

Złożoność algorytmów

- **Złożoność pamięciowa algorytmu**
wynika z liczby i rozmiaru struktur danych wykorzystywanych w algorytmie.
- **Złożoność czasowa algorytmu**
wynika z liczby operacji elementarnych wykonywanych w trakcie przebiegu algorytmu.

ZŁOŻONOŚĆ CZASOWA ALGORYTMU - **zależność** pomiędzy rozmiarem danych wejściowych a liczbą operacji elementarnych wykonywanych w trakcie przebiegu algorytmu (podawana jako funkcja rozmiaru danych, której wartości podają liczbę operacji)

np.

- w alg. sortowania bąbelkowego dla listy o długości N :
 $F(N)$ = liczba porównań par sąsiednich elementów (?)
- w alg. rozwiązywania problemu wież Hanoi dla N krążków:
 $F(N)$ = liczba przeniesień pojedynczego krążka z kołka na kołek (?)
- w alg. wyznaczania najdłuższej przekątnej wielokąta wypukłego o N wierzchołkach:
 $F(N)$ = liczba porównań długości dwóch odcinków-przekątnych (?)
- w alg. wyznaczania „najtańszej sieci kolejowej” dla N węzłów sieci i M możliwych do zbudowania odcinków:
 $F(N, M)$ = liczba porównań kosztów budowy dwóch odcinków (?)

W praktyce złożoność czasowa decyduje o przydatności algorytmów

Ponieważ

- trzeba rozwiązywać algorytmicznie coraz większe zadania:
 - w komputerowych systemach wspomagania decyzji,
 - przy komputerowych symulacjach i prognozach złożonych zjawisk.
- rozwijane są komputerowe systemy czasu rzeczywistego:
 - sterujące automatycznie złożonymi układami (transport, produkcja)

Chcemy zmniejszać czas wykonania algorytmu!

Przykład 1

Normalizacja wektora (tablicy jednowymiarowej) względem wartości maksymalnej; dane wejściowe zapisane w $V(1), V(2), \dots, V(N)$

Algorytm 1

- wyznacz w zmiennej MAX największą z wartości ;
- dla I od 1 do N wykonuj :**
 - $V(I) \leftarrow V(I) \cdot 100 / MAX$

Algorytm 2

- wyznacz w zmiennej MAX największą z wartości ;
- $ILORAZ \leftarrow 100 / MAX$;
- dla I od 1 do N wykonuj :**
 - $V(I) \leftarrow V(I) \cdot ILORAZ$

Jeśli operacją elementarną jest wyznaczenie wartości iloczynu lub ilorazu dwóch zmiennych,
to $F_1(N) = 2 \cdot N$ i $F_2(N) = N + 1$

Przykład 2

Wyszukiwanie liniowe elementu z listy o długości N ; dane wejściowe to lista i element do wyszukania:

Algorytm 1

- weź pierwszy element listy ;
 - wykonuj:**
 - sprawdź czy bieżący element jest tym szukanym ;
 - sprawdź czy osiągnąłeś koniec listy ;
 - weź następny element z listy
- aż** znajdziesz szukany element lub przejrzysz całą listę

Algorytm 2

1. dopisz szukany element na końcu listy ;
2. weź pierwszy element listy ;
3. **wykonuj:**
 - 3.1. sprawdź czy bieżący element jest tym szukany ;
 - 3.2. weź następny element z listy**aż** znajdziesz ;
4. sprawdź czy jesteś na końcu listy

Jeśli operacją elementarną jest sprawdź , to $F_1(N) = 2 \cdot N$ i $F_2(N) = N + 1$

Poprawianie złożoności algorytmu jest czymś więcej niż tylko zmniejszaniem czasu jego wykonania!

Jeżeli np. porównujemy dwa algorytmy wykonujące to samo zadanie za pomocą jednej pętli ograniczonej, w której liczba iteracji N jest wprost proporcjonalna do rozmiaru danych wejściowych, to liczby operacji elementarnych (czasy wykonania) tych algorytmów w funkcji rozmiaru danych wynoszą odpowiednio:

$$F_1(N) = K_1 + L_1 \cdot N$$

$$F_2(N) = K_2 + L_2 \cdot N$$

do ich porównania możemy wykorzystać iloraz $s(N) = \frac{F_1(N)}{F_2(N)}$

i wtedy spełnienie warunków:

$s(N) = 1$ oznaczałoby jednakową szybkość działania

$s(N) < 1$ oznaczałoby, że 1. algorytm jest szybszy.

Ale zauważmy, że $s(N) = 1$ zachodzi tylko wtedy, kiedy $K_1 = K_2$ i $L_1 = L_2$, co oznacza, że oba algorytmy są praktycznie identyczne.

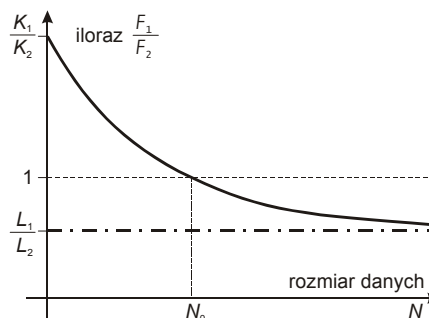
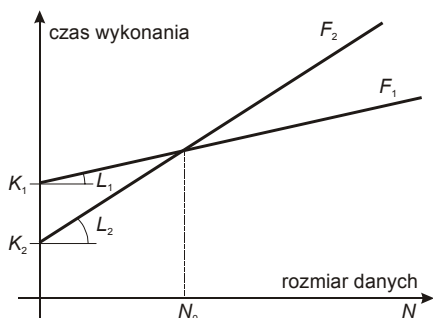
A co mamy powiedzieć, kiedy $s(N) \geq 1$ dla $N \leq N_0$, ale $s(N) < 1$ dla $N > N_0$.

Który algorytm jest lepszy?

Przyjęto, że porównując algorytmy badamy wartość ilorazu $s(N)$ dla bardzo dużych wartości N , czyli formalnie dla $N \rightarrow \infty$.

Zatem o wyniku porównania czasów działania dwóch algorytmów decyduje wartość: $\lim_{N \rightarrow \infty} s(N) = \frac{L_1}{L_2}$; jest to

tzw. *analiza asymptotyczna*.



Dwa algorytmy o czasach wykonania $F_1(N)$ i $F_2(N)$ mają **złożoność tego samego rzędu**, jeśli $\lim_{N \rightarrow \infty} \frac{F_1(N)}{F_2(N)} = C$,
gdzie $0 < C < \infty$

Spełnienie takiego warunku oznaczamy

$F_1(N) = \Theta(F_2(N))$

Jeśli zachodzi $F_1(N) = \Theta(F_2(N))$, to także $F_2(N) = \Theta(F_1(N))$

Algorytm o czasie wykonania $F_1(N)$ ma **nie wyższy rząd złożoności** od algorytmu o czasie wykonania $F_2(N)$,

jeśli $\lim_{N \rightarrow \infty} \frac{F_1(N)}{F_2(N)} = C < \infty$

Spełnienie takiego warunku oznaczamy

$F_1(N) = O(F_2(N))$

Jeśli zachodzi jednocześnie $F_1(N) = O(F_2(N))$ i $F_2(N) = O(F_1(N))$, to $F_1(N) = \Theta(F_2(N))$.
 Jeśli zachodzi $F_1(N) = \Theta(F_2(N))$, to także $F_1(N) = O(F_2(N))$.

- jeśli $C = 0$, to algorytm o czasie wykonania $F_1(N)$ ma niższy rząd złożoności (ma lepszą złożoność) od algorytmu o czasie wyk. $F_2(N)$; oznaczamy to symbolicznie $F_1(N) \prec F_2(N)$
- algorytm o czasie wykonania $F(N)$ ma **złożoność liniową**, jeśli $\lim_{N \rightarrow \infty} \frac{F(N)}{N} = C$, dla $0 < C < \infty$;
 złożoność rzędu N oznaczana jest symbolem $\Theta(N)$
 (ma on złożoność co najwyżej liniową, jeśli $C < \infty$; ozn. $O(N)$).
- algorytm o czasie wykonania $F(N)$ ma **złożoność kwadratową**, jeśli $\lim_{N \rightarrow \infty} \frac{F(N)}{N^2} = C$, dla $0 < C < \infty$;
 złożoność rzędu N^2 oznaczana jest symbolem $\Theta(N^2)$
 (ma on złożoność co najwyżej kwadratową, jeśli $C < \infty$; ozn. $O(N^2)$;
 zauważmy jeszcze, że $N \prec N^2$).
- sytuację, w której zachodzi warunek $\forall N: 0 \leq F(N) \leq C < \infty$ oznaczamy $F(N) = O(1)$.

Dopiero zmniejszenie rzędu złożoności algorytmu jest istotnym ulepszeniem rozwiązania problemu algorytmicznego!

Jeżeli algorytm wykonuje różną liczbę operacji elementarnych w zależności od konkretnych danych wejściowych (o tym samym rozmiarze) możemy badać czas wykonania **w najgorszym przypadku** (czyli skupiając się na takich przypadkach dopuszczalnych danych wejściowych, dla których ta liczba jest największa) - tzw. *analiza najgorszego przypadku* lub *pesymistyczna*.

W przykładach normalizacji wektora i wyszukiwania liniowego

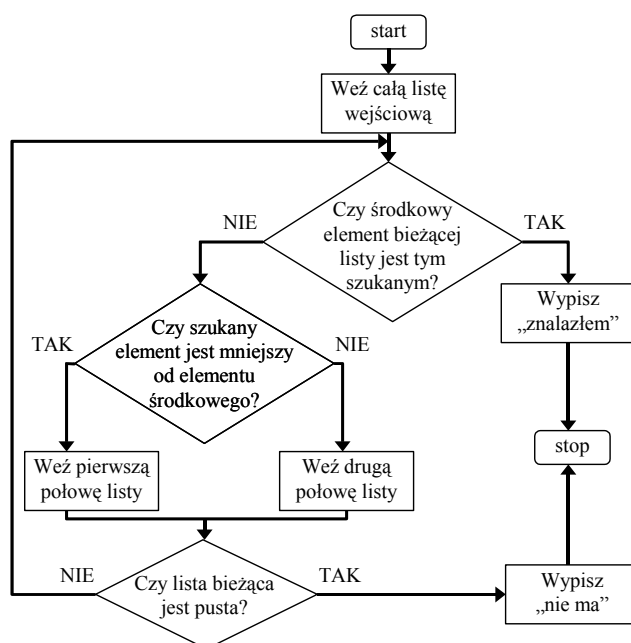
- ◇ czas wykonania Algorytmu 1. wynosi $F_1(N) = 2 \cdot N : F_1(N) = O(N)$
- ◇ czas wykonania Algorytmu 2. wynosi $F_2(N) = N + 1 : F_2(N) = O(N)$
- ◇ oba mają pesymistyczny czas wykonania $O(N)$ - mogą zdarzyć się takie dane wejściowe dla wyszukiwania liniowego, dla których trzeba będzie przejrzeć całą listę o długości N (jest tak, kiedy szukanego elementu nie ma w ogóle na liście)

Czy można zaproponować algorytm o lepszej złożoności dla wyszukiwania elementu na liście?

Szukamy zatem algorytmu o (pesymistycznym) czasie wykonania $F(N) \prec N$.

Wyszukiwanie **binarne** (przez połowienie) elementu z listy uporządkowanej:

Y_1, Y_2, \dots, Y_N (dla każdego $i < j$ zachodzi $Y_i \leq Y_j$)



Jeśli przyjmiemy, że operacją elementarną jest porównanie elementu szukanego z jednym z elementów listy (środkowym), to analiza złożoności polega na znalezieniu odpowiedzi na pytanie: *ile razy jest powtarzana w najgorszym przypadku iteracja w algorytmie?*

Odpowiedź: $1 + \log_2 N$ ($\log_2 N$ będziemy oznaczali $\lg N$)

Zatem pesymistyczna złożoność algorytmu wyszukiwania binarnego wynosi $O(\lg N)$

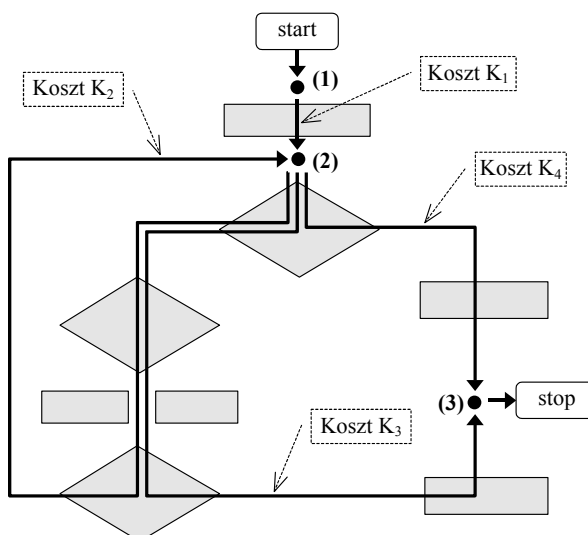
(mówimy, że algorytm ten ma złożoność logarytmiczną);

$$\lg N \prec N$$

Skala ulepszenia:

| N | $1 + \lfloor \lg N \rfloor$ |
|---------------------------|-----------------------------|
| 10 | 4 |
| 100 | 7 |
| 1 000 | 10 |
| 10 000 | 14 |
| 1 000 000 | 20 |
| 1 000 000 000 | 30 |
| 1 000 000 000 000 | 40 |
| 1 000 000 000 000 000 | 50 |
| 1 000 000 000 000 000 000 | 60 |

O złożoności czasowej algorytmu decyduje liczba wykonywanych iteracji



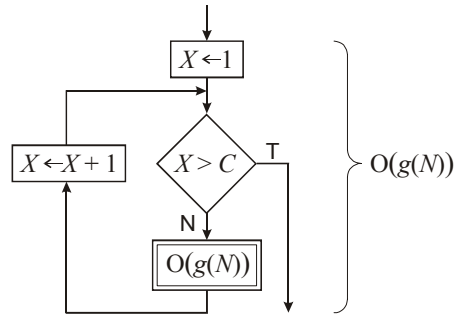
Całkowity koszt czasowy w najgorszym przypadku:

$$F(N) = K_1 + \max(K_3, K_4) + K_2 \cdot \lg N, \text{ a to oznacza } F(N) = O(\lg N)$$

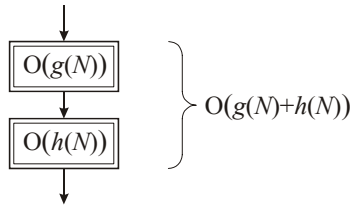
Właściwości notacji $O(\cdot)$ (tzw. rachunek $O(\cdot)$)

- jeśli $F(N)$ jest funkcją złożoności algorytmu, to spełnienie warunku $\lim_{N \rightarrow \infty} \frac{F(N)}{g(N)} = C < \infty$, jest zapisywane $F(N) = O(g(N))$; odczytujemy „algorytm ma złożoność rzędu nie wyższego niż $g(N)$ ” lub krócej „złożoność algorytmu jest $O(g(N))$ ”
- równość w zapisie $F(N) = O(g(N))$ powinna być rozumiana w ten sposób, że funkcja $F(N)$ jest jedną z funkcji, które spełniają powyższy warunek lub precyzyjniej, że funkcja $F(N)$ należy do zbioru wszystkich funkcji spełniających powyższy warunek
- $F(N) = O(F(N))$,
- $O(O(g(N))) = O(g(N))$,

- $C \cdot O(g(N)) = O(C \cdot g(N)) = O(g(N))$ (dla dowolnego $0 < C < \infty$),

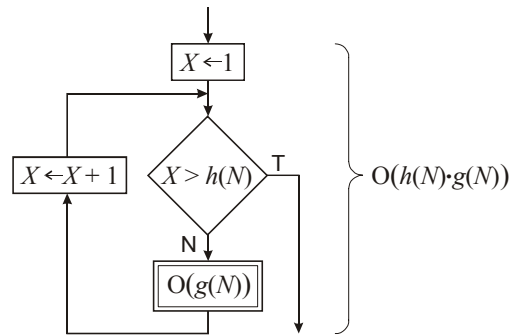
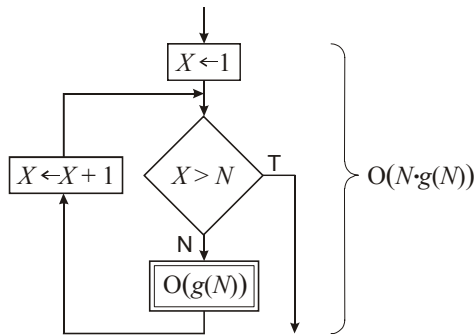


- $O(g(N)) + O(h(N)) = O(g(N) + h(N))$,

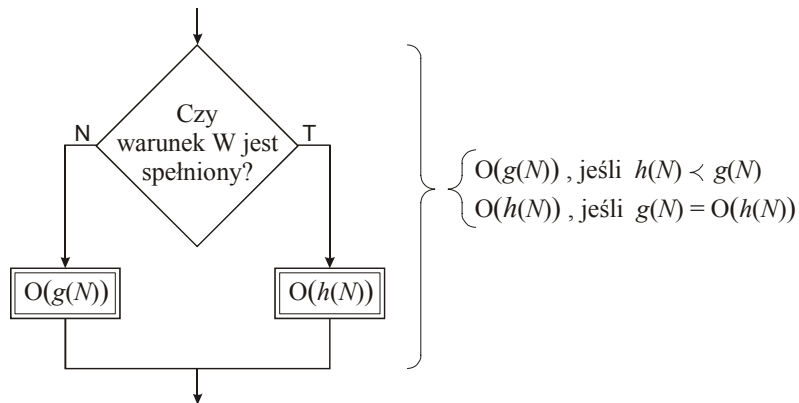


- jeśli zachodzi $\lim_{N \rightarrow \infty} \frac{g(N)}{h(N)} = 0$, czyli $g(N) \prec h(N)$,
to $O(g(N)) + O(h(N)) = O(g(N) + h(N)) = O(h(N))$

- $O(g(N)) \cdot O(h(N)) = O(g(N) \cdot h(N)) = g(N) \cdot O(h(N)) = h(N) \cdot O(g(N))$,



- przy analizie złożoności w najgorszym przypadku (dla warunku zależnego od danych wejściowych) zachodzi:



Złożoność czasowa przykładowych algorytmów

- sortowanie bąbelkowe w pierwotnej wersji (zagnieżdżone iteracje):
 1. wykonuj co następuje $N - 1$ razy:
 - 1.1. ... ;
 - 1.2. wykonuj co następuje $N - 1$ razy:
 - 1.2.1. ... ;

Całkowity koszt czasowy w najgorszym przypadku wynosi z dokładnością do stałych:
 $(N - 1) \cdot (N - 1) = N^2 - 2N + 1$; $1 \prec 2N \prec N^2$, czyli złożoność jest $O(N^2)$

- sortowanie bąbelkowe ulepszone (coraz krótsze przebiegi):
 Całkowity koszt czasowy w najgorszym przypadku wynosi:
 $(N - 1) + (N - 2) + (N - 3) + \dots + 2 + 1 = 0,5 \cdot N^2 - 0,5 \cdot N$, czyli także złożoność jest $O(N^2)$
- sumowanie zarobków pracowników - złożoność $O(N)$
- znajdowanie największej przekątnej w wielokącie wypukłym metodą naiwną - złożoność $O(N^2)$
- znajdowanie największej przekątnej w wielokącie wypukłym metodą jednokrotnego obiegu z parą prostych równoległych - złożoność $O(N)$
- rekurencyjny algorytm dla problemu wież Hanoi

*procedura **przenieś** N ;*

1. jeśli $N = 1$, to wypisz ruch i koniec;
2. w przeciwnym razie (tj. jeśli $N > 1$) wykonaj co następuje:
 - 2.1. wywołaj **przenieś** $N - 1$;
 - 2.2. wypisz ruch ;
 - 2.3. wywołaj **przenieś** $N - 1$;

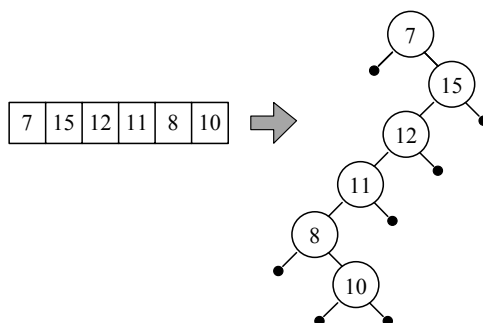
Oznaczmy nieznaną koszt czasowy przez $T(N)$ i ułożymy równania, które $T(N)$ musi spełniać (tzw. **równania rekurencyjne**):

$$T(1) = 1$$

$$T(N) = 2 \cdot T(N - 1) + 1$$

Spełnia je koszt $T(N) = 2^N - 1$, czyli złożoność jest $O(2^N)$

- sortowanie drzewiaste bez samoorganizacji drzewa:
 ma pesymistyczną (w najgorszym przypadku) złożoność $O(N^2)$, bo wprowadzanie lewostronne obejście drzewa ma złożoność liniową $O(N)$, ale konstrukcja binarnego drzewa poszukiwań ma pesymistyczną złożoność kwadratową



- sortowanie drzewiaste z samoorganizacją drzewa:
 ma złożoność $O(N \cdot \lg N)$, co daje wyraźną poprawę sprawności algorytmu sortowania w porównaniu z algorytmem bąbelkowym

| N | N^2 | $N \cdot \lg N$ |
|---------------|---------------------------|-----------------|
| 10 | 100 | 33 |
| 100 | 10 000 | 664 |
| 1 000 | 1 000 000 | 9 965 |
| 1 000 000 | 1 000 000 000 000 | 19 931 568 |
| 1 000 000 000 | 1 000 000 000 000 000 000 | 29 897 352 853 |

- sortowanie przez scalanie (rekurencyjne):

*procedura **sortuj-listę** L ;*

1. jeśli L zawiera tylko jeden element, to jest posortowana;
2. w przeciwnym razie wykonaj co następuje:
 - 2.1. ... ;
 - 2.2. wywołaj **sortuj-listę połowa** L ;
 - 2.3. wywołaj **sortuj-listę połowa** L ;
 - 2.4. scal posortowane połowy listy L w jedną posortowaną listę;

Oznaczmy nieznaną koszt czasowy przez $T(N)$ i ułożmy równania rekurencyjne, które $T(N)$ musi spełniać:

$$T(1) = 0$$

$$T(N) = 2 \cdot T(0,5 \cdot N) + N$$

Spełnia je koszt $T(N) = N \cdot \lg N$, czyli złożoność jest $O(N \cdot \lg N)$

Sortowanie przez scalanie jest jednym z najlepszych algorytmów sortowania (choć ma nie najlepszą złożoność pamięciową $O(N)$)

Średnia złożoność

Analiza najgorszego przypadku **lub** analiza średniego przypadku

W analizie średniego przypadku istotną rolę odgrywiają założenia o rozkładzie prawdopodobieństwa w zbiorze dopuszczalnych danych wejściowych.

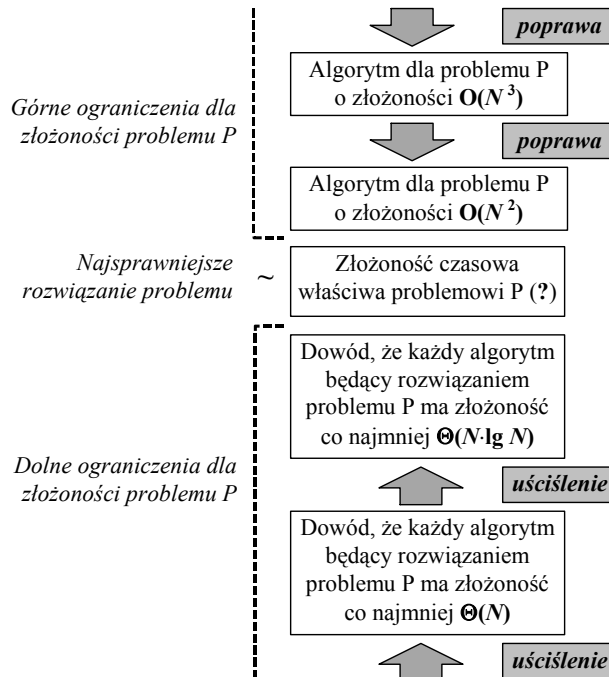
| algorytm | średni przyp. | najgorszy przyp. |
|--|--------------------|--------------------|
| sumowanie zarobków | $O(N)$ | $O(N)$ |
| sortowanie bąbelkowe | $O(N^2)$ | $O(N^2)$ |
| sortowanie drzewiaste z samoorganizacją drzewa | $O(N \cdot \lg N)$ | $O(N \cdot \lg N)$ |
| sortowanie przez scalanie | $O(N \cdot \lg N)$ | $O(N \cdot \lg N)$ |
| Quicksort | $O(N \cdot \lg N)$ | $O(N^2)$ |

Średni koszt czasowy algorytmu Quicksort wynosi tylko $1,4 \cdot N \cdot \lg N$

Dolne i górne ograniczenia złożoności problemów algorytmicznych

Czy można skonstruować jeszcze lepszy algorytm?

- złożoność poprawnego algorytmu znajdującego rozwiązanie danego problemu ustanawia **górne ograniczenie** złożoności dla tego problemu.
- **dolne ograniczenie** złożoności problemu (otrzymane w wyniku analizy samego problemu) określa zakres dalszej poprawy rzędu złożoności algorytmów rozwiązujących ten problem



Problemy zamknięte i luki algorytmiczne

| | | |
|---------|------------|------------|
| problem | dolne ogr. | górne ogr. |
|---------|------------|------------|

| | | |
|--|-------------------------|------------------------|
| przeszukiwanie listy nieuporządkowanej | $\Theta(N)$ | $O(N)$ |
| przeszukiwanie listy uporządkowanej | $\Theta(\lg N)$ | $O(\lg N)$ |
| sortowanie | $\Theta(N \cdot \lg N)$ | $O(N \cdot \lg N)$ |
| wyznaczanie najtańszej sieci kolejowej | $\Theta(N)$ | $O(f(N) \cdot N)^{1)}$ |

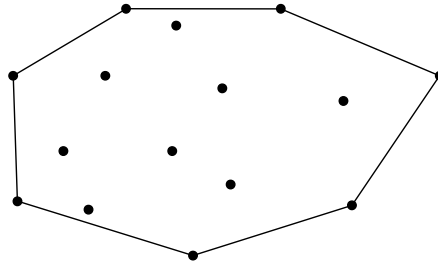
¹⁾ $f(N)$ - bardzo wolno rosnąca funkcja, np. dla $N = 64\ 000$ ma wartość 4

- problem zamknięty

- problem z luką algorytm.

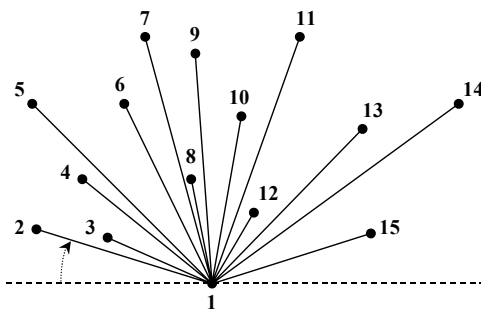
Przykład analizy złożoności algorytmu

Problem wyznaczania powłoki wypukłej dla zbioru N punktów na płaszczyźnie:

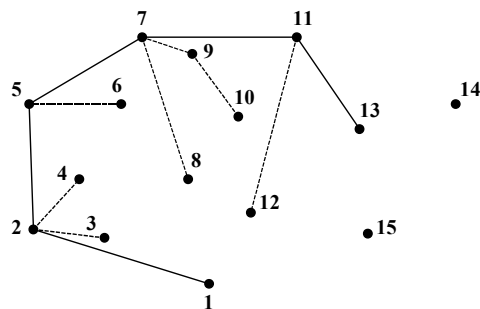


- algorytm „naiwny” ma złożoność $O(N^3)$; trzeba dla każdego z N punktów dobrać pozostałe $N-1$ do pary, która wyznacza prostą i dla każdej z tych prostych sprawdzić czy $N-2$ punkty leżą wszystkie po tej samej stronie prostej.
- algorytm o niższej złożoności (?):
 1. znajdź punkt „najniżej” położony - P_1 ;
 2. posortuj pozostałe punkty rosnąco według kątów tworzonych przez odcinki $\overline{P_1 P_j}$ z linią poziomą przechodzącą przez P_1 - powstanie lista P_2, \dots, P_N ;
 3. dołącz do powłoki punkty P_1 i P_2 ;
 4. **dla J od 3 do N wykonuj** ;
 - 4.1. dołącz do powłoki punkt P_J ;
 - 4.2. cofając się wstecz po odcinkach aktualnej powłoki, usuwaj z niej te punkty P_K , dla których prosta przechodząca przez P_K i P_{K-1} przecina odcinek $\overline{P_1 P_J}$, aż do napotkania pierwszego punktu nie dającego się usunąć ;

krok 2.



kolejne iteracje w kroku 4.



Podsumowanie złożoności algorytmu krok po kroku:

| | | |
|---------|---|--------------------------|
| Krok 1. | $O(N)$ | (wybór pierwszego) |
| Krok 2. | $O(N \cdot \lg N)$ | (sortowanie) |
| Krok 3. | $O(1)$ | (dołączenie 1 odcinka) |
| Krok 4. | $O(N)$ | (dołączanie z usuwaniem) |
| Razem | $O(N + N \cdot \lg N + 1 + N) = O(N \cdot \lg N)$ | |