Do czego potrzebne jest sortowanie

Gdy mamy jakiś zbiór danych, typowo nieuporządkowany, może zachodzić potrzeba jego uporządkowania, inaczej **posortowania**. Typowe przypadki:

- Potrzeba porządkowania może wynikać z wymagań prezentacji. Na przykład, zestawienie transakcji bankowych powinno być uporządkowane chronologicznie, bo inaczej nie da się prześledzić kolejnych transakcji, ani sprawdzić poprawności obliczenia salda.

- W uporządkowanych danych o wiele sprawniej można wykonywać **wyszukiwanie**. Jeżeli jakiś zbiór danych jest uporządkowany według pewnego **klucza**, to znalezienie pozycji z tym kluczem można wykonać metodą **wyszukiwania binarnego** (lub jakiejś jego odmiany). Takie wyszukiwanie ma efektywność asymptotyczną przypadku najgorszego (jak również średniego) $\Theta(\log_2 n)$, w odróżnieniu od efektywności $\Theta(n)$ dla wyszukiwania liniowego w zbiorze nieuporządkowanym.

  Rozważmy ponownie przykład tablicy z milionem elementów. Wyszukiwanie binarne w takiej tablicy, gdy jest ona uporządkowana, wymaga maksymalnie 20 porównań elementów z wyszukiwanym kluczem. Dla porównania, wyszukiwanie liniowe może wymagać maksymalnie miliona porównań, lub pół miliona dla przypadku średniego.

  Zwłaszcza gdy takich wyszukiwań trzeba realizować dużo, sumaryczny czas obliczeń będzie dramatycznie gorszy w przypadku nieuporządkowanym.

Przegląd algorytmów sortowania

Przyjrzymy się trzem najbardziej popularnych spośród „mocnych” algorytmów sortowania, to znaczy takich o asymptotycznym czasie działania $\Theta(n \lg n)$:

Algorytm	Czas przyp. najgorszego	Czas przyp. średniego	Pamięć dodatkowa	Stabilny
wstawianie	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$	tak
scalanie	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Omega(n)$	tak
kopcowanie	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(1)$	nie
quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$	$\Theta(1)/\Theta(\lg n)$	nie(tak?)
zliczanie	$\Theta(k+n)$	$\Theta(k+n)$	$\Omega(k+n)$	tak
pozycyjne	$\Theta(d(n+k))$	$\Theta(d(n+k))$	$\Omega(n+k)$	tak
kubełkowe	$\Theta(n^2)$	$\Theta(n)$	$\Omega(n)$	tak

Sortowanie w miejscu — gdy algorytm sortowania nie wykorzystuje dodatkowej pamięci, poza pamięcią samej sekwencji sortowanej, z wyjątkiem stałej liczby zmiennych pomocniczych.

Sortowanie stabilne — gdy algorytm sortowania gwarantuje, że elementy o równych wartościach klucza pozostają w tej samej kolejności w sekwencji wynikowej. Stabilność często jest cechą konkretnej implementacji, a nie samego algorytmu, jednak dla pewnych algorytmów jest bardzo trudna bądź niemożliwa do zachowania, a dla innych jest to proste.

Podejście dziel-i-rządź

Poznany wcześniej algorytm sortowania przez scalanie wykorzystywał podejście **inkrementacyjne**. To znaczy dla każdego elementu przetwarzał całą tablicę tak, aby ponownie była posortowana.

Inne podejście, bardzo często pojawiające się w algorytmach informatycznych, realizuje symboliczną zasadę: **dziel-i-rządź**. To znaczy, pierwotny problem jest w jakiś sposób dzielony na mniejsze podproblemy, następnie te są rozwiązywane rekurencyjnie, czyli tą samą metodą, a na koniec rozwiązania podproblemów są ponownie jakimś sposobem łączone w końcowe rozwiązanie oryginalnego problemu.

Oczywiście, w ogólnym przypadku nie jest oczywiste ani nawet jasne jak przeprowadzić każdą z tych części, to znaczy:

- podzielić oryginalny problem na podproblemy,
- rekurencyjnie rozwiązać podproblemy,
- połączyć rozwiązania podproblemów w końcowe rozwiązanie oryginalnego problemu.

Poznamy szereg takich algorytmów, za każdym razem analizując te trzy kluczowe elementy. Jednak ogólnie to podejście sprawdza się w znacznej części kluczowych algorytmów informatycznych.

Sortowanie przez scalanie

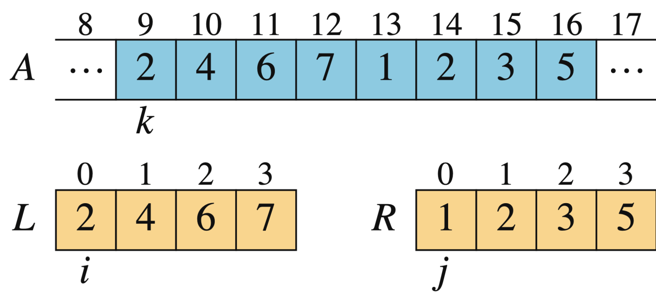
Idea wykorzystania podejścia dziel-i-rządź do sortowania zbioru elementów jest prosta: podziel zbiór na mniejsze podzbiory, posortuj je indywidualnie, i na końcu **scal** mniejsze posortowane zbiory w pełny posortowany zbiór.

Co więcej, dwa pierwsze pierwsze kroki są również względnie proste. Tablica jest dzielona na połówki, i każda jest sortowana rekurencyjnie. Poprawność rekurencji gwarantuje jej warunek stopu, to znaczy próba sortowania tablicy zredukowanej do jednego, lub nawet zera elementów, które nie mogą być nieposortowane.

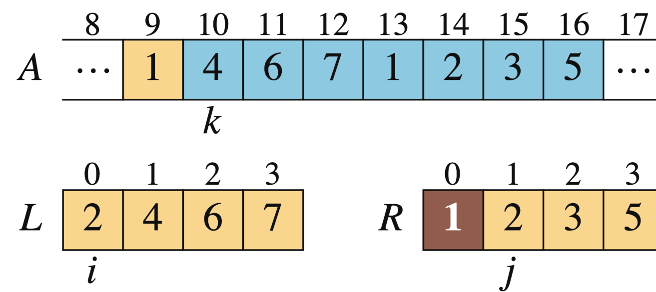
```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

Trochę trudniejszy jest ostatni krok, czyli połączenie posortowanych połówek w całość.

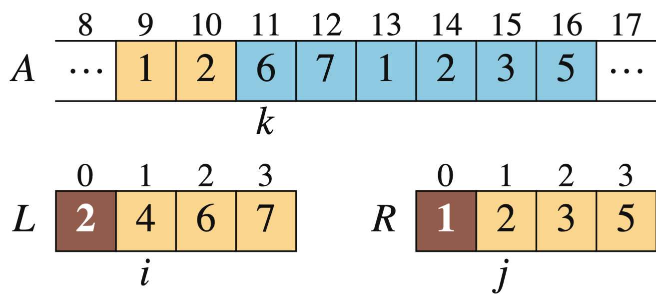
```
MERGE(A, p, q, r)
 1  n_L = q - p + 1              // length of A[p : q]
 2  n_R = r - q                  // length of A[q + 1 : r]
 3  let L[o : n_L - 1] and R[0 : n_R - 1] be new arrays
 4  for i = 0 to n_L - 1         // copy A[p : q] into L[0 : n_L - 1]
 5      L[i] = A[p + i]
 6  for j = 0 to n_R - 1         // copy A[q + 1 : r] into R[0 : n_R - 1]
 7      R[j] = A[q + j + 1]
 8  i = 0                        // i indexes the smallest remaining element in L
 9  j = 0                        // j indexes the smallest remaining element in R
10  k = 0                        // k indexes the location in A to fill
11  // As long as each of the arrays L and R contains an unmerged element,
    //     copy the smallest unmerged element back into A[p : r].
12  while i < n_L and j < n_R
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
18      k = k + 1
19  // Having gone through one of L and R entirely, copy the
    //     remainder of the other to the end of A[p : r]
20  while i < n_L
21      A[k] = L[i]
22      i = i + 1
23      k = k + 1
24  while j < n_R
25      A[k] = R[j]
26      j = j + 1
27      k = k + 1
```



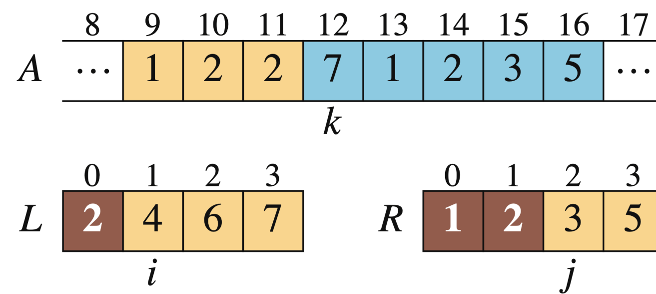
(a)



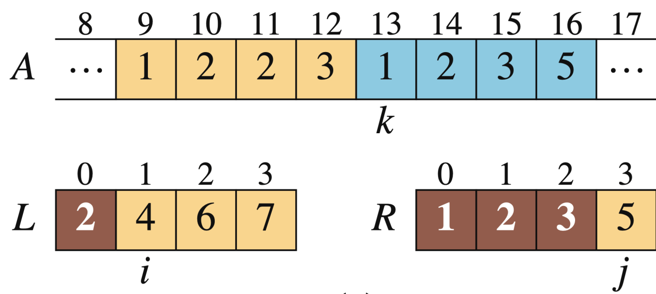
(b)



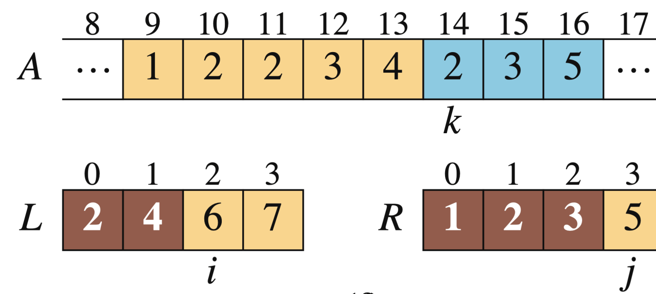
(c)



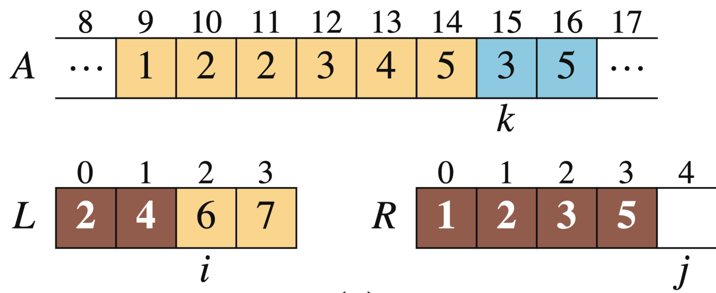
(d)



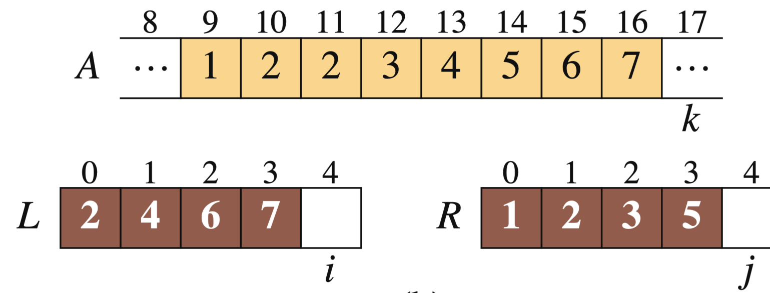
(e)



(f)



(g)



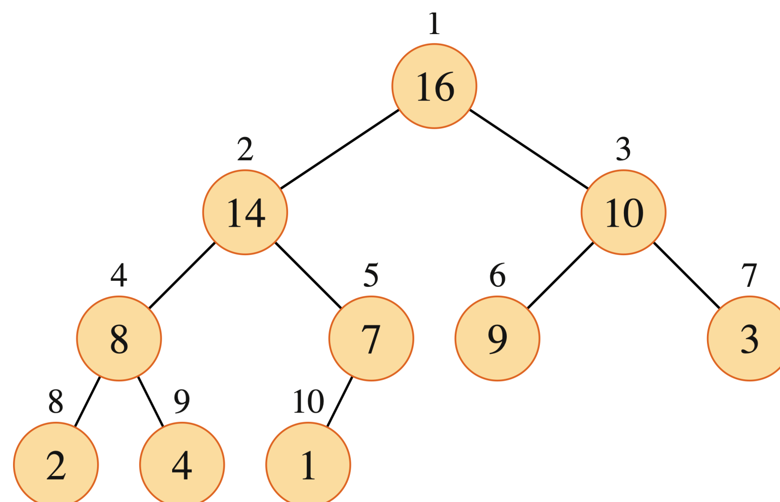
(h)

Kopce

Sortowanie kopcowe, któremu chcemy się obecnie przyjrzeć, wykorzystuje specjalną strukturę danych — drzewo binarne o pewnej szczególnej własności — zwane **kopcem** (*heap*). Przed analizą samego algorytmu sortowania, musimy zapoznać się z kopcami, ich właściwościami, i sposobem ich budowy.

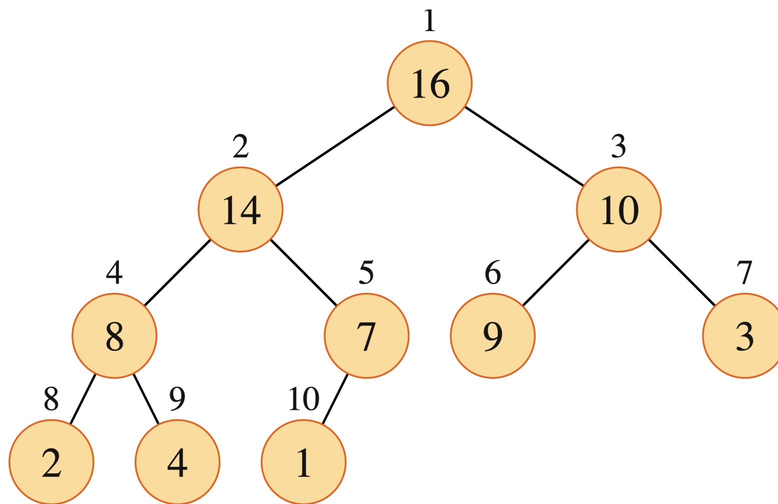
Tym samym wyprzedzimy tu nieco analizę struktur danych, w szczególności drzew binarnych (i nie tylko), dla celów tego szczególnego algorytmu sortowania. Do kopców wrócimy później aby poznać ich inne zastosowania.

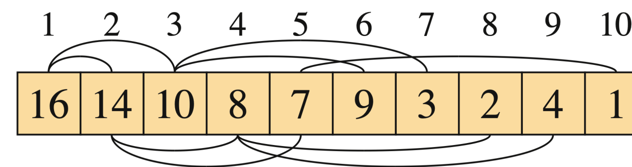
Kopcem nazywamy drzewo binarne, w którym klucze nie maleją wzdłuż każdej ścieżki od liści do korzenia drzewa. Podstawową własnością kopca jest, że jego maksymalnym elementem jest jego korzeń. Inną ciekawą własnością jest, że każde poddrzewo kopca jest również kopcem.



Reprezentacja kopca

Kopce, podobnie jak inne drzewa używane w roli struktur danych, można budować jako struktury danych wskaźnikowe. Jednak istnieje konstrukcja tablicy, którą można uważać za reprezentację kopca.





Elementy kopca z drzewa po lewej są upakowane w tablicy warstwami, od lewej do prawej (numerki przy węzłach drzewa odpowiadają indeksom elementów tablicy).

Zachodzą własności:

$$
\begin{aligned}
\textsc{Lewy-Potomek}(i) &= 2i \\
\textsc{Prawy-Potomek}(i) &= 2i+1 \\
\textsc{Rodzic}(i) &= \lfloor i/2 \rfloor
\end{aligned}
$$

Podane wzory pozwalają łatwo poruszać się po kopcach zapisanych w tablicach zgodnie z powyższym schematem.

Budowa kopca (1)

Przedstawiona procedura budowy kopca zakłada, że tablica A zawiera elementy kopca w zakresie indeksów od 1 do A.*heap-size*, w dowolnej kolejności. Najpierw rozważymy procedurę pomocniczą MAX-HEAPIFY, która pracując na elementach tablicy, i zamieniając jej elementy miejscami w razie potrzeby, docelowo tworzy w niej kopiec.

O takim sposobie pracy mówimy, że przetworzenie tablicy w kopiec odbywa się **w miejscu**.

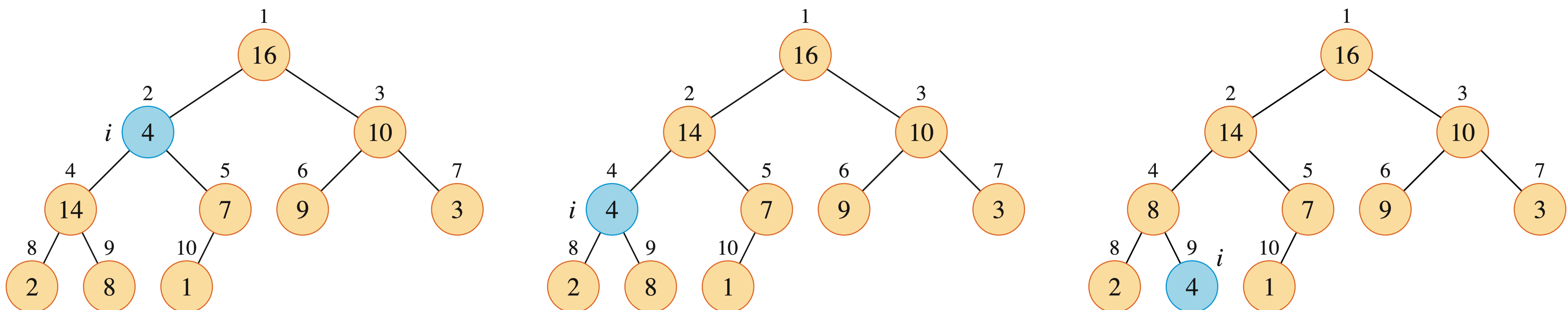
Procedura jest rekurencyjna i zakłada, że w chwili wywołania dla węzła i kopca oba poddrzewa węzła i są poprawnymi kopcami (jeśli są niepuste). Jedyne naruszenie własności kopca może występować w samym korzeniu i. To znaczy, wartość w korzeniu może być mniejsza od wartości w którymś z potomków (lub od wartości obu).

Procedura sprawdza to, i w przypadku naruszenia własności kopca zamienia wartość z korzenia z większym spośród potomków. Wartość w korzeniu wtedy staje się większa, więc nie koliduje z drugim z potomków. Jednak w poddrzewie z zamienioną wartością jego korzenia własność kopca mogła zostać naruszona, zatem po takiej zamianie procedura wywołuje sama siebie rekurencyjnie, aby przywrócić własność kopca w tym jednym potomku.

Czas pracy tej procedury w najgorszym przypadku wynosi $O(\log_2 n)$.

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)                                         // equal to 2i
 2  r = RIGHT(i)                                        // equal to 2i + 1
 3  if l ≤ A.heap-size and A[l] > A[i]
 4      largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7      largest = r
 8  if largest ≠ i
 9      exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

Przykład: wywołanie MAX-HEAPIFY dla $i = 2$ wykrywa naruszenie własności kopca ($largest = 4$). Po zamianie $A[2]$ z $A[4]$ rekur. wywołanie dla $i = 4$ ponownie wykrywa naruszenie ($largest = 9$). Po zamianie $A[4]$ z $A[9]$ własność kopca jest przywrócona.



Budowa kopca (2)

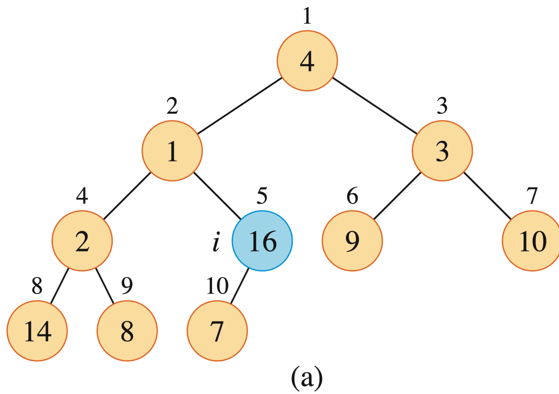
Mamy już gotową procedurę budującą poprawne kopce w tablicy, gdzie poddrzewa niższego rzędu są już kopcami. Aby zbudować pełny kopiec w całej tablicy wystarczyłoby wywołać tę procedurę dla kolejnych elementów kopca od najniższych poziomów drzewa, czyli jego **liści**, w górę. Albo, posługując się terminologią reprezentacji tablicowej, od końca tablicy, gdzie zapisane są liście, do jej początku, gdzie w węźle numer 1 zapisany jest korzeń kopca.

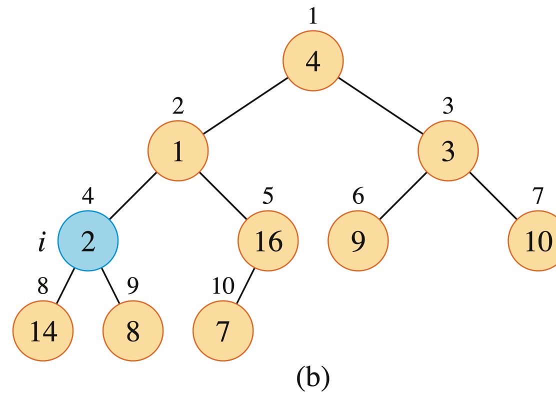
Ten proces można jeszcze minimalnie usprawnić przez zauważenie (patrz literatura), że w każdym zapisanym w tablicy kopcu zawierającym n elementów, jej elementy od $A[\lfloor n/2 \rfloor + 1]$ do $A[n]$ są wszystkie liśćmi kopca. Zatem są też kopcami (trywialnymi), i budowanie kopca można rozpocząć od elementu $\lfloor n/2 \rfloor$ idąc w górę kopca, czyli w dół sekwencji elementów tablicy:

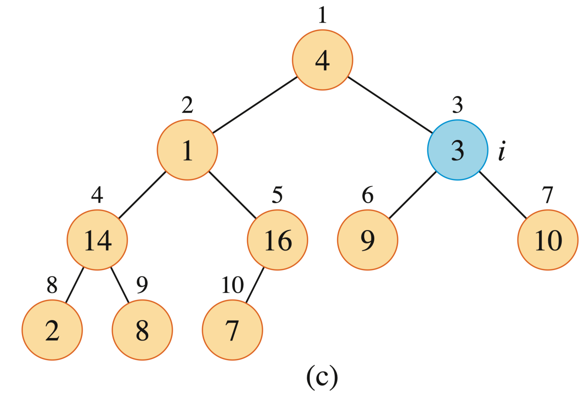
```
BUILD-MAX-HEAP(A, n)
1  A.heap-size = n
2  for i = ⌊n/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

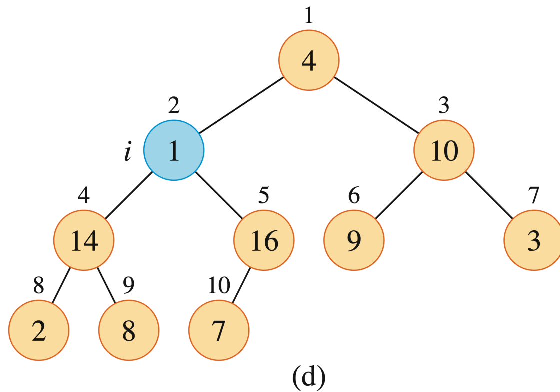
Budowa kopca — przykład

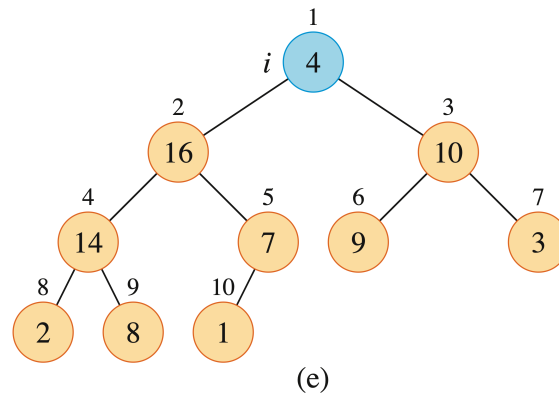
A	4	1	3	2	16	9	10	14	8	7

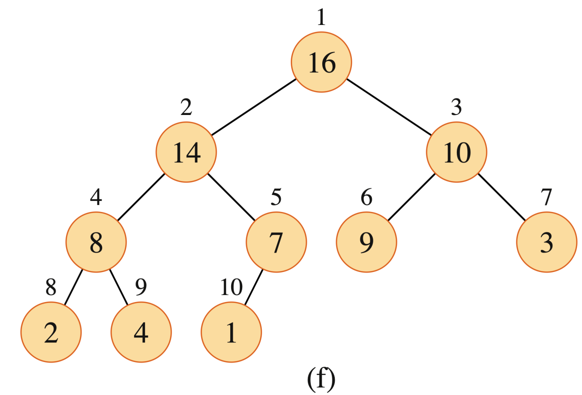

(a)


(b)


(c)


(d)


(e)


(f)

Rysunki pokazują tylko wywołania procedury BUILD-MAX-HEAP dla kolejnych węzłów kopca — elementów tablicy od 5 do 1. Wynik każdego wywołania widać na kolejnej postaci kopca.

Sortowanie przez kopcowanie

Po zastosowaniu przedstawionych algorytmów budowy kopca otrzymujemy w tablicy poprawny kopiec, z największym elementem w pozycji $A[1]$.
Jak stąd przejść do tablicy posortowanej w porządku rosnącym?
Czy nie należało zbudować kopca w odwrotnym porządku?

Okazuje się, że nie, i droga do posortowania tablicy jest już bardzo krótka.

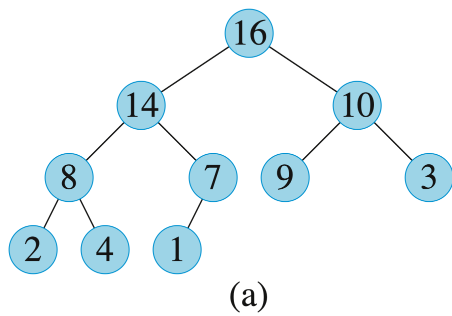
Mając maksymalny element kopca (tablicy) w pozycji $A[1]$ wystarczy umieścić go na swoim właściwym miejscu, to znaczy w pozycji $A[n]$. Natomiast dotychczasowy element $A[n]$ umieścić w ... jedynej wolnej obecnie pozycji, czyli $A[1]$.

To naruszy poprawność kopca, ale tylko w korzeniu, i tę poprawność można przywrócić procedurą Max-Heapify, zauważając, że oba poddrzewa kopca są nadal poprawne.

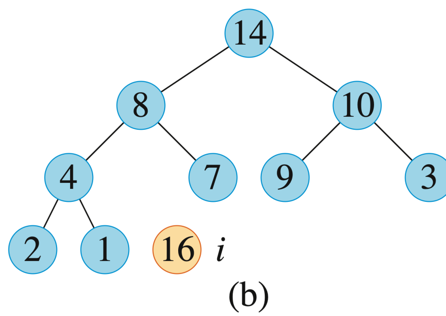
To znaczy, z wyjątkiem elementu $A[n]$, ale ten element jest już w swojej właściwej pozycji w docelowej tablicy, i najlepiej byłoby go już w ogóle nie ruszać. Jednocześnie, ten element jest „ostatnim" liściem kopca, i można łatwo wyłączyć go z kopca, skracając wielkość kopca o jeden element.

Gdy to zrobimy, i przywrócimy poprawność zmniejszonemu kopcowi, to w pozycji $A[1]$ znajdzie się drugi największy element kopca (tablicy), który docelowo powinien znaleźć się na pozycji $A[n-1]$. Możemy teraz powtórzyć poprzednią procedurę, to znaczy, zamienić miejscami $A[1]$ z $A[n-1]$, skrócić kopiec znowu o 1, i znów przywrócić mu własność kopca. I tak dalej, do ostatniego elementu.

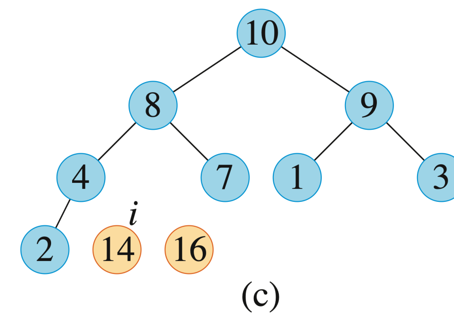
```
HEAPSORT(A, n)
1  BUILD-MAX-HEAP(A, n)
2  for i = n downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```



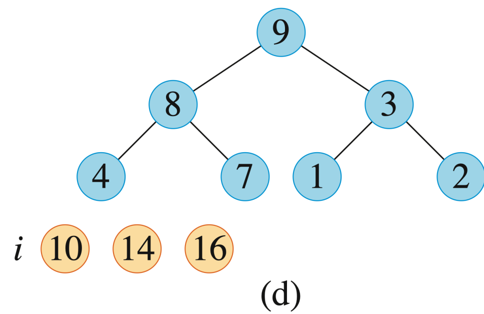
(a)



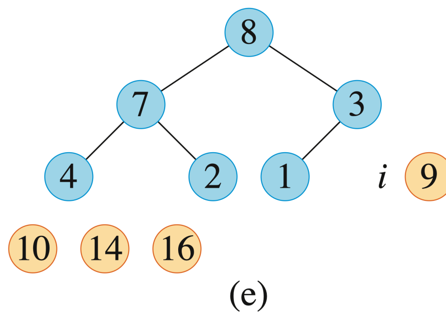
(b)



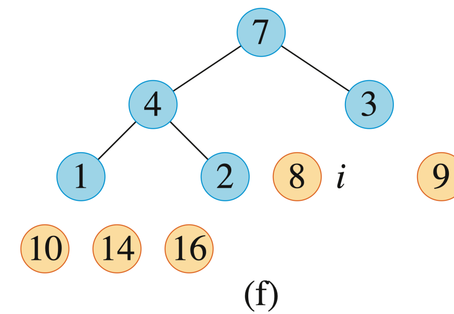
(c)



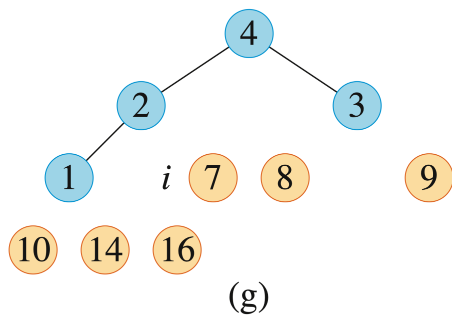
(d)



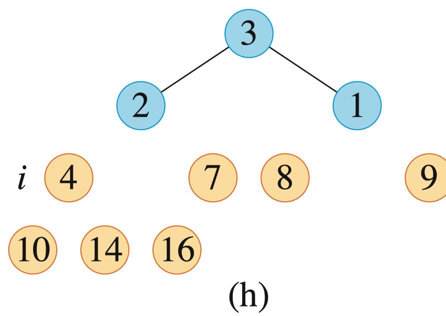
(e)



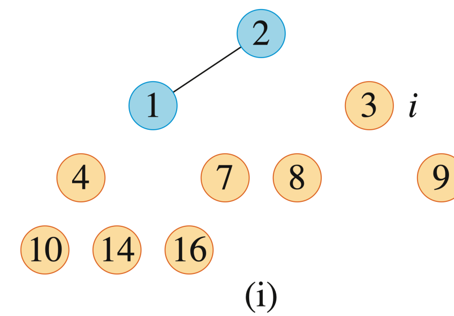
(f)



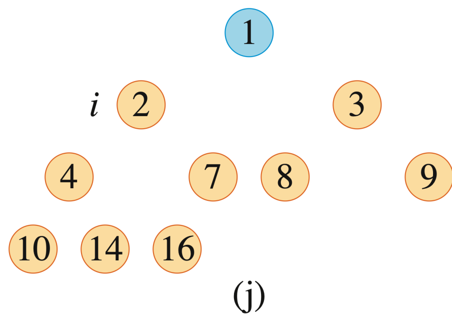
(g)



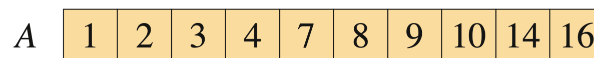
(h)



(i)



(j)



(k)

Kopce — uwagi terminologiczne

Pojęcie kopca związane jest z kilkoma dwuznacznościami terminologicznymi.

Pierwsza dwuznaczność istnieje już w języku angielskim. Słowo *heap* używane jest w kontekście struktur danych jako określenie drzewa binarnego zgodnego z powyższą definicją (kopca). Jednocześnie w kontekście struktur alokacji pamięci oznacza ono zakres pamięci przydzielany w sposób nieuporządkowany, w odróżnieniu od **stosu** (*stack*). W tym drugim kontekście, słowo *heap* jest najczęściej tłumaczone na polski jako **sterta**.

W języku polskim istnieją dodatkowe dwuznaczności. Struktura danych określona w tej prezentacji mianem kopca (*heap*) bywa/bywała w polskiej literaturze informatycznej nazywana **stogiem** a także **stertą**.

Sortowanie szybkie quicksort

Quicksort jest algorytmem o bardzo szczególnych własnościach. W przypadku średnim algorytm ma czas działania $\Theta(n \log n)$ podobnie jak dwa poprzednie, ale stałe, których nie widać w tej złożoności asymptotycznej, są w nim szczególnie małe. A ponieważ większość praktycznych zadań sortowania wykonywanych jest na skończonych zbiorach danych, algorytm bardzo często okazuje się najlepszy. A ponieważ sortuje **w miejscu**, to bardzo dobrze sprawdza się w wielu różnych aplikacjach.

Jednak w przypadku najgorszym podstawowa wersja algorytmu działa w czasie $\Theta(n^2)$!! Przyczyna tego zachowania kryje się w kluczowym dla algorytmu pomyśle, by podzielić sortowaną tablicę na dwie części: elementów mniejszych i większych, i zacząć od wyboru **elementu rozdzielającego** (ang. *pivot*). Gdy ten element zostanie wybrany nieszczęśliwie, i np. podzieli tablicę z milionem elementów w ten sposób, że w części mniejszej znajdzie się tylko kilka elementów, a cała reszta w drugiej części tablicy, i następnie dalsze podziały w tym wywołaniu sortowania będą równie nieszczęśliwe, to czas pracy algorytmu na całej tablicy będzie $\Theta(n^2)$.

Zatem kluczowa dla tego algorytmu jest metoda wyboru punktu rozdzielającego. Z tego względu powstało wiele wersji algorytmu z różnymi metodami jego wyboru. Te metody mają różne własności, jednak żadna nie przenosi algorytmu do klasy $\Theta(n \log n)$ w przypadku najgorszym. Stabilność algorytmu zależy od tej metody, ale dla większości z nich algorytm jest niestabilny.

Quicksort — podstawowy algorytm

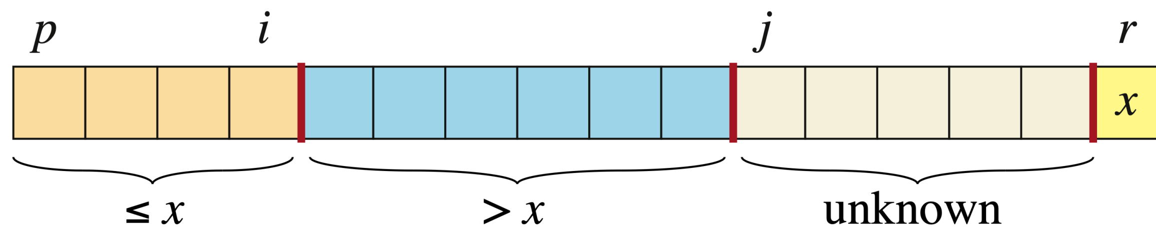
Quicksort jest kolejnym przypadkiem algorytmu typu dziel-i-rządź. Idea działania polega na tym, by podzielić tablicę na elementy „mniejsze” od pewnej wybranej wartości (pivot) i przenieść je do początkowej części tablicy, oraz elementy „większe”, które znajdą się w końcowej części tablicy. Następnie obie części są rekurencyjnie sortowane. A ponieważ znajdują się już w „swoich” zakresach docelowej tablicy, to tych posortowanych części nie trzeba już łączyć, w odróżnieniu od sortowania przez scalanie. Zatem na tym działanie algorytmu się kończy, ponieważ cała tablica jest posortowana.

```
QUICKSORT(A, p, r)
1  if p < r
2        // Partition the subarray around the pivot, which ends up in A[q].
3        q = PARTITION(A, p, r)
4        QUICKSORT(A, p, q − 1)  // recursively sort the low side
5        QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

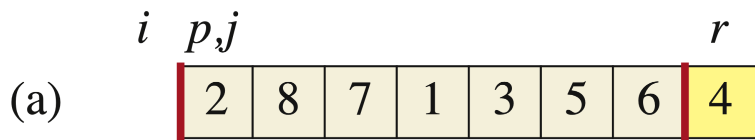
Jak wspomnieliśmy wcześniej, i co widać w postaci algorytmu, cała tajemnica działania algorytmu kryje się w procedurze PARTITION.

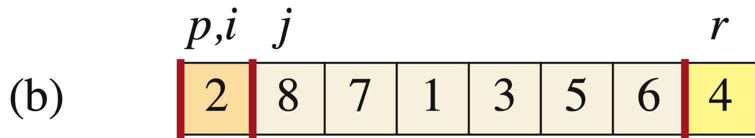
Procedura musi wybrać punkt podziału, oraz poprzenosić elementy większe od niego na koniec tablicy, a mniejsze na początek. Ta wersja nie podejmuje problemu „dobrego” wyboru elementu rozdzielającego, i wykorzystuje w tej roli element ostatni.

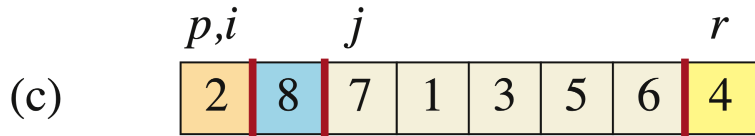
Ideę działania algorytmu PARTITION ilustruje poniższy rysunek, który można zarazem traktować jako warunek niezmiennika pętli algorytmu:

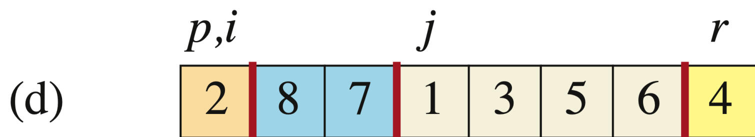


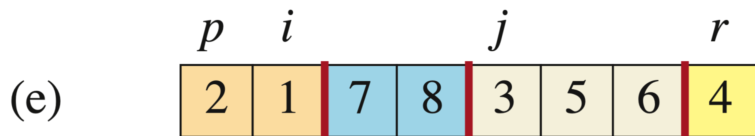

```
PARTITION(A, p, r)
1  x = A[r]                          // the pivot
2  i = p - 1                         // highest index into the low side
3  for j = p to r - 1                // process each element other than the pivot
4      if A[j] ≤ x                   // does this element belong on the low side?
5          i = i + 1                 // index of a new slot in the low side
6          exchange A[i] with A[j]   // put this element there
7  exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
8  return i + 1                      // new index of the pivot
```

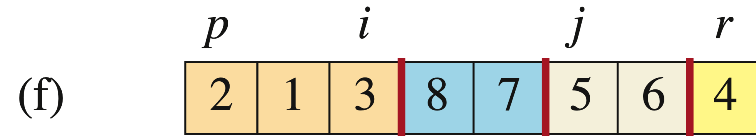

i p,j r
(a)
2 8 7 1 3 5 6 4

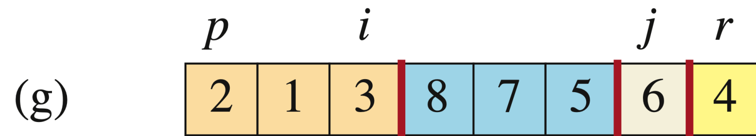

p,i j r
(b)
2 8 7 1 3 5 6 4

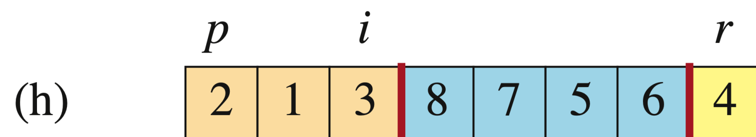

p,i j r
(c)
2 8 7 1 3 5 6 4

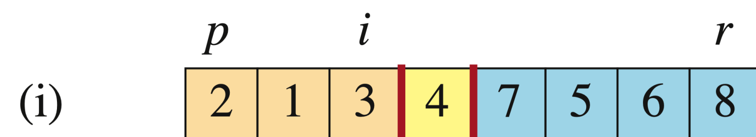

p,i j r
(d)
2 8 7 1 3 5 6 4


p i j r
(e)
2 1 7 8 3 5 6 4


p i j r
(f)
2 1 3 8 7 5 6 4


p i j r
(g)
2 1 3 8 7 5 6 4


p i r
(h)
2 1 3 8 7 5 6 4


p i r
(i)
2 1 3 4 7 5 6 8

Quicksort — wybór elementu rozdzielającego

Wybór elementu rozdzielającego (*pivot*) jest krytyczny dla efektywności algorytmu, ponieważ to on decyduje, czy jego czas działania będzie się utrzymywał w „normie", czyli zgodnie ze średnią, czy też będzie się degenerował.

Dla zbiorów danych losowych, arbitralny wybór *pivot*, na przykład taki jak w przedstawionej wyżej wersji procedury PARTITION da wynik średni, czyli pożądany.

Jednak już, na przykład, dla zbioru wartości identycznych, algorytm się zdegeneruje. Podobnie będzie dla zbioru już posortowanego, obojętnie rosnąco, czy malejąco.

Najlepszą sytuacją dla algorytmu byłoby wybieranie elementu równego **medianie** zbioru wartości, czyli wartości dzielącej zbiór na dwie równe (lub prawie) części. Dlatego często stosowane implementacje procedury PARTITION, w celu zwiększenia szansy na „trafienie" w taki element pivot, stosują wyrafinowane heurystyki (zgadywanie) tego wyboru, jak na przykład środkowa wartość z elementów: pierwszego, ostatniego, i środkowego, tablicy, a ponadto elementy randomizacji (losowości). Dodatkowo, specjalne procedury wyboru stosowane są dla ustrzeżenia się przed degeneracją w przypadku wszystkich równych elementów.

Krótkie podsumowanie — pytania sprawdzające

1. Wzorując się na rysunku 2.4 z podręcznika CLRS przedstaw działanie algorytmu MERGE-SORT na sekwencji $\langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

2. Jaka jest minimalna i jaka maksymalna liczba elementów, jakie mogą być zapisane w kopcu o wysokości h (zawierającego h poziomów)?

3. Czy tablica liczb posortowanych malejąco jest zawsze kopcem?

4. Wzorując się na rysunkach z przykładu na stronie 10, przedstaw działanie procedury MAX-HEAPIFY na tablicy $A = \{27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0\}$.

5. Wzorując się na rysunkach z przykładu na stronie 12, przedstaw działanie procedury BUILD-MAX-HEAP na tablicy $A = \{5, 3, 17, 10, 84, 19, 6, 22, 9\}$.

6. Wzorując się na rysunkach z przykładu na stronie 15, przedstaw działanie procedury HEAPSORT na tablicy $A = \{5, 13, 2, 25, 7, 17, 20, 8, 4\}$.

7. Wzorując się na rysunkach z przykładu na stronie 15, przedstaw działanie procedury PARTITION na tablicy $A = \{13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11\}$.

Literatura i materiały pomocnicze

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L Rivest, Clifford Stein: Wprowadzenie do algorytmów, PWN, 2024, rozdziały 2.3, 6, i 7.