Efektywna metoda sortowania – sortowanie przez scalanie

Rekurencja

Dla rozwiązania danego problemu, algorytm wywołuje sam siebie przy rozwiązywaniu podobnych podproblemów.

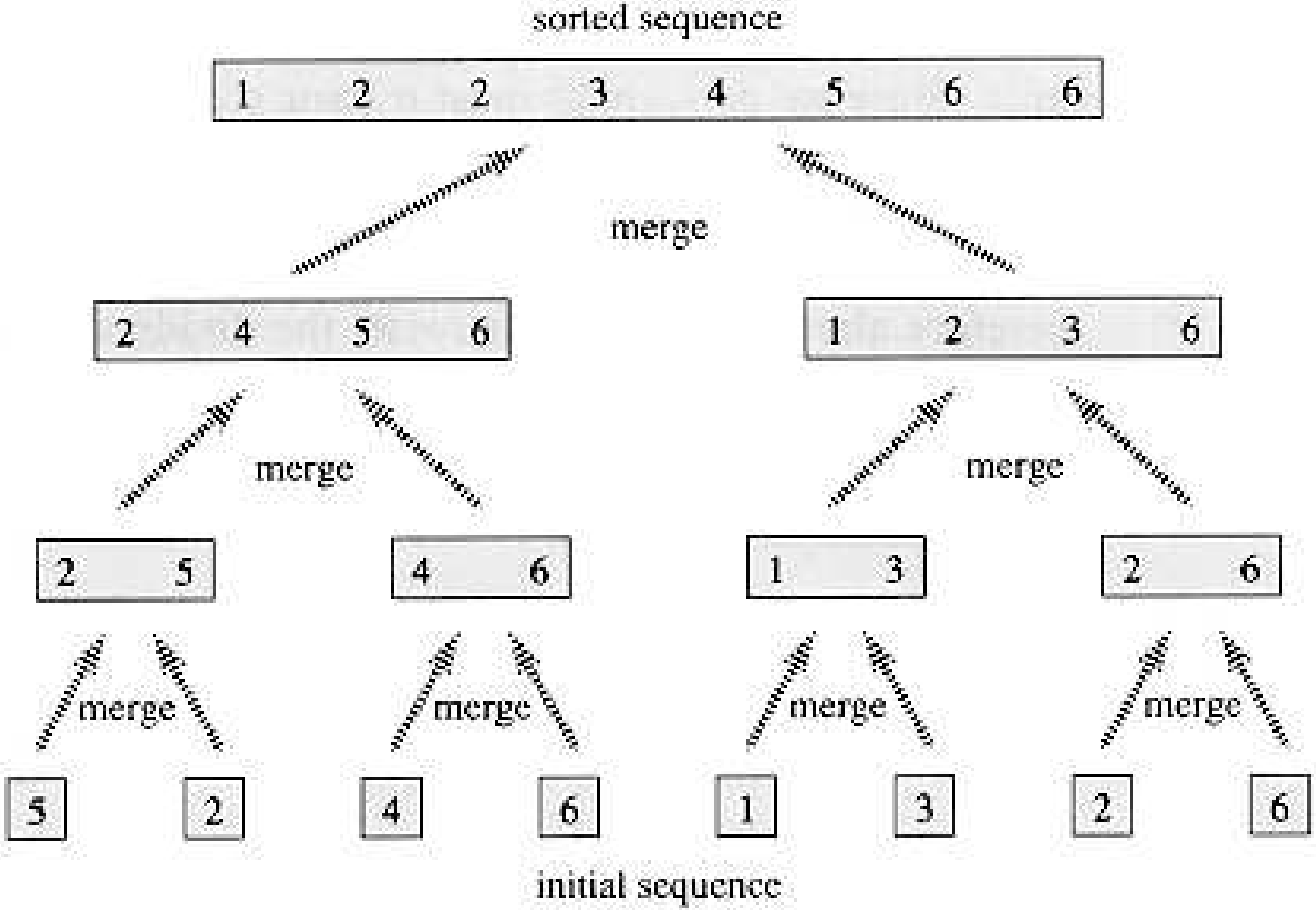
Metoda dziel i zwyciężaj

Dzielimy problem na kilka mniejszych podproblemów podobnych do problemu wyjściowego i rozwiązujemy je rekurencyjnie. Na koniec rozwiązania są łączone w celu utworzenia rozwiązania całego problemu.

Dziel: Dzielimy problem na podproblemy
Zwyciężaj: Rozwiązujemy podproblemy rekurencyjnie, o ile nie są zbyt małego rozmiaru. W takim przypadku używamy metod bezpośrednich.
Połącz: Łączymy rozwiązania podproblemów, aby otrzymać rozwiązanie całego problemu.

Metoda dziel i zwyciężaj dla sortowania przez scalanie:

Dziel: Dzielimy n-elementowy ciąg na dwa podciągi, po $n/2$ elementów każdy.
Zwyciężaj: Sortujemy otrzymane podciągi, używając rekurencyjnie sortowania przez scalanie.
Połącz: Łączymy posortowane podciągi w jeden posortowany ciąg.


sorted sequence
1 2 2 3 4 5 6 6
merge
2 4 5 6
1 2 3 6
merge
merge
2 5
4 6
1 3
2 6
merge
merge
merge
merge
5
2
4
6
1
3
2
6
initial sequence

Procedura scalania

Mamy dwa posortowane podciągi, chcemy je połączyć w jeden posortowany ciąg. W tym celu cyklicznie bierzemy mniejszą z liczb znajdujących się na początku obu podciągów i wstawiamy ją do ciągu posortowanego. Operacja ta odbywa się w czasie $\Theta(n)$.

Pseudokod

```
mergesort(T,p,r)
   jeżeli p<r
      q = zaokrągl_w_dół( (p+r)/2 )
      mergesort(T,p,q)
      mergesort(T,q+1,r)
      merge(T,p,q,r)
```

Początkowe wywołanie procedury: `mergesort(T,1,n)`

Analiza algorytmu

Dla tablicy jednoelementowej sortowanie działa oczywiście w czasie stałym $\Theta(1)$.
Załóżmy, że $n>1$. Niech $F(n)$ będzie czasem potrzebnym na rozwiązanie problemu o rozmiarze n.
Dla sortowania przez scalanie: $F(n) = \Theta(1) + 2F(n/2) + \Theta(n) =$
(dziel) (zwyciężaj) (połącz)
$= 2F(n/2) + \Theta(n) = \Theta(n \log_2 n)$

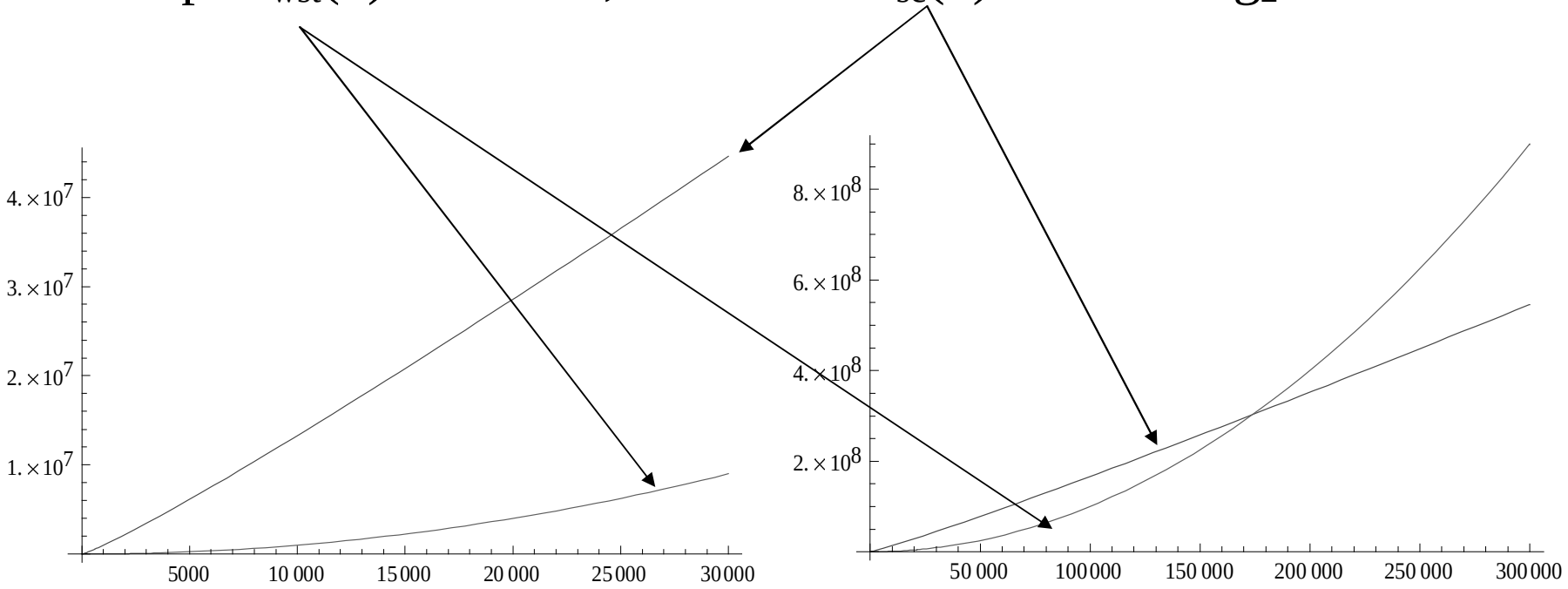
Przypomnienie: sortowanie przez wstawianie działa w czasie $\Theta(n^2)$.

Porównanie czasów działania algorytmów sortowania przez wstawianie i scalanie

Załóżmy, że możemy wykonać dane zadanie przy użyciu dwóch algorytmów: jednego o złożoności czasowej $\Theta(n^2)$, a drugiego o złożoności $\Theta(n \log_2 n)$ – na przykład sortowanie n-elementowej tablicy metodą przez wstawianie i scalanie.

Sortowanie przez wstawianie wykonajmy na szybkim komputerze (mała stała przed Θ) a sortowanie przez scalanie na bardzo wolnym (duża stała przed Θ).

Niech np. $F_{wst}(n) = 0.01\ n^2$, natomiast $F_{sc}(n) = 100\ n \log_2 n$.



Wniosek: szybszy algorytm wygrywa dla odpowiednio dużego rozmiaru problemu, mimo iż został uruchomiony na 10 000 razy wolniejszym komputerze!

Rozsądny a nierozsądny czas działania

Liczba operacji potrzebna do rozwiązania problemu o danej złożoności obliczeniowej

złożoność / n	**10**	**50**	**100**	**300**	**1 000**
log n	3	5	6	8	10
n	10	50	100	300	1 000
n log n	33	282	665	2 469	9 966
n^2	100	2 500	10 000	90 000	1 000 000
n^3	1 000	125 000	1 000 000	27 000 000	1 000 000 000
2^n	1 024	liczba 16-cyfr.	liczba 31-cyfr.	liczba 91-cyfr.	liczba 302-cyfr.
n!	3 600 000	liczba 65-cyfr.	liczba 161-cyfr.	liczba 623-cyfr.	niewyobrażalnie duża
n^n	10 000 000 000	liczba 85-cyfr.	liczba 201-cyfr.	liczba 744-cyfr.	niewyobrażalnie duża

Dla porównania: liczba protonów we Wszechświecie – liczba 126-cyfrowa; liczba mikrosek. od wielkiego wybuchu – liczba 24-cyfrowa

Maksymalny rozmiar problemu możliwego do rozwiązania w danym czasie

(przy założeniu 1 000 000 operacji na sekundę)

złożoność / czas	**1 s**	**1 min**	**1 godz**	**1 rok**
log n	$2^{1\ 000\ 000}$	$2^{60\ 000\ 000}$	$2^{36\ 000\ 000\ 000}$	$2^{300\ 000\ 000\ 000\ 000}$
n	10^6	6×10^7	3.6×10^9	3×10^{13}
n log n	62 746	2.8×10^6	1.3×10^8	8×10^{11}
n^2	1 000	7 746	60 000	5.6×10^6
2^n	19	25	31	44
n!	9	11	12	16

Wniosek: algorytmy o rozsądnym czasie wykonania to algorytmy o co najwyżej wielomianowej złożoności obliczeniowej.

Sortowanie szybkie – quicksort

(szczegółowo omówione „na tablicy”)

W metodzie tej dzieli się tablicę na dwie części (np. ze względu na pierwszy jej element), a następnie sortuje rekurencyjnie każdą z nich.

Szkielet kodu procedury sortującej:

```
procedure qsort(d,g)
  jeżeli fragment tablicy składa się z co najmniej 2 elementów, to:
     podziel elementy tablicy np. ze względu na wartość T[d]
     // w wyniku podziału wartość T[d] powinna znaleźć się
     // na właściwym miejscu s wewnątrz tablicy
     // oraz powinien spełniony być warunek:
     // T[d] ... T[s-1] <= T[s] <= T[s+1] ... T[g]
     qsort(d, s-1) // posortuj dolną część tablicy
     qsort(s+1, g) // posortuj górną część tablicy
```

Przykładowy szkielet kodu rozdzielającego elementy tablicy:

```
element_graniczny = T[d]  // rozdzielamy względem pierwszego elementu
srodek = d
pętla od i = d+1 do i = g
  jeżeli element T[i] < element_graniczny
    srodek = srodek+1
    zamień T[srodek] z T[i]
zamień T[d] z T[srodek]
```

Optymalizacja

1. Udoskonalenie metody znajdowania elementu środkowego, według którego rozdzielane są elementy tablicy (należy najpierw zastanowić się, kiedy przyjęta wyżej metoda rozdzielania względem pierwszego elementu tablicy jest mało efektywna?);
2. Poszukanie efektywniejszych metod rozdzielania elementów tablicy lub usprawnienie zaproponowaną wyżej

3. Zoptymalizowanie kodu programu (np. rozwijanie kodu funkcji wewnątrz pętli zamiast jej wywołań);
4. Zastosowanie sortowania przez wstawianie dla wstępnie posortowanych metodą szybką małych fragmentów tablic.

Bisekcja

(szczegółowo omówiona „na tablicy”)

Metoda służy do wyznaczenia miejsca zerowego danej funkcji i polega na cyklicznym połowieniu zadanego z góry przedziału (w którym znajduje się pierwiastek) aż do osiągnięcia zadanej dokładności.