Teoretyczne podstawy informatyki

Repetytorium:
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_0 takie że:

$f(n) < c \bullet g(n)$ dla wszystkich $n \geq n_0$.

Przykład:

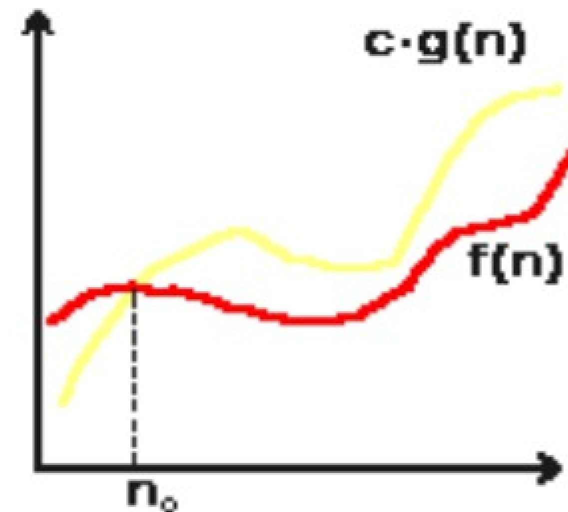
$f(n) = n^2 + 100n + \log_{10} n + 1000$

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 $\mathbf{O(\log_2 \log_2 n)}$ czy $\mathbf{O(1)}$ 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_0 takie że, f(n) $\geq$ c g(n) dla wszystkich n $\geq n_0$.

Równoważność

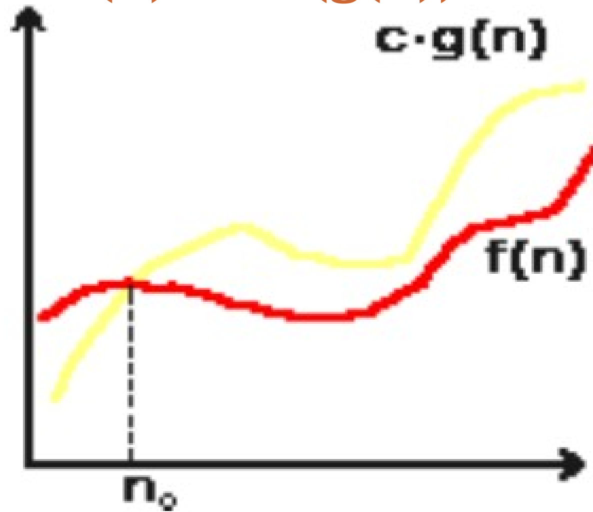
f(n) jest Ω(g(n)) wtedy i tylko wtedy, gdy g(n) jest *O*(f(n))

Definicja

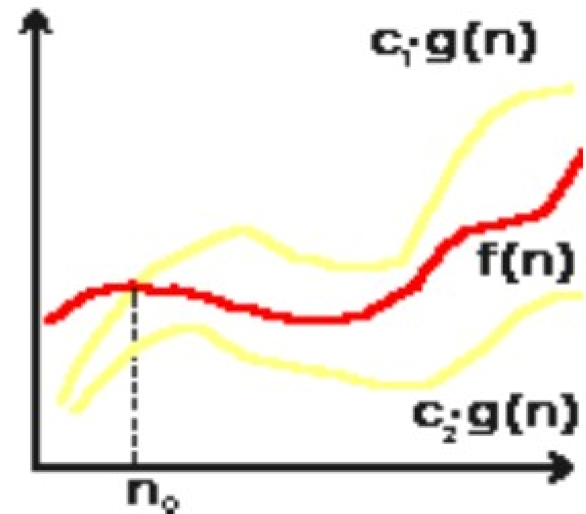
f(n) jest Θ(g(n)), jeśli istnieją takie liczby dodatnie c_1, c_2 i n_0 takie że, c_1g(n) $\leq$ f(n) $\leq$ c_2g(n) dla wszystkich n $\geq n_0$.

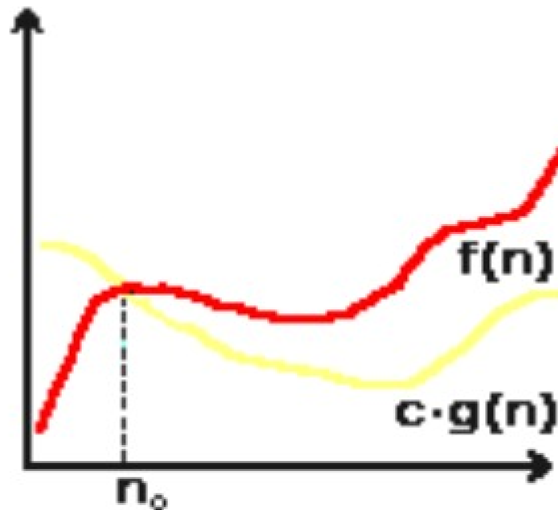
Notacja O, Ω i Θ

$f(n) = O\ (g(n))$



$f(n) = \Theta\ (g(n))$





$f(n) = \Omega\ (g(n))$

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 zastanawiając się nad doborem struktury danych.

Najcześciej spotykane złożoności

log(n)- złożoność logarytmiczna

n - złożoność liniowa

nlog(n) - złożoność liniowo-logarytmiczna

n^2 - złożoność kwadratowa

n^k - złożoność wielomianowa

2^n - złożoność wykładnicza

n! - złożoność wykładnicza, ponieważ $n! > 2^n$ już od n=4

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.

Funkcje niewspółmierne

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.

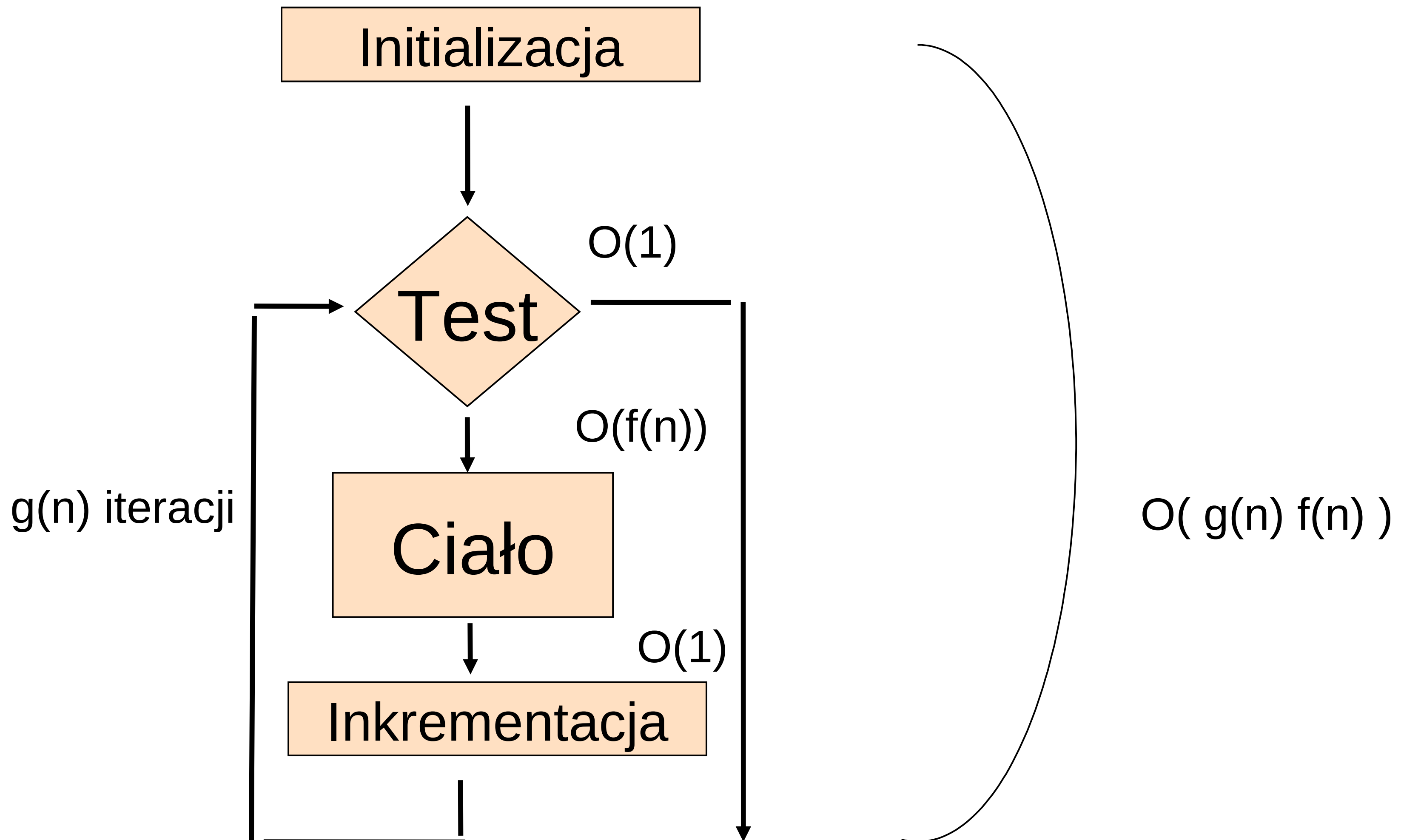
Czas działania pętli „for”

- **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: $\mathbf{1+3n+2(1+2+\ldots+n-1) = 1+3n+n(n-1) = O(n)+O(n^2) = O(n^2)}$ 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 podtablicę zawierającą liczby uporządkowane rosnąco.

  for (i=0; len=1; i<n-1; i++) {

  for (i_1=i_2=k=i; k<n-1 && a[k]<a[k+1]; k++,i_2++);

  if(len < i_2-i_1+1) len=i_2-i_1+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 **1-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 $i \in \{1,\ldots,n-1\}$.
  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”**,
  - w **„przypadku pesymistycznym”**
  - oraz w **„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:

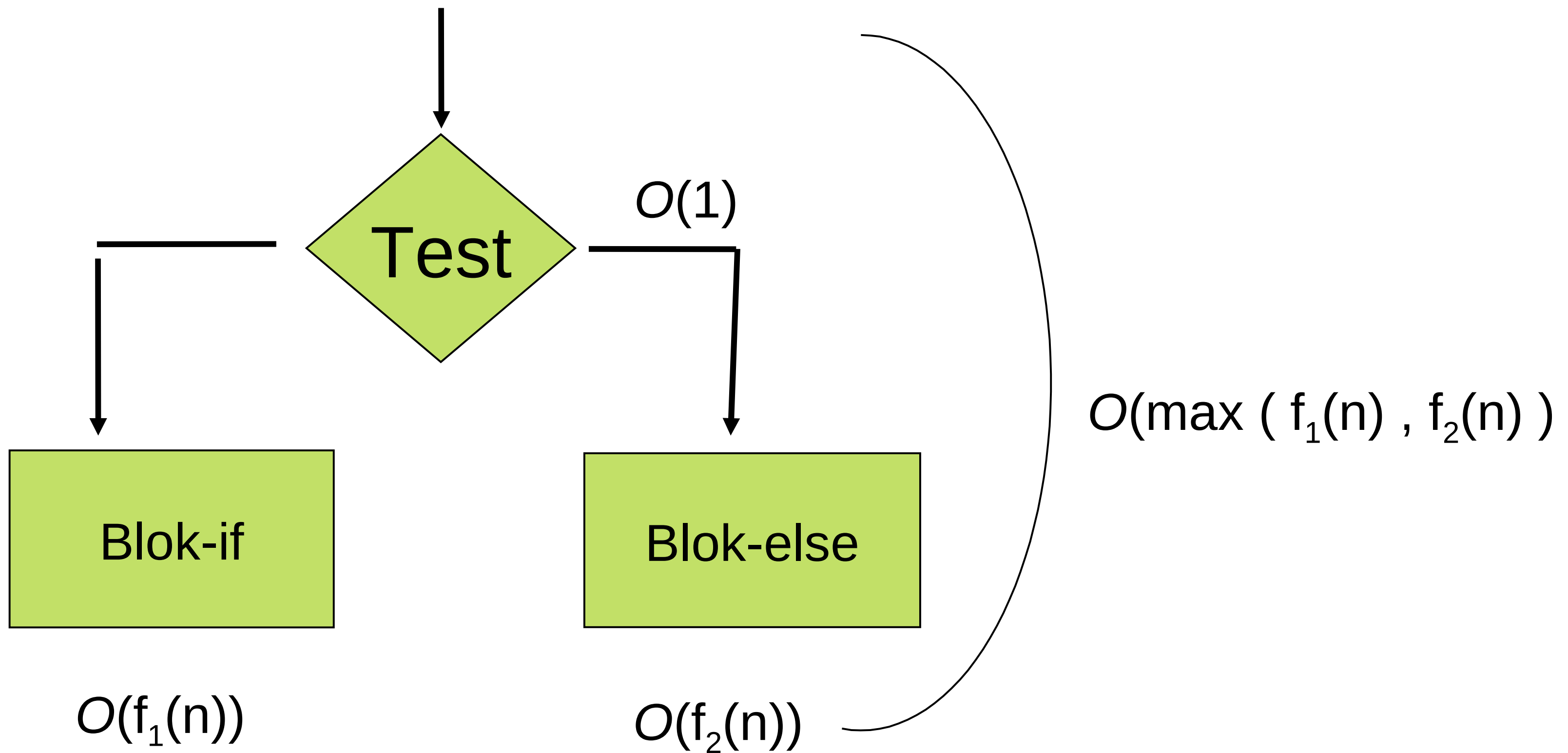
```
if (<warunek>)
    <blok-if>
else
    <blok-else>
```

Gdzie

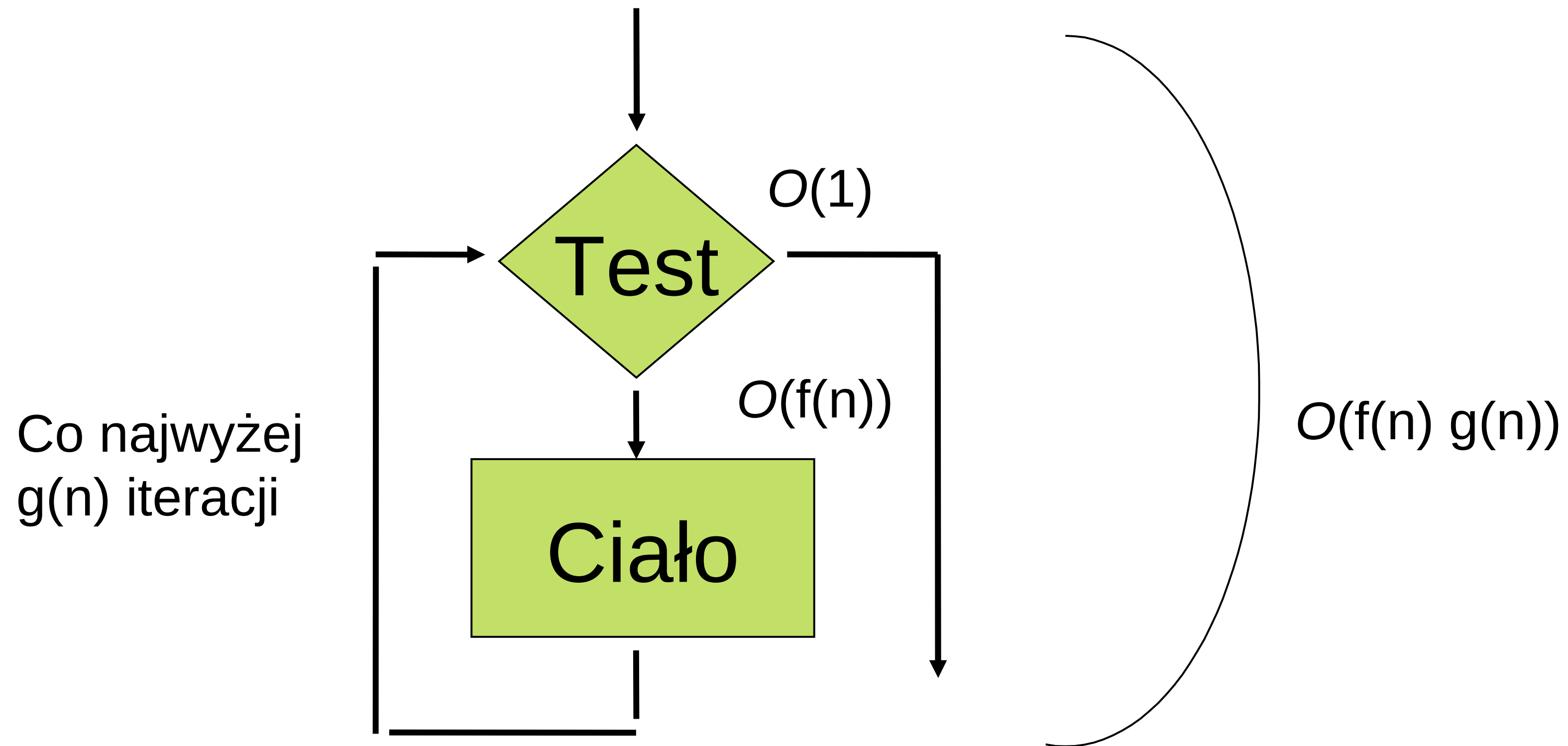
- **<warunek>** jest wyrażeniem które trzeba obliczyć. Warunek niezależnie od tego jak skomplikowany wymaga wykonania stałej liczby operacji (więc czasu ***O*(1))** chyba ze zawiera wywołanie funkcji, .
- **<blok-if>** zawiera instrukcje wykonywane tylko w przypadku gdy warunek jest prawdziwy, czas działania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

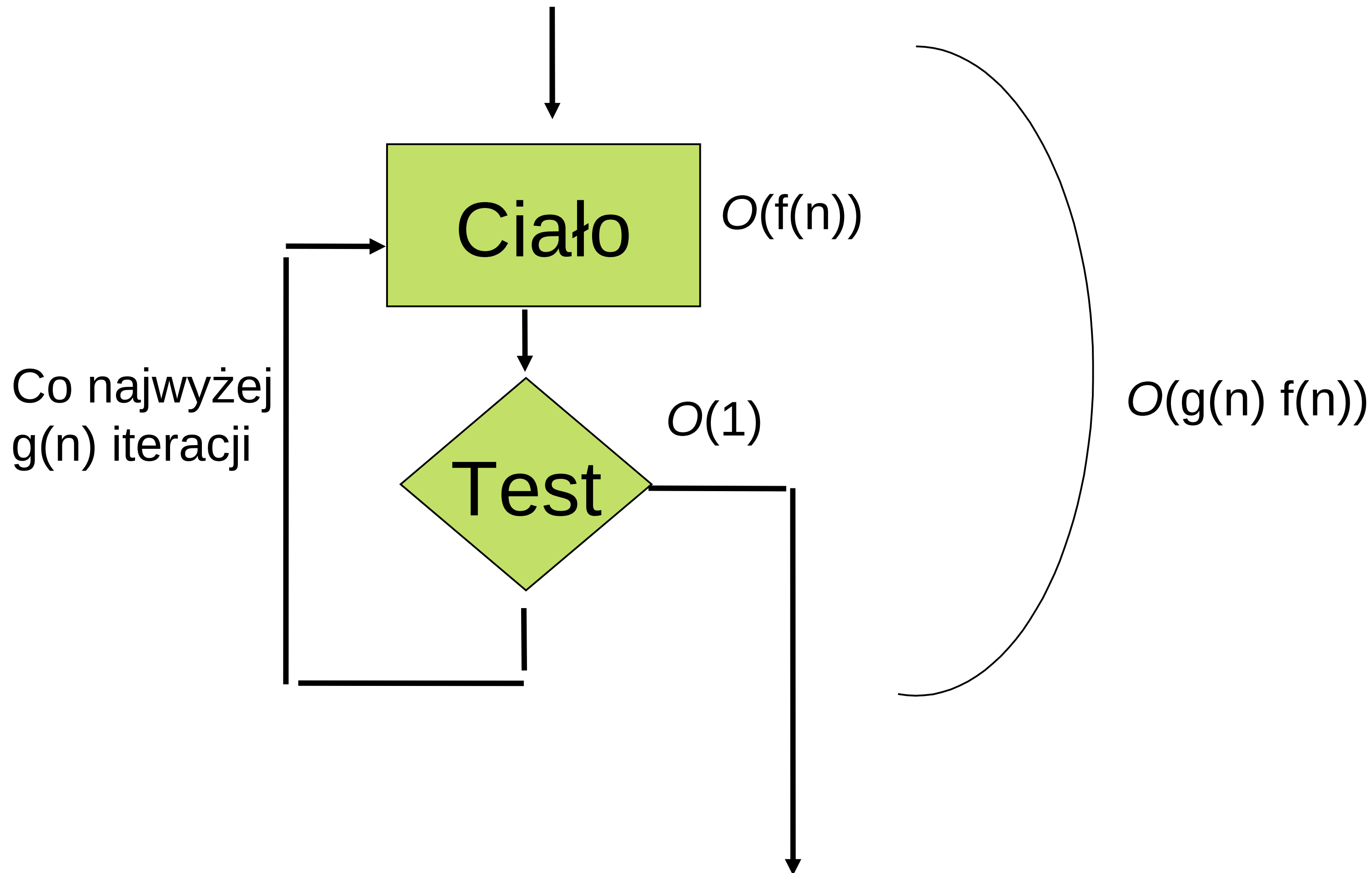
Czas działania instrukcji „if”



Czas działania instrukcji „while”



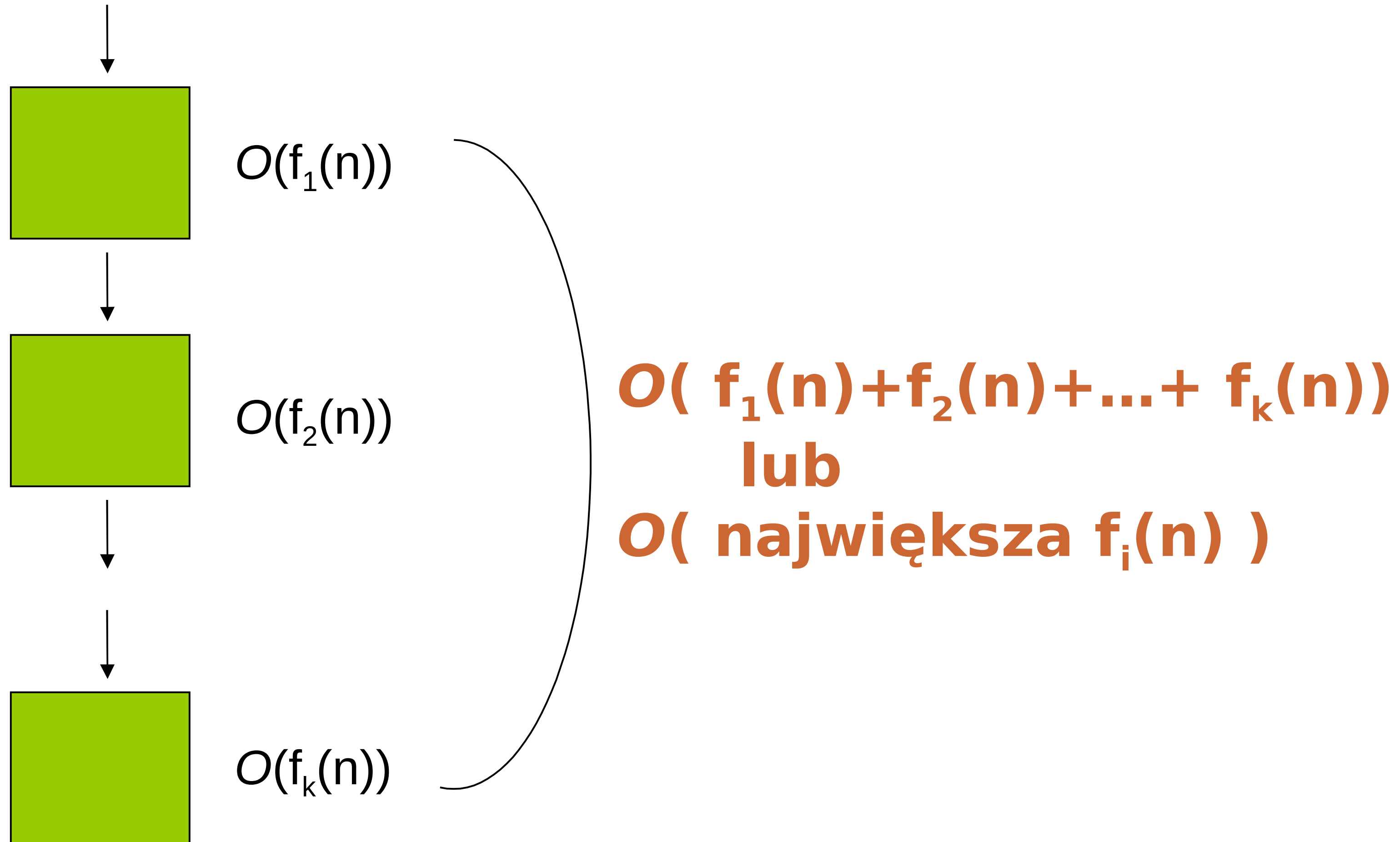
Czas działania instrukcji „do-while”



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 obliczymy sumując górne ograniczenia czasów wykonania poszczególnych instrukcji, które należą do tego bloku.

Czas działania bloku instrukcji

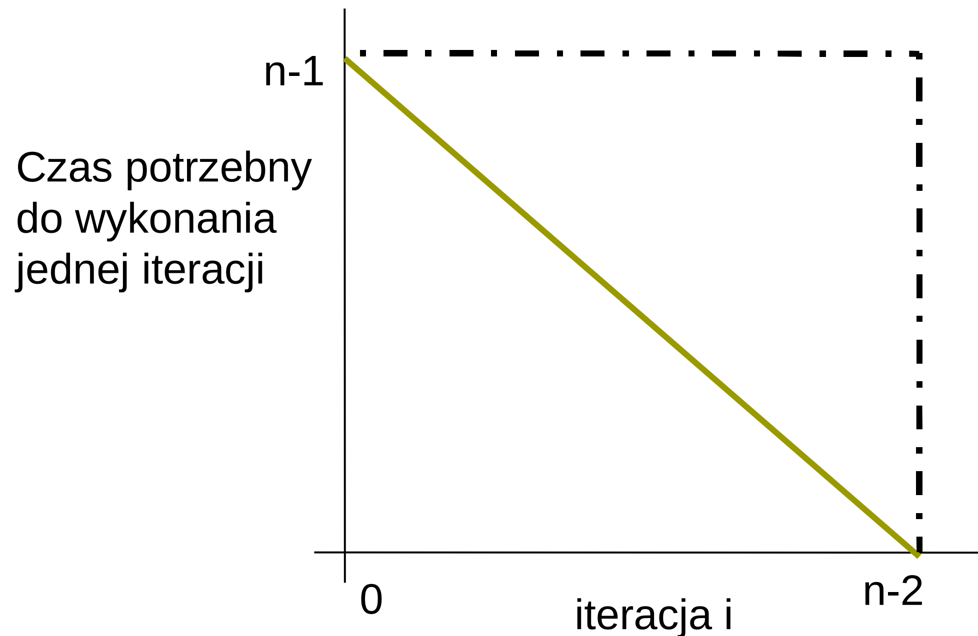


Proste lub precyzyjne ograniczenie

- 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:

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

Proste lub precyzyjne ograniczenie



Górne ograniczenie czasu niezbędnego do wykonania wszystkich iteracji wynosi:

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

Efektywność algorytmu

- **Czas działania:**
  - Oznaczamy przez funkcje **T(n)** liczbę jednostek czasu, które zajmuje wykonanie programu lub algorytmu w przypadku problemu o rozmiarze **n**.
  - Funkcje te nazywamy **czasem działania**. Dość często czas działania zależy od konkretnych danych wejściowych, nie tylko ich rozmiaru. W takim przypadku, funkcje **T(n)** definiuje się jako **najmniej korzystny przypadek** z punktu widzenia kosztów czasowych. Inną wyznaczaną wielkością jest też **czas średni**, czyli średni dla różnych danych wejściowych.

Rekurencja

Rekurencje były badane już w 1202 roku przez L. Fibonacciego, od którego nazwiska pochodzi nazwa liczb Fibonacciego.

A. De Moivre w 1730 roku wprowadził pojęcie funkcji tworzących do rozwiązywania rekurencji.

Czas działania programu

- Dla konkretnych danych wejściowych jest wyrażony liczba wykonanych prostych (elementarnych) operacji lub "kroków". Jest dogodne zrobienie założenia 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 wyrażającą czas dla problemu rozmiaru **n** za pomocą czasu dla podproblemów mniejszych rozmiarów.
- Możemy wiec użyć narzędzi matematycznych aby **„rozwiązać rekurencje"** i w ten sposób otrzymać oszacowania czasu działania algorytmu.

Rekurencja dla alg. typu “dziel i zwyciężaj”

- 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 ≤ c** dla pewnej stałej **c**, to przyjmujemy że jego rozwiązanie zajmuje stały czas, co zapiszemy jako **Θ(1)**.
  - Załóżmy że dzielimy problem na **a** podproblemów, każdy rozmiaru **n/b**. Jeśli **D(n)** jest czasem dzielenia problemu na podproblemy, a **C(n)** jest czasem scalania rozwiązań podproblemów w pełne rozwiązanie dla oryginalnego problemu, to otrzymujemy rekurencje

T(n) = Θ(1) jeśli n ≤ c

T(n) = a T(n/b) + D(n) + C(n) w przeciwnym przypadku

Rekurencja dla alg. “dziel i zwyciężaj”

- **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 wiec **$C(n)=\Theta(n)$**.
- **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 **$T(n) = \Theta(n \log n)$**.

Jak to sprawdzić?

Metody rozwiązywania rekurencji

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. Następnie korzystamy z gotowego rozwiazania.

Metoda podstawiania

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

Metoda podstawiania

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:

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

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

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

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

spełnione dla **c>=1**;

Metoda iteracyjna

- 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

Przykład:

- Postać rekurencji:
  $T(n) = 3T(n/4) + n$
- Iterujemy:
  $T(n) = n + 3T(n/4) = n + 3((n/4) +3T(n/16))$
  $= n + 3 (n/4) + 9T(n/16)$
  $= n + 3 n/4 + 9 n/16 + 27 T(n/64)$
- 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 **$i > \log_4(n)$**).
  $T(n) \leq n + 3n/4 + 9n/16 + 27n/64 + ... + 3^{\log_4 n} \Theta(1)$
  $T(n) \leq 4n + 3^{\log_4 n} \Theta(1) = \Theta(n)$

Metoda iteracyjna

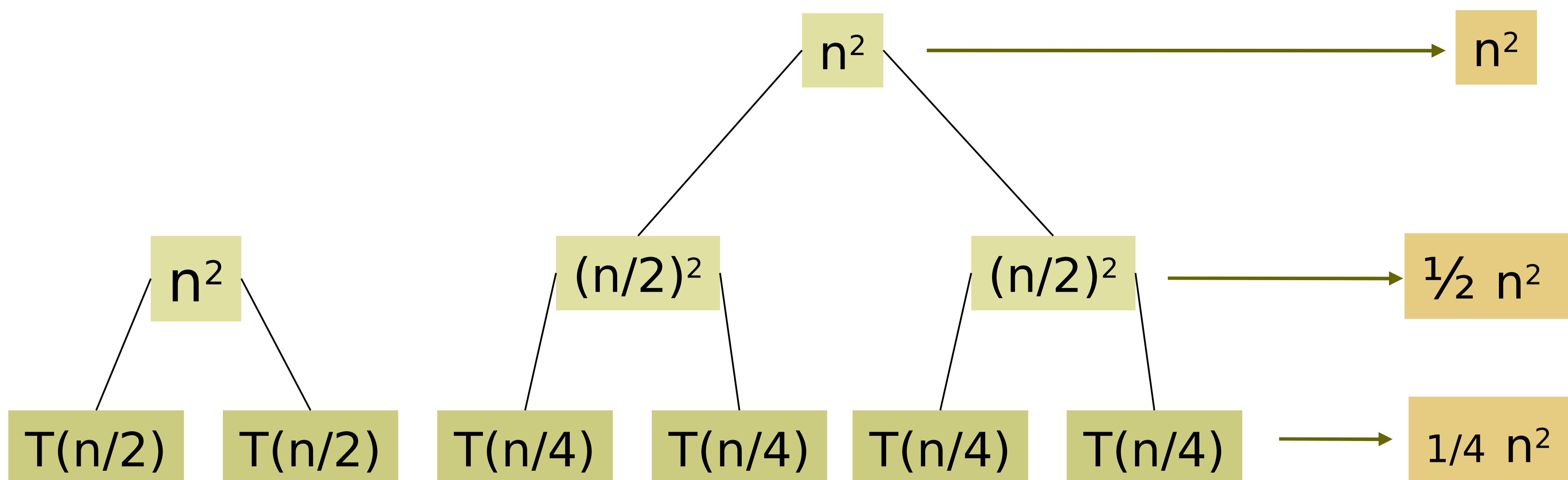
- Metoda iteracyjna jest zazwyczaj związana z dużą ilością przekształceń algebraicznych, więc zachowanie prostoty nie jest łatwe.
- Punkt kluczowy to skoncentrowanie się na dwóch parametrach:
  - liczbie iteracji koniecznych do osiągnięcia warunku brzegowego
  - oraz sumie składników pojawiających się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

Drzewa rekursji

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

Drzewo rekursji dla alg. „dziel i zwyciężaj

$T(n) = 2\, T(n/2) + n^2$

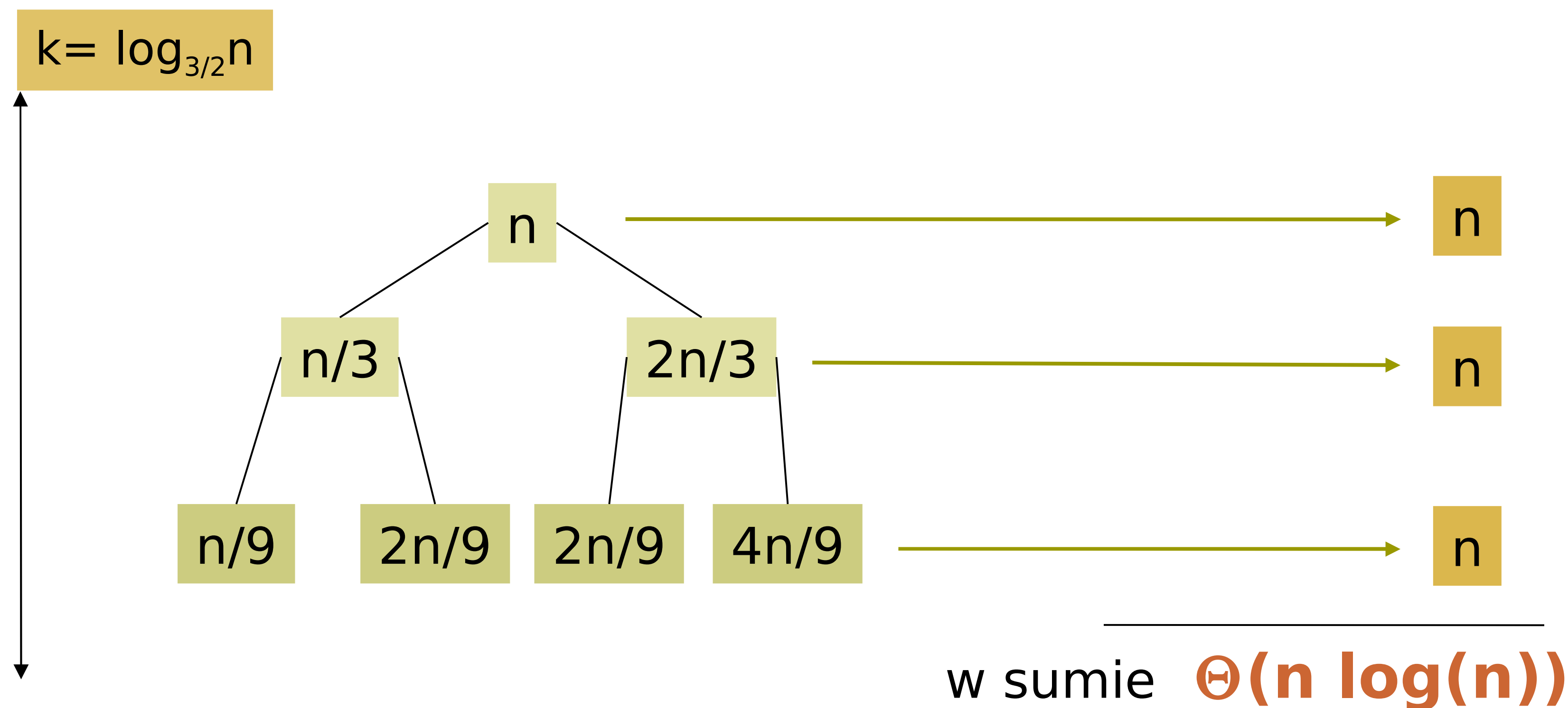


ostateczny wynik: $T(n) = \Theta(n^2)$

w sumie: $\Theta(n^2)$

Drzewa rekursji

$T(n) = T(n/3) + T(2n/3) + n$



Dzielimy tak długo aż $n (2/3)^k = 1 \Rightarrow n = (3/2)^k$

$\Rightarrow k = \log_{3/2}(n) = \log_{3/2}(2) * \log(n)$

ostateczny wynik: $T(n) = \Theta(n \log(n))$

Metoda rekurencji uniwersalnej

- 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 uniwersalnej

- 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 funkcja **f(n)**.

Twierdzenie o rekurencji uniwersalnej

- 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 **$f(n) = O(n^{\log_b a-\varepsilon})$** dla pewnej stałej **$\varepsilon>0$**, 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 **$f(n) = n^{\log_b a+\varepsilon}$** dla pewnej stałej **$\varepsilon>0$** 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

"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

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

$a=9$, $b=3$,

$f(n)=n$,

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

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

Przykład

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

$a=1, b=3/2,$

$f(n)=1,$

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

Stosujemy przypadek 2, gdyż

$f(n) = \Theta(n^{\log_b a}) = \Theta(1)$,

a zatem rozwiązaniem rekurencji jest

$T(n) = \Theta(\log n)$.

Przykład

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

$a=3$, $b=4$,
$f(n)=n \log n$,
a zatem $n^{\log_b a} = n^{\log_4 3} = O(n^{0,793})$.

Ponieważ $f(n) = \Omega(n^{\log_4 3+\varepsilon})$, gdzie $\varepsilon \approx 0.2$, wiec stosuje się tutaj przypadek 3, jeśli możemy pokazać że dla $f(n)$ zachodzi warunek regularności.
Dla dostatecznie dużych n:

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

dla $c=3/4$.

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

Przykład

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

$a=2$, $b=2$,
$f(n)=n \log n$,

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^{ε}** dla każdej dodatniej stałej **ε**.

W konsekwencji rekurencja ta “wpada” w lukę miedzy przypadkiem 2 i 3.

Złożoność zamortyzowana

- W wielu sytuacjach na strukturach danych działają nie pojedyncze operacje ale ich sekwencje.
- Jedna z operacji takiej sekwencji może wpływać na dane w sposób powodujący modyfikacje czasu wykonania innej operacji.

  Jednym ze sposobów określania czasu wykonania w przypadku pesymistycznym dla całej sekwencji jest dodanie składników odpowiadających wykonywaniu poszczególnych operacji.
- Jednak wynik tak uzyskany może być zbyt duży w stosunku do rzeczywistego czasu wykonania. Analiza amortyzacji pozwala znaleźć bliższą rzeczywistej złożoność średnią.

Złożoność zamortyzowana

- Analiza z amortyzacją polega na analizowaniu kosztów operacji, zaś pojedyncze operacje są analizowane właśnie jako elementy tego ciągu.
  Koszt wykonania operacji w sekwencji może być różny niż w przypadku pojedynczej operacji, ale ważna jest też częstość wykonywania operacji.
- Jeśli dana jest sekwencja operacji op_1, op_2, op_3, … to analiza złożoności pesymistycznej daje daje złożoność obliczeniowa równa:

  $$C(op_1, op_2, op_3, \dots) = C_{pes}(op_1) + C_{pes}(op_2) + C_{pes}(op_3) + \dots$$

  dla złożoności średniej uzyskujemy

  $$C(op_1, op_2, op_3, \dots) = C_{śre}(op_1) + C_{śre}(op_2) + C_{śre}(op_3) + \dots$$

- Nie jest analizowana kolejność operacji, “sekwencja” to po prostu “zbiór” operacji.

Złożoność zamortyzowana

- Przy analizie z amortyzacją zmienia się sposób patrzenia, gdyż sprawdza się co się stało w danym momencie sekwencji i dopiero potem wyznacza się złożoność następnej operacji:

  $$\mathbf{C(op_1, op_2, op_3, \dots) = C(op_1) + C(op_2) + C(op_3) + \dots}$$

  gdzie **C** może być złożonością optymistyczna, średnią, pesymistyczna lub jeszcze inną - w zależności od tego co działo się wcześniej.
- Znajdowanie złożoności zamortyzowanej tą metoda może być zanadto skomplikowane.
  Znajomość natury poszczególnych procesów oraz możliwych zmian struktur danych używane są do określenia funkcji **C**, którą można zastosować do każdej operacji w sekwencji. Funkcja jest tak wybierana aby szybkie operacje były traktowane jak wolniejsze niż w rzeczywistości, zaś wolne jako szybsze.
- Sztuka robienia analizy amortyzacji polega na znalezieniu funkcji **C**; takiej która dociąży tanie operacje dostatecznie aby pokryć niedoszacowanie operacji kosztownych.

Przykład

- **Przykład:**
  - dodawanie elementu do wektora zaimplementowanego jako elastyczna tablica
- **Przypadek optymistyczny:** wielkość wektora jest mniejsza od jego pojemności, dodanie elementu ogranicza się do wstawienia go do pierwszej wolnej komórki.
  Koszt dodania nowego elementu to ***O*(1)**.
- **Przypadek pesymistyczny:** rozmiar jest równy pojemności, nie ma miejsca na nowe elementy. Konieczne jest zaalokowanie nowego obszaru pamięci, skopiowanie do niego dotychczasowych elementów i dopiero dodanie nowego. Koszt wynosi wówczas ***O*(rozmiar (wektor))**. Pojawia się nowy parametr, bo pojemność można zwiększać o więcej niż jedną komórkę wtedy przepełnienie pojawia się tylko “od czasu do czasu”.

Przykład

- **Analiza z amortyzacją:** badane jest jaka jest oczekiwana wydajność szeregu kolejnych wstawień. Wiadomo, że we przypadku optymistycznym jest to ***O*(1)**, a w przypadku pesymistycznym ***O*(rozmiar)**, ale przypadek pesymistyczny zdarza się rzadko.
- Należy przyjąć pewna hipotezę:
  - kosztAmort(push(x)) = 1
    niczego nie zyskujemy, łatwe wstawienia nie wymagają poprawek, nie pojawia się jednak zapas na kosztowne wstawienia
  - kosztAmort(push(x)) = 2
    zyskujemy zapas na łatwych wstawieniach, ale czy wystarczający…?
    Zależy to od rozmiaru wektora…

- Analiza z amortyzacją: badane jest jaka jest oczekiwana wydajność szeregu kolejnych wstawień. Wiadomo, że we przypadku optymistycznym jest to O(1), a w przypadku pesymistycznym O(rozmiar), ale przypadek pesymistyczny zdarza się rzadko.
- Należy przyjąć pewna hipotezę:

  kosztAmort(push(x)) = 1

  niczego nie zyskujemy, łatwe wstawienia nie wymagają poprawek, nie pojawia się jednak zapas na kosztowne wstawienia

  kosztAmort(push(x)) = 2

  zyskujemy zapas na łatwych wstawieniach, ale czy wystarczający…? Zależy to od rozmiaru wektora…

Przykład dla założenia że koszt amortyzowany to 2. Operacje prawie cały czas są “na minusie” co jest niedopuszczalne

Rozmiar	Pojemność	Koszt Amortyzowany	Koszt	Zapas
0	0			
1	1	2	0+1	1
2	2	2	1+1	1
3	4	2	2+1	0
4	4	2	1	1
5	8	2	4+1	-2
6	8	2	1	-1
7	8	2	1	0
8	8	2	1	1
9	16	2	8+1	-6
10	16	2	1	-5
…	…	…	…	…
16	16	2	1	1
17	32	2	16+1	-14
18	32	2	1	-13

Przykład dla założenia że koszt amortyzowany to 3.

Rozmiar	Pojemność	Koszt Amortyzowany	Koszt	Zapas
0	0			
1	1	3	0+1	2
2	2	3	1+1	3
3	4	3	2+1	3
4	4	3	1	5
5	8	3	4+1	3
6	8	3	1	5
7	8	3	1	7
...	...	...	...	...
16	16	3	1	17
17	32	3	16+1	3
18	32	3	1	5

a) kosztAmort(push(x)) = 3

- Nigdy nie pojawia się “debet”, zaoszczędzone jednostki są niemal w całości zużywane gdy pojawi się kosztowna operacja.

Złożoność zamortyzowana

- W przedstawionym przykładzie wybór funkcji stałej był słuszny, ale zwykle tak nie jest.
- Niech funkcja przypisująca liczbę do konkretnego stanu struktury danych **ds** będzie nazywana **funkcją potencjału**. Koszt amortyzowany definiuje się następująco:

  $$\text{koszAmort}(op_i) = \text{koszt}(op_i) + \text{f.potencjału}(ds_i) - \text{f.potencjału}(ds_{i-1})$$

- Jest to faktyczny koszt wykonania operacji op_i powiększony o zmianę potencjału struktury danych **ds** po wykonaniu tej operacji.
  Definicja ta obowiązuje dla pojedynczej operacji.

  $$\text{koszAmort}(op_1, op_2, op_3, \ldots, op_m) = \sum_{i=1}^{m} (\text{koszt}(op_i) + \text{f.potencjału}(ds_i) - \text{f.potencjału}(ds_{i-1}))$$

- W większości przypadków funkcja potencjału początkowo jest zerem, nigdzie nie jest ujemna, tak że czas amortyzowany stanowi kres górny czasu rzeczywistego.

Kontynuacja przykładu

- f.potencjału ($vector_i$) =

  = 0 (jeśli $rozmiar_i$ = $pojemność_i$ czyli vector jest pełny)

  = 2 $rozmiar_i$ - $pojemność_i$ (w każdym innym przypadku)

- Można sprawdzić że przy tak zdefiniowanej **f.potencjału, koszAmort(op_i)** jest faktycznie równy **3** w każdej konfiguracji (tanie wstawianie, kosztowne, tanie po kosztownym)

Struktury danych i algorytmy obróbki danych zewnętrznych

- Podstawowy czynnik rzutujący na różnice między obróbką danych wewnętrznych (czyli przechowywanych w pamięci operacyjnej) a obróbką danych zewnętrznych (czyli przechowywanych w pamięciach masowych) jest specyfika dostępu do informacji.
- Mechaniczna struktura dysków sprawia, że korzystnie jest odczytywać dane nie pojedynczymi bajtami, lecz w większych **blokach**. Zawartość pliku dyskowego można traktować jako listę łączoną poszczególnych bloków, bądź też jako drzewo którego liście reprezentują właściwe dane, a węzły zawierają informację pomocniczą ułatwiającą zarządzanie tymi danymi.

Struktury danych i algorytmy obróbki danych zewnętrznych

- Załóżmy że:

  adres bloku = 4 bajty

  długość bloku = 4096 bajtów

- Czyli w jednym bloku można zapamiętać adresy do 1024 innych bloków. Czyli informacja pomocnicza do 4 194 304 bajtów będzie zajmować 1 blok.
- Możemy też budować strukturę wielopoziomową, w strukturze dwupoziomowej blok najwyższy zawiera adresy do 1024 bloków pośrednich, z których każdy zawiera adresy do 1024 bloków danych. Maksymalna wielkość pliku w tej strukturze 1024 * 1024 * 1024 = 4 294 967 296 bajtów = 4GB, informacja pomocnicza zajmuje 1025 bloków.

- Nieodłącznym elementem współpracy pamięci zewnętrznej z pamięcią operacyjną są **bufory**, czyli zarezerwowany fragment pamięci operacyjnej, w której system operacyjny umieszcza odczytany z dysku blok danych lub z którego pobiera blok danych do zapisania na dysku.
- **Miara kosztu dla operacji na danych zewnętrznych.**
  - Głównym składnikiem czasu jest czekanie na pojawienie się właściwego sektora pod głowicami. To może być nawet kilkanaście milisekund...
    Co jest ogromnie długo dla procesora taktowanego kilku-gigahercowym zegarem.
  - Zatem "merytoryczna jakość" algorytmu operującego na danych zewnętrznych będzie zależna od liczby wykonywanych **dostępów blokowych** (odczyt lub zapis pojedynczego bloku nazywamy dostępem blokowym).

Sortowanie

- **Sortowanie zewnętrzne** to sortowanie danych przechowywanych na plikach zewnętrznych.
- Sortowanie przez łączenie pozwoli na posortowanie pliku zawierającego **n** rekordów, przeglądając go jedynie ***O*(log n)** razy.
- Wykorzystanie pewnych mechanizmów systemu operacyjnego – dokonywanie odczytów i zapisów we właściwych momentach – może znacząco usprawnić sortowanie dzięki zrównoległowieniu obliczeń z transmisją danych.

Sortowanie przez łączenie

- Polega na organizowaniu sortowanego pliku w pewna liczbę serii, czyli uporządkowanych ciągów rekordów. W kolejnych przebiegach rozmiary serii wzrastają a ich liczba maleje, ostatecznie (posortowany) plik staje się pojedynczą serią.
- Podstawowym krokiem sortowania przez łączenie dwóch plików, f_1 **i** f_2, jest zorganizowanie tych plików w serie o długości **k**, tak że:
  - liczby serii w plikach f_1, f_2, z uwzględnieniem "ogonów" różnią się co najwyżej o jeden
  - co najwyżej w jednym z plików f_1, f_2, może się znajdować ogon
  - plik zawierający "ogon" ma poza nim co najmniej tyle serii ile jego partner.
- Prosty proces polega na odczytywaniu po jednej serii (o długości **k**) z plików f_1, f_2, łączenia tych serii w dwukrotnie dłuższą i zapisywania tak połączonych serii na przemian do plików g_1, g_2.
- Całkowita liczba dostępów blokowych w całym procesie sortowania jest rzędu $O((n \log n)/b)$ gdzie **b** jest liczba rekordów w jednym bloku.

Przykład

Pliki oryginalne:	28	3	93	10	54	65	30	90	10	69	8	22
	31	5	96	40	85	9	39	13	8	77	10	

Pliki zorganizowane w serie o długości 2:	28 31	93 96	54 85	30 39	8 10	8 10
	3 5	10 40	9 65	13 90	69 77	22

Pliki zorganizowane w serie o długości 4	3 5 28 31	9 54 65 85	8 10 69 77
	10 40 93 96	13 30 39 90	8 10 22

Pliki zorganizowane w serie o długości 8:	3 5 10 28 31 40 93 96	8 8 10 10 22 69 77
	9 13 30 39 54 65 85 90	

Pliki zorganizowane w serie o długości 16:	3 5 9 10 13 28 30 31 39 40 54 65 85 90 93 96
	8 8 10 10 22 69 77

Pliki zorganizowane w serie o długości 32:	3 5 8 8 9 10 10 10 13 22 28 30 31 39 40 54 65 69 77 85 90 93 96

Podsumowanie

- *„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, ze ulepszenia sprzętowe uczynią prace nad efektywnymi algorytmami zbyteczna.
- Istnieją problemy których rozwiązanie za pomocą zasobów komputerowych jest teoretycznie możliwe, ale praktycznie przekracza możliwości istniejących technologii. Przykładem takie problemu jest rozumienie języka naturalnego, przetwarzanie obrazów (do pewnego stopnia oczywiście) czy “inteligentna” komunikacja. Pomiędzy komputerami a ludźmi na rozmaitych poziomach.
- Kiedy pewne problemy staja się “proste”… Nowa grupa wyzwań, które na razie można sobie tylko próbować wyobrażać, wytyczy nowe granice możliwości wykorzystania komputerów.