Podstawy złożoności obliczeniowej

Podsumowanie poprzedniego wykładu

Algorytmika to jedna z najważniejszych dziedzin informatyki i matematyki. Bez algorytmiki nie potrafilibyśmy stworzyć żadnego porządnego programu komputerowego.

Algorytm w matematyce, informatyce oraz w życiu potocznym to skończony uporządkowany zupełnie zbiór jasno zdefiniowanych czynności koniecznych do wykonania pewnego zadania w skończonej liczbie kroków.

Algorytm dokładny przepis podający sposób rozwiązania określonego problemu w postaci skończonej liczby uporządkowanych operacji.

Cechy algorytmów:

- poprawność (algorytm daje dobre wyniki, odzwierciedlające rzeczywistość)
- jednoznaczność (brak rozbieżności wyników przy takich samych danych, jednoznaczne opisanie każdego kroku)
- skończoność (wykonuje się w skończonej ilości kroków)
- sprawność (czasowa - szybkość działania i pamięciowa - "zasobożerność")
- prostota wykonania (operacje powinny być jak najprostsze)

Zależnie od potrzeby, rozwiązanie problemu może być zapisane w postaci:

- **opisu słownego**
- **schematu blokowego**
- **pseudokodu**
- **języka programowania wysokiego poziomu, np. Pascal lub C**

Aby algorytm rozwiązywał poprawnie pewne zagadnienia niezbędna są dane początkowe i zadanie ograniczeń np. warunki brzegowe, jednym słowem należy przedstawić rzeczywistość poprzez:

- zdefiniowanie zadania
- wprowadzenie założeń i ograniczeń
- zdefiniowanie algorytmu (przepisu) rozwiązania

Teoria obliczeń

Teoria obliczeń to dział informatyki.

Jedną z gałęzi tego działu jest teoria złożoności obliczeniowej. W uproszczeniu można powiedzieć, że zajmuje się ona oszacowaniem wydajności czasowej i pamięciowej algorytmów.

Teoria złożoności obliczeniowej bazuje na wielu modelach, które służą do łatwego porównywania algorytmów.

Dlaczego używamy złożoności obliczeniowej

Komputerów na świecie są miliony. Wiele z nich bardzo się od siebie różni. Mają różny procesor, inny moduł RAM. Część z nich używa bardziej wydajnych dysków, które pozwalają na szybszy dostęp do danych.

Dla części z nich dane dostępne są na zdalnych maszynach, do których trzeba łączyć się przez sieć. Są też mega-komputery, maszyny o ogromnej mocy obliczeniowej, czy smartfony w kieszeniach.

W związku z tą różnorodnością pojawia się potrzeba wspólnej miary. Miary, która jest niezależna od zmiennych czynników sprzętowych.

Może ona pomóc zorientować się w **wydajności danego algorytmu**, przyporządkować go do **zdefiniowanej klasy algorytmów**. Tutaj w grę wkraczają **modele**. **Modele** te upraszczają zawiłości związane z różnorodnością sprzętu.

Mamy zatem wspólną bazę – **model**.

Dalej jednak pozostaje pytanie: w jaki sposób mierzyć wydajność poszczególnych algorytmów?

Mierzenie czasu jest mało praktyczne. Mierzenie czasu nie ma większego sensu na komputerze z powodu różnorodności sprzętu. Otrzymane wyniki nie byłyby miarodajne w przypadku innego komputera.

Wyjściem jest mierzyć zatem liczbę operacji wykonanych na **modelu**.

Następnie próbujemy znaleźć **funkcję**, która będzie opisywała liczbę operacji w zależności od wejścia algorytmu. Funkcje te możemy porównywać ze sobą.

Przykład wyznaczania złożoności obliczeniowej

Załóżmy, że chcemy policzyć sumę elementów tablicy. Może nam w tym pomóc następujący algorytm:

```
#include <stdio.h>
#define ROZMIAR 4
int sum();
int number[ROZMIAR] = {3,6,8,10};
int main()
{
int i, wynik ;
wynik = sum(ROZMIAR);
printf ("Druk tablicy tab:\n");
for (i=0; i<ROZMIAR; ++i) {
printf ("Element numer %d = %d\n", i, number[i]);
}
printf ("suma szeregu = %d",wynik);
return 0;
}
    int sum(numbers)
    {
    int sum1 = 0;
    for (int n  = 0; n < numbers; ++n) {
        sum1 += number[n];
    }
    return sum1;
}
```

Ile mamy w nim operacji? int sum = 0; przypisanie to jedna operacja.

Następnie mamy pętlę for. Jej ciało zawiera jedną operację. Sama pętla wykona się dokładnie tyle razy ile jest elementów tablicy number. Liczbę tych elementów określmy, jako n.

Na końcu mamy instrukcję return sum;. Jest to ostatnia operacja.

Dodając te operacje do siebie otrzymujemy wzór:

$$f(n)=1+n+1=n+2$$

Zgodzą się wszyscy że złożoność obliczeniowa naszego algorytmu opisana jest przez funkcję

$$f(n) = n + 2$$

Złożoność obliczeniowa

Złożoność obliczeniową określamy jako **pewną funkcję** rozmiaru danych wejściowych algorytmu.

Teoretycznie powinno się ją wyznaczać, jak opisano w poprzednim punkcie – licząc operacje.

O ile dla naukowców znalezienie dokładnej funkcji może być bardzo istotne, to w **praktyce wystarczą jej oszacowania**. Takie oszacowanie nazywamy w skrócie „**rząd złożoności obliczeniowej**”

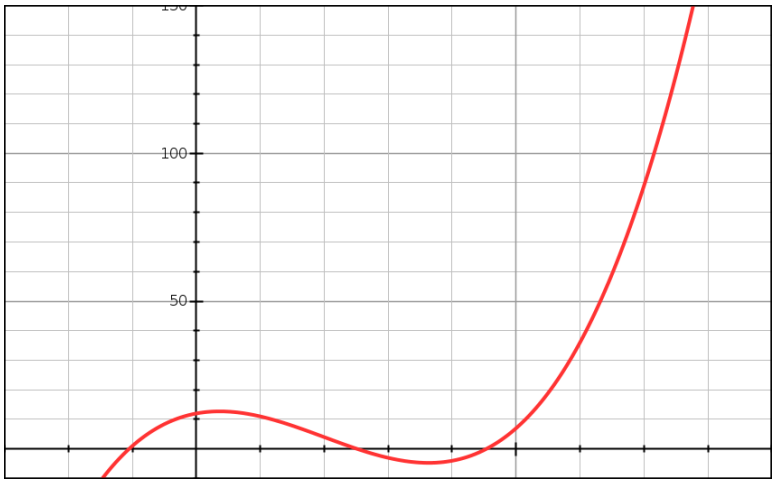
Teoretycznie mamy oszacowania od góry, od dołu i połączenie dwóch poprzednich zwane jako **notacja O** (omikron), **notacja Ω** (omega) i **notacja Θ (**theta).

Oszacowania rzędu złożoności

Weźmy przykładową funkcję opisującą złożoność obliczeniową jakiegoś problemu:

$$f(n)=n^3-6n^2+4n+12$$

Możemy założyć, że funkcja ta opisuje złożoność obliczeniową jakiegoś algorytmu. Argument n to rozmiar danych wejściowych do algorytmu. Wykres tej funkcji wygląda następująco:



Wykres funkcji `f(n) = n^3 - 6n^2 + 4n + 12`

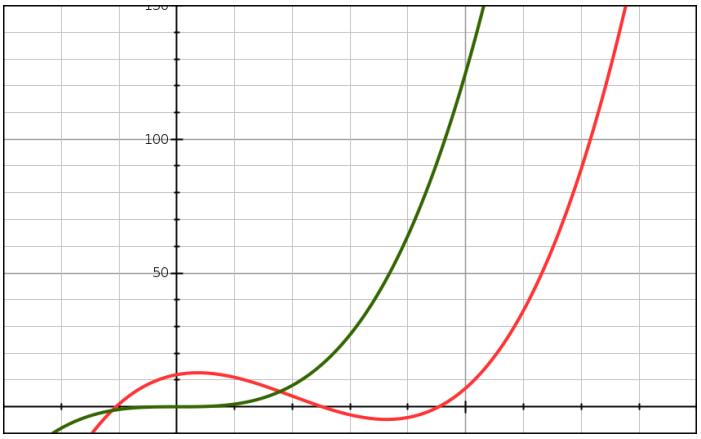
Popatrzmy jak wyglądają oszacowania tego hipotetycznego problemu w **notacji O** (omikron), **notacji Ω** (omega) i **notacji Θ (**theta).

Notacja O (omicron)

Notacja ta zakłada, że istnieje funkcja g(n), dla której spełniona jest poniższa własność:

$$\forall n \geqslant n_0 : f(n) \leqslant c*g(n)$$

Własność ta oznacza, że wynik funkcji g(n) pomnożony przez jakąś stałą c będzie większy bądź równy wynikowi funkcji f(n). Własność ta jest spełniona dla wszystkich n, które będą większe od n_0. Jeszcze łatwiej wygląda to na wykresie:



Oszacowanie z góry, notacja O.

Powyższy wykres pokazuje dwie funkcje. Pierwszą, którą już znamy z poprzedniego wykresu.

Druga to wykres funkcji g(n) = n^3. Jak widać od pewnego punktu zielona linia jest zawsze ponad czerwoną linią. To nic innego jak oszacowanie z góry. To właśnie jest notacja O. Zatem w naszym przypadku nasza funkcja f(n) ma złożoność O(n^3).

Notacja O jest najczęściej spotykana do określania złożoności algorytmów.

Notacja O jest oszacowaniem z góry. Zatem można powiedzieć, że jeśli algorytm ma złożoność O(n^2) to ma także złożoność O(n^3) czy nawet O(n!). Jednak O(n^2) może być najlepszym oszacowaniem złożoności danego algorytmu.

Z racji tego, że jest to oszacowanie pomijamy w nim wszelkiego rodzaju stałe.

Zatem O(2n + 123), O(2n) i O(n) to ta sama złożoność obliczeniowa O(n). Stałe te i tak nie mają znaczenia przy odpowiednio dużych wartościach n.

Notacja Ω (omega)

Notacja ta różni się od poprzedniej własnością, którą spełnia nowa funkcja:

$$\forall n \geqslant n_0 : f(n) \geqslant c*g(n)$$

Własność ta oznacza, że wynik funkcji g(n) pomnożony przez jakąś stałą c będzie mniejszy bądź równy wynikowi funkcji f(n). Własność ta jest spełniona dla wszystkich n, które będą większe od n_0.

Ponownie wykres pomoże to zrozumieć:

Oszacowanie z dołu, notacja Ω.

Na wykresie widoczne są dwie funkcje. Pierwszą znamy. Druga to wykres funkcji g(n) = n^2. “Ostatni” punkt przecięcia tych dwóch wykresów, to n_{02}.

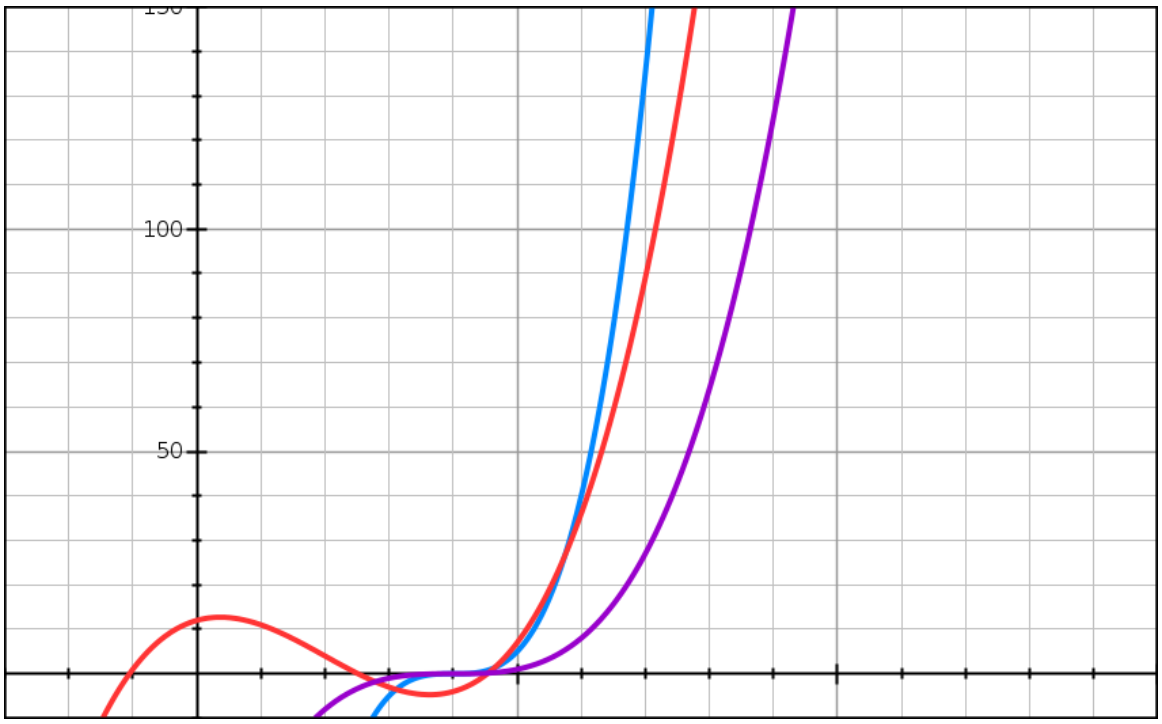
Od tego miejsca wykres funkcji g(n) jest zawsze pod wykresem funkcji f(n). Możemy powiedzieć, że funkcja f(n) ma złożoność Ω(n^2).

Notacja Θ (theta)

Można powiedzieć, że notacja Θ to połączenie notacji O i Ω. W tym przypadku funkcja użyta do oszacowania musi spełniać zależność:

$$\forall n \geqslant n_0 : c1 * g(n) \geqslant f(n) \geqslant c2 * g(n)$$

Można powiedzieć, że wynik funkcji g(n) pomnożony przez stałą c1 będzie większy bądź równy wartości funkcji f(n). Jednocześnie będzie mniejszy bądź równy wartości funkcji f(n) jeśli pomnożymy go przez stałą c2. Ponownie wykres może pomóc to zrozumieć:



Dokładne oszacowanie rzędu, notacja Θ.

W naszym przypadku funkcję g(n) możemy opisać wzorem g(n) = (n-4)^3. Stałe mają odpowiednio wartości c1 = 5, c2 = 1. Wykres oznaczony kolorem niebieskim to wynik funkcji g(n) przemnożony przez stałą c1. Wykres oznaczony kolorem fioletowym to wynik funkcji g(n) przemnożony przez stałą c2.

Jak wcześniej wspomniano notacja O jest najczęściej spotykana. W dalszej części będziemy odnosić się tylko do tej notacji.

Rząd złożoności obliczeniowej

Tutaj skupię się na przykładzie wzoru używanego wcześniej:

$$f(n) = n^3 - 6n^2 + 4n + 12$$

Jak wspomniałem wcześniej w praktyce nie potrzebujemy tak dokładnego wzoru, bowiem wystarczy jedynie zgrubne oszacowanie, które uwzględnia najbardziej istotny element funkcji.

Który element funkcji jest najbardziej istotny? Ten, który ma największy wpływ na ostateczny wynik funkcji. Jak to sprawdzić? Wystarczy pod n podstawić bardzo dużą liczbę i zobaczyć, który element będzie miał największą wartość. Na przykład:

Element	Wartość przy `n = 1'000'000'000`
`n^3`	1′000′000′000′000′000′000′000′000′000
`6n^2`	6′000′000′000′000′000′000
`4n`	4′000′000′000
`12`	12

Jak widać, przy odpowiednio dużych wartościach n część “elementów równania” jest mniej istotna. W przypadku badanej funkcji, najszybciej rosnącym elementem jest n^3. Ma on największy wpływ na ostateczny wynik funkcji.

Z powyższych rozważań wynika, że funkcja **f(n) ma złożoność O(n^3)**. Jest to tak zwana **złożoność wielomianowa**.

Istnieje kilka popularnych rzędów złożoności obliczeniowej.

O(1)

Złożoność stała jest niezależna od liczby danych wejściowych. Mówimy, że problem o złożoności O(1) możemy rozwiązać w stałym czasie niezależnie od wielkości danych wejściowych.

Przykład problemu, dla którego istnieje algorytm złożoności obliczeniowej O(1):

Na wejściu programu jest tablica liczb o długości N. Liczby są posortowane rosnąco. Pomiędzy dwoma sąsiadującymi liczbami różnica jest stała. Znajdź sumę liczb tej tablicy.

Problem to nic innego jak obliczenie sumy ciągu arytmetycznego. Istnieje na to wzór, który można zaimplementować:

```
#include <stdio.h>
#define ROZMIAR 4
int sum();
int number[ROZMIAR] = {3,6,8,10};
int main()
{
int i, wynik ;
wynik = sum(ROZMIAR);
printf ("Druk tablicy tab:\n");
for (i=0; i<ROZMIAR; ++i) {
printf ("Element numer %d = %d\n", i, number[i]);
}
printf ("suma szeregu = %d",wynik);
return 0;
}
int sum(numbers) {
    if ( numbers == 0)          return 0;

    return (number[0] + number[numbers - 1]) * numbers / 2;

}
```

W tym przypadku nie potrzebujemy iterować po elementach tablicy. Niezależnie od wielkości tablicy wejściowej możemy obliczyć sumę ciągu w stałym czasie.

O(n)

Złożoność liniowa. Jest to specyficzny przypadek złożoności wielomianowej. Czas rozwiązania problemu jest wprost proporcjonalny do wielkości danych wejściowych. Przykład problemu, dla którego istnieje algorytm O(n):

Na wejściu programu jest tablica liczb o długości N. Znajdź sumę wszystkich liczb w tablicy wejściowej.

```
#include <stdio.h>
#define ROZMIAR 4
int sum();
int number[ROZMIAR] = {3,6,8,10};
int main()
{
int i, wynik ;
wynik = sum(ROZMIAR);
printf ("Druk tablicy tab:\n");
for (i=0; i<ROZMIAR; ++i) printf ("Element numer %d = %d\n", i, number[i]);

wynik = sum(ROZMIAR);
printf ("suma szeregu = %d",wynik);
return 0;
}
int sum( int numbers) {
   int sum1 = 0, i;
    if ( numbers == 0)         return 0;

        for (i=0; i <= ROZMIAR - 1; i++) {
        sum1 = sum1 + number[i];
        printf("sum1 %d,", sum1);
    }
    return sum1;

}
```

Aby znaleźć tę sumę należy sprawdzić wszystkie elementy tablicy. Musimy zatem odbyć N kroków.

O(log(n))

Złożoność logarytmiczna, czas rozwiązania zależy od wyniku logarytmu z wielkości danych wejściowych. Przykład problemu, dla którego istnieje algorytm O(log(n)):

Przykład: *Na wejściu programu jest posortowana tablica liczb o długości N. Sprawdź czy liczba x istnieje w tablicy wejściowej.*

To popularny algorytm **przeszukiwania binarnego**. Jego nazwa pochodzi od tego, że przy każdej iteracji algorytmu dzielimy przeszukiwany zbiór na dwie równe części. Algorytmy, które dzielą w ten sposób problem na mniejsze problemy przeważnie są zależne od logarytmu wielkości danych wejściowych.

```
#include <stdio.h>
#define ROZMIAR 4
int sum();
int number[ROZMIAR] = {3,6,8,10};
int main()
{
int i, wynik ;

printf ("Druk tablicy tab:\n");
for (i=0; i<ROZMIAR; ++i) printf ("Element numer %d = %d\n", i, number[i]);

wynik = binarySearch(ROZMIAR,8);
printf ("indeks liczby = %d",wynik);
return 0;
}
 int binarySearch( int n, int liczba) {
    int indexLow = 0;
    int indexHigh = n - 1;
    while (indexHigh - indexLow > 1) {
        int indexMiddle = indexLow + (indexHigh - indexLow) / 2;

        if (liczba < number[indexMiddle]) {
            indexHigh = indexMiddle - 1;
        }
        else if (liczba > number[indexMiddle]) {
            indexLow = indexMiddle + 1;
        }
printf ("indeks liczby = %d %d \n",indexLow,indexHigh);
    }
    return indexLow;
}
```

O(nlog(n))

Złożoność liniowo-logarytmiczna. Czas rozwiązania problemu jest wprost proporcjonalny do iloczynu wielkości danych wejściowych i ich logarytmu. Przykładem problemu dla którego istnieje algorytm o złożoności O(nlog(n)) jest:

Na wejściu programu jest tablica liczb. Zwróć tablicę, która będzie zawierała te same elementy, które są w tablicy wejściowej. Tablica wynikowa powinna być posortowana w porządku rosnącym.

Powyższy problem to sortowanie. Jeden ze standardowych problemów w informatyce. Algorytmem sortującym, który ma złożoność obliczeniową O(nlog(n)) jest sortowanie przez scalanie (ang. merge sort):

Algorytm dzieli tablicę na części do czasu aż każda z nich będzie miała długość 1. Następnie scala je ze sobą. Każde takie scalenie to koszt O(n). W związku z tym, że tablicę wejściową dzieliliśmy za każdym razem na pół takich scaleń mamy log(n). Zatem wynikowa złożoność algorytmu to O(nlog(n)).

```
public static int[] sort(int[] numbers) {
    if (numbers.length <= 1) {
        return numbers;
    }
    int[] first = new int[numbers.length / 2];
    int[] second = new int[numbers.length - first.length];
    for (int i = 0; i < first.length; i++) {
        first[i] = numbers[i];
    }
    for (int i = 0; i < second.length; i++) {
        second[i] = numbers[first.length + i];
    }
    return merge(sort(first), sort(second));
}

private static int[] merge(int[] first, int[] second) {
    int[] merged = new int[first.length + second.length];
    for (int indexFirst = 0, indexSecond = 0, indexMerged = 0; indexMerged
< merged.length; indexMerged++) {
        if (indexFirst >= first.length) {
            merged[indexMerged] = second[indexSecond++];
        }
        else if (indexSecond >= second.length) {
            merged[indexMerged] = first[indexFirst++];
        }
        else if (first[indexFirst] <= second[indexSecond]) {
            merged[indexMerged] = first[indexFirst++];
        }
        else {
            merged[indexMerged] = second[indexSecond++];
        }
    }
    return merged;
}
```

O(n^2)

Złożoność kwadratowa. Jest to specyficzny przypadek złożoności wielomianowej. Przykładowy problem może być ten, który użyłem wyżej – posortowanie tablicy. Tym razem jednak algorytm jest mniej wydajny. Sortowanie bąbelkowe charakteryzuje się złożonością obliczeniową O(n^2):

```
#include <stdio.h>
#define ROZMIAR 10
int sum();
int number[ROZMIAR] = {10,9,8,6, 5,4,3,2,1,0};
int main()
{
int i, wynik ;

wynik = sort(ROZMIAR);
printf ("liczba operacji = %d \n",wynik);

printf ("Druk tablicy tab:\n");
for (i=0; i<ROZMIAR; ++i) printf ("Element numer %d = %d\n", i, number[i]);
return 0;
}
 int sort(int n) {
     int l = 0, k = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n - 1; j++) {
            if (number[j] > number[j + 1]) {
                int temp = number[j + 1];
                number[j + 1] = number[j];
                number[j] = temp;
                l++;
            }
            k++;
        }
        printf ("liczba operacji = %d, %d \n",k,l);
    }
    return k;
}
```

Mamy tutaj dwie zagnieżdżone pętle. Każda z nich iteruje po n elementach. W związku z tym otrzymujemy złożoność O(n^2).

O(n^x)

Złożoność wielomianowa. Jak już wiemy złożoność liniowa i złożoność kwadratowa są specyficznymi przypadkami złożoności wielomianowej. Ze względu na częste występowanie wyszczególniłem je jako osobne rzędy złożoności. Przykłady problemów i rozwiązań znajdziesz w poprzednich punktach.

O(x^n)

Jest to złożoność wykładnicza, jej przykładem może być O(2^n). Problemem, który ma rozwiązanie o złożoności co najmniej O(2^n) jest:

Na wejściu programu jest tablica unikalnych liczb. Zwróć tablicę, która będzie zawierała wszystkie możliwe podzbiory elementów tablicy wejściowej.

Wynika to z faktu, że wszystkich możliwych podzbiorów zbioru, który ma n elementów jest dokładnie 2^n. Poniższy algorytm ma złożoność O(log(n)2^n).

```
public static int[][] powerSet(int[] numbers) {
    int two_pow_n = 1 << numbers.length;

    int[][] powerSet = new int[two_pow_n][];
    for (int subsetIndex = 0; subsetIndex < two_pow_n; subsetIndex++) {
        powerSet[subsetIndex] = pickNumbers(subsetIndex, numbers);
    }
    return powerSet;
}

private static int[] pickNumbers(int subsetIndex, int[] numbers) {
    int howManyOnes = 0;
    int temp = subsetIndex;
    while (temp > 0) {
        if (temp % 2 == 1) {
            howManyOnes++;
        }
        temp >>= 1;
    }

    int[] subset = new int[howManyOnes];

    for (int charIndex = 0, lastElementIndex = 0; subsetIndex > 0;
charIndex++) {
        if (subsetIndex % 2 == 1) {
            subset[lastElementIndex++] = numbers[charIndex];
        }
        subsetIndex >>= 1;
    }

    return subset;
}
```

Wynika to z faktu, że pętla wewnątrz metody powerSet wywołana jest dokładnie 2^n razy. Natomiast wewnątrz metody pickNumbers są dwie pętle. Każda z nich ma złożoność O(log(n)). Zatem finalna złożoność algorytmu to O(log(n)2^n).

Spróbuj uruchomić ten kod z tablicą wejściową z 30 elementami, życzę powodzenia ;).

O(n!)

Jest to złożoność typu silnia. Przykładem problemu, dla którego istnieje naiwny algorytm o tej złożoności to problem komiwojażera:

Na wejściu programu jest n miast oraz odległości pomiędzy każdą parą miast. Zakładając, że komiwojażer zaczyna z miasta A i ma dojść do miasta B jaką trasę powinien pokonać aby była ona najkrótsza?

Nie silę się nawet na naiwny algorytm dla tego problemu, nie jest on możliwy do uruchomienia na dzisiejszych komputerach dla problemów odpowiednio dużych. Wyobraź sobie, skalę możliwych rozwiązań. (60 - 1)!/2 ~= 6,9 * 10^795. Szacowana liczba atomów wodoru w widzialnym wszechświecie to około 10^80. Przekładając to na problem wyżej, możliwych dróg pomiędzy 60 miastami jest tylko 31% mniej niż atomów wodoru w widzialnym wszechświecie ;).

Najlepszy, średni i najgorszy przypadek

Ten sam algorytm może zachowywać się zupełnie inaczej w przypadku innych danych wejściowych. Nie mówię tu o wielkości problemu, wielkości danych wejściowych. A o instancji problemu.

Jeśli algorytm sortujący, jako dane wejściowe przyjmuje tablicę liczb o rozmiarze n = 5, to wielkością problemu może być teoretycznie O(n) = 5 (jeśli podana tablica jest już posortowana.

Natomiast instancji tego problemu jest nieskończenie wiele, np: [1, 2, 3, 4, 5], [-1, 2, -3, 4, -5] czy [5, 4, 3, 2, 1]. Określony algorytm może mieć złożoność obliczeniową określoną w notacji O w zależności od instancji problemu. Są także algorytmy, których złożoność obliczeniowa jest niezależna od instancji problemu.

W zależności od **wymagań** w wyborze algorytmu bierze się pod uwagę złożoność odpowiedniego przypadku.

Podsumowanie

Wraz z rozwojem informatyki, a tym samym algorytmiki, narastała potrzeba oceny jakości algorytmów. Na przełomie wielu lat pojawiło się bowiem wiele algorytmów rozwiązujących te same problemy na różne sposoby.

Narastająca potrzeba oceny jakości algorytmów przyczyniła się do powstania teorii złożoności obliczeniowej, która na dzień dzisiejszy jest powszechnie stosowaną i uznawaną techniką do porównywania wydajności algorytmów. Wydajność algorytmów z kolei rozpatrywana jest obecnie według dwóch wiodących kryteriów tj. wg złożoności czasowej oraz wg złożoności pamięciowej.

Złożoność algorytmów

Niezależnie od rozpatrywanego rodzaju złożoności obliczeniowej, tj. złożoności czasowej czy też złożoności pamięciowej; u podstaw znajduje się teoria będąca częścią wspólną dla obu wymienionych zagadnień.

Celem złożoności obliczeniowej jest możliwość oszacowania teoretycznej pesymistycznej, średniej oraz optymistycznej wydajności danego algorytmu. Aby istniała możliwość szacowania złożoności obliczeniowej, koniecznym było zdefiniowanie wspólnej miary złożoności algorytmów, która byłaby niezależna od zastosowanego języka programowania, sprzętu czy też kompilatora. Miara taka została zdefiniowana i jest nią liczba operacji przez algorytm, w celu założonego wyniku.

Złożoność czasowa

Kluczową rolę w dziedzinie algorytmiki pełni złożoność czasowa algorytmów. Za pomocą niej określamy jak długo dany algorytm musi pracować aby wykonał swoje zadanie. Czas pracy algorytmu nie jest wyrażany w sekundach, lecz w jednostkach, którym nie jest przypisana żadna rzeczywista miara.

Czas pracy algorytmu jest więc w praktyce liczbą, której wartość jest funkcją danych wejściowych oraz zasad opisujących sposób ich przetwarzania przez dany algorytm. Warto również wiedzieć, że złożoność czasową algorytmów **szacuje się zgrubnie** przy pomocy tzw. **notacji omikron O(n).**

Złożoność pamięciowa

Celem złożoności pamięciowej jest oszacowanie zużycia pamięci przez dany algorytm. Złożoność pamięciową podobnie jak w przypadku złożoności obliczeniowej szacuje się na podstawie liczby danych wejściowych, przekazanych do algorytmu oraz sposobu ich przetwarzania przez dany algorytm. W przypadku szacowania złożoności pamięciowej rozpatruje się tylko i wyłącznie pamięć, którą należy dodatkowo zaalokować w trakcie pracy algorytmu tak, aby było możliwe jego wykonanie zgodnie z zasadami określającymi sposób

działania algorytmu. Do złożoności pamięciowej nie wlicza się więc rozmiaru danych wejściowych, które zostały przekazane do danego algorytmu.

Rzędy złożoności czasowej

Złożoność czasowa	Opis	Szybkość algorytmu
O(**1**)	**stała złożoność** – algorytm wykonuje się w stałej ilości operacji, niezależnie od liczby danych wejściowych.	najszybszy
O(**$\log_2(n)$**)	**złożoność logarytmiczna** – algorytm wykonuje logarytmiczną ilość operacji w stosunku do liczby danych wejściowych. **n** jest to liczba danych wejściowych; podstawa logarytmu zazwyczaj wynosi **2**, jednak czasami może być większa (np. w B-drzewach).	bardzo szybki
O(**n**)	**złożoność liniowa** – algorytm wykonuje wprost proporcjonalną ilość operacji do liczby danych wejściowych.	szybki
O(**$n*\log_2(n)$**)	**złożoność liniowo-logarytmiczna**	szybki
O(**n^2**)	**złożoność kwadratowa** – ilość operacji algorytmu jest wprost proporcjonalna do liczby danych wejściowych podniesionej do potęgi drugiej.	niezbyt szybki
O(**n^X**)	**złożoność wielomianowa** – ilość operacji algorytmu jest wprost proporcjonalna do liczby danych wejściowych podniesionej do potęgi **X**.	wolny
O(**X^n**)	**złożoność wykładnicza** – ilość operacji algorytmu jest wprost proporcjonalna do stałej **X** większej lub równej **2**, podniesionej do potęgi równej liczbie danych wejściowych. **X** jest stałą większą niż **2**.	

Klasy problemów NP

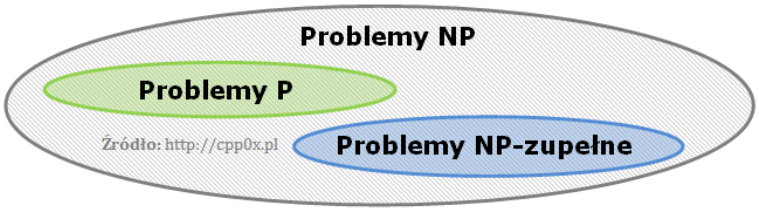
Cała klasa problemów o złożoności czasowej wielomianowej lub większej, uznawana jest za ciekawy obszar do prowadzenia wszelkich badań naukowych. Dla wspomnianych problemów postanowiono więc utworzyć kolejne zagadnienia, które umożliwiłyby precyzyjniejsze określanie poziomu trudności danego problemu. Zagadnienia te przyjęły następujące nazewnictwo:

- **problem P;**
- **problem NP;**
- **problem NP-zupełny;**
- **problem NP-trudny.**

Znaczenie poszczególnych zagadnień zostanie wyjaśnione w dalszej części niniejszego rozdziału.

Problem NP

Problemami klasy NP nazywamy **problemy obliczeniowe** w których znalezienie poprawnego rozwiązania wymaga co najmniej złożoności obliczeniowej wielomianowej. Warto również wiedzieć, że w klasie problemów NP zawierają się problemy P oraz problemy NP zupełne.



Problem P

Problemami klasy P nazywamy **problemy obliczeniowe** w których znalezienie rozwiązania wymaga złożoności obliczeniowej wielomianowej. Problemy P należą do zbioru problemów NP.

Problem NP-zupełny

Problemami klasy NP-zupełnej nazywamy **problemy obliczeniowe** w których znalezienie rozwiązania nie jest możliwe w czasie wielomianowym. Problemy NP-zupełne należą do zbioru problemów NP jak również do zbioru problemów NP-trudnych.

Problemy NP-zupełne można zaprezentować następująco:

Problem NP-trudny

Problemami klasy NP-trudnej nazywamy **problemy obliczeniowe** dla których znalezienie rozwiązania problemu nie jest możliwe ze złożonością wielomianową oraz sprawdzenie rozwiązania problemu jest co najmniej tak trudne jak każdego innego problemu NP. Problemy NP-Trudne obejmują zarówno **problemy decyzyjne** jak również **problemy przeszukiwania** czy też **problemy optymalizacyjne**.

Podsumowanie problemów NP

Klasy problemów NP obejmują najtrudniejsze problemy złożoności czasowej algorytmów. Podsumowaniem tematyki klasy problemów NP jest tabela reprezentująca ich zależności w stosunku do klas złożoności czasowej.

<table>
<tr><th rowspan="2">Klasa problemu NP</th><th colspan="2">Klasy złożoności czasowej</th></tr>
<tr><th>Sprawdzenie rozwiązania</th><th>Znalezienie rozwiązania</th></tr>
<tr><td>problem P</td><td rowspan="3">złożoność wielomianowa:
P = O(n^x)</td><td>złożoność wielomianowa:
NP = O(n^x)</td></tr>
<tr><td>problem NP</td><td>złożoność wielomianowa
lub wykładnicza:
NP = O(n^x)
NEXPTIME = O(Xⁿ)</td></tr>
<tr><td>problem NP-zupełny</td><td rowspan="2">złożoność wykładnicza:
NEXPTIME = O(Xⁿ)

Źródło: http://cpp0x.pl</td></tr>
<tr><td>problem NP-trudny</td><td>złożoność wielomianowa
lub wykładnicza:
P = O(n^x)
EXPTIME = O(Xⁿ)</td></tr>
</table>

Metody rozwiązywania problemów NP

Problemy klasy NP obarczone są dużą złożonością czasową. Złożoność czasowa wpływa z kolei na realny czas wykonywania danego algorytmu, który z założenia powinien być maksymalnie krótki, a jeżeli jest to niemożliwe – mieścił się w określonym czasie.

Istotnym problemem dla klasy problemów NP-Trudnych jest fakt, że znalezienie rozwiązania wymaga złożoności czasowej wykładniczej, a zatem niemożliwym jest efektywne przetwarzanie danych wejściowych na współczesnych komputerach.

Fakt ten wymusza z kolei dostosowywanie algorytmów do realnych możliwości komputerów, a zatem koniecznym jest redukowanie kosztów czasowych algorytmów, często poświęcając przy tym jakość uzyskiwanych wyników, czyli uzyskiwanych rozwiązań. Niniejszy rozdział będzie się więc koncentrował na przeglądzie ogólnych metod rozwiązywania problemów klasy NP.

Metody przeszukiwania brute-froce

U podstaw wszystkich problemów NP-Trudnych znajduje się metoda rozwiązywania problemów poprzez zastosowanie techniki przeszukiwania brute-force. Metoda ta polega na sprawdzeniu wszystkich możliwości w celu znalezienia najlepszego możliwego rozwiązania problemu dla zadanych danych wejściowych. Technika przeszukiwania brute-force ma dwie kluczowe zalety:

- zawsze zostanie znalezione najlepsze możliwe rozwiązanie (jeżeli rozwiązanie istnieje);

- jest łatwa do zaimplementowania.

Metoda przeszukiwania brute-force nadaje się zatem przy pierwszym rozpoznaniu danego problemu bowiem mamy możliwość napisania prostego algorytmu, rozwiązującego dany problem małym nakładem pracy.

Wspomniana metoda jest jednak obarczona bardzo poważną wadą tj. złożoność czasowa algorytmu rośnie wykładniczo, a zatem sprawdzać się będzie tylko i wyłącznie dla bardzo małej liczby danych wejściowych, nieprzekraczających rzędu 15-40 rekordów. Problem wykładniczej złożoności czasowej obrazuje poniższa tabela:

Moc obliczeniowa	3,00 GHz ≈ 3 000 000 000 liczba operacji/sek.	
Liczba rekordów $O(2^n)$	**Konieczna liczba operacji**	**Przybliżony czas trwania obliczeń**
10	1 024	0 ms
20	1 048 576	0,3 ms
30	1 073 741 824	358 ms
40	1 099 511 627 776	6 min
50	1 125 899 906 842 620	4 dni
60	1 152 921 504 606 850 000	12 lat

Z powyższej tabeli wynika więc, że posiadając algorytm o złożoności czasowej $O(2^n)$ czas pracy algorytmu wyniesie co najmniej 12 lat przy 60 rekordach w danych wejściowych.

Warto w tym miejscu również dodać, że w rzeczywistości czas ten będzie znacznie dłuższy, ponieważ przedstawiona tabela zakłada, że każda operacja algorytmu wymaga jednego cyklu procesora, co w rzeczywistości jest nieosiągalne.

W praktyce więc ‘konieczną liczbę operacji’ oraz ‘przybliżony czas trwania obliczeń’ z powyżej tabeli można przemnożyć przez wartość wynoszącą co najmniej 50 cykli lub większą, co da znacznie bardziej realne czasy wykonywania algorytmów, uruchamianych na współczesnych komputerach.

Metody heurystyczne

Kolejną metodą rozwiązywania problemów NP-Trudnych poszukiwanie rozwiązania metodą heurystyczną. Metoda heurystyczna jest to metoda polegająca na poszukiwaniu rozwiązania problemu NP-Trudnego ze złożonością co najwyżej wielomianową.

Algorytm heurystyczny charakteryzuje się tym, że z dużym prawdopodobieństwem rozwiązanie optymalne problemu nie zostanie znalezione, jednak znalezione rozwiązanie będzie bliskie optymalnemu. Bliskie optymalnemu z kolei oznacza, że w wyniku działania algorytmu heurystycznego powinieneś uzyskać rozwiązanie względnie satysfakcjonujące zużywając przy tym mniejszą ilość zasobów czasowych niż przy algorytmie o złożoności wykładniczej.

Algorytmy heurystyczne zazwyczaj mają wyznaczoną pesymistyczną granicę błędu w stosunku do rozwiązania optymalnego, także w wielu przypadkach znajdują one swoje zastosowanie w codziennym życiu.

Metody genetyczne

Najmniej przewidywalną metodą rozwiązywania problemów NP-Trudnych ze względu na jakość uzyskiwanych rozwiązań jest metoda genetyczna. Metoda genetyczna w dużym uproszczeniu polega na zaimplementowaniu algorytmu, który potrafi wygenerować rozwiązanie, a następnie stara się to rozwiązanie poprawić. Wynik każdego rozwiązania dokonanego w przeszłości wpływa na wyniki uzyskiwane w przyszłości. Celem algorytmów genetycznych jest poprawianie jakości uzyskiwanych rozwiązań wraz z przyrostem liczby przeprowadzonych treningów na bazie danych testowych. Największymi problemami algorytmów genetycznych są:

- algorytm genetyczny w wyniku treningu bardzo często traci zdolność znajdywania rozwiązania optymalnego;
- przetrenowanie algorytmu genetycznego prowadzi do wyraźnego spadku zdolności rozwiązywania problemów na które system nie był trenowany;
- nie da się określić pesymistycznej granicy błędu w stosunku do rozwiązania optymalnego;
- sposób rozwiązywania problemu jest niepowtarzalny tj. trenując algorytm tymi samymi danymi w innej kolejności uzyskamy zupełnie inny genotyp, który może generować zarówno lepsze jak i gorsze wyniki niż wcześniej wytrenowany algorytm;
- algorytm genetyczny może przestać generować wyniki satysfakcjonujące dla danych, które wcześniej były dla nas akceptowalne.

Pomimo, iż liczba wad wynikająca z rozwiązywania problemów metodą genetyczną jest przytłaczająca, to mimo wszystko jest ona interesującym obszarem badań naukowych ze względu na małą złożoność czasową jaką można uzyskać w tego typu algorytmach.

Metody losowe

Uproszczoną wersją metody genetycznej jest rozwiązywanie problemów metodą losową. Metoda losowa w przeciwieństwie do metody genetycznej wymaga znacznie mniejszego nakładu pracy, a ponadto nie wyklucza znalezienia optymalnego rozwiązania.

Istotnym problemem algorytmów wykorzystujących metodę losową jest fakt, że wraz ze wzrostem liczby danych wejściowych, drastycznie maleje szansa na znalezienie rozwiązania optymalnego. Pomimo, iż znalezienie rozwiązania optymalnego może okazać się niemożliwe przy zastosowaniu tej metody to metoda losowa nie wyklucza znajdywania rozwiązań bliskich optymalnemu.