W. 2
Złożoność obliczeniowa algorytmu
(Wstęp do teorii)

Problem

- Potrzebujemy miary „efektywności” działania algorytmu która umożliwi porównanie „efektywności” jego działania z innymi algorytmami tego typu.
- Wielkość ta nie powinna zależeć od komputera na którym się algorytm (program implementujący algorytm) wykonuje. Innymi słowy, powinna to być uniwersalna miara „efektywności” działania.
- Czym jest „efektywność”?
  - Czas działania dla jednostkowej porcji danych w problemie.
  - Objętość w pamięci komputera w jednostkach umownych na porcję przetwarzanych danych przez algorytm.
- Widzimy więc, że potrzebujemy miary „porcji danych” (**n)** przetwarzanych przez algorytm, a chcemy otrzymać funkcję, która mierzy „efektywność” w jednostkach (czas, miejsce w pamięci,...) na przetworzenie „porcji danych” - **T(n)**.

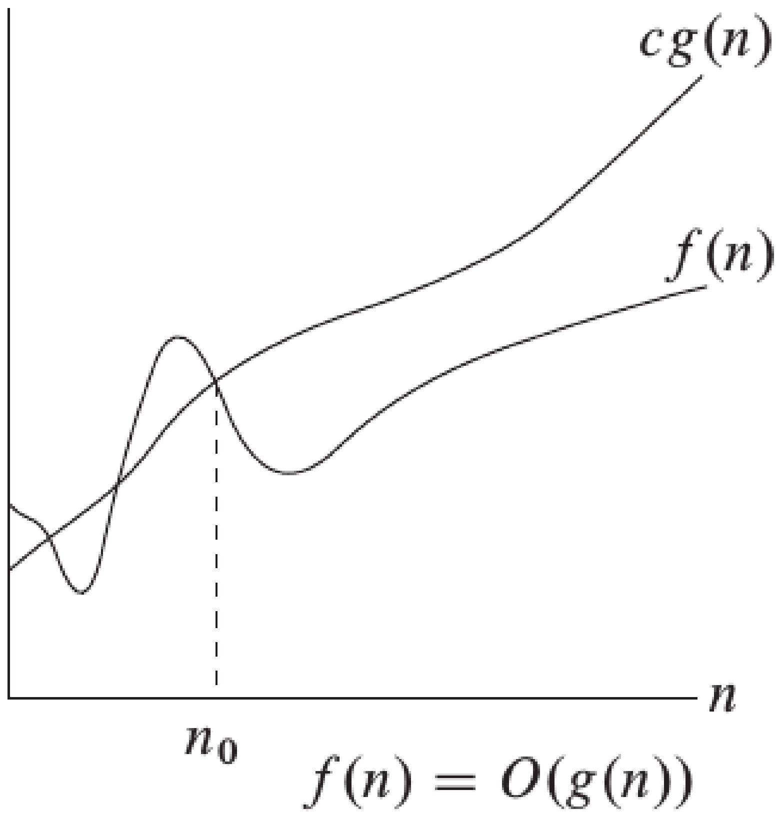
Definicje

- „Porcja danych”:
  - **Rozmiar danych n –** mierzy istotną wielkość danych przetwarzanych przez algorytm.
    - Przykłady:
      - Liczba elementów w przetwarzanej tablicy.
      - Liczba słów w przetwarzanym tekście.
      - Liczba obrazów do przetwarzania.
- „Efektywność”:
  - **Złożoność obliczeniowa algorytmu** jest to funkcja **T(n)**, która informuje o zmianie wybranej charakterystyki działania algorytmu, gdy zmieniamy rozmiar danych **n**.
    - Przykłady:
      - **Czasowa złożoność obliczeniowa algorytmu** – określa jak zmienia się czas przetwarzania danych o rozmiarze **n**. Jest to najczęściej wykorzystywana miara efektywności działania algorytmu.
      - **Pamięciowa złożoność obliczeniowa** – określa wielkość wykorzystywanej pamięci komputera. Jest ona ważna w przypadku wybranych algorytmów. Jej analiza pozwala np. stwierdzić, czy podczas przetwarzania danych wystąpi przepełnienie stosu (stack overflow). Jest ona rzadziej wykorzystywana jako miara efektywności, aczkolwiek również ważna w przypadku problemów wymagających dużo miejsca w pamięci.
- Będziemy głównie skupiać się na czasowej złożoności obliczeniowej, gdyż w przypadku prostych problemów jest ważniejsza (ile czasu należy czekać na przetworzenie danych). **Dlatego poniżej złożoność obliczeniowa będzie oznaczać złożoność czasową.**

Notacja asymptotyczna

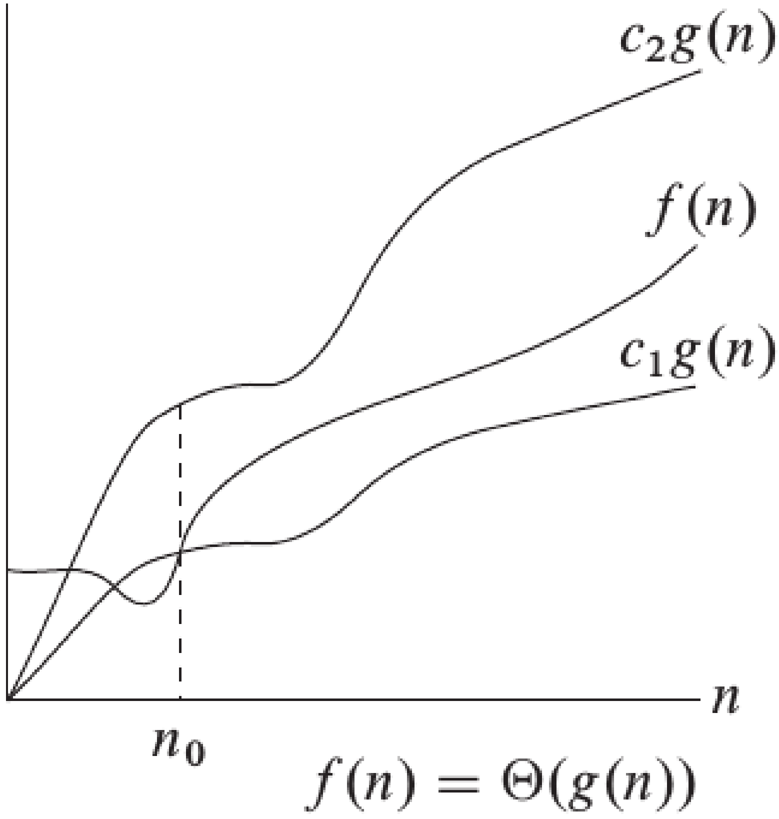
- Motywacja:
  - Często nie jesteśmy w stanie dokładnie policzyć czasowej złożoności **T(n)**, gdyż nie jesteśmy w stanie określić wszystkich narzutów czasowych lub wręcz jest to trudne do określenia/zlokalizowania.
  - Dlatego skupiamy się na głównych fragmentach algorytmu, które mają znaczący wpływ na czas przetwarzania. Zatem **nie jesteśmy** zainteresowani pytaniem: „Ile sekund muszę czekać na przetworzenie porcji danych?”
  - Jesteśmy raczej zainteresowani odpowiedzią na pytanie: „**Ile razy** wydłuży się czas pracy programu, gdy porcję danych zwiększymy, np. z **n** do **2n**.”
  - Znając czas przetwarzania jednostkowej porcji danych n=1, możemy **oszacować** jak zwiększy się (skaluje się) czas przetwarzania n>1 porcji danych.
  - Podsumowując:
    - Często trudno jest oszacować dokładną liczbę jednostek czasu (np. sekund) potrzebną na przetworzenie danych.
    - Chcemy wiedzieć jak skaluję się czas (w umownych jednostkach) gdy rozmiar danych się zmienia. Innymi słowy chcemy znać znaczący wkład (asymptotykę) funkcji **T(n)**.

Notacja - O



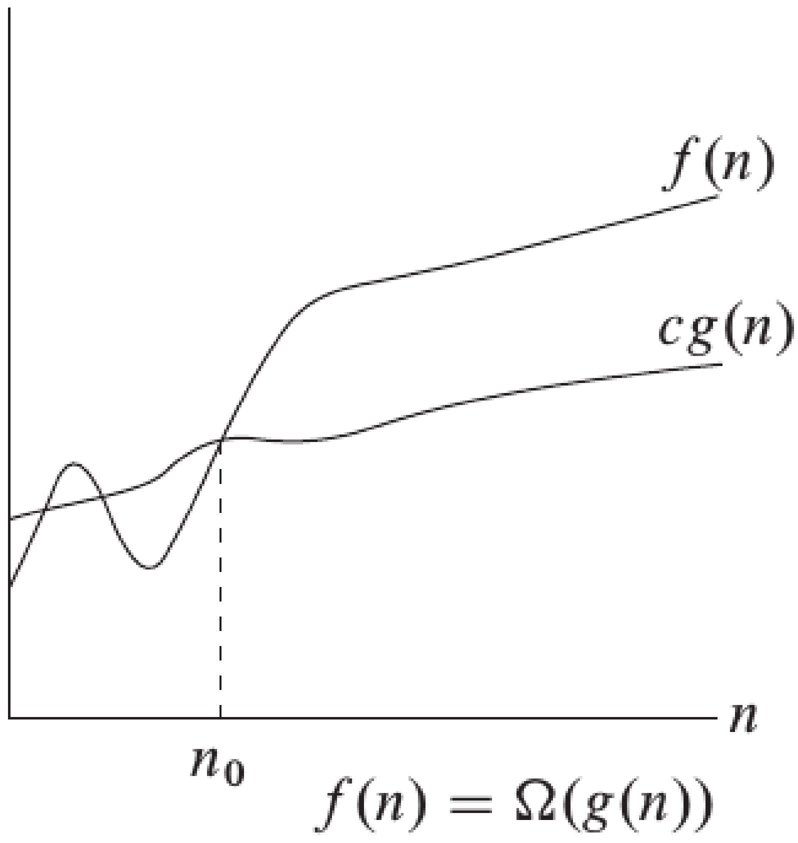
- **Definicja** (O – górne asymptotyczne ograniczenie):
  - **O(g(n))={ f(n) | istnieje dodatnia stała c i n0, że zachodzi 0 <= f(n) <= c g(n) , dla n > n0}.**
- Jest to górne ograniczenie (z dokładnością do stałej multiplikatywnej c) na szybkość wzrostu funkcji.
- Proszę zauważyć, że O(g(n)) jest **zbiorem!!!**
- Przykład:
  - an^2 +bn+c jest klasy O(n^2)
  - a 2^n + bn+c jest klasy O(2^n), gdyż wzrost potęgowy (bn) jest zaniedbywalny w stosunku do wykładniczego (2^n).
  - 10 n oraz 100 n są klasy O(n), zatem przynależność do klasy nie determinuje czynnika mnożącego najszybciej rosnący składnik!!!
- Jest to najczęściej wykorzystywana miara określająca klasę złożoności algorytmu.
- Warto zwrócić uwagę, że jest to tylko górne ograniczenie dla dużych wartości n.

Notacja Θ (grecka duża litera theta)



- **Definicja (**Θ- mocne związanie asymptotyczne)
  - **Θ(g(n)) = { f(n) | istnieje stałe c1,c2 >0 oraz n0, takie, że 0<= c1 g(n) <= f(n) <= c2 g(n), dla n>n0}**.
- Ponownie, jest to **zbiór**.
- Przykłady:
  - 2n^2-3n jest klasy Θ(n^2), gdyż dla n>1 istnieją stałe c1, c2>0, że zachodzi
    - c1 < 2-(1/n) < c2 (narysować !)
- Funkcje należące do tej klasy mają zarówno górne jak i dolne ograniczenie, więc są znacznie lepiej określone niż w przypadku notacji O.

Notacja Ω (grecka duża litera omega)



- **Definicja** (asymptotyczna dolna granica)
  - **Ω(g(n))={f(n) | istnieje stała c >0 oraz n0 >0, że zachodzi 0<= cg(n) <= f(n), dla n>n0}**
  - Jest to również zbiór!
  - Funkcja f(n) należąca do tej klasy Ω(g(n)) nie rośnie wolniej niż g(n) z dokładnością do pewnej multiplikatywnej stałej.

Notacja o (małe o)

- Definicja:
  - o(g(n)) = { f(n) | dla **każdej** dodatniej stałej c >0 oraz dla stałej n0 >0 zachodzi 0<= f(n) **<** c g(n), dla n>n0}.
- f(n) jest klasy o(g(n)), gdy zachodzi: $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$
- Zatem w nieskończoności f(n) jest nieistotna w porównaniu do funkcji g(n).
- Przykład: n^2 jest klasy o(n^3), ale n^2 **nie** jest klasy o(n^2).
- Proszę porównać z definicją O:
  - O(g(n))={ f(n) | **istnieje** dodatnia stała c i n0, że zachodzi 0 <= f(n) **<=** c g(n) , dla n > n0}.

Twierdzenie

- Jeżeli $f(n)$ jest klasy $\Theta(g(n))$, to również jest klasy $O(g(n))$ oraz $\Omega(g(n))$.
  - Dowód:
    - Wynika to z tego, że jeżeli funkcja należy do zbioru $\Theta(g(n))$, to należy do części wspólnej zbiorów $O(g(n))$ oraz $\Omega(g(n))$.

Użyteczne własności

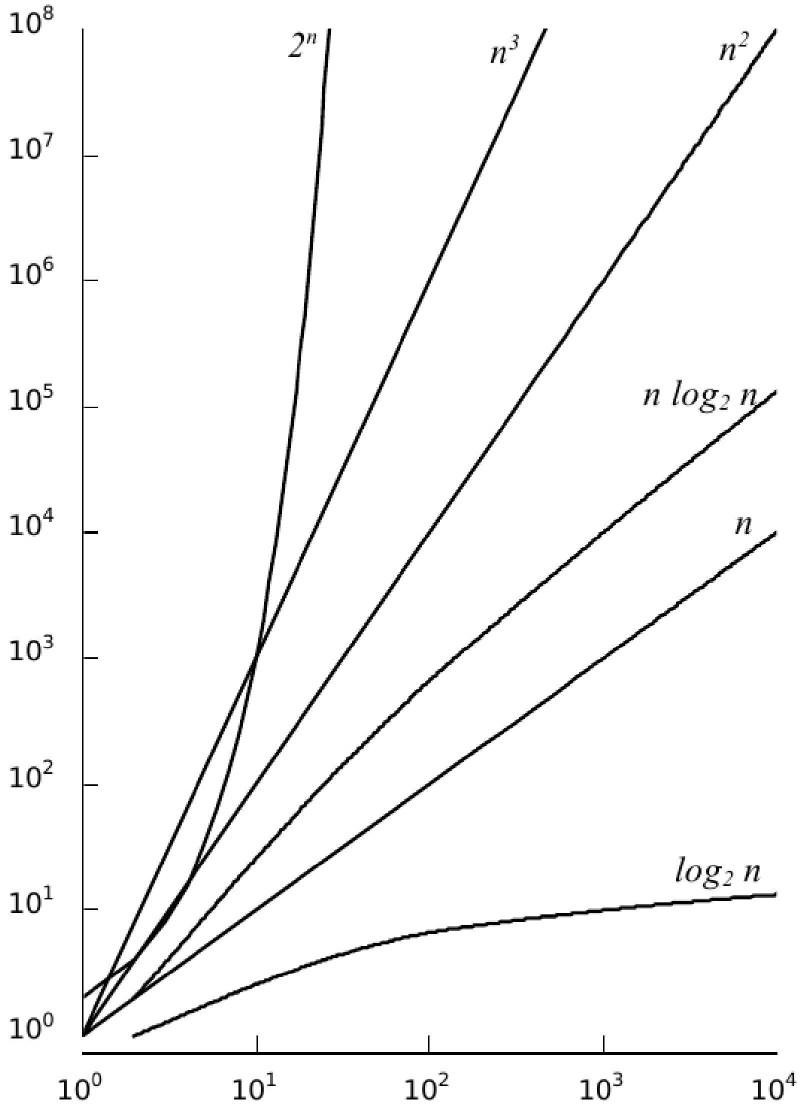
- $cO(f(n)) = O(f(n))$, $c>0$ stała
- $O(f(n))+O(f(n)) = O(f(n))$
- $O(O(f(n)))= O(f(n))$
- $O(f(n))\, O(g(n)) = O(f(n)\, g(n))$
- $O(f(n)\, g(n)) = f(n)\, O(g(n))$

- Proszę udowodnić te własności używając definicji zbioru $O(f(n))$.

Wybrane typy złożoności

- **Klasa O(1)** – złożoność algorytmu jest niezależna od wielkości zadania **n**.
- **Klasa O(log(n))** – złożoność logarytmiczna. Jest lepsza od złożoności liniowej, gdyż log(n) rośnie wolniej od funkcji liniowej n. Jeżeli zadanie zwiększymy 100 razy, to czas obliczeń zwiększy się 2x. Często jednak log w informatyce oznacza logarytm przy podstawie 2, a nie przy podstawie 10 – należy to każdorazowo sprawdzać.
- **Klasa O(n)** – złożoność liniowa: jeżeli zadanie zwiększymy, np. 2x to czas przetwarzania wzrośnie również 2x.
- **Klasa O(n*(log(n))^k)** dla całkowitego k>0 – złożoność kwaziliniowa (quasilinear). Jest to gorsza złożoność od liniowej. Najlepsze algorytmy sortujące (heap sort – sortowanie przez kopcowanie) są klasy O(n*log(n)).
- **Klasa O(n^p)** dla całkowitego p>1 – klasa problemów wielomianowych (P - polynomial). Algorytmy tej klasy są raczej niepraktyczne dla dużego n.
- **Klasa O(a^n)** dla a>0 – algorytmy potęgowe (jedna z podklas NP- non-polynomial – niewielomianowe). Zazwyczaj niepraktyczne, ale czasami użyteczne, gdy dają wynik w sensownym czasie, np. dla małej porcji danych, a nie istnieje inny algorytm rozwiązujący dany problem szybciej.

Wybrane typy złożoności



Algorytmy typu dziel i rządź

Motywacja

- Algorytmy typu dziel i rządź (divide and conquer):
  - Algorytmy te dzielą zadanie przetworzenia **n** danych na zadania mniejsze polegające na tym samym ale przetwarzające mniejsze porcje danych.
  - Dlatego nazywamy je ‘dziel i rządź’.
  - Ich złożoność obliczeniowa **T(n)** często spełnia równanie (rekurencyjne) typu:
    - **$T(n) = a\,T(n/b) + c$**
    - gdzie b jest liczbą podziałów zadania, c - czasem przetworzenia jednej porcji, a – liczba przetwarzanych mniejszych porcji (często b=a, ale nie zawsze).
- Podejście typu dziel i rządź jest również często wykorzystywane w socjologii/polityce do spolaryzowania/podzielenia społeczeństwa na grupy zwolenników i przeciwników wybranych (czasami nieistotne dla nich) poglądów. Takimi grupami łatwiej się manipuluje wyprowadzając na ulice aby zademonstrować siłę danej frakcji lub szantażować przeciwną frakcję. Poza tym grupa lepiej scala i utrwala przynależność osób do danej frakcji.

Równania rekurencyjne
Rozwiązanie przez podstawianie

Rozwiązanie przez podstawianie

- Wiele prostych równań rekurencyjnych możemy rozwiązać rozwijając rekurencję od T(n), aż do T(1), a następnie sumujemy wszystkie wyrazy.
- Często w procesie sumowania wykorzystuje się np.
  - wzory na sumę szeregu geometrycznego
  - oszacowuje się sumę skończoną z góry przez nieskończoną sumę gdy możemy takie nieskończone sumowanie wykonać, a nas interesuje jedynie asymptotyczne oszacowanie – klasa O(...).
- Przedstawimy kilka prostych przykładów.

Silnia

- Algorytm rekurencyjny obliczania silni:
  - Silnia(0)=1
  - Silnia(n)=n*Silnia(n-1)

- Jeżeli czas pojedynczego obliczenia to
  - **T(1) = a**
- To czas kroku rekurencyjnego spełnia równanie:
  - **T(n)=T(n-1)+a**

Równanie rekurencyjne (Silnia)

- Mamy równanie:
  - **T(1)=a**
  - **T(n) = T(n-1) +a**
  - Oblicz klasę funkcji T(n).
- Rozwiązanie równania:
  - Zapisz:
    - T(n)-T(n-1) =a
    - …
    - T(2)-T(1) = a
  - Dodaj stronami:
    - T(n)-T(1) = (n-1)a
  - Wykorzystując T(0)=a, mamy
  - T(n)=an
- **Rozwiązanie:**
  - **T(n) =na, czyli T(n) jest klasy O(n).**

Równanie rekurencyjne (problemy wykładnicze 1)

- Mamy równanie:
  - **T(1)=a**
  - **T(n)= b*T(n-1)**
  - Oblicz klasę T(n).
- Rozwiązanie równania:
  - T(n) = b*T(n-1)=b*b*T(n-2)=...= (b^(n-1))T(1)=(b^n)(a/b)
- **Rozwiązanie:**
  - **T(n) jest klasy O(b^n), zatem jest to problem wykładniczy!!!**
- Z problemami tego typu spotykam się, gdy problem dla n redukuje się do b x problemów rozmiaru (n-1).

Wieże Hanoi

- Algorytm rekurencyjny dla problemu Wież Hanoi:
  - Przesuń(**n**):
    - Przesuń(**n-1**)
    - Przesuń(**1**)
    - Przesuń(**n-1**)

- Mamy więc równanie rekurencyjne:
  - Przesunięcie jednego krążka:
    - **T(1)= c**
  - Równanie rekurencyjne:
    - **T(n)=2*T(n-1)+c**

Równanie rekurencyjne (Wieże Hanoi)

- Mamy równanie:
  - **T(1)= c**
  - **T(n)=2*T(n-1)+c**
  - Oblicz klasę T(n).
- Rozwiązanie równania rekurencyjnego:
  - T(n) = 2*T(n-1)+c = 2*(2*T(n-2)+c)+c =... =
  - =(2^(k))T(n-k)+ c(2^(k-2)+2^(k-3)+...+1)= (2^(n-1))T(1) + c(2^(n-2)+2^(n-3)+...+2^0) = c (2^(n-1)+2^(n-2)+...+1)=|szereg geometryczny| =c (2^n-1)/(2-1)= **c(2^n-1)**
- **Rozwiązanie:**
  - **T(n)=c(2^n-1) jest klasy O(2^n), zatem algorytm jest klasy wykładniczej – jednej z najgorszych dla dużego n.**
  - Dla c=1 (jednostka czasu na wywołanie T(1)) otrzymujemy: T(n)=2^n-1.
  - Liczby postaci 2^n – 1 nazywają się liczbami Mersenne i dla pewnych wartości n są liczbami pierwszymi:
    - https://en.wikipedia.org/wiki/Mersenne_prime
  - Analiza liczby ruchów pokazuje, że problemu nie daje się rozwiązać szybciej niż w wyniku 2^n-1 ruchów, więc algorytm rekurencyjny o złożoności wykładniczej jest optymalny. Jest to problemy typu NP (non-polynomial).
  - Ciekawostka:
    - Problem możemy również „zaprogramować” i rozwiązać przy użyciu organizmów żywych. Przykładem jest rozwiązanie problemu Wież Hanoi przy użyciu kolonii mrówek argentyńskich:
      - https://jeb.biologists.org/content/214/1/50
    - Innym przykładem jest optymalizacja sieci połączeń w Tokijskim metrze przy użyciu (pleśni – Śluzowca) Physarum polycephalum. Jest to problem komiwojażera (travelling salesman problem, TSP) i również należy do problemów z klasy NP (Więcej na ten temat przy omawianiu grafów) :
      - https://science.sciencemag.org/content/327/5964/439.abstract
    - Proszę więc zapamiętać, że komputerem może być dowolna rzecz, która ma sprecyzowany zbiór reguł, które można wykorzystać, by zaimplementować algorytm.

Równania rekurencyjne
Rozwiązanie przez drzewo rekurencyjne

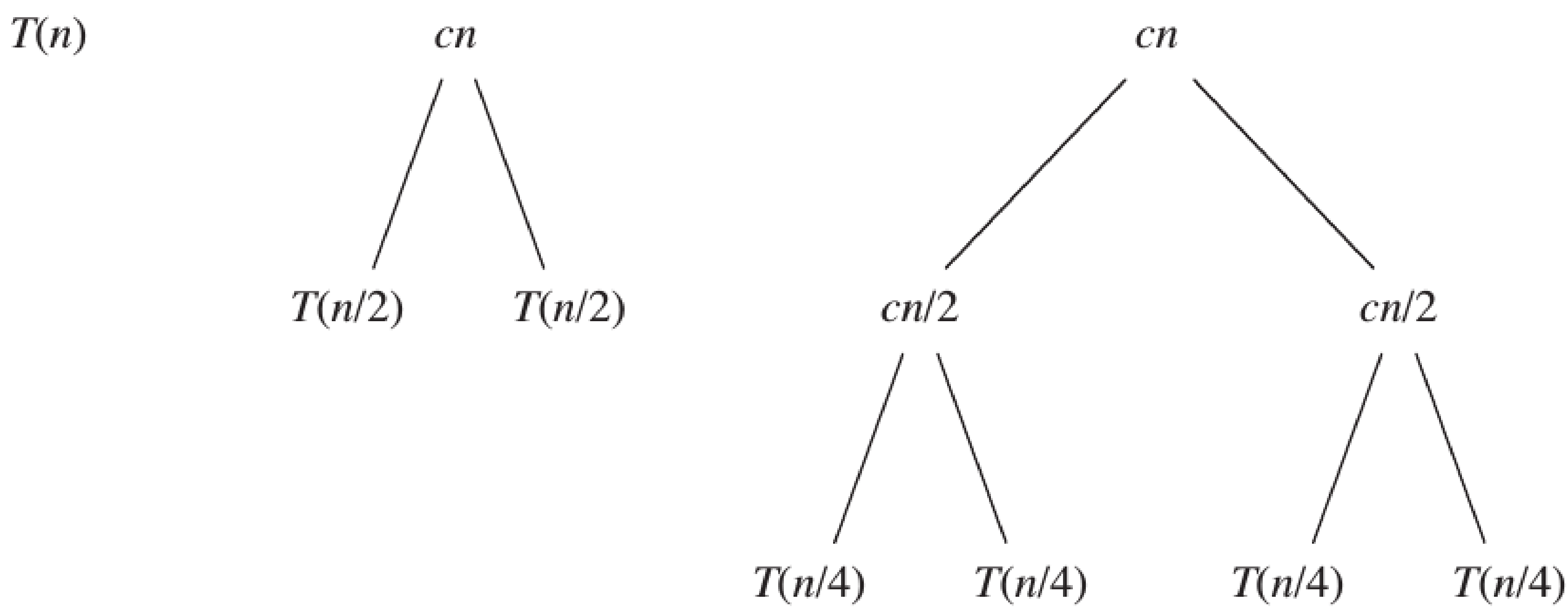
Rozwiązanie przez drzewo rekurencyjne

- Niektóre równania rekurencyjne opisują procesy podziału zadania na kilka mniejszych w sposób rekurencyjny.
- Wówczas problem możemy zwizualizować na strukturze drzewa podziału/wywołań rekurencyjnych.
- Drzewo „spłaszcza” wielkość wielkość problemu, a przez to możemy przyśpieszać rozwiązanie problemu.
- Wiele najefektywniejszych algorytmów ma takie zachowanie.
- Wskazówka:
  - Jeżeli przetwarzanie w gałęzi jest niezależne od innych gałęzi to sugeruje, że przetwarzanie w gałęziach można przyśpieszyć przez przetwarzanie równoległe.

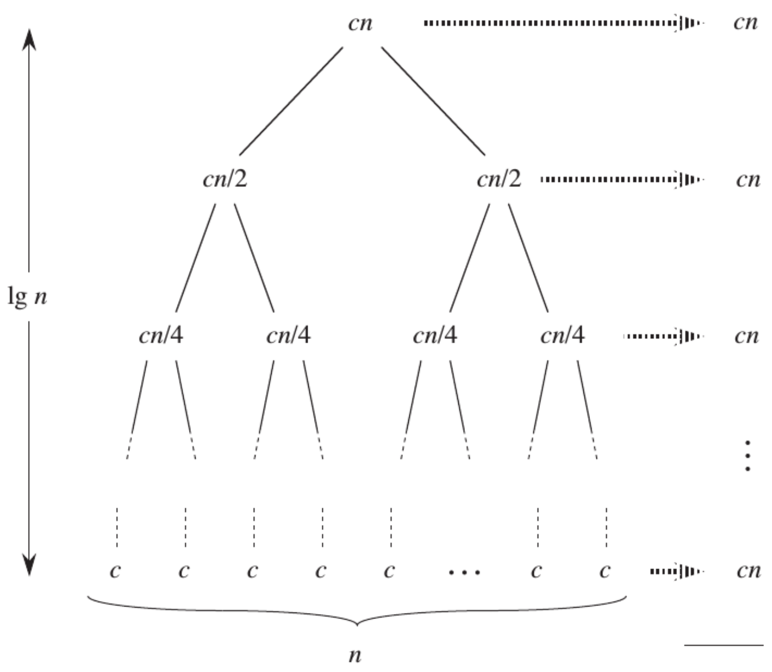
Równanie rekurencyjne 1

- Mamy równanie:
  - **$T(1) = c$**
  - **$T(n)= 2T(n/2)+cn$**
  - Podaj klasę $T(n)$.
- Zauważ, że tego typu problemy w każdym kroku dzielą się na dwa mniejsze o pół ($T(n/2)$), a dodatkowo wykonujemy n zadań o czasie $c=T(1)$.
- Z takim problemem spotkamy się podczas sortowania.

Równanie rekurencyjne 1
drzewo – krok 1



Równanie rekurencyjne 1 drzewo – wszystkie kroki

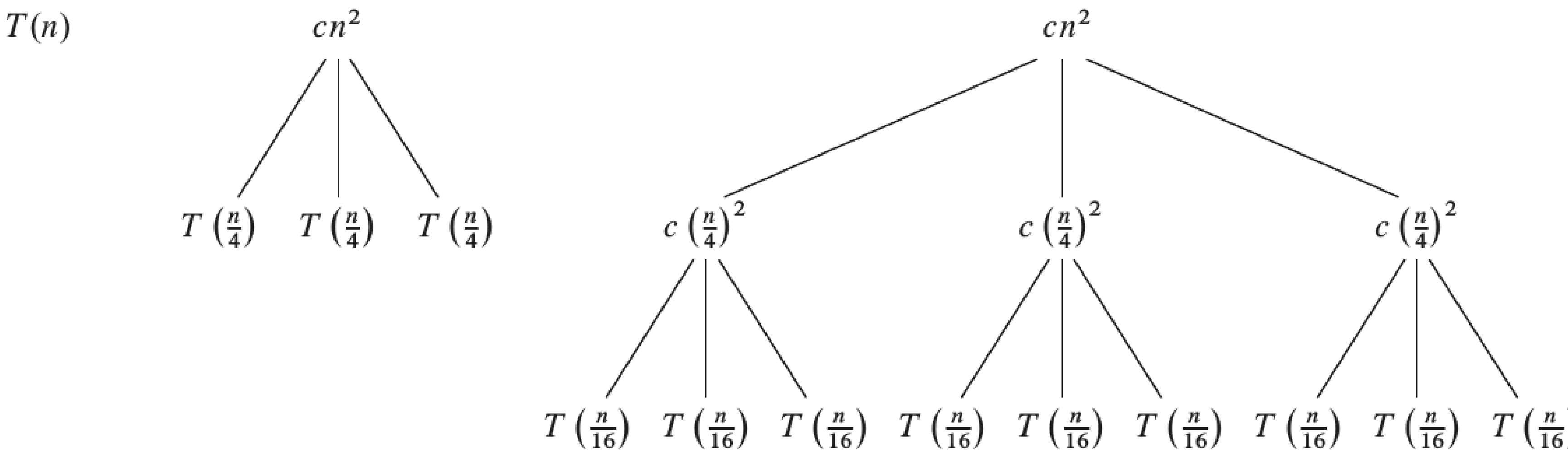


- Mamy drzewo o ln(n) poziomach.
- Na każdym poziomie mamy złożoność c*n.
- Zatem:
  - **T(n)=c*n*ln(n)**
- **Klasą T(n) jest O(n*ln(n))** – klasa kwaziliniowa (quasilinear).
- Klasa ta jest gorsza niż O(n), ale ciągle lepsza niż O(n^2). Proszę narysować!

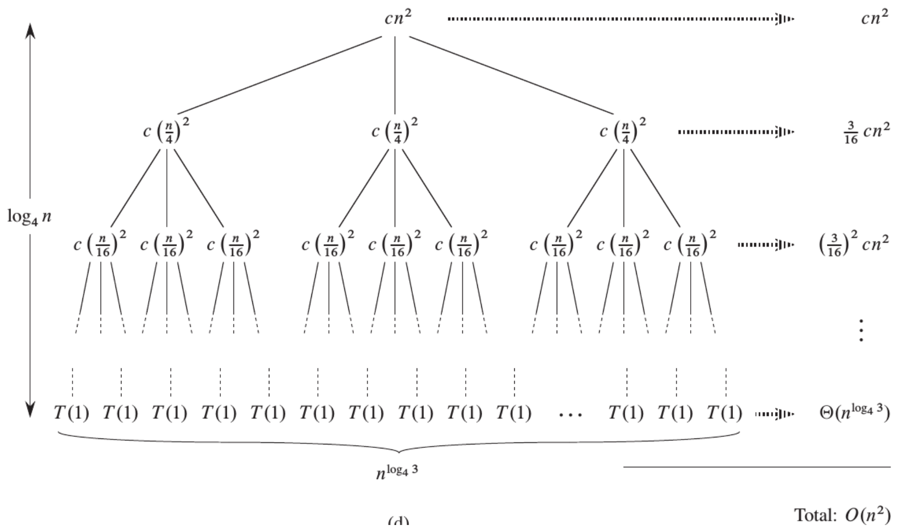
Równanie rekurencyjne 2

- Mamy równanie:
  - **T(1)=c**
  - **T(n)=3T(n/4) + cn^2**
  - Podaj klasę funkcji T(n).
- Równanie to pojawia się, gdy w algorytmie problem rozdziela się na 3 problemy wielkości n/4, a dodatkowo jest wykonywanych c*n^2 obliczeń przy redukcji.

Równanie rekurencyjne 2 drzewo – początkowe kroki



Równanie rekurencyjne 2 drzewo – wszystkie kroki



- Drzewo ma głębokość $\log_4 n$, gdyż w każdym kroku dzielimy wielkość danych na 4 części.
- Najniższy poziom ma $3^\wedge(\log_4 n)=n^\wedge(\log_4 3)$ elementów T(1), tj. 3 podziały na każdy poziom drzewa.
- Złożoność ostatniego poziomu: $3^\wedge(\log_4 n)*T(1)=n^\wedge(\log_4 3)*O(1)=O(n^\wedge(\log_4 3))$.

Równanie rekurencyjne 2 drzewo – oszacowanie

$$T(n)=cn^2+\left(\frac{3}{16}\right)cn^2+\ldots+\left(\frac{3}{16}\right)^{\log_4 3-1}+O(n^{\log_4 3})=cn^2\sum_{i=1}^{\log_4 3-1}\left(\frac{3}{16}\right)^i+O(n^{\log_4 3})<cn^2\sum_{i=1}^{\infty}\left(\frac{3}{16}\right)^i+O(n^{\log_4 3})=\frac{1}{1-3/16}cn^2+O(n^{\log_4 3})=\frac{16}{13}cn^2+O(n^{\log_4 3})$$

Zatem T(n) jest klasy O(n^2).

Ogólne twierdzenie o równaniach rekurencyjnych (Master theorem)

Twierdzenie główne (master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Dowód: T.Cormen et al. ‘Introduction to Algorithms’ (Polskie wydanie: „Wprowadzenie do algorytmów”).

Analiza probabilistyczna
i
Randomizowanie

Idea

- Niektóre problemy, np. poszukiwanie elementu w tablicy, dają różne czasy w zależności od położenia elementu.
- Wówczas możemy podać jedynie losową charakterystykę czasu działania, wartość najlepszą (best case scenario) i najgorszą (worst case scenario).
- W ogólności złożoność obliczeniowa ma pewien rozkład prowdopodobieństwa dla różnego rozkładu danych. Możemy więc jedynie podać charakterystyki(momenty) tego rozkładu – średnią, odchylenie od średniej, itd.
- Często takie rozkłady złożoności obliczeniowej generuje się sprawdzając czasy działania algorytmu na (randomizowanych) losowych danych wejściowych.

Liniowe przeszukiwanie listy (element po elemencie)

- Algorytm liniowego przeszukiwania:
  - Mamy listę n elementów.
  - Szukamy elementu ‘a’, przeszukując komórki jedna po drugiej przesuwając się od początku listy.
- Proszę zaimplementować ten algorytm na dwa sposoby:
  - Iteracja po tablicy (while)
  - Rekurencyjne szukanie (rozwiązanie na następnym slajdzie)

- Jeżeli element ‘a’ znajduje się na początki listy ([a,x,...]), to czas potrzebny na odnalezienie tego elementu to:
  - $T(n)=1$,
- Jeżeli element znajduje się na końcu listy ([…,a]), to czas odnalezienia elementu to
  - $T(n)=n$.
- Zatem jeżeli element jest umieszczony na losowym miejscu w tablicy, to (średni) czas będzie średnią arytmetyczną tych czasów. Zatem średnio:
  - $T(n)=(n+1)/2$, czyli jest to klasa $O(n)$.
- Takie losowe dane często nazywamy danymi randomizowanymi (losowymi).

Liniowe rekurencyjne przeszukiwanie tablicy (Python)

- # Recursive function to search x in arr[l..r]
- def **recSearch**(arr, l, r, x):
  - if r < l:
    - return -1
  - If arr[l] == x:
    - return l
  - if arr[r] == x:
    - return r
  - return **recSearch**(arr, l+1, r-1, x)

- # Driver Code
- arr = [12, 34, 54, 2, 3]
- n = len(arr)
- x = 3
- index = recSearch(arr, 0, n-1, x)
- if index != -1:
  - print "Element", x,"is present at index %d" %(index)
- else:
  - print "Element %d is not present" %(x)

- **Zaimplementuj algorytm w języku C++.**

Problem sekretarki/łowcy posagów/wyboru partnera (on-line hiring problem)

- Problem:
  - Mamy listę **n** kandydatów.
  - Po rozmowie kwalifikacyjnej możemy kandydata wybrać lub odrzucić.
  - **Nie możemy powrócić do odrzuconego kandydata** – możemy się przesuwać tylko do przodu listy kandydatów. Jest to sensowne założenie w przypadku wyboru partner życiowego lub łowcy posagów, gdy nie możemy wrócić do wcześniejszego kandydata.
  - Jak wybrać najlepszego z listy kandydata na stanowisko?

- Załóżmy, że przesłuchujemy k<n osób i je automatycznie odrzucamy – chcemy tylko wyrobić sobie zdanie na temat jakości kandydatów.
- Następnie dla osób o indeksach z zakresu [k+1, n] wybieramy najlepszego kandydata, lepszego od odrzuconych do tej pory.
- Jak wybrać k, aby otrzymać najlepszego kandydata?
- Zauważ, że jeżeli najlepszy kandydat był w pierwszych k, odrzuconych to już nie wybierzemy najlepszego kandydata. Wówczas jesteśmy skazani na ostatniego kandydata z listy.
- Chcemy, aby dobrać takie k, aby sukces był najbardziej **prawdopodobny**. Jednak nie zawsze możemy osiągnąć sukces w takim podejściu.

Przykładowy algorytm iteracyjny

- On-Line-Maximum(k,n):
  - bestscore=- ∞ #najmniejsza możliwa ocena
  - for i =1 to k: #dla pierwszych k kandydatów tylko rejestruj najlepszy wynik i odrzucaj kandydatów
    - If score(i) > bestscore:
      - bestscore = score(i)
  - for i=k+1 to n: #dla kolejnych kandydatów wybierz pierwszego najlepszego
    - ff score(i) > bestscore:
      - return(i) #wybór najlepszego
  - return(n) #zwróć ostatniego jeżeli nie znalazłeś najlepszego w zakresie [k+1,n]

Jak wybrać k optymalnie?

- Załóżmy, że idealny kandydat jest na pozycji k<a <= n. Gdyż tylko wówczas mamy szansę na wybór najlepszego.
- Niech S(i) oznacza, że najlepszy kandydat jest na miejscu i. Prawdopodobieństwo sukcesu (niezależne zdarzenia) to:
  - P(sukces) = P(S(1))+P(S(2))+...+P(S(n)) = P(S(k+1))+...+P(S(n)), gdyż nie odniesiemy sukcesu jeżeli idealny kandydat ma indeks i < k.
- Prawdopodobieństwo sukcesu P(S(i)), i>k jest wówczas gdy spełnione są dwa niezależne zdarzenia:
  - B(i) – najlepszy kandydat jest na pozycji i>k; Ponieważ rozkład najlepszego kandydata jest jednorodny na przedziale [1,n] więc P(B(i))=1/n.
  - O(i) – **drugi** najlepszy kandydat nie jest wybrany, t.j., jest na pozycji automatycznie odrzuconej [1,k]. Zatem prawdopodobieństwo jest stosunkiem długości przedziałów [1,k], [1,i] (pozycja najlepszego), czyli P(O(i)) =k/i.
  - P(S(i))=P(B(i))*P(O(i)) = k/(i*n).
- Sumując po wszystkich prawdopodobieństwach mamy:
  - P(sukces) = k/n (1/(k+1) + 1/(k+2)+ … + 1/n).

Jak wybrać k optymalnie?

- Dla dużych **n** zamieniamy sumę na całkę:
- $P(sukces) \sim \frac{k}{n} \int_k^n \frac{1}{x} dx = \frac{k}{n} \ln\left(\frac{n}{k}\right)$
- Ekstremum funkcji (1/x)*ln(x) jest dla x=e, zatem
- k = n/e
- **Najefektywniejszy algorytm:**
  - **Przeprowadź wywiad odrzucając 1/e ≈ 37% wszystkich kandydatów.**
  - **Następni wybierze najlepszego od wcześniej odrzuconych.**
- Dokładniejsza analiza wszystkich wariantów:
  - T. Cormen et al. „Wstęp do algorytmów” - rozdział 5
  - Wykład na temat problemu sekretarki: Bogdan Miś, „Nowe Ślady Pitagorasa”:
    - https://www.wykop.pl/link/1431111/ile-trzeba-ich-zaliczyc-zeby-potem-wiedziec-ze-to-ta-jedna/

Optymalizacja algorytmów
Praktyczne wskazówki

Wskazówki

- Wybieraj najbardziej optymalne algorytmy do danego zadania.
  - Przemyśl strukturę programu i postaraj się podzielić i rozplanować zadanie tak, aby wykorzystać optymalne podejścia do każdego kawałka.
  - Wiele optymalnych algorytmów do konkretnych zadań jest znanych i już zaimplementowanych w konkretnych językach. Dlatego należy samodzielnie zgłębiać wiedzę z tej dużej rozległej dziedziny aby wiedzieć jaki algorytm wybrać w konkretnej sytuacji.
  - Jak je znaleźć?
    - Literatura:
      - T.Cormen et al. ‘Introduction to Algorithms’ (Polskie wydanie: „Wprowadzenie do algorytmów”).
      - Donald Knuth, „The Art of Computer Programming”
    - Internet:
      - NIST Dictionary of Algorithms and Data Structures: https://xlinux.nist.gov/dads/
      - Rosetta Code: http://www.rosettacode.org/wiki/Rosetta_Code
      - Wikipedia: https://en.wikipedia.org/wiki/List_of_algorithms
    - Inne.
    - Podstawowe algorytmy poznamy na tym kursie, ale jest to wierzchołek góry lodowej.

Wskazówki c.d.

- Wykorzystaj pamięć podręczną – cache - (programowanie dynamiczne) aby nie liczyć wielokrotnie tej samej rzeczy.
- Eliminuj lub optymalizuj operacje czasochłonne, np. wejście-wyjście.
- Przesuwaj część logiki aplikacji w miejsca bardziej odpowiednie do tego, np. wstępne przetwarzanie danych w bazie danych przed ich pobraniem z bazy danych do naszego programu.
- Używaj przetwarzania równoległego. Szczególnie, gdy widzisz, że zadanie dzieli się na części, które możesz przetwarzać niezależnie od siebie. Techniki programowania równoległego nauczymy się na kursie Technik komputerowych w fizyce.
- Używaj flag kompilacji, aby optymalizować kod wynikowy pod konkretną architekturę. Czasami jest to niemożliwe jeżeli aplikacja jest przeznaczona pod kilka różnych typów procesorów i musi być skompilowana na najszerszą gamę procesorów (bez optymalizacji).

Literatura dodatkowa

- Rozdział 3 - Algorytmy. Struktury danych i techniki programowania.
- Rozdziały 3 i 4 – T. Cormen, ‘Wprowadzenie do algorytmów’, PWN
- Rozdział 4 - Data Structures and Algorithms using Python.
- Rozdział 5 - T. Cormen, ‘Wprowadzenie do algorytmów’, PWN (randomizacja)

Więcej?

- Więcej przykładów pojawi się przy omawianiu konkretnych algorytmów.

Koniec

Dodatek

Notacja O w matematyce

- W ogólnym przypadku mówimy, że funkcja f(x) w punkcie x0 jest klasy O(g(x)), gdy
  - $\lim_{x \to x_0} \left| \frac{f(x)}{g(x)} \right| < \infty$
- Notacja O pozwala nam wydobyć istotną w danym punkcie x0 asymptotykę funkcji i ukryć nieistotne szczegóły.
- W przypadku algorytmów interesuje nas granica w x0 będącym nieskończonością.
- Przykład:
  - W x0=0 mamy sin(x) jest klasy O(1), ale również O(x), ale **nie** O(x^2).
  - Szereg Taylora często zapisujemy przy użyciu notacji O. Dla przykładu, szereg Taylora (x0=0) dla funkcji sin(x) to:
    - sin(x) = x + O(x^3)
    - To pozwala liczyć granice w których występują symbole nieoznaczone:
      - Granica sin(x)/x w x0=0 jest granicą wyrażenia: (x+O(x^3))/x= 1+ O(x^2), które w granicy x=x0 to 1.