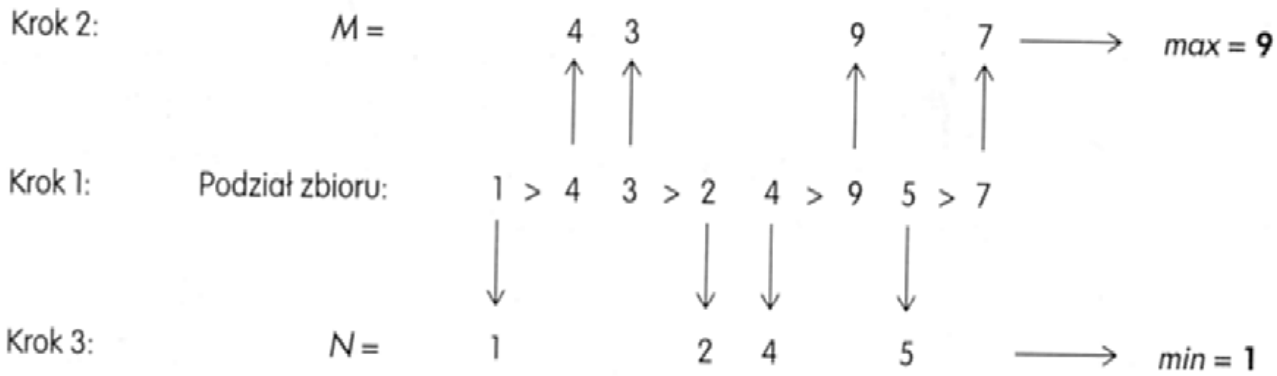
Metoda "DZIEL i ZWYCIĘŻAJ"

W metodzie najczęściej wyróżnić można trzy podstawowe kroki:

1. Podziel zestaw danych na dwie, równe części (w przypadku nieparzystej liczby wyrazów jedna część będzie o 1 wyraz większa)
2. Zastosuj algorytm dla każdej z nich oddzielnie, chyba że pozostał już tylko jeden element
3. Połącz, jeśli potrzeba, rozwiazania dla każdej z cześci w całość

Najmniejszy i największy element zbioru n-elementowego

Aby znaleźć rozpiętość zbioru należy znaleźć maksimum i minimum, a potem policzyć różnicę.
Można wykorzystać dwukrotnie metodę przeszukiwania liniowego (algorytmy Min, Max) - *2*(n-1)* porównań.
Algorytmy Min, Max można połączyć, aby poszukiwania były bardziej efektywne, zgodnie z przepisem:



Krok 1: {podział zbioru na dwa podzbiory}
Z elementów zbioru utworzyć pary (dla nieparzystego *n* pozostaje dodatkowy element *z*). Porównać elementy w parach i gdy $x > y$ to dołączyć x do zbioru (M) kandydatów na maksimum, y do zbioru (N) kandydatów na minimum. W przeciwnym razie postąpic odwrotnie.

Krok 2: Znaleźć maksimum w zbiorze M za pomocą algorytmu Max.
Krok 3: Znaleźć minimum w zbiorze N za pomocą algorytmu Min.
Krok 4: Jeśli *n* jest nieparzyste to - jeśli $z < min$ to $min = z$,
w przeciwnym razie - jeśli $z > max$ to $max = z$.

W powyższej formie algorytm można zrealizować wykorzystując trzy podprogramy:
i) służący do podziału zbioru na dwa mniejsze - krok 1,
ii) liczący maksimum (Max) - krok 2,
iii) liczący minimum (Min) - krok 3

Można wprowadzić dwa usprawnienia :
1) Zamiast tworzyć zbiory M i N można przestawiać pary (np. jeśli dla pary zachodzi $x > y$) - w ten sposób po pierwszym kroku na nieparzystych miejscach tablicy będą elementy zbioru N, na parzystych elementy zbioru M.

2) Jeśli *n* nieparzyste to przedłużyć ciąg dublując ostatni element tablicy - w ten sposób można pozbyć się kroku 4 i ujednolicić kroki 2 i 3 (minimum jest szukane w połowie komórek tablicy o indeksach nieparzystych, a maksimum w drugiej połowie o indeksach parzystych).

Dla n parzystego algorytm wykonuje $3*n/2 - 2 = 2*(n-1) - n/2$ porównań (w sosunku do algorytmu liniowego mniej o $n/2$ porównań).

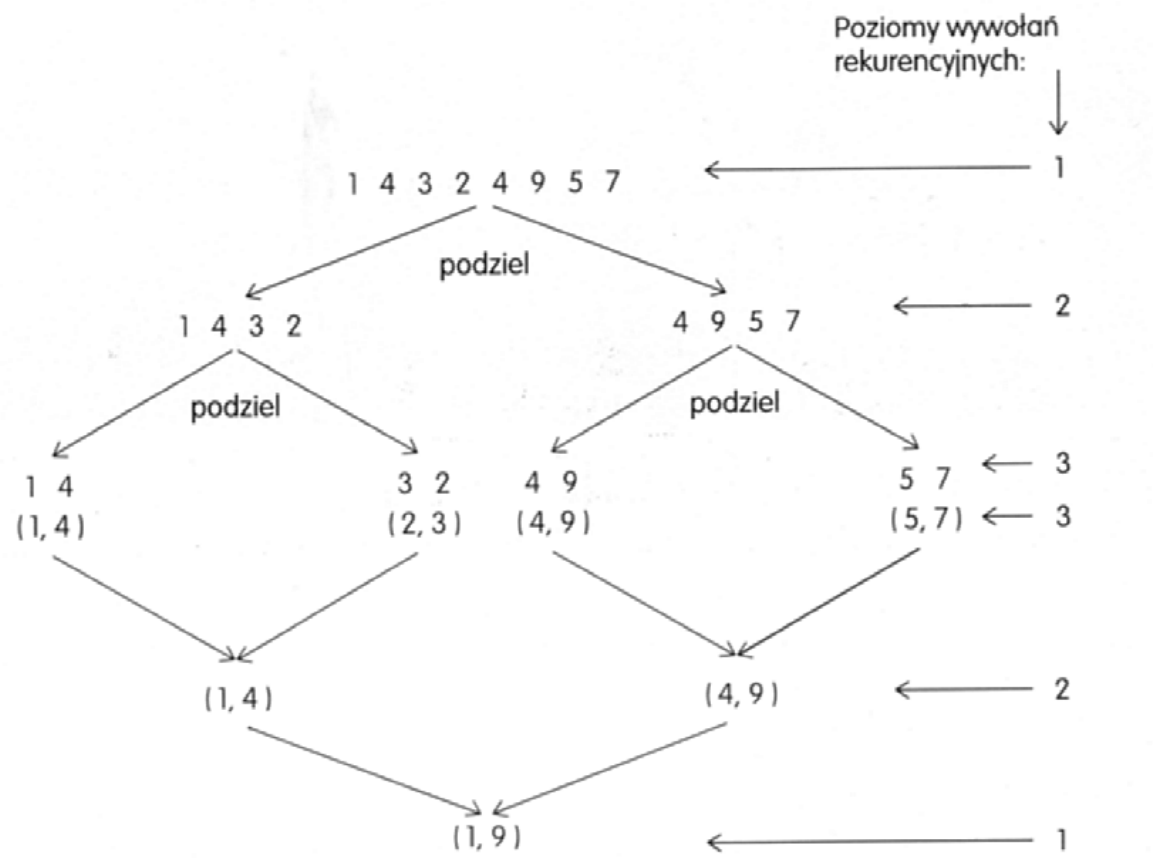
Algorytm Min-Max jest przykładem zastosowania metody "dziel i zwyciężaj" i można w nim wyróżnić trzy etapy:
1) Podziel problem na pod-problemy.
2) Znajdź rozwiązania pod-problemów.
3) Połącz rozwiązania pod-problemów w rozwiązanie głównego problemu.

Dodatkowo pod-problemy powinny mieć następujące własności:
1a. Problem jest dzielony na takie same lub bardzo podobne pod-problemy.

1b. Liczba pod-problemów wynosi conajmniej 2.
1c. Pod-problemy są rozwiązywane na podzbiorach zbioru danych, w których liczba elementów jest niemal jednakowa i stanowi stałą część całego zbioru.

W metodzie "dziel i zwyciężaj" najczęściej pod-problemy, na które rozkładamy problem, są tym samym problemem, ale dla mniejszego zbioru danych. Można więc zastosować algorytm rekurencyjny.

Algorytm Min-Max wykorzystuje metodę dziel i zwyciężaj jedynie w pierwszym kroku, ale wersja rekurencyjna wykorzystuje metodę "dziel i zwyciężaj" w każdym kroku.



Algorytm Min_Max_Rek(Z,max,min) - wersja rekurencyjna
{Z- zbiór liczb; max, min -największy, najmniejszy element}

Krok 1: Jeśli zbiór Z składa się z jednego elementu (z), to *min* = *z*, *max* = *z*.
Jeśli zbiór Z składa się z dwóch elementów, to wartość większego z nich przypisz do max, a wartość mniejszego

z nich do min.

Krok 2: W przeciwnym razie:

2a: Podziel zbiór Z na dwa podzbiory Z_1 i Z_2 o tej samej lub niemal tej samej liczbie elementów.

2b: Wykonaj ten sam algorytm dla (Z_1 , max_1, min_1)

2c: Wykonaj ten sam algorytm dla (Z_2 , max_2, min_2)

2d: Wartość większej z liczb max_1, max_2 przypisz do *max,* wartość mniejszej z liczb min_1 , min_2 przypisz do *min*.

Bardzo dobrym przykładem metody "dziel i zwyciężaj" jest algorytm przeszukiwania binarnego.

Istotnym uogólnieniem tego algorytmu jest schemat binarnego umieszczania, w którym wstawiamy dodatkowy element do ciągu w taki sposób by ciąg pozostał uporządkowany.

Algorytm umieszczania binarnego

Problem: wstawić do uporządkowanego ciągu nowy element tak aby ciąg pozostał uporządkowany

Dane wejściowe: uporządkowany ciąg liczb w tablicy $a[k..l]$, $k \le l$, to znaczy $a_k \le a_{k+1} \le ... \le a_l$ oraz element $y \ge a_k$.

Wynik: miejsce dla y w ciągu $a[k..l]$, tzn. największe r takie, że $a_r \le y \le a_{r+1}$, jeśli $k \le r \le l-1$ (miejsce znalezione), lub $r = l$, gdy $a_r \le y$ (brak miejsca w zakresie $k..l$).

Krok 1. $lewy = k$, $prawy = l$

Krok 2. $s = \lceil (lewy+prawy)/2 \rceil$

Krok 3. Jeśli $a_s \le y$, to lewy = s, a w przeciwnym razie $prawy = s-1$.

Krok 4. Jeśli $lewy=prawy$, to zakończ algorytm - wtedy $r = lewy$, a w przeciwnym razie powtórz krok 2.

Przeszukiwanie interpolacyjne

Mając za zadanie znaleźć jakiś element w książce telefonicznej, w książce adresowej czy też w jakimkolwiek zbiorze posortowanych elementów można zastosować przeszukiwanie jeszcze efektywniejsze niż binarne zwane **interpolacyjnym**.

W **przeszukiwaniu binarnym** następny punkt w przedziale o końcach *lewy* i *prawy* znajdujemy ze wzoru: $s = (lewy + prawy)/2$ lub inaczej $s = lewy + 1/2 * (prawy - lewy)$. We wzorach tych nie uwzględniamy wartości poszukiwanego elementu y .

W **przeszukiwaniu interpolacyjnym** we wzorze tym zastąpimy proporcję 1/2 stosunkiem odległości elementu y od elementu na lewym końca przedziału poszukiwań do całego przedziału,
$s = lewy + (y-a_{lewy})/(a_{prawy} - a_{lewy}) *(prawy - lewy)$.

Algorytm będący modyfikacją algorytmu binarnego umieszczania nazywa się **algorytmem interpolacyjnego umieszczania**.

Wymaga on kilku modyfikacji związanych za sposobem obliczania *s:*

1. Wartość s nie może być większa niż prawy koniec przedziału *prawy.*
2. Obliczenia kończymy, gdy $a_{lewy} = y$ (wtedy s nie ulega już zmianie w następnym kroku).
3. Nie wolno dzielić przez zero, tzn. przeszukiwany ciąg nie zawiera takich samych elementów.

Algorytm interpolacyjnego umieszczania

Problem: wstawić do uporządkowanego ciągu nowy element, tak aby ciąg pozostał uporządkowany, wykorzystać wartość elementu czyli użyć metody interpolacyjnej

Dane wejściowe: uporządkowany ciąg różnych liczb w tablicy $a[k..l]$, $k \le l$, to znaczy $a_k \le a_{k+1} \le ... \le a_l$ oraz element $y \ge a_k$.

Wynik: miejsce dla y w ciągu $a[k..l]$, czyli takie s, że $a_s = y$ jeśli y jest równe jakiemuś elementowi przeszukiwanego ciągu, lub s jest największą liczbą taką, że $a_s \leq y$.

Krok 1. *lewy = k, prawy = l* {początkowe końce przedziału poszukiwań}
Krok 2. $s=min\{lewy+\lceil (y-a_{lewy})/(a_{prawy}-a_{lewy})*(prawy - lewy) \rceil, prawy\}$
Krok 3. Jeśli $a_s \leq y$, to *lewy = s,* a w przeciwnym razie *prawy = s-1*.
Krok 4. Jeśli *lewy=prawy* lub $a_{lewy} = y$ to zakończ algorytm, a w przeciwnym razie powtórz krok 2.

Porównanie efektywności algorytmów binarnego i interpolacyjnego umieszczania dla n=50 losowych (uporządkowanych) liczb całkowitych z przedziału [1,..,100] i dla y=10,30,49,70,95.

y	Binarne umieszczanie					Interpolacyjne umieszczanie				
10	6	6	6	6	6	2	3	2	2	2
30	6	6	6	6	5	3	2	2	2	2
49	5	6	6	6	6	3	3	3	2	2
70	6	5	5	6	6	3	3	3	2	1
95	6	6	6	6	6	2	2	3	1	3

Ilość iteracji

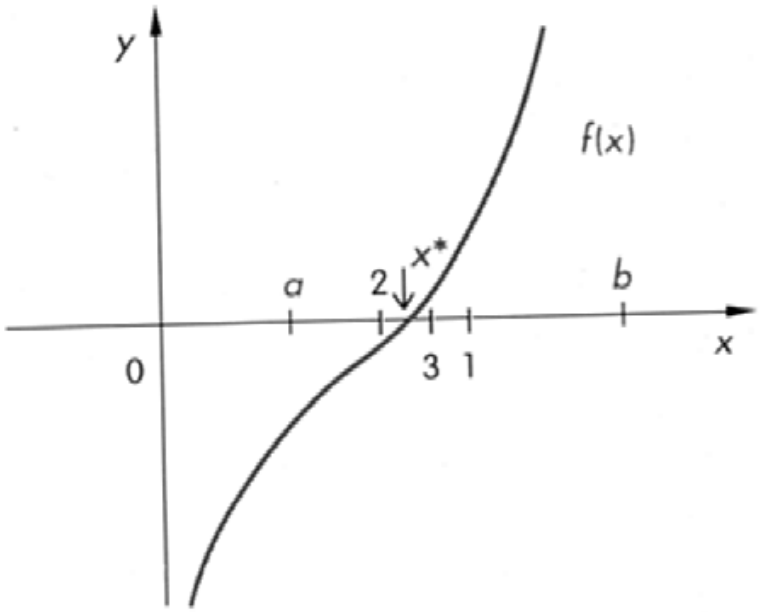
W analizie algorytmów pojawiają się często funkcje **ograniczenie dolne** $\lfloor x \rfloor$ i **ograniczenie górne** $\lceil x \rceil$ **,** które przyporządkują zmiennej rzeczywistej x najbliższe jej wartości liczby całkowite.
Ograniczenie dolne x jest największą liczbą całkowitą niewiększą niż x. **Ograniczenie górne** jest najmniejszą liczbą całkowitą nie mniejszą niż x.

Znajdowanie zera funkcji metodą połowienia przedziału

Niech f(x) będzie funkcją ciągłą w przedziale domkniętym [a,b] oraz spełniony jest warunek f(a)*f(b) < 0 (na końcach przedziału funkcja ma różne znaki).

Oznacza to, że w przedziale [a,b] jest punkt x^* spełniający warunek $f(x^*) = 0$ czyli będący miejscem zerowym funkcji f(x).

Metoda znajdowania x^* polega na:
- podziale przedziału punktem znajdującym się w połowie
- znalezieniu znaku funkcji f w tym punkcie
- wyborze z dwóch pod-przedziałów tego na którego końcach funkcja f ma nadal przeciwne znaki



Jeśli przez $[a_i, b_i]$ oznaczymy ciąg kolejnych przedziałów generowanych w tej metodzie to mamy do wyboru trzy kryteria przerwania obliczeń:

1. Różnica między kolejnymi przybliżeniami położenia wartości zera funkcji jest mniejsza niż przyjęta dokładność obliczeń *Eps*, czyli $b_i - a_i < Eps$.

2. Liczba wykonanych iteracji osiągnęła określoną przez nas granicę MaxIter.
3. Wartość funkcji w kolejnym przybliżeniu jest dostatecznie bliska zeru, czyli mniejsza niż zadana liczba *Eps1* ($|f((a_i + b_i)/2)| \leq Eps1$.

Widać, że warunki 1 i 2 są zależne i na podstawie *Eps* można znaleźć MaxIter które wynosi co najmniej $log_2((b-a)/Eps)$.

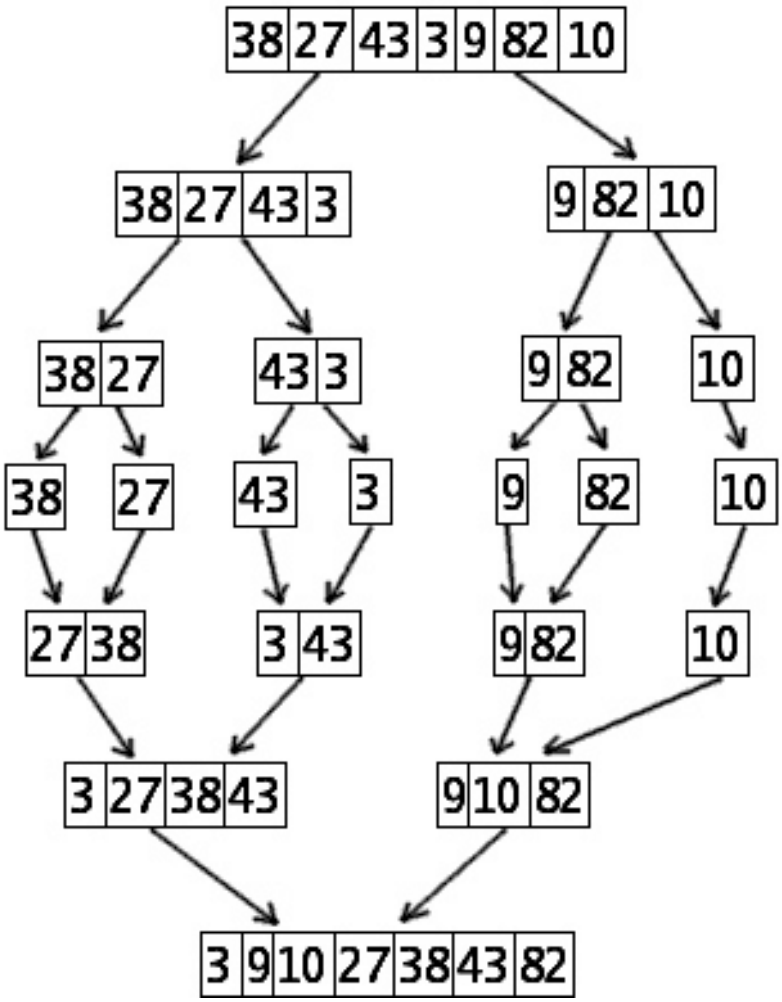
Zapis algorytmu jest formalnością.

Sortowanie przez scalanie

Procedura scalania dwóch ciągów *A[1..n]* i *B[1..m]* do ciągu *C[1..m+n]*:

1. Utwórz liczniki ustawione na początki ciągów *A* i *B* -> *i=0, j=0*
2. Jeżeli *A*[*i*] < = *B*[*j*] wstaw *A[i]* do *C* i zwiększ *i* o jeden, w przeciwnym przypadku wstaw *B[j]* do *C* i zwiększ *j* o jeden
3. Powtarzaj krok 2 aż wszystkie wyrazy *A* i *B* trafią do *C*

Scalenie wymaga *n+m* operacji porównań elementów i wstawienia ich do tablicy wynikowej.



Dowód przez indukcję względem długości *n* tablicy elementów do posortowania.

1) n=2

Algorytm podzieli dane wejściowe na dwie części, po czym zastosuje dla nich scalanie do posortowanej tablicy

2) Założenie: dla ciągów długości *k*, *k<n* algorytm mergesort prawidłowo sortuje owe ciągi.

Dla ciągu długości *n* algorytm podzieli ten ciąg na *dwa ciągi* długości *n/2*. Na mocy założenia indukcyjnego ciągi te zostaną prawidłowo podzielone i scalone do dwóch posortowanych ciągów długości *n/2*.

Ciągi te zostaną natomiast scalone przez procedurę scalającą do *jednego*, posortowanego ciągu długości n.

Algorytm sortowania przez scalanie

Problem: posortuj *n* kluczy w ciąg niemalejący.

Dane wejściowe: dodatnia liczba całkowita *n*, tablica kluczy *S* indeksowana od 1 do *n*.

Wynik: tablica *S*, zawierająca klucze w porządku niemalejącym.

```
void mergesort(int n, keytype S[])
{
 if(n>1) {
    const int h= ⌊n/2⌋, m=n-h;
    keytype U[1..h], V[1..m];
    skopiuj S[1] do S[h] na miejsce U[1] do U[h];
    skopiuj S[h+1] do S[n] na miejsce V[1] do v[m];
    mergesort(h,U);
    mergesort(m,V);
    merge(h,m,U,V,S);
 }
}
```

Agorytm scalania

Problem: scal dwie posortowane tablice w jedną posortowaną tablicę.

Dane wejsciowe: dodatnie liczby całkowite *h* i *m* , tablica posortowanych kluczy *U* indeksowana od 1 do *h*, tablica posortowanych kluczy V indeksowana od 1 do *m*.

Wynik: tablica S, indeksowana od 1 do $h+m$, zawierająca klucze z tablic U i V w ramach pojedynczej posortowanej tablicy.

```
void merge(int h, int m, const keytype U[],
                         const keytype V[],
                               keytype S[])
{
  index i,j,k;

  i=1; j=1; k=1;
  while (i<=h && j<=m) {
     if(U[i]<V[j]) {
        S[k]=U[i];
        i++;
     }
     else {
        S[k]=V[j];
        j++;
     }
     k++;
  }
  if(i>h)
     skopiuj V[j] do V[m] na miejsce S[k] do S[h+m];
  else
     skopiuj U[i] do U[h] na miejsce S[k] do S[h+m];
}
```

---

k	U	V	S(wynik)	
1	**10** 12 20 27	**13** 15 22 25	10	
2	10 **12** 20 27	**13** 15 22 25	10 12	
3	10 12 **20** 27	**13** 15 22 25	10 12 13	
4	10 12 **20** 27	13 **15** 22 25	10 12 13 15	
5	10 12 **20** 27	13 15 **22** 25	10 12 13 15 20	
6	10 12 20 **27**	13 15 **22** 25	10 12 13 15 20 22	
7	10 12 20 **27**	13 15 22 **25**	10 12 13 15 20 22 25	
-	10 12 20 27	13 15 22 25	10 12 13 15 20 22 25 27	<wartości
	i	*j*		<końcowe

Wartości porównywane.

Sortowanie w miejscu

Algorytm sortowania, który nie wykorzystuje żadnej dodatkowej przestrzeni pamięciowej, poza wymaganą do przechowywania danych wejsciowych. W algorytmie przeprowadza się pewne działania na tablicy wyjściowej S.

Algorytm sortowanie w miejscu

Problem: posortuj *n* kluczy w ciąg niemalejacy.

Dane wejściowe: dodatnia liczba całkowita *n*, tablica kluczy *S* indeksowana od 1 do *n*.

Wynik: tablica *S*, zawierająca klucze w porządku niemalejącym.

```
void mergesort2(index low, index high)
{
  index mid;

  if(low<high) {
     mid = ⌊(low+high)/2⌋;
     mergesort2(low,mid);
     mergesort2(mid+1,high);
     merge2(low,mid,high);
  }
}
```

Algorytm scalania

Problem: scal dwie posortowane podtablice tablicy *S*, utworzone w funkcji *mergesort2*.

Dane wejściowe: indeksy *low, mid i high* oraz podtablica *S* indeksowana od *low* do *high*. Klucze w tablicy od pozycji *low* do *mid*

są już posortowane w porządku niemalejącym, podobnie jak klucze w tablicy od pozycji *mid+1* do *high*.

Wynik: podtablica tablicy S indeksowana od *low* do *high*, zawierająca klucze w porzadku niemalejacym.

```
void merge2(index low, index mid, index high)
{
 index i,j,k;
 keytype U[low..high]; // Tablica lokalna do scalania
 i=low;j=mid+1;k=low;

 while(i <= mid && j <= high) {
     if(S[i]<S[j]) {
        U[k] = S[i];
        i++;
     }
     else {
        U[k] = S[j];
        j++;
     }
     k++;
 }
 if(i>mid)
 przenies S[j] do S[high] na miejsce U[k] do U[high];
 else
 przenies S[i] do S[mid] na miejsce U[k] do U[high];

przenies U[low] do U[high] na miejsce S[low] do
S[high];
}
```

Sortowanie przez scalanie – wersja nierekurencyjna

Podstawową wersję algorytmu sortowania przez scalanie można uprościć. Pomysł polega na odwróceniu procesu scalania serii. Ciąg danych możemy wstępnie podzielić na *n* serii długości *1*, scalić je tak, by otrzymać $\frac{n}{2}$ serii długości *2*, scalić je otrzymując $\frac{n}{4}$, serii długości *4*... Złożoność obliczeniowa jest taka sama jak w przypadku klasycznego *mergesort*, w tym przypadku jednak nie korzystamy z

rekursji, a więc zaoszczędzamy czas i pamięć potrzebną na jej obsłużenie.

```
typedef unsigned int TYP;

void MergeSort(TYP* A, unsigned n) // n==Length(A)
{
   unsigned d=1, // dlugosc aktualnej serii
            i,j;
   while(d<n)
   {
        i=0;j=i+d;
        while(j<n)
        {
        Merge(A[i..i+d-1],A[j,min(n,j+d-1)]) //wynik w
A[i..2d-1]
           i+=2d;
           j+=2d;
        }
        d=d*2;
   }
}
```

Szybki algorytm porządkowania (quicksort)

Podstawowy krok polega na podziale porządkowanego ciągu wybranym elementem v na dwie części, takie że w jednej znajdują się liczby nie większe niż v, a w drugiej liczby nie mniejsze niż v.

Po wykonaniu tego kroku element v można umieścić między podciągami, a następnie przejść do porządkowania obu podciągów tą samą metodą, aż do momentu gdy podciągi będą złożone tylko z jednego elementu..

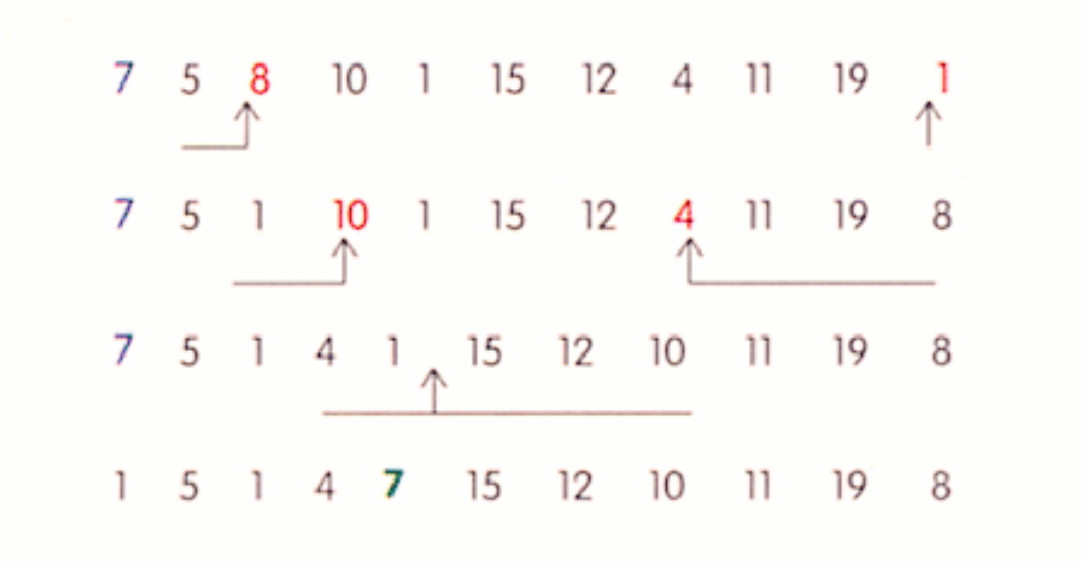
Dodatkowo możemy uściślić algorytm podziału:

1. Przyjmuje się, że v jest pierwszym elementem dzielonego ciągu.
2. Podział ciągu rozpoczynamy od obu jego końców, posuwając się w kierunku środka ciągu.

3. W ruchu od lewej strony zatrzymujemy się na elemencie większym od v, zaś w ruchu od prawej zatrzymujemy się na elemencie mniejszym od v.
4. Parę znalezionych elementów zamieniamy miejscami.

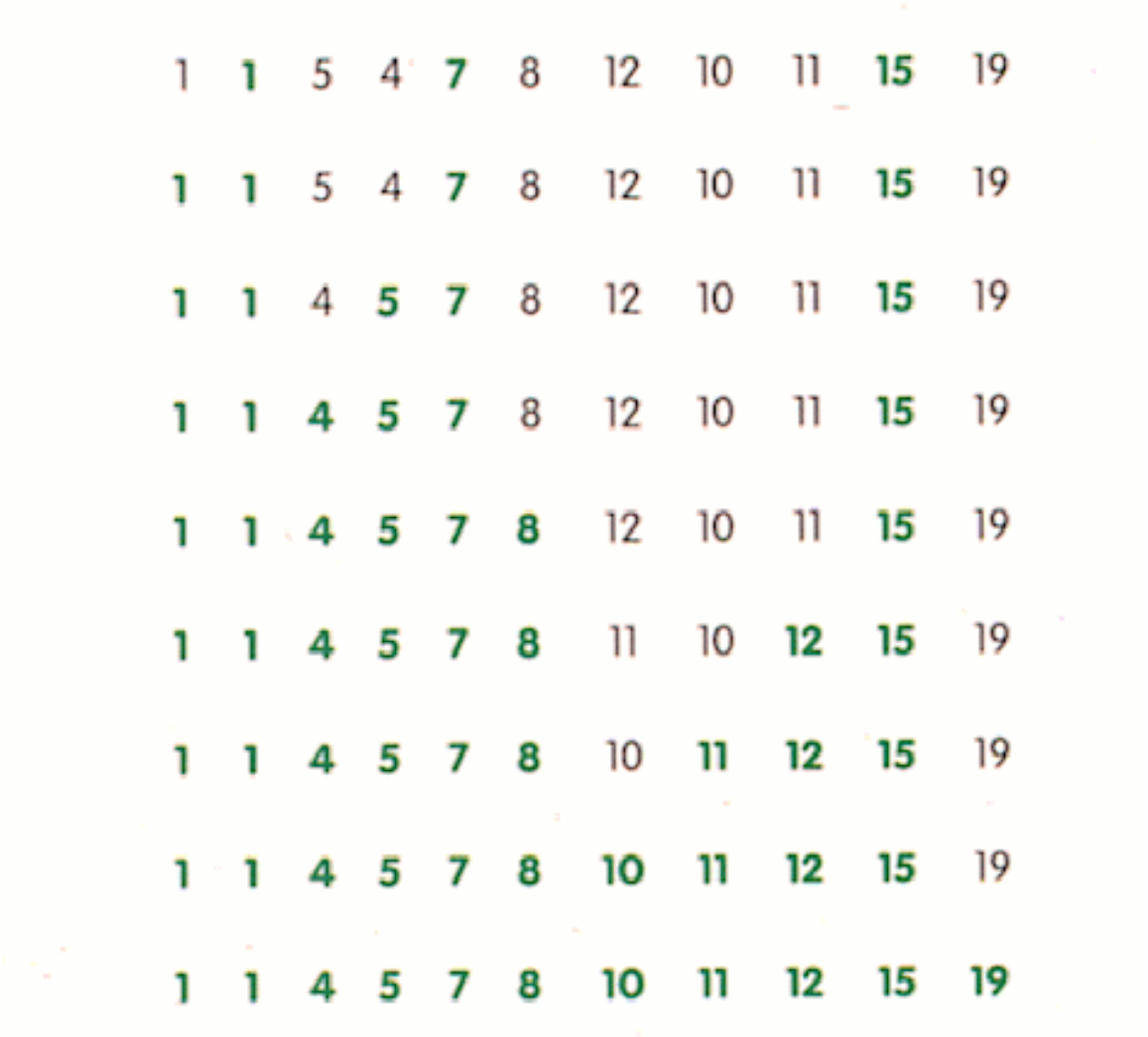
Przykład:

Etap 1



Etap 2

Etap 3 i kolejne.



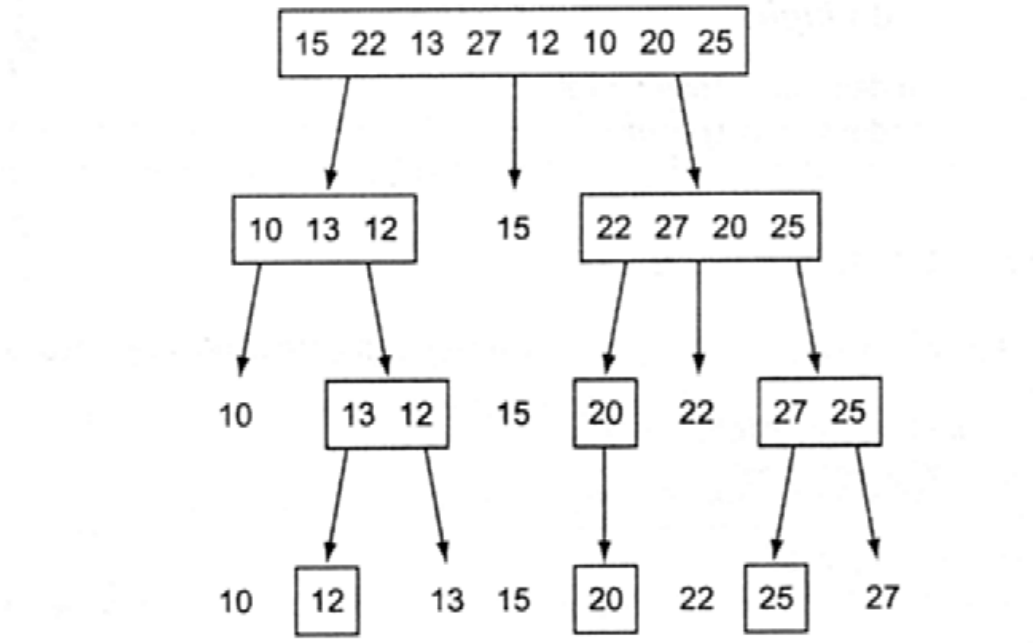
Algorytm: {Dane to ciąg liczb $x_l, x_{l+1}, ..., x_p$ }

Krok1. Jeśli $l<p$, to przyjąć za element podziału $v= x_l$, i podzielić tym elementem dany ciąg.
{oznacza to, że *v* znajdzie się na pozycji elementu x_k, dla pewnego k takiego, że $l \leq k \leq p$ i elementy na lewo będą od niego nie większe, a na prawo - nie mniejsze}

Krok2. Zastosuje ten sam algorytm do *(l,k-1,x)*
Krok3. Zastosuj ten sam algorytm do *(k+1,p,x)*

Technicznie podział można również przeprowadzić nieco inaczej przesuwając się z dwoma indeksami cały czas od lewej strony (pierwszy indeks oznacza każdy kolejny element, drugi pozycję na którą należy go przenieść).

Schematycznie całość algorytmu można zobrazować:



Podział tablicy można przedstawić.

Algorytm podziału tablicy danych

Problem: podziel tablicę *S* dla celów sortowania szybkiego

Dane wejściowe: dwa indeksy *low* i *high*, oraz podtablica tablicy *S* indeksowana od *low* do *high*.

Wynik: *pivotpoint*, punkt odniesienia dla podtablicy indeksowanej od *low* do *high*.

```
void partition(index low, index high,
               index &pivotpoint)
{
  index i,j;
  keytype pivotitem;

  pivotitem = S[low];
  j=low;
  for(i=low+1;i<=high;i++)
      if(S[i]<pivotitem) {
         j++;
         zamień S[i] z S[j];
      }
  pivotpoint=j;
//umieszczenie pivotitem na pozycji pivotpoint
  zamień S[low] z S[pivotpoint];
}
```

Procedura *partition* sprawdza każdy element w tablicy po kolei. Znaleziony element o wartości mniejszej niż element odniesienia zostaje przeniesiony na lewą strone tablicy – zgodnie z opisem poniżej:

i	***j***	***S[1]***	***S[2]***	***S[3]***	***S[4]***	***S[5]***	***S[6]***	***S[7]***	***S[8]***
-	-	15	22	13	27	12	10	20	25
2	1	**15**	**22**	13	27	12	10	20	25
3	2	**15**	22	**13**	27	12	10	20	25
4	2	**15**	**13**	**22**	**27**	12	10	20	25
5	3	**15**	13	22	27	**12**	10	20	25
6	4	**15**	13	**12**	27	**22**	**10**	20	25
7	4	**15**	13	12	**10**	22	**27**	**20**	25
8	4	**15**	13	12	10	22	27	20	**25**
-	4	**10**	13	12	**15**	22	27	20	25

Zapis algorytmu rekurencyjnego dla sortowania szybkiego jest formalnością.

Algorytm Strassena mnożenia macierzy

W standardowym mnożeniu macierzy A i B o rozmiarze $n \times n$ robimy n^3 operacji mnożenia, zaś operację dodawania/odejmowania wykonujemy $n^3 - n^2$ razy. Algorytm Strassena robi to bardziej wydajnie.

Załóżmy, że chcemy otrzymac iloczyn C dwóch macierzy A i B o wymiarach 2×2, to znaczy:

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

Możemy określić następujące zależności:

$$\begin{aligned} m_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ m_2 &= (a_{21} + a_{22})b_{11} \\ m_3 &= a_{11}(b_{12} - b_{22}) \\ m_4 &= a_{22}(b_{21} - b_{11}) \\ m_5 &= (a_{11} + a_{12})b_{22} \\ m_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\ m_7 &= (a_{12} - a_{22})(b_{21} + b_{22}) \end{aligned}$$

Za ich pomocą można określić iloczyn macierzy:

$$C = \begin{vmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{vmatrix}$$

W przypadku mnożenia macierzy o wymiarach 2×2 metoda Strassena wymaga 7 mnożeń i 18 operacji dodawania/odejmowania, metoda standardowa 8 mnożeń i 4 operacji dodawania/odejmowania. Zysk dla macierzy 2×2 jest niewielki. Dla wiekszych macierzy zysk jest wiekszy.

Zakładamy, że n jest potęgą liczby 2 i tworzymy podmacierze macierzy A w postaci:

$$A_{11} = \begin{bmatrix} a_{11} & a_{12} \cdots a_{1,n/2} \\ a_{21} & a_{22} \cdots a_{2,n/2} \\ & \vdots \\ a_{n/2,1} & \cdots a_{n/2,n/2} \end{bmatrix}$$

$$\overset{\leftarrow n/2 \rightarrow}{\underset{n/2}{\updownarrow}} \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Korzystając z metody Strassena można w pierwszej kolejności policzyć:

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

gdzie wykonywane operacje to teraz dodawanie i mnożenie macierzy. Podobnie liczymy M_2 do M_7 , a następnie:

$$C_{11} = M_1 + M_4 – M_5 + M_7$$

oraz C_{12} , C_{21} , C_{22}.

Jeśli założymy, że:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix} \quad B = \begin{bmatrix} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{bmatrix}$$

to obliczenia przebiegają następująco:

$$M_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$$
$$= \left(\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \right) \times \left(\begin{bmatrix} 8 & 9 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 9 & 1 \\ 4 & 5 \end{bmatrix} \right)$$
$$= \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix}$$

$$\overset{\leftarrow 2 \rightarrow}{\underset{2}{\updownarrow}} \left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array} \right] = \left[\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{array} \right] \times \left[\begin{array}{cc|cc} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ \hline 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{array} \right]$$

Kiedy macierze są małe, mnożenie wykonujemy w standardowy sposób (w naszym przykładzie dla n=2). Stąd:

$$M_1 = \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix}$$
$$= \begin{bmatrix} 3 \times 17 + 5 \times 7 & 3 \times 10 + 5 \times 9 \\ 11 \times 17 + 13 \times 7 & 11 \times 10 + 13 \times 9 \end{bmatrix} = \begin{bmatrix} 86 & 75 \\ 278 & 227 \end{bmatrix}$$

W ten sam sposób możemy policzyć wartości od M_2 do M_7 , a potem wartości C_{11} , C_{12} , C_{21} , C_{22}. Połączone dają C.

Algorytm mnożenia macierzy kwadratowych - metoda Strassena.

Problem: określ iloczyn dwóch macierzy o wymiarach $n \times n$, gdzie n jest potegą liczby 2

Dane wejściowe: liczba całkowita n, będąca potęgą liczby 2 oraz dwie macierze A i B o wymiarach $n \times n$.

Wyniki: iloczyn C macierzy A i B.

```
void strassen (int n,
               matrix A,
               matrix B,
               matrix& C)
{
  if(n<=threshold)
    oblicz C=A×B za pomocą algorytmu standard;
  else
    podziel A na cztery podmacierze A11,A12,
    A21, A22;
    podziel B na cztery podmacierze B11,B12,
    B21, B22;
    oblicz C=A×B przy użyciu metody Strassena;
    // np. strassen(n/2,A11+A22,B11+B12,M1);
}
```

Wartość zmiennej *threshold* to punkt, w którym uznajemy, że bardziej wydajne jest użycie algorytmu standardowego.

Elementarna liczba operacji mnozenia oraz dodawania/odejmowania jest $\sim n^{2.81}$. Opracowano również algorytm z liczbą operacji $\sim n^{2.38}$, ale wartość zmiennej *threshold* jest duża.

Przeciwskazania do używania metody dziel i zwyciężaj

Należy unikać metody *dziel i zwyciężaj* gdy:

- Realizacja o rozmiarze *n* jest dzielona na dwie lub większą liczbę realizacji, z których niemal każda ma rozmiar *n*
- Realizacja o rozmiarze *n* jest dzielona na niemal *n* realizacji o rozmiarze *n/c*, gdzie *c* jest stałą.

Przykładowo algorytm liczenia *n*-tego wyrazu ciagu Fibonacciego (wersja rekurencyjna), jest algorytmem typu *dziel i zwyciężaj,* który dzieli realizacje obliczającą *n*-ty wyraz na dwie pod-realizacje, które obliczają odpowiednio *(n-1)*–ty i *(n-2)*–ty wyraz. Liczba wyrazów obliczanych przez ten algorytm jest wykładnicza w stosunku do *n* , natomiast liczba wyrazów obliczanych przez wersje iteracyjna jest linowa w stosunku do *n*.

Z drugiej strony jeśli problem ma charakter wykładniczy to nie ma powodu by unikać prostego rozwiązania typu *dziel i zwyciężaj,* jak chociażby w problemie weż Hanoi.