Algorytm i złożoność obliczeniowa algorytmu

Złożoność obliczeniowa to ilość zasobów niezbędna do wykonania algorytmu. Wyróżnia się dwa rodzaje:

- Złożoność czasową (obliczeniową)
- Złożoność pamięciową

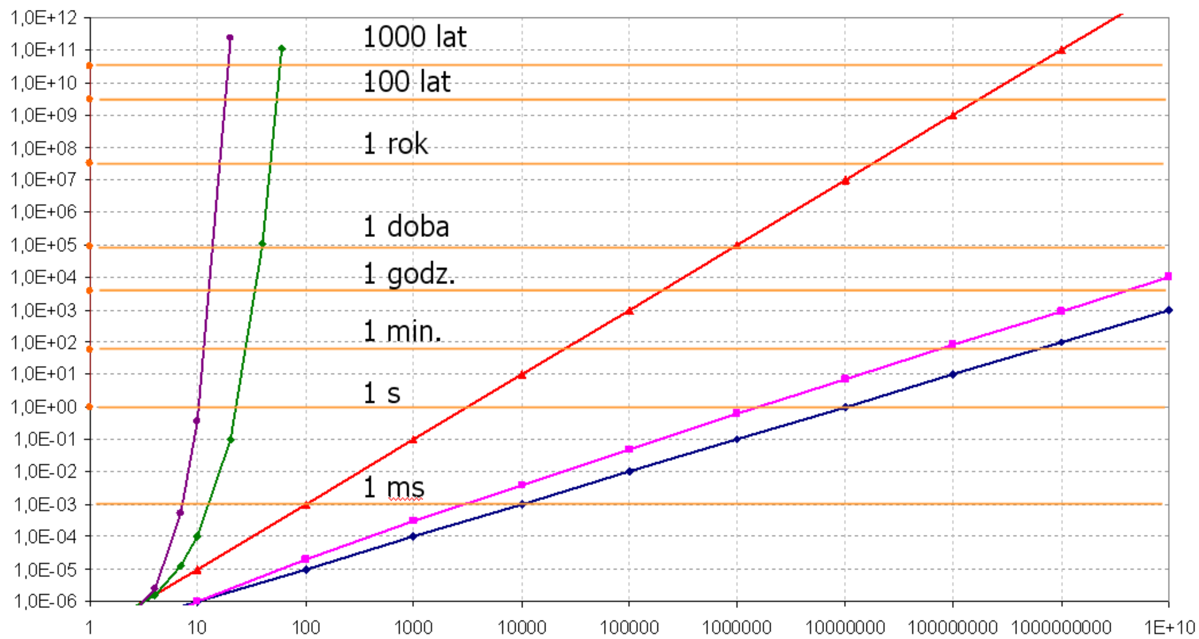
Złożoność zależy od rozmiaru danych wejściowych oraz od algorytmu ich przetwarzania

Istnieje wiele miar złożoności obliczeniowej i wiele notacji. W większości miar najważniejsze jest asymptotyczne tempo wzrostu (tj. jak złożoność rośnie ze wzrostem rozmiaru danych). Do porównywania algorytmów najczęściej używa się notacji O (dużego O), która podaje **asymptotyczne ograniczenie górne**. W notacji O uwzględnia się tylko funkcję dominującą i pomija się współczynniki liczbowe, ponieważ dla odpowiednio dużych zbiorów danych współczynniki stałe mają mniej istotne znaczenie niż rodzaj funkcji dominującej, np.

- f(n) = 100 n2 + 50n   => O(n2)
- f(n) = 1000 n   => O(n)

Przykładowe złożoności algorytmów, w porządku rosnącym:

- *O(1)* – złożoność stała,
- *O(logn)* – złożoność logarytmiczna,
- *O(n)* – złożoność liniowa,
- *O(nlogn)* – złożoność liniowo-logarytmiczna,
- *O(n2)* – złożoność kwadratowa,
- *O(nk)* – złożoność wielomianowa (*k* jest stałą),
- *O(kn)* – złożoność wykładnicza (*k* jest stałą),
- *O(n!)* – złożoność rzędu silnia



Sortowanie

Istnieje wiele algorytmów sortowania; niektóre mają bardzo specyficzne lub ograniczone zastosowanie (np. stosowane głównie w systemach wbudowanych albo kiedy możliwość ich użycia zależy od statystycznych cech sortowanego zbioru), inne są uniwersalne.
Pierwsze algorytmy sortowania pojawiły się w latach 50 ub. wieku (m.in. bublesort 1956, quicksort 1959), ale ciągle są opracowywane nowe użyteczne algorytmy, które znajdują zastosowania (np. timsort, opracowany w 2002 r., jako hybrydowy algorytm merge sort oraz insertion sort, zastosowany m.in. jako wbudowany algorytm sortowania w językach Python i GNU Octave, oraz na platformach Java SE i Android). Duże zainteresowanie algorytmami sortowania wynika z powszechności zastosowań, dużej złożoności obliczeniowej (oraz potrzeby jej zmniejszenia, ze względu na powszechne i częste zastosowania)
Algorytmy sortowania są też wdzięcznym tematem nauki programowania (algorytmy jako takie, ale też np. złożoność obl.)

Właściwości i kryteria oceny algorytmów sortowania

- **Złożoność obliczeniowa**
  Nie zawsze można ją określić jednoznacznie, ponieważ w wielu algorytmach zależy od rozkładu elementów – dlatego wyznacza się złożoność optymistyczną (best), średnią i pesymistyczną (worst);
- **Złożoność pamięciowa**
  Algorytmy sortowania "w miejscu" ("in-place") zamieniają elementy miejscami i nie potrzebują dodatkowej pamięci na same elementy, jednak czasami zużywają inne zasoby;
- **Rekursja**
  Użycie rekursji zużywa zasoby (stos systemowy), co może ograniczać zastosowania algorytmów – np. quicksort i syst. wbudowane;
  Istnieją algorytmy rekusywne, nierekursywne oraz mieszane
- **Stabilność**
  Algorytm sortowania jest stabilny, jeżeli "równe" elementy po sortowaniu zachowują swój porządek sprzed sortowania;
- **Mechanizm sortowania** (sposób działania)
  Wyróżnia się algorytmy sortowania przez porównanie ("comparison sort") oraz takie, które porównania nie używają ("integer sort");
- **Metoda sortowania**
  Wyróżnia się kilka ogólnych metod: sortowanie przez wstawianie, zamiany, wybór, łączenie – konkretne algorytmy należą do jednej z metod ogólnych, np. bublesort jest sortowaniem przez zamiany
- **Adaptacyjność**
  Zdolność do skrócenia czasu sortowania, jeżeli zbiór jest częściowo posortowany (ma część elementów we właściwej kolejności)
  – np. bublesort ma śrenią złożoność O(n2) oraz optymistyczną O(n), natomiast insertion sort zawsze O(n2)
- **Złożoność implementacji**
  Niektóre hybrydowe algorytmy sortowania mają bardzo złożone algorytmy, a ich implementacja może być kłopotliwa dla początkującego programisty; inne, jak bublesort, są dziecinnie proste

Sortowanie bąbelkowe

Sortowanie bąbelkowe jest chyba najprostszym pojęciowo oraz najłatwiejszym w implementacji algorytmem sortowania.
Polega na porównywaniu sąsiednich elementów i w razie potrzeby ich zamianie; Przy N elementach trzeba N-1 porównań, a cały proces trzeba powtórzyć co najwyżej N-1 razy

```
FOR n=0, 1, …, N-2
  FOR i=0, 1, …, N-2
    IF t_i > t_i+1
      t_i ↔ t_i+1
```

Udoskonalenia
(1) Algorytm można zakończyć, jeżeli wewnętrzna pętla nie wykona ani jednej zamiany (wtedy zewnętrzna pętla może być nieskończona)

```
FOR ∞
  Z = false
  FOR i=0, 1, …, N-2
    IF t_i > t_i+1
      t_i ↔ t_i+1
      Z = true
  IF NOT(Z) BREAK
```

(2) Po pierwszym wykonaniu pętli wewnętrznej element największy znajdzie się na samym końcu ("wygra" wszystkie porównania) → liczba porównań pętli wewnętrznej może być zmniejszana; Można połączyć z (1)

```
FOR n=0, 1, …, N-2
  FOR i=0, 1, …, N-n-2
    IF t_i > t_i+1
      t_i ↔ t_i+1
```

Alternatywna wersja

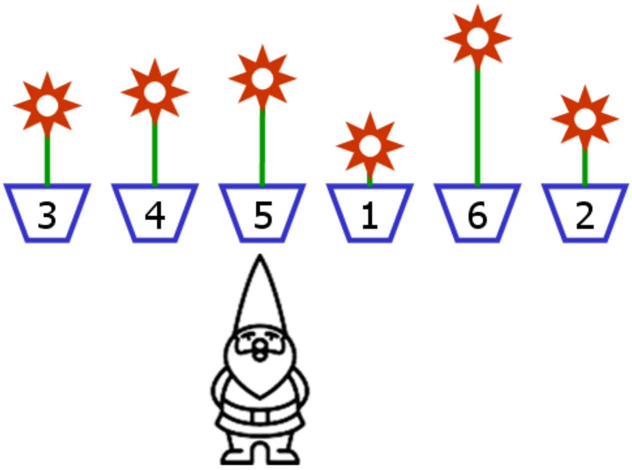
```
FOR n=N-2, N-1, …, 2
  FOR i=0, 1, …, n
    IF t_i > t_i+1
      t_i ↔ t_i+1
```

(3) Jeżeli zbiór będzie prawie posortowany, z wyjątkiem ostatniego elementu (np. { 2, 3, 4, 5, 6, 1 }), to algorytm będzie miał złożoność O(n2), chociaż mógłby mieć O(n) gdyby odwrócić kierunek sortowania – aby takiemu efektowi zapobiec, wewnętrzna pętla powinna na zmianę biec w górę i w dół zbioru, po coraz krótszym jego fragmencie

```
FOR ∞
  l = 1
  r = N-2
  FOR i = l, l+1, …, r
    IF t_i > t_i+1 => t_i ↔ t_i+1
  r ← r-1
  FOR i = r, r-1, …, l
    IF t_i > t_i+1 => t_i ↔ t_i+1
  l ← l+1
```

Sortowanie gnoma

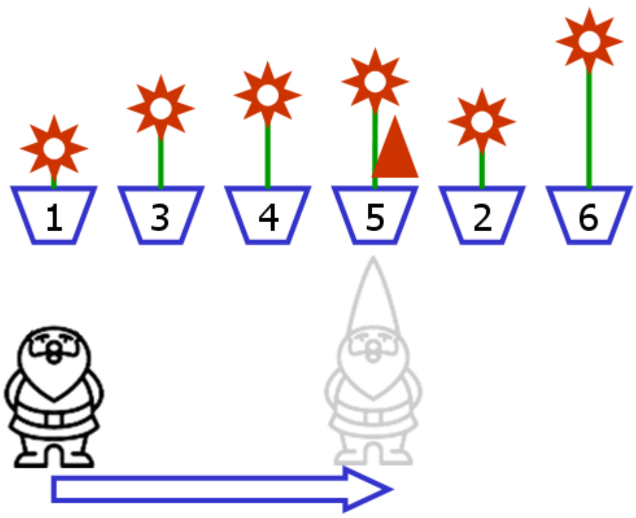
Ciekawy wariant sortowania bąbelkowego, przedstawiany w formie dykteryjki: gnom (niezbyt inteligentny) ma w ogrodzie poukładać doniczki z kwiatami wg wysokości kwiatów; zaczyna od jednej strony (np. od lewej) i porównuje kwiat, przy którym stoi z następnym. Jeżeli są we właściwej kolejności, to przechodzi dalej, jeżeli nie, to zamienia je miejscami i cofa się (chyba że już jest na samym początku)




```
p = 0
WHILE p<N-1
  IF t_p > t_p+1
    t_p ↔ t_p+1
    IF p>0 p ← p-1
  ELSE
    p ← p+1
```

Ulepszenie

Wadą algorytmu jest jałowe powtarzanie porównań podczas poruszania się w prawo, po cofnięciu się w lewo. Wiadomo, że od bieżącego miejsca aż do miejsca, gdzie dotarł poprzednio, kwiaty są posortowane => powinien to miejsce zaznaczyć (powiesić tam swoją czapkę?) i od razu tam wrócić



Sortowanie przez wstawianie

Przypomina układanie kart w ręku gracza: dzielimy elementy zbioru na część posortowaną (z lewej) i nieposortowaną (z prawej); należy pobrać pierwszy element z części nieposortowanej i – przesuwając się w lewo – znaleźć miejsce, gdzie powinien się znaleźć. Elementy posortowane, od tego miejsca aż do części nieposortowanej, należy przesunąć w prawo, żeby zrobić miejsce dla elementu wstawianego

```
FOR n = 1, 2, …, N-1
  temp = tn
  i = n – 1
  WHILE i>=0 AND ti>temp
    t_{i+1} ← t_i
    i ← i-1
  t_{i+1} = temp
```

Sortowanie przez wybieranie

Koncepcyjnie zbliżony do sortowania przez wstawianie, ale niestabilny. Najprostsza wersja polega na szukaniu minimum dla nieposortowanej przez kolejne porównania i (ewentualnie) zamiany elementu z początku tej części ze wszystkimi kolejnymi elementami:

```
FOR n = 0, 1, …, N-2
  FOR i = n+1, n+2, … N-1
  IF t_i<t_n
    t_i ↔ t_n
```

Metoda jest niewydajna, ponieważ elementy w nieposortowanej części mają tendencję do układania się w porządku malejącym

Najlepszy wariant sortowania przez wybieranie polega na szukaniu minimum w nieposortowanej (prawej) części zbioru i **jednorazowej** zamianie miejscami minimum oraz pierwszego elementu tej części:

```
FOR n = 0, 1, …, N-2
  min = t_n
  poz = n
  FOR i = n+1, n+2, … N-1
    IF t_i < min
      min ← t_i
      poz = i
    t_n ↔ t_poz
```

Można nawet zrezygnować ze zmiennej *min* i ograniczyć się do korzystania z samej tablicy

```
FOR n = 0, 1, …, N-2
  poz = n
  FOR i = n+1, n+2, … N-1
    IF t_i < t_poz
      poz = i
    t_n ↔ t_poz
```

Sortowanie przez zliczanie

Może być stosowana dla zbiorów o równomiernym rozkładzie kluczy należących do skończonego zbioru – wówczas osiąga złożoność O(n+k) oraz złożoność pamięciową O(n+k), gdzie *n* – oznacza liczebność zbioru, *k* – rozpiętość wartości kluczy. Polega na utworzeniu histogramu, tj. policzeniu liczby wystąpień kluczy o tej samej wartości. Później wartości są wkładane wprost na pozycję wynikającą z histogramu

W skrajnym przypadku (kiedy klucze nie powtarzają się) ma złożoność obliczeniową O(n) i pamięciową O(k). Nie nadaje się do sortowania złożonych danych (np. dane osobowe, alfabetycznie) – co prawda klucze da się wyznaczyć, ale rozpiętość jest zbyt duża i złożoność pamięciowa jest nie do przyjęcia

Sortowanie kubełkowe

Wariant koncepcji poprzedniej – należy podzielić zakres rozpiętości kluczy na k przedziałów (kubełków), następnie przeglądając elementy zbioru przypisywać je do odpowiedniego kubełka na podstawie wartości klucza (złożoność O(n)). Niepuste kubełki trzeba posortować (stosując mniejsze kubełki) itd.
Przy równomiernym rozkładzie kluczy całkowitych osiąga złożoność O(n+k) oraz złożoność pamięciową O(n+k) i jest stabilny

Bogosort

Jeden z algorytmów zaprojektowanych tak, aby były jak najwolniejsze. Polega na losowym ustawianiu elementów zbioru przy każdej iteracji i sprawdzaniu, czy przypadkiem kolejność nie jest właściwa (niemalejąca). Możliwych permutacji zbioru n elementów jest n!, stąd złożoność obliczeniowa O(n!). Już przy kilkudziesięciu elementach czas sortowania trzeba liczyć w miliardach lat (np. $25! \approx 1{,}5\cdot10^{25}$, $50! \approx 3\cdot10^{64}$)

```
FOR ∞
  gotowe = true
  FOR i=0, 1, …, N-1
    IF t_i>t_i+1
      gotowe = false;
      BREAK
  IF gotowe BREEAK
  FOR i=N-1, N-2, …, 1
    t_i ↔ t_random(i)
```

Zadania

Proszę napisać program, który…

1. Tworzy tablicę o rozmiarze podanym przez użytkownika, wypełnia ja wylosowanymi liczbami i wyświetla na ekranie konsoli. Zakres wartości liczb powinien być w rozsądnej relacji do rozmiaru tablicy, tak aby liczby nie powtarzały się zbyt często (2-5 x rozmiar)
2. Sortuje tablicę z punktu 1 metodą bąbelkową (najprostszy wariant), wyświetla wartości posortowane oraz wykonuje test uporządkowania (sprawdza czy porządek liczb jest niemalejący) i wyświetla wynik tego testu
3. Wykonuje zadania z punktu 2, jednak bez wyświetlania tablicy przed i po sortowaniu (tylko wynik testu uporządkowania) oraz mierzy czas sortowania tablicy; Zmierzyć czas sortowania tablic o rozmiarze rosnącym potęgowo (100, 1000, 10000, …) i ocenić typ złożoności obliczeniowej (można nanieść wyniki na wykres, na końcu instrukcji)
Wskazówka: kod odpowiedzialny za wyświetlanie tablicy należy ukryć w komentarzu, a nie kasować, może się jeszcze przydać.
4. * Wykonuje zadania z punktu 3 usprawnioną metodą sortowania bąbelkowego. Powtórzyć pomiary czasu sortowania dla tych samych liczności zbioru liczb i ocenić stopień poprawy (czy zmienił się typ złożoności obliczeniowej czy tylko współczynnik skali?)
Wskazówka: podczas przerabiania programu może się przydać wyświetlanie zawartości tablicy, ukryte w komentarzu
5. Wykonuje zadania z **punktu 3** za pomocą algorytmu gnoma, w wersji podstawowej
6. * Wykonuje zadania z **punktu 4** za pomocą usprawnionego algorytmu gnoma
7. Wykonuje zadania z **punktu 3** za pomocą algorytmu wybierania, w wersji podstawowej
8. * Wykonuje zadania z **punktu 4** za pomocą usprawnionego algorytmu wybierania
9. * Wykonuje zadania z **punktu 3** za pomocą algorytmu bogosort
10. * Wykonuje zadania z **punktu 2** za pomocą algorytmu zliczania