1)$.
2. **<blok-if>** zawiera instrukcje wykonywane tylko w przypadku gdy warunek jest prawdziwy, czas działania **f(n)**.
3. **<blok-else>** wykonywany jest tylko w przypadku gdy warunek jest fałszywy, czas działania **g(n)**.

Czas działania instrukcji warunkowej należy zapisać jako $O(\max(f(n), g(n)))$

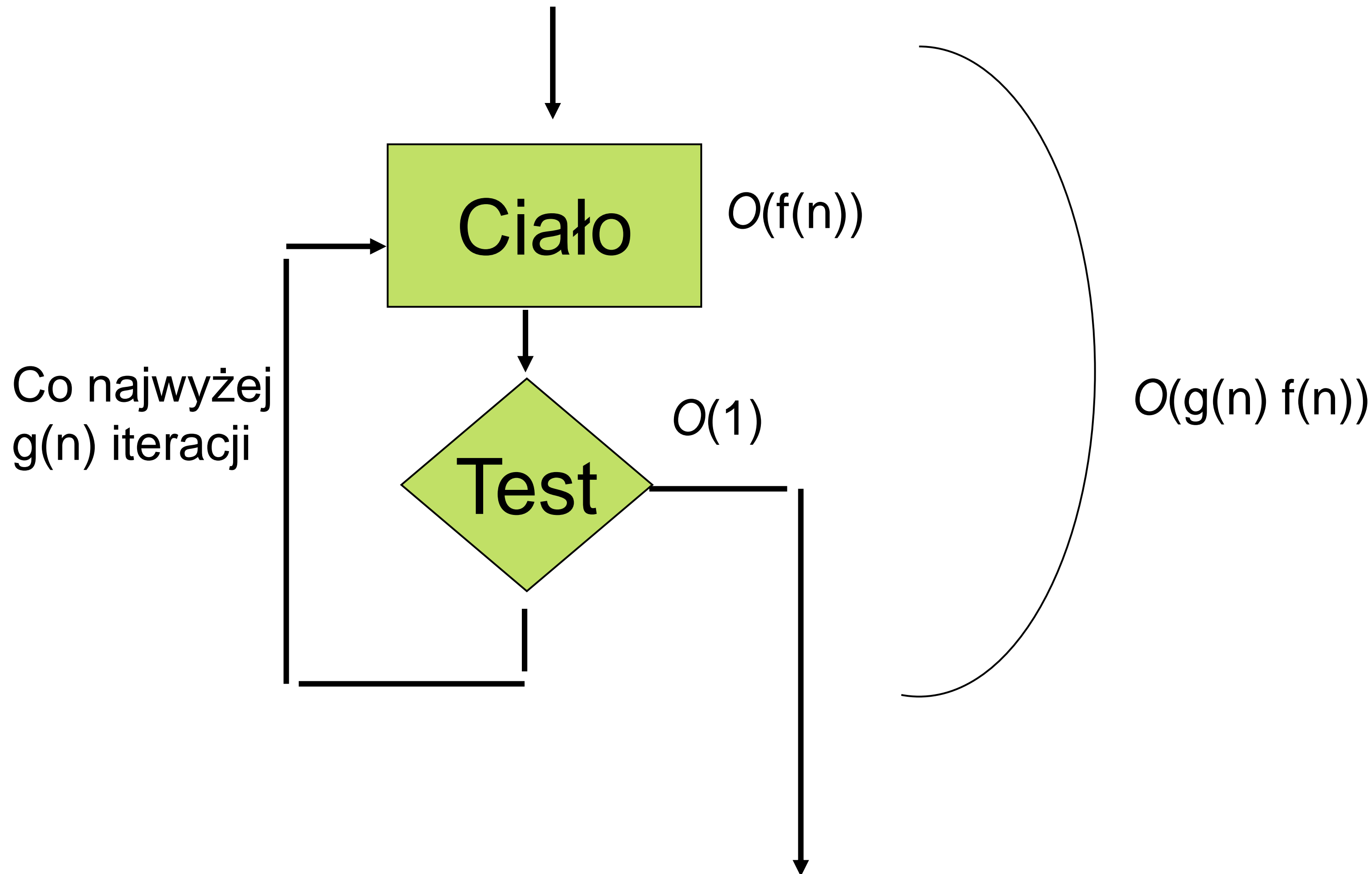
Czas działania instrukcji „if"



Czas działania instrukcji „while”



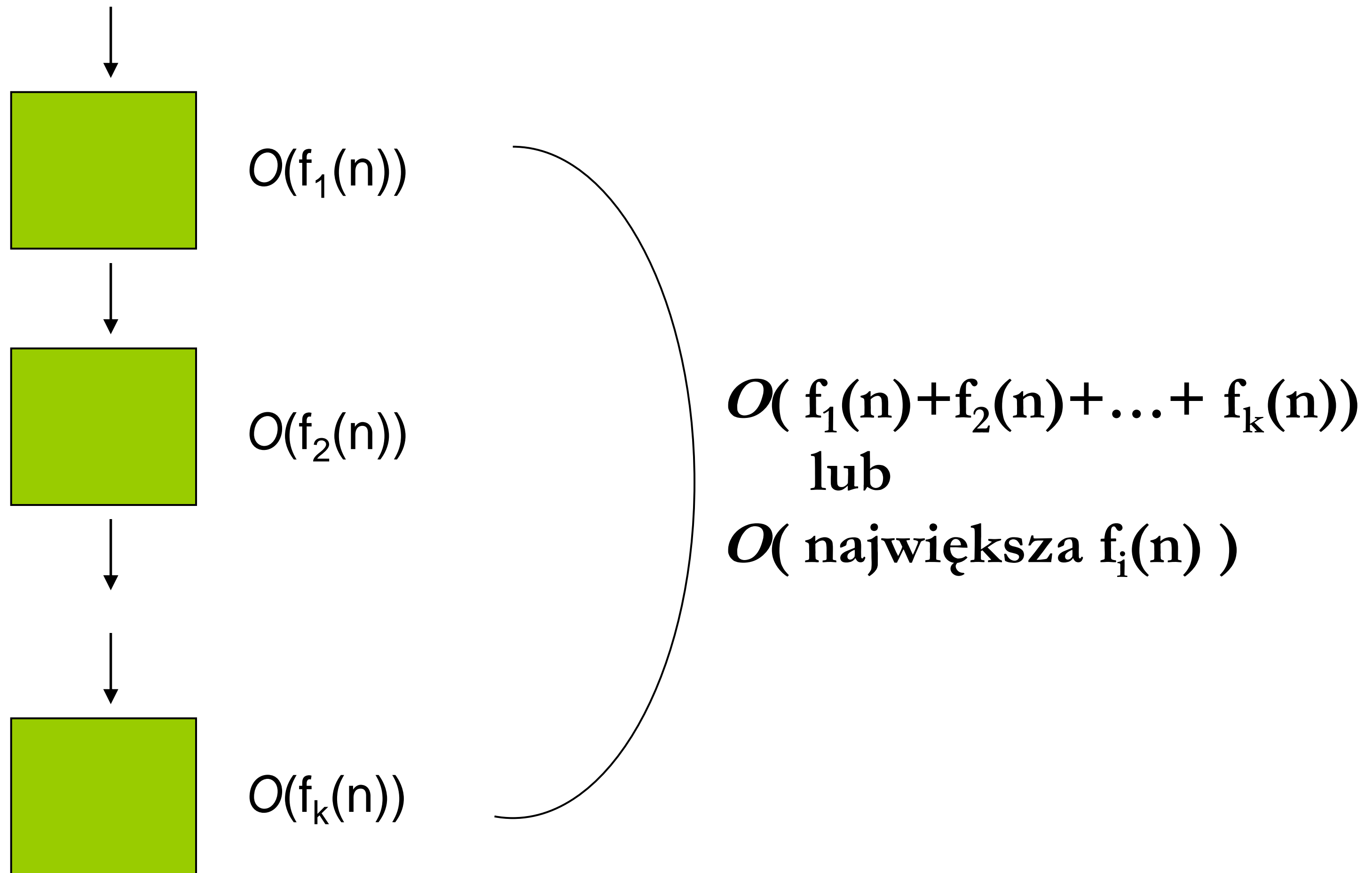
Czas działania instrukcji „do-while"



Czas działania bloków

- Sekwencja instrukcji przypisań, odczytów i zapisów, z których każda wymaga czasu ***O*(1)**, potrzebuje do swojego wykonania łącznego czasu ***O*(1)**.
- Pojawiają się również instrukcje złożone, jak instrukcje warunkowe i pętle.
- Sekwencję prostych i złożonych instrukcji nazywa się blokiem.
- Czas działania bloku obliczmy sumując górne ograniczenia czasów wykonania poszczególnych instrukcji, które należą do tego bloku.

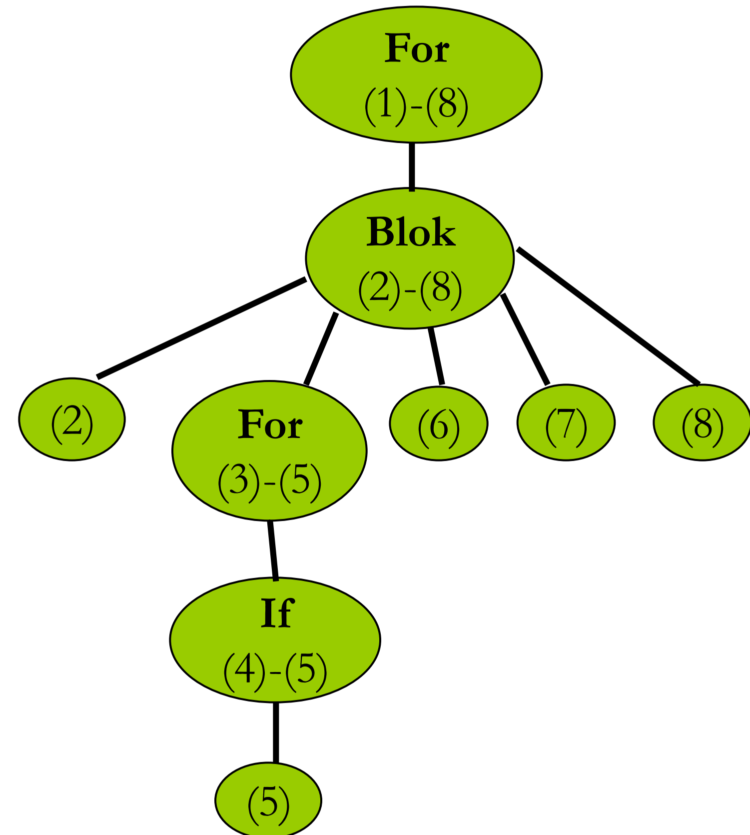
Czas działania bloku instrukcji



Przykład: „sortowanie przez wybieranie"

Drzewo reprezentujące grupowanie instrukcji

```
1.  for (i=0; i< n-1; i++ ) {
2.      small = i;
3.      for (j=i+1; j<n; j++ )
4.              if( A[j] < A[small] )
5.              small =j;
6.      temp = A[small];
7.      A[small] = A[i];
8.      A[i] = temp; }
```

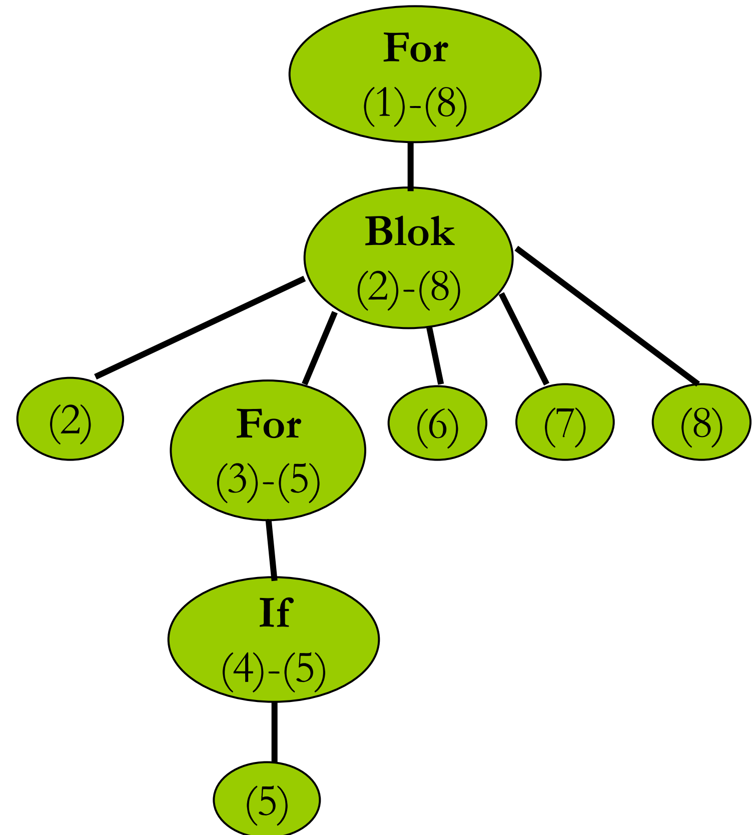


Przykład: „sortowanie przez wybieranie"

- Rozpoczynamy analizę „od liścia do korzenia"
  - Każda instrukcja przypisania (liście) wymaga czasu *O*(1)
  - Instrukcja „if" (4-5) wymaga czasu *O*(1)
  - Instrukcja 'for" (3)-(5) wymaga czasu *O*(n-i-1) oraz i<n
  - Instrukcja „for" (2)-(8) może być dalej ograniczona przez *O*(n-1)
  - Instrukcja „for" (1)-(8) może być ograniczona przez *O*(n (n-1)),

Odrzucając wyraz mniej znaczący otrzymujemy oszacowanie czasu działania jako **$O(n^2)$**.

Drzewo reprezentujące grupowanie instrukcji

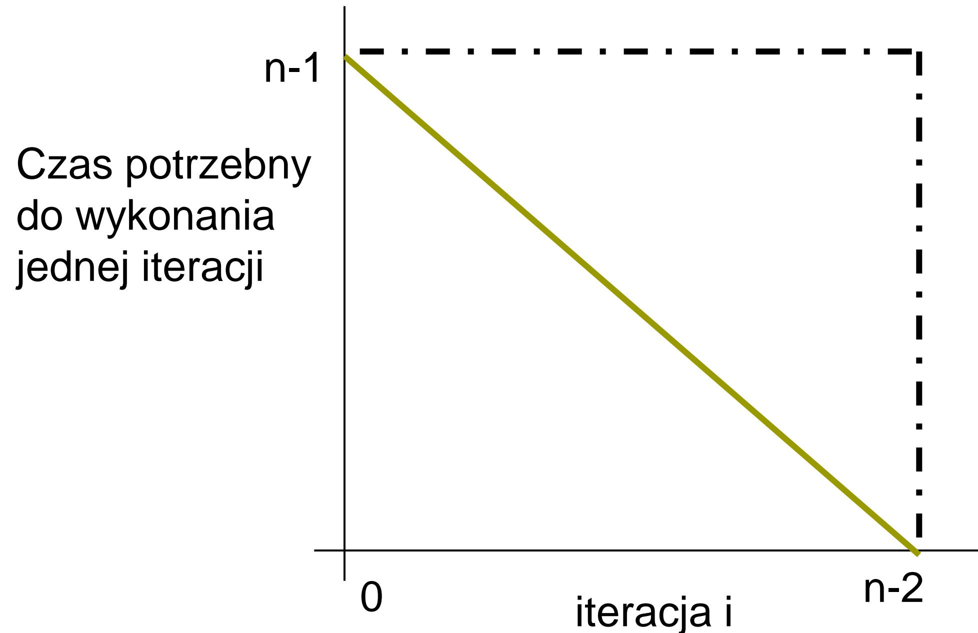


Proste lub precyzyjne ograniczenie

- Dotychczas rozważaliśmy szacowanie czasu działania pętli używając ujednoliconego górnego ograniczenia, mającego zastosowanie w każdej iteracji pętli.
- Dla sortowania przez wybieranie, takie proste ograniczenie prowadziło do szacowania czasu wykonania **O(n^2)**.
- Można jednak dokonać bardziej uważnej analizy pętli i traktować wszystkie jej iteracje osobno. Można wówczas dokonać sumowania górnych ograniczeń poszczególnych iteracji. Czas działania pętli z wartością ***i*** zmiennej indeksowej ***i*** wynosi ***O(n-i-1)*****,** gdzie ***i*** przyjmuje wartości od ***0*** do ***n-2.***
- Górne ograniczenie czasu niezbędnego do wykonania wszystkich iteracji wynosi:

$$O\left(\sum_{n=0}^{n-2} (n-i-1) \right) = O(\, n(n-1)/2 \,)$$

Proste lub precyzyjne ograniczenie



Górne ograniczenie czasu niezbędnego do wykonania wszystkich iteracji wynosi:

$$O\left(\sum_{n=0}^{n-2} (n-i-1) \right) = O(n(n-1)/2)$$

Nie przejmuj się efektywnością algorytmu… wystarczy poczekać kilka lat.

- Taki pogląd funkcjonuje w środowisku programistów, nie określono przecież granicy rozwoju mocy obliczeniowych komputerów. Nie należy się jednak z nim zgadzać w ogólności. Należy zdecydowanie przeciwstawiać się przekonaniu o tym, że ulepszenia sprzętowe uczynią pracę nad efektywnymi algorytmami zbyteczną.
- Istnieją problemy których rozwiązanie za pomocą zasobów komputerowych jest teoretycznie możliwe, ale praktycznie przekracza możliwości istniejących technologii. Przykładem takiego problemu jest rozumienie języka naturalnego, przetwarzanie obrazów (do pewnego stopnia oczywiście) czy "inteligentna" komunikacja pomiędzy komputerami a ludźmi na rozmaitych poziomach.
- Kiedy pewne problemy stają się "proste"… Nowa grupa wyzwań, które na razie można sobie tylko próbować wyobrażać, wytyczy nowe granice możliwości wykorzystania komputerów.

- Do problemu systematycznej analizy czasu działania programu powrócimy jeszcze na wykładzie w styczniu…

Kombinatoryka

Kombinatoryka i prawdopodobieństwo

- Często spotykamy się z problemem obliczenia wartości wyrażającej prawdopodobieństwo zajścia określonych zdarzeń.
- Dziedzina matematyki zajmująca się tą tematyką to **kombinatoryka**.
- Pojęcia związane z próbami szacowania prawdopodobieństwa występowania zdarzeń definiuje **teoria prawdopodobieństwa**.

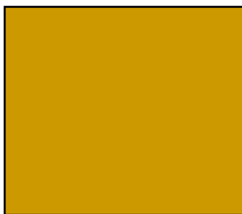
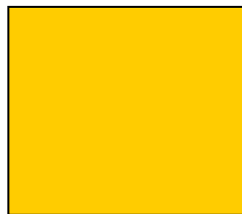
- Zacznijmy od kombinatoryki...

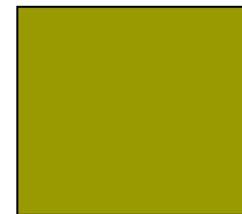
Wariacje z powtórzeniami

- Jednym z najprostszych, ale też najważniejszych problemów jest analiza listy elementów, z których każdemu należy przypisać jedną z wartości należących do stałego zbioru.
- Należy określić możliwą liczbę różnych przyporządkowań (*wariacji z powtórzeniami*) wartości do elementów.
- Przykład:

  4 kwadraty, każdy można pokolorować jednym z 3 kolorów.

  Ile możliwych pokolorowań? $3 \cdot 3 \cdot 3 \cdot 3 = 3^4 = 81$



Wariacje z powtórzeniami

- Mamy listę **n**-elementów. Istnieje zbiór **k**-wartości z których każda może być przyporządkowana do jakiegoś elementu. Przyporządkowanie jest listą **n** wartości (n_1, n_2,...n_n). Gdzie każda z n_1, n_2,...n_n jest jedną z wartości **k**.
- Istnieje k^n różnych przyporządkowań.
- **Twierdzenie:**
  S(n): liczba możliwych sposobów przyporządkowania dowolnej z k wartości do każdego z n elementów wynosi k^n.

Wariacje z powtórzeniami

- **Podstawa:**

  Przypadek podstawowy to **n=1**. Jeżeli mamy **1** element możemy wybrać dla niego dowolną spośród **k** wartości. Istnieje wiec **k** różnych przyporządkowań. Ponieważ $k^1=k$, podstawa indukcji jest prawdziwa.

- **Indukcja:**

  Załóżmy że **S(n)** jest prawdziwe i rozważmy **S(n+1)**,określające że istnieje k^{n+1} możliwych przyporządkowań jednej z **k** wartości do każdego z **n+1** elementów.

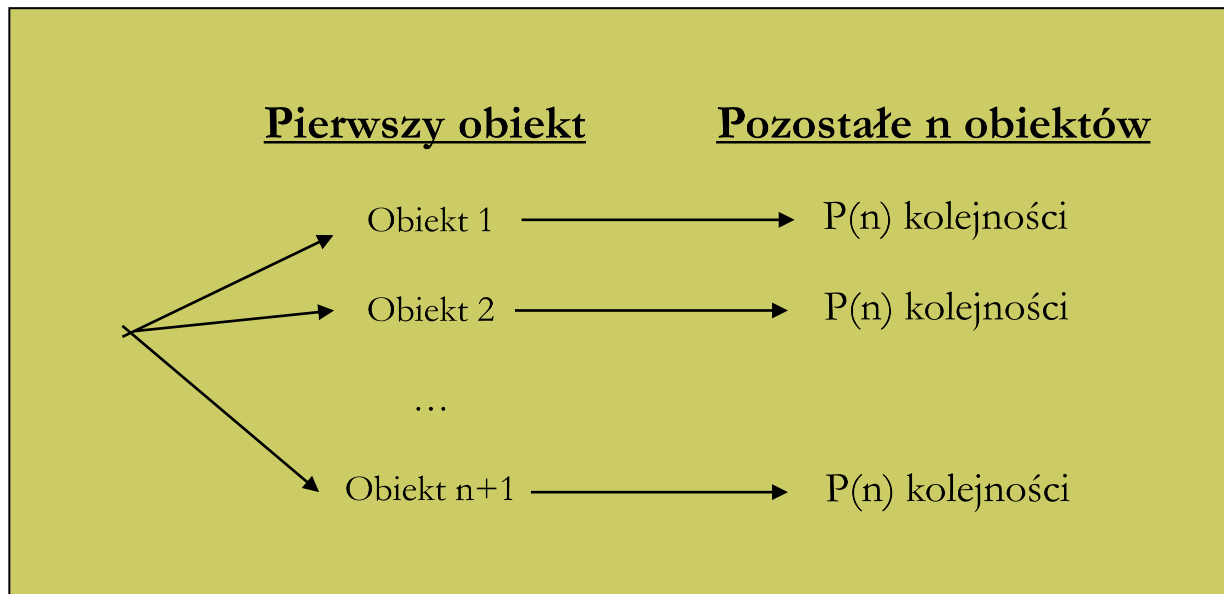
  Wiemy, że istnieje **k** możliwości doboru wartości dla pierwszego elementu. Zgodnie z hipotezą indukcyjną, istnieje k^n przyporządkowań wartości do pozostałych **n** elementów. Łączna liczba przyporządkowań wynosi $k \bullet k^n = k^{n+1}$. Cnd.

Permutacje

- Mając **n** różnych obiektów, na ile różnych sposobów można je ustawić w jednej lini?
- Takie uporządkowanie nazywamy permutacją.
- Liczbę permutacji **n** obiektów zapisujemy jako **P(n)**.

Jak obliczyć P(n+1)?

- Problem: mamy **n+1** obiektów ($\mathbf{a_1, a_2, \ldots, a_n, a_{n+1}}$) które mają zostać ustawione.
- Ilość możliwych wyników ustawień jest **P(n+1)**



Permutacje n+1 obiektów

Jak obliczyć P(n+1)?

- **Twierdzenie**:
  $P(n) = n!$ dla wszystkich **$n \geq 1$**

- **Podstawa**:
  Dla **$n=1$**, **$P(1)=1$** określa że istnieje jedna permutacja dla jednego obiektu.

- **Indukcja**:
  Załóżmy ze **$P(n) = n!$**
  Wówczas wg. naszego twierdzenia: **$P(n+1)=(n+1)!$**
  Rozpoczynamy od stwierdzenia że **$P(n+1)=(n+1) \cdot P(n)$**
  Zgodnie z hipotezą indukcyjną **$P(n)=n!$**, zatem **$P(n+1)=(n+1) \cdot n!$**
  Zatem **$P(n+1)=(n+1) \cdot n! =(n+1) \cdot n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1 = (n+1)!$**, czyli nasze twierdzenie jest poprawne. Cnd.

- Jednym z interesujących zastosowań wzoru na liczbę permutacji jest dowód na to że **algorytmy sortujące musza działać w czasie co najmniej proporcjonalnym do (n log n), dla n elementów do posortowania**, chyba że wykorzystują jakieś specjalne własności sortowanych elementów.

Wariacje bez powtórzeń

- Niekiedy chcemy wybrać tylko niektóre spośród elementów zbioru i nadać im określony porządek.
- Uogólniamy opisaną poprzednio funkcję P(n) reprezentującą liczbę permutacji, aby otrzymać **dwuargumentową funkcje P(n,m),** którą definiujemy jako ilość możliwych sposobów wybrania **m elementów z n-elementowego zbioru**, przy czym **istotną** rolę odgrywa **kolejność wybierania elementów**, natomiast nieważne jest uporządkowanie elementów nie wybranych.
- Zatem **P(n) = P(n,n).**

Wariacje bez powtórzeń

Przykład:

- Ile istnieje sposobów utworzenia sekwencji **m** liter ze zbioru **n** liter, jeżeli żadna litera nie może występować więcej niż raz?
- Na sam początek możemy zauważyć warunek, by zadanie miało sens: **$n \geq m$**
- Pierwszą literę możemy wybrać na n sposobów (wybieramy ze zbioru **n**-elementowego), drugą na **n-1** sposobów (gdyż nie możemy wybrać tej samej litery co poprzednio), trzecią na **n-2** sposoby…
- Ostatnią na **n-(m-1)** sposobów.

Twierdzenie: $P(n,m) = n \bullet (n-1) \bullet \ldots \bullet n-(m-1)$ dla wszystkich $m \leq n$

Twierdzenie: $P(n,m) = \frac{n!}{(n-m)!}$ dla wszystkich $m \leq n$

Kombinacje

- Kombinacja to każdy podzbiór zbioru skończonego. Kombinacją ***m***-elementową zbioru ***n***-elementowego **A** nazywa się każdy ***m***-elementowy podzbiór zbioru **A** ($0 \le m \le n$). Używa się też terminu "kombinacja z ***n*** elementów po ***m*** elementów" lub wręcz "kombinacja z ***n* po *m***".
- Taką funkcję zapisujemy jako: $\binom{n}{m}$

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

Wyznaczanie liczby kombinacji

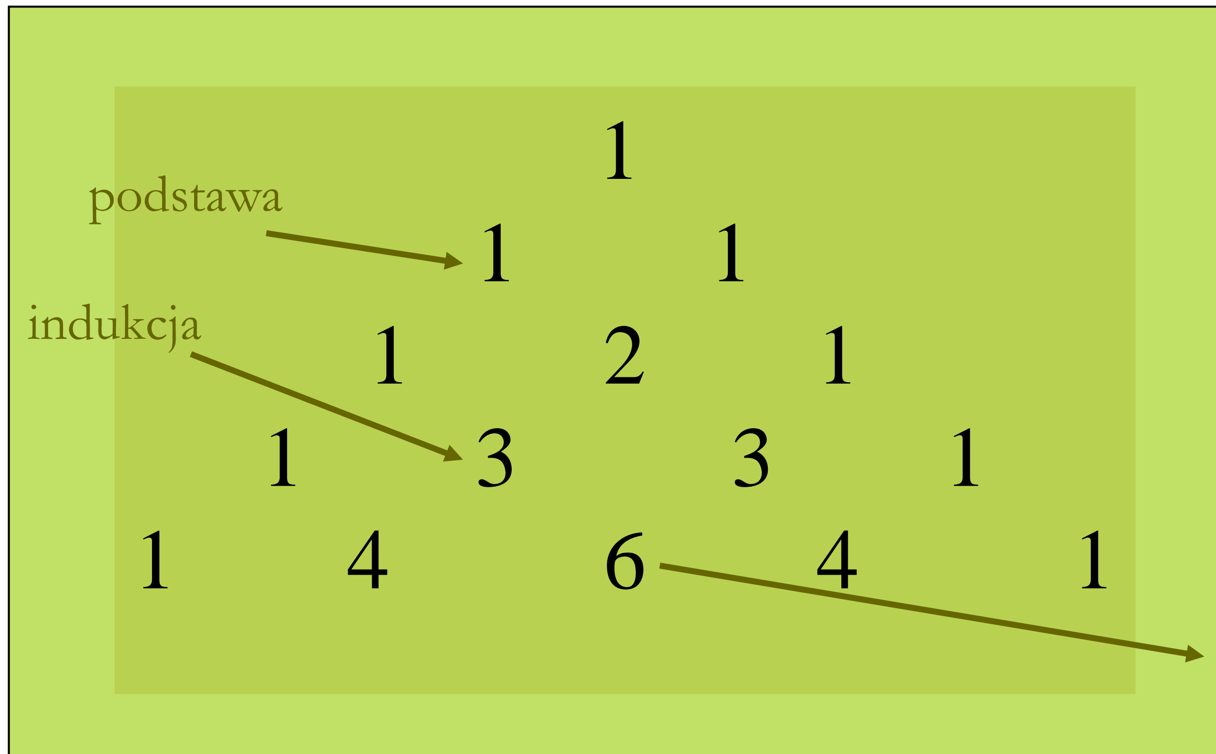
- **Rekurencyjny algorytm: (ilustruje tzw. trójkąt Pascala)**

- **Podstawa**: $\binom{n}{0}=1$ dla dowolnego n≥1.
  Oznacza to że istnieje tylko jeden sposób wybrania **zero** elementów ze zbioru **n**-elementowego – wybranie niczego.
  Także $\binom{n}{n}=1$, ponieważ jedynym sposobem wybrania **n**-elementów ze zbioru **n**-elementowego jest wybranie ich wszystkich.

- **Indukcja**: Jeśli O< m< n, to $\binom{n}{m}=\binom{n-1}{m}+\binom{n-1}{m-1}$.
  Oznacza to, że jeżeli chcemy wybrać **m** elementów ze zbioru **n**-elementowego, możemy albo:
  - nie wybrać pierwszego elementu, po czym wybrać m elementów z pozostałych n-1 elementów. Taką liczbę możliwości wyraża $\binom{n-1}{m}$.
  - wybrać pierwszy element, po czym wybrać m-1 elementów z pozostałych n-1 elementów. Taka liczbę możliwości wyraża $\binom{n-1}{m-1}$.

Trójkąt Pascala

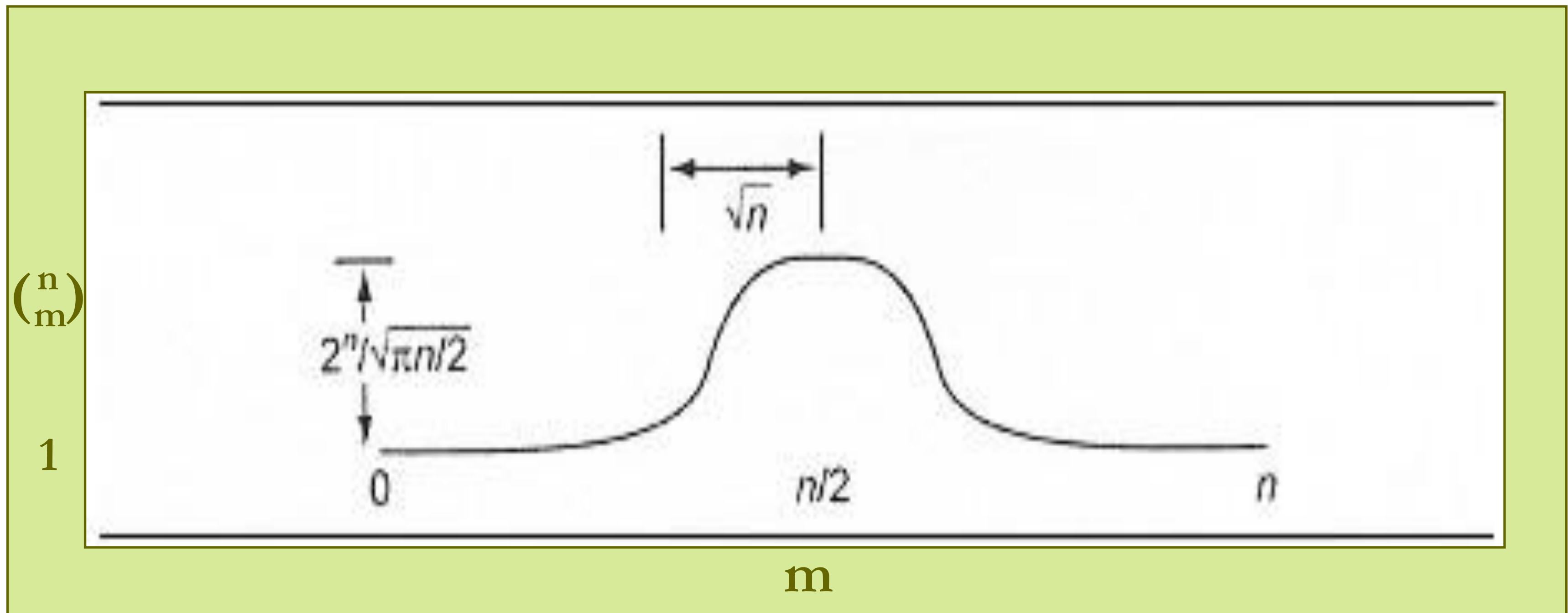
- Rekurencję przy obliczaniu liczby kombinacji często ilustruje się przy pomocy **trójkąta Pascala**.
- $\binom{n}{m}$ = **(m+1)** liczba w **(n+1)** wierszu



$\binom{4}{2}$ = 4!/(2! x 2!) = 6

Interesujące własności funkcji $\binom{n}{m}$

- To również są współczynniki rozkładu dwuwyrazowego wielomianu (dwumianu) $(x+y)^n$
- $\sum_{m=0}^{n} \binom{n}{m} = 2^n$
- Wykres funkcji $\binom{n}{m}$ dla stałej dużej wartości **n**:



Permutacje z powtórzeniami

- **Twierdzenie**:
  Jeżeli istnieje **n** elementów podzielonych na **k** grup o rozmiarach równych odpowiednio $i_1, i_2, i_3, \ldots, i_k$ gdzie elementy jednej grupy są identyczne, ale elementy różnych grup różnią się od siebie, liczba uporządkowań tych elementów wynosi
  $S(k) = n!/ \Pi_{j=1}^{k} i_j!$

- **Podstawa**:
  Dla **k=1**, istnieje tylko jedna grupa zawierająca identyczne elementy, które możemy uporządkować tylko w jeden sposób, niezależnie od liczności tego zbioru. Jeśli **k=1**, to $i_1=n$, zatem $S(1)=n!/n!=1$ jest prawdziwe.

- **Indukcja**:
  Załóżmy że **S(k)** jest prawdziwe i rozważmy sytuacje, w której mamy **k+1** grup. Niech ostatnia grupa składa się z $m=i_{k+1}$ elementów, występujących na **m** pozycjach,
  z których możemy je wybierać na $\binom{n}{m}$ sposobów.
  Stosując hipotezę indukcyjną otrzymujemy że $S(k+1)= \binom{n}{m} \cdot (n-m)!/ \Pi_{j=1}^{k} i_j!$ co łatwo można przekształcić (pamiętając że $m=i_{k+1}$) do postaci:
  $S(k+1)= n! / \Pi_{j=1}^{k+1} i_j!$ a więc cnd.

- Jest to typowy problem przy układania anagramów

Łączenie reguł kombinatorycznych

- Typowy problem kombinatoryczny wymaga łączenia przedstawionych reguł (cegiełek) w bardziej skomplikowane struktury.
- Techniki których używamy to:
  - prowadzenie obliczeń jako sekwencji wyborów;
  - prowadzenie obliczeń jako różnicy innych obliczeń (np. wszystkich wyborów – nieprawidłowych wyborów);
  - prowadzenie obliczeń jako sumy rozwiązań dla podprzypadków które są wzajemnie rozłączne.