Podstawy Informatyki
Wydajność algorytmów
IT, IO, WIMiIP

Danuta Szeliga

AGH Kraków

Spis treści

Analiza algorytmów

Analiza algorytmów

- Każde wykonanie algorytmu na komputerze wymaga wykonania pewnej pracy obliczeniowej oraz pewnej ilości miejsca w jego pamięci
- Zatem projektując algorytm powinniśmy odpowiedzieć na pytanie:

  > Czy nasz komputer umożliwia stosowanie opracowanego algorytmu dla danych o przewidywanym rozmiarze?

- Okazuje się bowiem, że dla wielu znanych algorytmów czas ich działania rośnie zbyt szybko wraz ze wzrastem rozmiaru danych wejściowych

Analiza algorytmów

Analiza algorytmu

polega na określeniu zasobów, jakie potrzebne są do jego wykonania

Zasoby

- czas obliczeń $\rightarrow$ złożoność obliczeniowa
- pamięć $\rightarrow$ złożoność pamięciowa lub wymagania pamięciowe
- szerokość kanału komunikacyjnego
- sprzęt komputerowy

Wszystkie te zasoby są wyrażane jako funkcja rozmiaru danych wejściowych

Analiza algorytmów

Czas działania

Zachowanie algorytmu może być różne dla różnych możliwych danych wejściowych $\rightarrow$ potrzebujemy środków do wyrażania zachowania algorytmów w postaci prostych, łatwych do zrozumienia formuł

- Czas obliczeń mierzy się zazwyczaj liczbą (dominujących) operacji elementarnych wykonywanych przez procesor w celu rozwiązania danego problemu za pomocą opracowanego algorytmu
- Zakłada się, że wykonanie pojedynczej i-tej operacji wymaga czasu c_i, który jest stały dla danego komputera

Analiza algorytmów

Zużycie pamięci

- Zużycie pamięci mierzy się liczbą zmiennych oraz liczbą i rodzajem struktur danych użytych przez dany algorytm (z uwzględnieniem ich rozmiaru)
- Definicja rozmiaru danych wejściowych zależy istotnie od rozważanego problemu
- Z reguły rozmiarem danych wejściowych jest długość (liczba elementów) ciągu wejściowego
- Jednostka: słowo pamięci
- Przykłady definicji rozmiaru problemu
  - sortowanie tablicy: długość tablicy
  - problemy grafowe: ilość węzłów i krawędzi
  - operacje na wielomianach: stopień wielomianu
  - operacje na macierzach: rozmiary macierzy
  - operacje arytmetyczne: całkowita liczba bitów

Wpływ danych na działanie algorytmu

Założenie: we wszystkich przypadkach rozmiar danych wejściowych jest taki sam

- Przypadek optymistyczny
  dane wejściowe są takie, że dany algorytm znajduje rozwiązanie w minimalnej liczbie kroków
- Przypadek pesymistyczny
  dane wejściowe są takie, że dany algorytm znajduje rozwiązanie wykonując największą możliwą dla danego rozmiaru danych liczbę operacji
- Przypadek średni (oczekiwany)
  statystycznie najbardziej prawdopodobny — dla losowo wybranych danych

Złożoność obliczeniowa

Złożoność obliczeniowa

- W celu określenia złożoności obliczeniowej algorytmu, wyrażamy ilość operacji potrzebnych do rozwiązania danego problemu przez algorytm zależością matematyczną (np. $f(n) = an^2 + bn + c$).
- W praktyce interesuje nas dominujący składnik we wzorze na $f(n)$ — zachowanie asymptotyczne

Rząd wielkości funkcji $f(n)$

to dominujący składnik w $f(n)$ z pominięciem stałych współczynników

- Rząd wielkości funkcji określa, jak szybko rośnie funkcja, gdy rośnie argument n $(n \to \infty)$
- Przeważnie mówimy, że dany algorytm jest lepszy od innego, jeśli jego pesymistyczny czas działania jest funkcją niższego rzędu (ALE: nie musi to być słuszne dla małych rozmiarów danych wejściowych!)

Notacja asymptotyczna

Asymptotyczna złożoność algorytmu

to określenie rzędu wielkości czasu działania algorytmu, tzn. określenie szybkości wzrostu czasu działania algorytmu, gdy rozmiar danych dąży do nieskończoności

- W notacji asymptotycznej czas działania algorytmów opisywany jest przez funkcje określone na zbiorze liczb naturalnych $\mathcal{N}$
- Argument funkcji jest najczęściej rozmiarem danych wejściowych

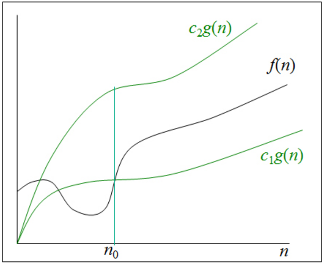
Notacja asymptotyczna

Notacja Θ

Dla danej funkcji $g(n) : \mathcal{N} \to \mathcal{R}$ oznaczamy przez $\Theta(g(n))$ klasę równoważności funkcji:

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2 > 0 \;\; \exists n_0 \in N : c_1 g(n) \leqslant f(n) \leqslant c_2 g(n) \; \forall n \geqslant n_0\}$$

- Formalnie zachodzi $f(n) \in \Theta(g(n))$
- Zazwyczaj piszemy $f(n) = \Theta(g(n))$
- Notacja Θ jest asymptotycznie dokładnym oszacowaniem: ogranicza funkcję od góry i od dołu



Notacja asymptotyczna

Przykład

- Dana jest funkcja

$$f(n) = an^2 + bn + c$$

gdzie $a > 0$, b i c są stałymi

- Odrzucając składniki niższego rzędu otrzymujemy

$$f(n) = \Theta(n^2)$$

- Wynika to z następującego faktu:

$$c_1 = a/4 \quad c_2 = 7a/4 \quad n_0 = 2max(|b|/a, \sqrt{|c|/a})$$

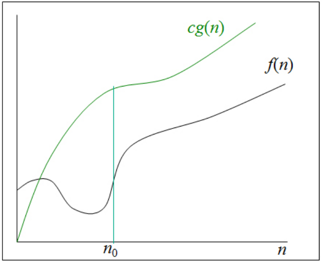
Notacja asymptotyczna

Notacja O

Dla danej funkcji $g(n) : \mathcal{N} \to \mathcal{R}$ oznaczamy przez $\mathrm{O}(g(n))$ klasę równoważności funkcji:

$$\mathrm{O}(g(n)) = \{f(n) : \exists c > 0 \ \ \exists n_0 \in N : 0 \leqslant f(n) \leqslant cg(n) \ \forall n \geqslant n_0\}$$

- Notacja O jest asymptotyczną granicą górną: szacuje pesymistyczny czas działania algorytmu
- $\Theta(g(n)) \subseteq \mathrm{O}(g(n))$



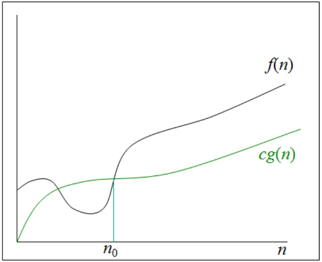
Notacja asymptotyczna

Notacja Ω

Dla danej funkcji $g(n) : \mathcal{N} \to \mathcal{R}$ oznaczamy przez $\Omega(g(n))$ klasę równoważności funkcji:

$$\Omega(g(n)) = \{f(n) : \exists c > 0 \ \ \exists n_0 \in N : 0 \leqslant cg(n) \leqslant f(n) \ \forall n \geqslant n_0\}$$

- Notacja Ω jest asymptotyczną granicą dolną: szacuje optymistyczny czas działania algorytmu
- $\Theta(g(n)) \subseteq \Omega(g(n))$



Notacja asymptotyczna

Własności

Przechodność

$$f(n) = \Theta(g(n)) \quad \wedge \quad g(n) = \Theta(h(n)) \quad \Rightarrow \quad f(n) = \Theta(h(n))$$
$$f(n) = \mathrm{O}(g(n)) \quad \wedge \quad g(n) = \mathrm{O}(h(n)) \quad \Rightarrow \quad f(n) = \mathrm{O}(h(n))$$
$$f(n) = \Omega(g(n)) \quad \wedge \quad g(n) = \Omega(h(n)) \quad \Rightarrow \quad f(n) = \Omega(h(n))$$

Zwrotność

$$f(n) = \Theta(f(n)) \qquad f(n) = \mathrm{O}(f(n)) \qquad f(n) = \Omega(f(n))$$

Symetria

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

Symetria transpozycyjna

$$f(n) = \mathrm{O}(g(n)) \quad \Leftrightarrow \quad g(n) = \Omega(f(n))$$

Notacja asymptotyczna

Twierdzenie

Dla każdych dwóch funkcji $f(n)$ i $g(n)$ zachodzi

$$f(n) = \Theta(g(n)) \quad \Leftrightarrow \quad f(n) = \mathrm{O}(g(n)) \quad \wedge \quad f(n) = \Omega(g(n))$$

Twierdzenie

Dla każdego wielomianu $f(n) \in \mathrm{P}_k$ zachodzi

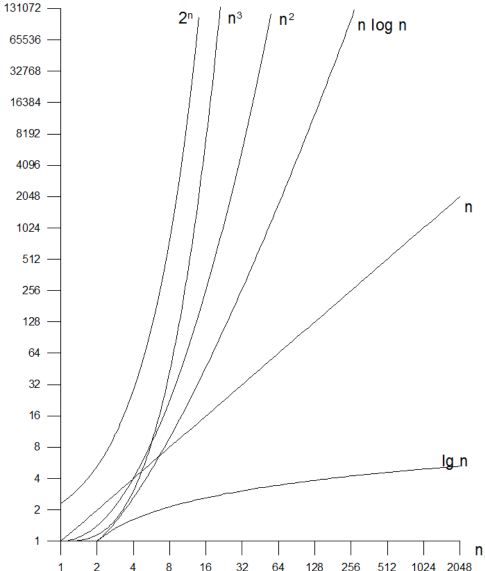
$$f(n) = \Theta(n^k) \qquad f(n) = \mathrm{O}(n^k) \qquad f(n) = \Omega(n^k)$$

Twierdzenie

$$\mathrm{O}(1) \subset \mathrm{O}(\lg n) \subset \mathrm{O}(n) \subset \mathrm{O}(n \lg n) \subset \mathrm{O}(n^2) \subset \mathrm{O}(2^n) \subset \mathrm{O}(n!) \subset \mathrm{O}(n^n)$$

Notacja asymptotyczna

Porównanie rzędów wielkości



Porównanie szybkości wzrostu funkcji

Założenie: operacja dla $n = 1$ wykonuje się w czasie $0.001\mu s$

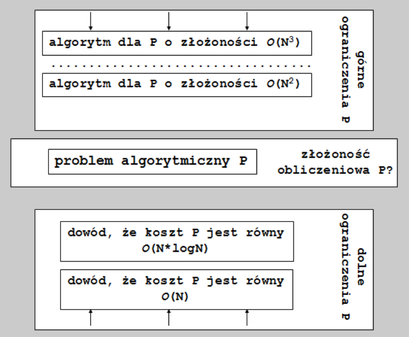
n	$\log n$	n	$n \log n$	n^2	n^3	2^n
10	$0.003\mu s$	$0.01\mu s$	$0.033\mu s$	$0.10\mu s$	$1.0\mu s$	$1.02\mu s$
20	$0.004\mu s$	$0.02\mu s$	$0.086\mu s$	$0.40\mu s$	$8.0\mu s$	$1.048ms$
30	$0.005\mu s$	$0.03\mu s$	$0.147\mu s$	$0.90\mu s$	$27.0\mu s$	$1.07s$
40	$0.005\mu s$	$0.04\mu s$	$0.213\mu s$	$1.60\mu s$	$64.0\mu s$	$18.3min$
50	$0.006\mu s$	$0.05\mu s$	$0.282\mu s$	$2.50\mu s$	$125.0\mu s$	$13.03d$
10^2	$0.007\mu s$	$0.10\mu s$	$0.664\mu s$	$10\mu s$	$1.0ms$	$4 \cdot 10^{13} l$
10^3	$0.010\mu s$	$1.0\mu s$	$9.966\mu s$	$1.0ms$	$1.0s$	
10^4	$0.013\mu s$	$10.0\mu s$	$133\mu s$	$100ms$	$16.7min$	
10^5	$0.017\mu s$	$100.0\mu s$	$1.66ms$	$10s$	$11.6d$	
10^6	$0.020\mu s$	$1.0ms$	$19.93ms$	$16.67min$	$31.7d$	
10^7	$0.023\mu s$	$10ms$	$0.232s$	$1.16d$	$31709l$	
10^8	$0.027\mu s$	$100ms$	$2.66s$	$115.74d$	$3.17 \cdot 10^7 l$	
10^9	$0.030\mu s$	$1s$	$29.9s$	$31.71l$		

Podstawowe złożoności obliczeniowe

$\log n$	algorytmy, w których zadanie o rozmiarze n sprowadzane jest do zadania rozmiaru $n/2$, plus pewna stała liczba działań
n	algorytmy, w których dla każdego z n elementów (danych wejściowych) wykonywana jest stała liczba działań
$n \log n$	algorytmy, w których zadanie o rozmiarze n zostaje sprowadzone do dwóch podzadań rozmiaru $n/2$, plus pewna liczba działań liniowa względem rozmiaru n, potrzebna do wykonania najpierw podzielenia, a następnie scalenia rozwiązań podzadań rozmiaru $n/2$ w rozwiązanie rozmiaru n
n^2	algorytmy, w których jest wykonywana pewna stała liczba działań dla każdej pary danych wejściowych (podwójna iteracja)
n^k	algorytmy o k wzajemnie zagnieżdżonych pętlach
2^n	algorytmy, w których jest wykonywana stała liczba działań dla każdego podzbioru danych wejściowych
$n!$	algorytmy, w których jest wykonywana stała liczba działań dla każdej permutacji danych wejściowych

Dolne i górne ograniczenia

- Znalezienie algorytmu rozwiązania danego problemu ustanawia górne ograniczenie dla tego zadania algorytmicznego
- Jeżeli dolne i górne ograniczenia są sobie równe z dokładnością do stałych, to problem w sensie notacji O jest zamknięty
- Jeżeli dolne i górne ograniczenia są różne, to mówimy o istnieniu luki algorytmicznej



Przykład problemu, który nie jest zamknięty

- Problem minimalnego drzewa rozpinającego
- Udowodniono, że zadanie to wymaga $O(n)$ czasu, gdzie n jest liczbą krawędzi grafu, ale nie ma algorytmu, który realizowałby to zadanie w czasie liniowym

Przykłady obliczania złożoności

Zagnieżdżone pętle

Sortowanie bąbelkowe (bubble sort)

```
bubble_sort(<type> t[N]){
 for(i=0; i < N; ++i)           //(1)
     for(j = N-1; j > i; --j)   //(2)
        if(t[j] < t[j-1])
                swap(t[j], t[j-1]);
}
```

Analiza kosztu czasowego algorytmu

- pętla zewnętrzna (1) wykona się N razy
- pętla wewnętrzna (2) wykona się średnio $(N-1)/2$ razy
- $\rightarrow$ koszt czasowy wykonania algorytmu jest równy

$$T(N) = N \cdot (N-1)/2 \quad \rightarrow \quad T(N) = \mathrm{O}(N^2)$$

Zagnieżdżone pętle

Sortowanie przez wstawianie (insert sort)

```
insert_sort(<type> t[N]){
  for(i=1; i<N; ++i){                    // (1), c1
    x=t[i];                              //      c2
    for(j=i-1; j>=0 && t[j]>x; --j)      // (2), c3
      t[j+1]=t[j];                       //      c4
    t[j+1]=x;                            //      c5
  }
}
```

Analiza kosztu czasowego algorytmu

- pętla zewnętrzna (1) wykona się $N-1$ razy
- pętla wewnętrzna (2) wykona się w i-tej iteracji $t_i \leq i$ razy
- $\rightarrow$ koszt czasowy wykonania algorytmu jest równy

$$T(N) = (N-1)(c_1 + c_2 + c_5) + (c_3 + c_4)\sum_{i=1}^{N-1} t_i$$

Zagnieżdżone pętle

Sortowanie przez wstawianie (insert sort) - cd

Analiza kosztu czasowego algorytmu

- Przypadek optymistyczny - tablica wstępnie posortowana. Dla każdego $i > 0$ zachodzi `t[i-1] <= x` $\rightarrow t_i = 1$, czyli

$$T(N) = (N-1)(c_1 + c_2 + c_5) + (c_3 + c_4)\sum_{i=1}^{N-1} 1$$
$$= (N-1)(c_1 + c_2 + c_5) + (N-1)(c_3 + c_4)$$
$$\rightarrow T(N) = \mathrm{O}(N)$$

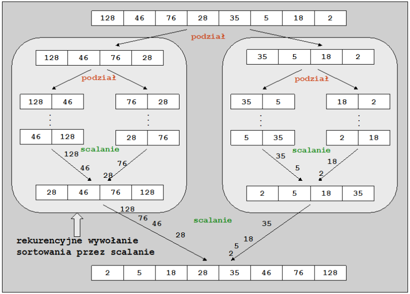
- Przypadek pesymistyczny - tablica posortowana odwrotnie. Dla każdego i zachodzi `t[j] > x` dla j od 1 do $i-1 \rightarrow t_i = i$, czyli

$$T(N) = (N-1)(c_1 + c_2 + c_5) + (c_3 + c_4)\sum_{i=1}^{N-1} i =$$
$$= (N-1)(c_1 + c_2 + c_5) + (c_3 + c_4)N(N-1)/2$$
$$\rightarrow T(N) = O(N^2)$$

Rekurencja

Sortowanie przez scalanie (merge-sort)

```
merge_sort(<type> t[],
            int p,int k){
  if(!(p<k)) return;
  q=(p+k)/2;
  merge_sort(t,p,q);
  merge_sort(t,q+1,k);
  merge(t,p,q,k);
}
```



- Jeżeli rozmiar danych jest wystarczająco mały, $n \leq c$, to $T(n) = \Theta(1)$
- Problem jest podzielony na 2 podproblemy, każdy o rozmiarze $n/2$

$$T(n) = \begin{cases} \Theta(1) & n \leq c \\ 2T(n/2) + \Theta(1) + \Theta(n) & n > c \end{cases}$$

- $D(n) = \Theta(1)$ – czas podziału na podproblemy
- $C(n) = \Theta(n)$ – czas scalania rozwiązań podproblemów w pełne rozwiązanie

Twierdzenie o rekurencji uniwersalnej

Równanie rekurencyjne

$$T(n) = aT(n/b) + f(n) \qquad a \geq 1 \quad b > 1 \quad f(n) > 0$$

- Algorytm dzieli problem rozmiaru n na a podproblemów, każdy o rozmiarze n/b
- Każdy podproblem jest rozwiązywany rekurencyjnie w czasie $T(n/b)$
- Koszt dzielenia problemu oraz łączenia wyników częściowych jest opisany funkcją $f(n)$

Twierdzenie o rekurencji uniwersalnej

$$f(n) = \mathrm{O}(n^{\log_b a - \varepsilon}) \Rightarrow T(n) = \Theta(n^{\log_b a})$$

$$f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$$

$$f(n) = \Omega(n^{\log_b a + \varepsilon}) \wedge af(n/b) \leq cf(n) \Rightarrow T(n) = \Theta(f(n))$$

gdzie $\varepsilon > 0$, $c < 1$, a ostatni warunek zachodzi dla wszystkich dostatecznie dużych n

Twierdzenie o rekurencji uniwersalnej

Interpretacja

Twierdzenie o rekurencji uniwersalnej

$$
\begin{array}{ll}
f(n) = \mathrm{O}(n^{\log_b a - \varepsilon}) & \Rightarrow T(n) = \Theta(n^{\log_b a}) \\
f(n) = \Theta(n^{\log_b a}) & \Rightarrow T(n) = \Theta(n^{\log_b a} \log n) \\
f(n) = \Omega(n^{\log_b a + \varepsilon}) \wedge \ af(n/b) \leq cf(n) & \Rightarrow T(n) = \Theta(f(n))
\end{array}
$$

gdzie $\varepsilon > 0$, $c < 1$, a ostatni warunek zachodzi dla $\forall n > n_0$

- W każdym z trzech przypadków porównujemy $f(n)$ z funkcją $n^{\log_b a}$ — większa funkcja decyduje o złożoności algorytmu rekurencyjnego
- W pierwszym przypadku $f(n)$ musi być wielomianowo mniejsza niż $n^{\log_b a}$
- W trzecim przypadku $f(n)$ musi być wielomianowo większa niż $n^{\log_b a}$ oraz spełniać warunek regularności $af(n/b) \leq cf(n)$
- Jest pewna luka pomiędzy przypadkami 1 i 2 oraz 2 i 3 (funkcje, które nie są wielomianowo mniejsze) — wtedy nie można zastosować twierdzenia o rekurencji uniwersalnej

Merge sort

Złożoność obliczeniowa

- Sortowanie przez scalanie

$$T(n) = \begin{cases} \Theta(1) & n \leq c \\ 2T(n/2) + \Theta(1) + \Theta(n) & n > c \end{cases}$$

- Mamy zatem: $a = b = 2$ oraz $f(n) = cn$, $n^{\log_2 2} = n^1 = n$, czyli $f(n) = \Theta(n) \Rightarrow$ mamy przypadek drugi
- Zatem

$$T(n) = \Theta(n^{\log_2 2} \log n) = \Theta(n \log n)$$

Równania rekurencyjne

- Rozpatrzmy równanie postaci $T(n) = 2T(n/2) + n^2$
- Równanie jest postaci $T(n) = aT(n/b) + f(n)$
  $\Rightarrow$ korzystamy z twierdzenia o rekurencji uniwersalnej

$$a = b = 2 \quad \Rightarrow \quad n^{\log_b a} = n^{\log_2 2} = n$$
$$f(n) = n^2$$

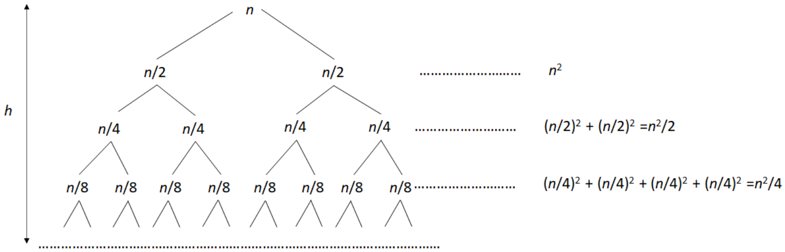
Funkcja $f(n) = n^2$ jest wielomianowo większa od funkcji n:
$n^2 = \Omega(n^{1+\epsilon})$
$\Rightarrow$ 3. przypadek twierdzenia o rekurencji uniwersalnej i sprawdzamy warunek regularności:

$$a f(n/b) \leq c\, f(n)$$
$$2(n/2)^2 \leq c\, n^2 \quad \rightarrow \quad 1/2 \leq c$$

Istnieje więc stała dodatnia $c < 1$ taka, że warunek regularności jest spełniony
Zatem rozwiązaniem równania jest $T(n) = \Theta(n^2)$

Drzewa rekursji

- Rozpatrzmy równanie postaci $T(n) = 2T(n/2) + n^2$
- Złożoność obliczeniową można oszacować wykorzystując drzewa rekursji



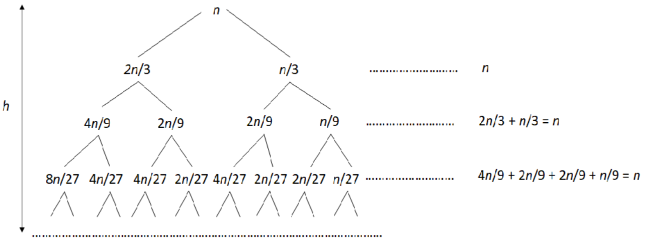
$$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \frac{n^2}{8} + \cdots = n^2\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right) \leq 2n^2$$

$\Rightarrow$

$T(n) = \Theta(n^2)$

Drzewa rekursji

- Rozpatrzmy równanie postaci $T(n) = T(2n/3) + T(n/3) + n$
- Szacujemy złożoność obliczeniową wykorzystując drzewa rekursji



Złożoność jest równa $T(n) = n \cdot h$

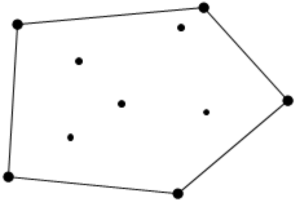
Wysokość drzewa (najdłuższa ścieżka od korzenia do liścia):

$$n \cdot \underbrace{\frac{2}{3} \cdot \frac{2}{3} \cdot \dots}_{h} = 1 \quad \Rightarrow \quad n \cdot \left(\frac{2}{3}\right)^h = 1 \quad \Rightarrow \quad h = \log_{3/2} n$$

$$\Rightarrow T(n) = n \cdot \log_{3/2} n = \Theta(n \lg n)$$

Powłoka wypukła n punktów na płaszczyźnie

- Jedno z podstawowych zadań geometrii obliczeniowej (w grafice komputerowej)
- Uwaga: Niech L będzie odcinkiem łączącym dwa punkty. Odcinek L stanowi część powłoki wypukłej wtw gdy wszystkie pozostałe punkty leżą po tej samej stronie przedłużenia odcinka L do prostej
  - Algorytm I. Dla n punktów na płaszczyźnie weź każdy potencjalny odcinek i sprawdź, czy wszystkie pozostałe $n-2$ punkty leżą po tej samej jego stronie



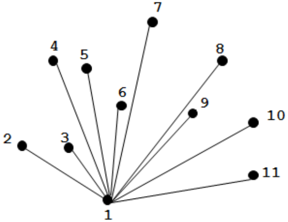
Powłoka wypukła n punktów na płaszczyźnie

cd

- Algorytm II
  1. Znajdź „najniższy” punkt P_1
  2. Posortuj pozostałe punkty wg kąta, jaki tworzą te punkty połączone z P_1 z linią poziomą. Niech $P_2 \ldots P_n$ będzie tak powstałą listą
  3. Zacznij od punktów P_1 i P_2 jako należących do bieżącej powłoki
  4. Powtarzaj dla $i = 3 \ldots n$
     1. Dodaj na próbę punkt P_i do bieżącej powłoki
     2. Przejdź wstecz przez odcinki bieżącej powłoki, usuwając punkty P_j, jeśli dwa punkty P_1 i P_i, znajdują się po przeciwległych stronach prostej przechodzącej przez P_{j-1} i P_j i kończąc proces przechodzenia wstecz w momencie napotkania takiego j, dla którego punktu P_j nie trzeba usuwać

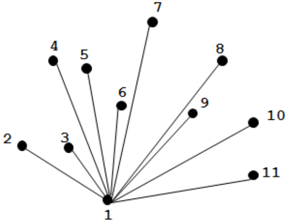
Powłoka wypukła *n* punktów na płaszczyźnie

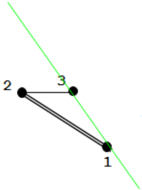
Działanie algorytmu (1)



Powłoka wypukła n punktów na płaszczyźnie

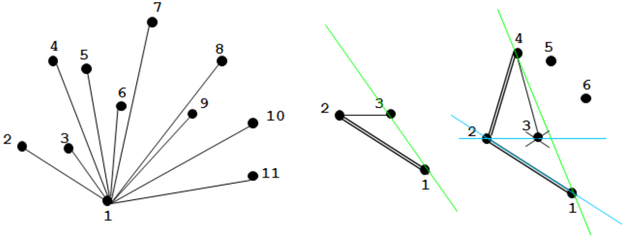
Działanie algorytmu (2)





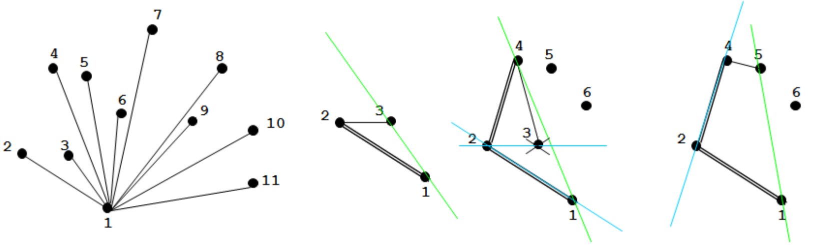
Powłoka wypukła n punktów na płaszczyźnie

Działanie algorytmu (3)



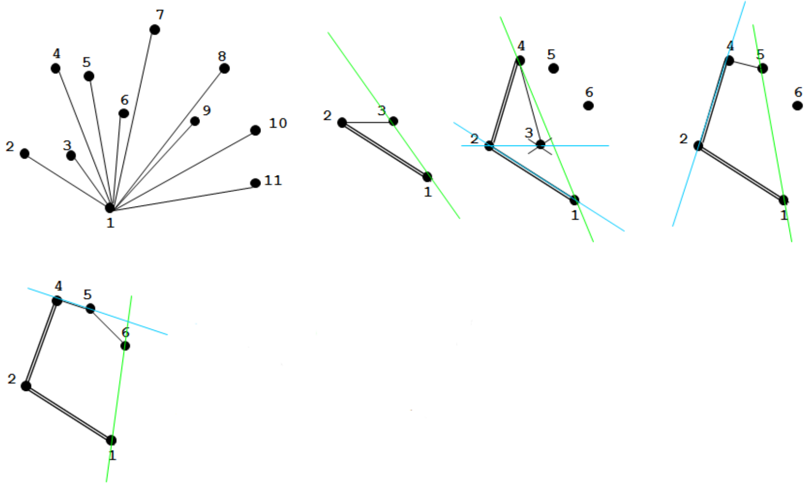
Powłoka wypukła n punktów na płaszczyźnie

Działanie algorytmu (4)



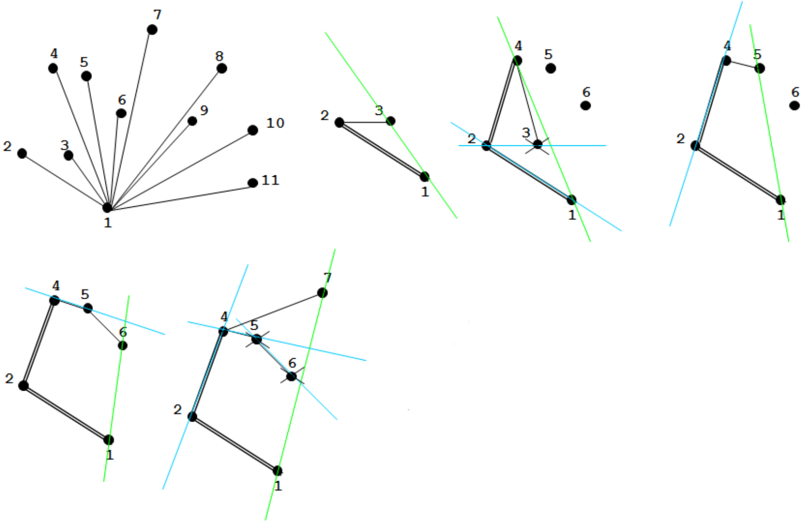
Powłoka wypukła n punktów na płaszczyźnie

Działanie algorytmu (5)



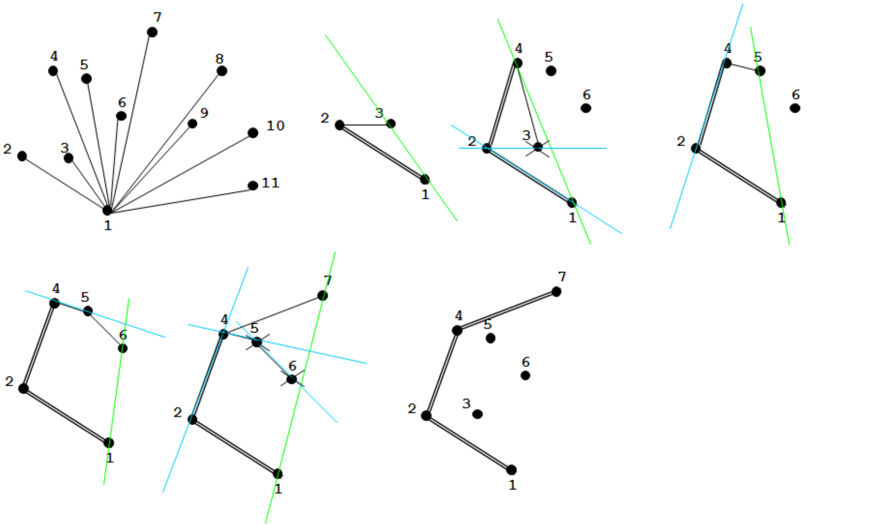
Powłoka wypukła *n* punktów na płaszczyźnie

Działanie algorytmu (6)



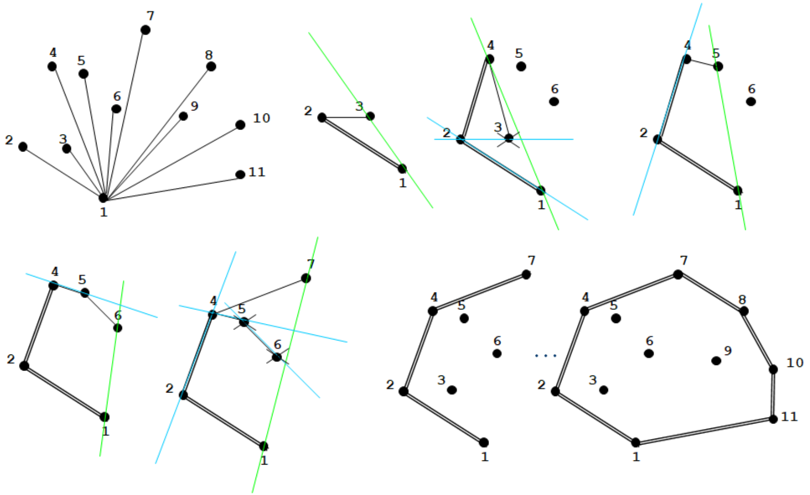
Powłoka wypukła n punktów na płaszczyźnie

Działanie algorytmu (7)



Powłoka wypukła n punktów na płaszczyźnie

Działanie algorytmu (8)



Powłoka wypukła n punktów na płaszczyźnie

Czas działania algorytmów

- Algorytm I
  - Danych jest n punktów, dla których istnieje $n^2/2$ odcinków i z każdym z nich należy sprawdzić $n-2$ punkty
    $\Rightarrow$ całkowity czas działania: $\mathrm{O}(n^3)$
- Algorytm II
  1. Wyszukiwanie — $\mathrm{O}(n)$
  2. Sortowanie — $\mathrm{O}(n \log n)$
  3. $\mathrm{O}(1)$
  4. $\mathrm{O}(n)$ - punkt może być usunięty co najwyżej raz; pętla wewnętrzna zatrzymuje się, gdy napotka punkt, którego nie trzeba usuwać
     $\Rightarrow$ całkowity czas działania: $\mathrm{O}(n \log n)$