W. 3
Sortowanie

Problem

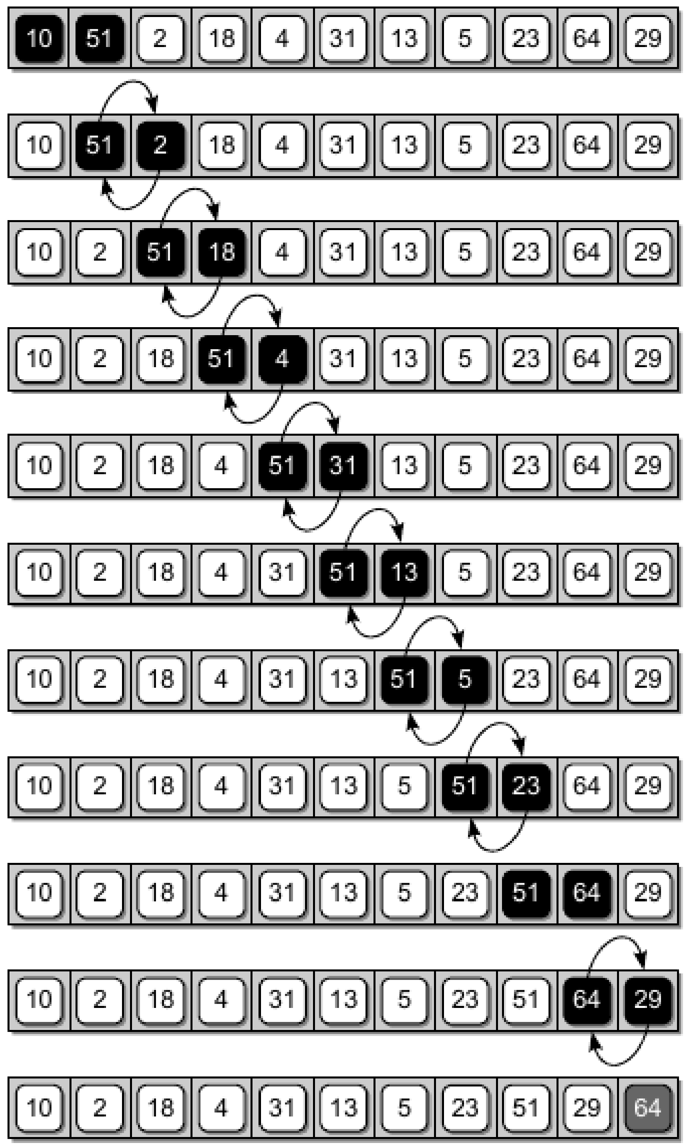
- Należy uporządkować tablicę liczb, aby wszystkie sąsiednie pary elementów spełniały pewną relację porządku (np. relację mniejszości).
- Złożoność obliczeniowa algorytmów sortowania:
  - O(n*ln(n)) – najszybsze;
  - O(n^2) – typowe;
- Uwaga:
  - Istnieją algorytmy sortowania w czasie liniowy O(n). Zobacz Rozdział 8 w książce T. Cormen, „Wstęp do algorytmów”.

Sortowanie bąbelkowe (Bubble sort)

Opis

- Sortowanie bąbelkowe polega na iteracji po tablicy i zamianie kolejności elementów pary, gdy są w niewłaściwym porządku.
- Sortowanie kończy się, gdy w kolejnym przebiegu nie dokonano zamiany elementów.
- Proces przypomina unoszenie się bąbelków w cieczy.
- Przykład działania:
  - https://www.youtube.com/watch?v=lyZQPjUT5B4

Jeden przebieg (bąbelek)



Implementacja (Python)

- def **bubbleSort**(theSeq):
- n = len(theSeq)
- for i in range(n - 1):
- for j in range(i, n - 1) :
- if theSeq[j] > theSeq[j + 1] :
- tmp = theSeq[j]
- theSeq[j] = theSeq[j + 1]
- theSeq[j + 1] = tmp
-
- a = [2,3,5,8,7]
- print("before sorting ", a)
- **bubbleSort**(a)
- print("after sorting ", a)

- Przebieg sortowania:
  - ('before sorting ', [2, 3, 5, 8, 7])
  - ('array before ', 0, 'run: ', [2, 3, 5, 8, 7])
  - ('swap ', 8, ' with ', 7)
  - ('array before ', 1, 'run: ', [2, 3, 5, 7, 8])
  - ('array before ', 2, 'run: ', [2, 3, 5, 7, 8])
  - ('array before ', 3, 'run: ', [2, 3, 5, 7, 8])
  - ('after sorting ', [2, 3, 5, 7, 8])

Implementacja (C++)

- void bubble(int tab[])
- {
- for (int i=1; i<n; i++)
- for (int j=n-1; j>=i; j--)
- if (tab[j]<tab[j-1])
- { // zamiana
-   int tmp=tab[j-1];
-   tab[j-1]=tab[j];
-   tab[j]=tmp;
- }
- }

Złożoność obliczeniowa (najgorszy przypadek)

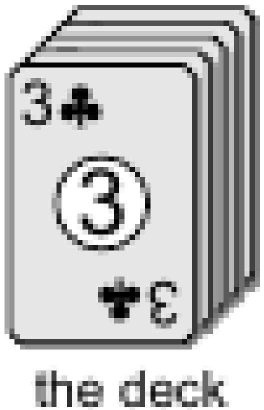
- Zewnętrzna pętla wykonuje się (n-1) razy.
- Wewnętrzna pętla iteruje po (n-1), następnie (n-2), następnie (n-3)....aż w końcu 1 elemencie.
- Zatem:
  - $T(n)=1+2+...+(n-1) = n(n-1)/2$
  - T(n) jest klasy O(n^2).

Sortowanie przez wstawianie
(Insertion sort)

Opis

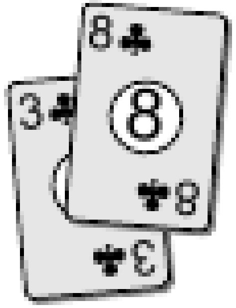
- Sortowanie przez wstawianie naśladuje sortowanie talii kart.
- Przykład działania:
  - https://www.youtube.com/watch?v=ROalU379l3U

Krok 1

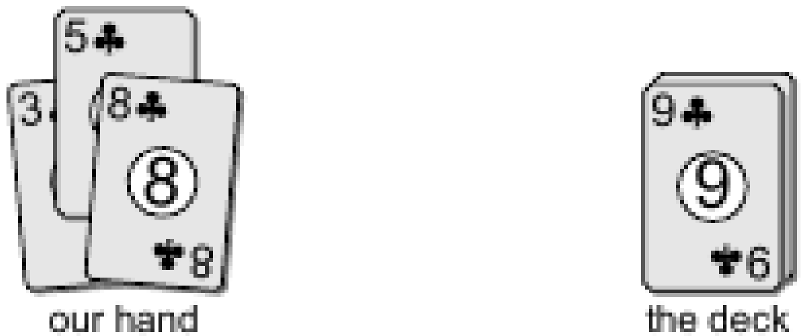


Krok 2

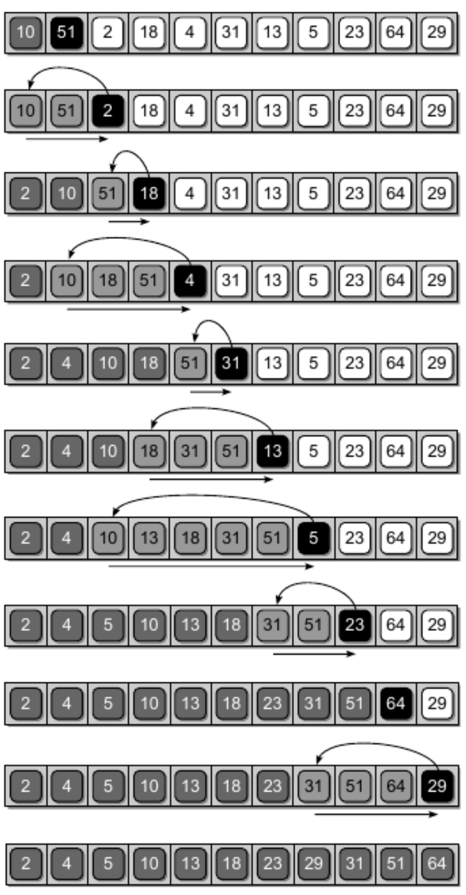
Krok 3



Krok 4



Pełny przebieg



Implementacja (Python)

```
def insertionSort( theSeq ):
    n = len( theSeq )
    for i in range( 1, n ) :
        value = theSeq[i]
        pos = i
        while pos > 0 and value < theSeq[pos - 1] :
            theSeq[pos] = theSeq[pos - 1]
            pos -= 1
            theSeq[pos] = value

a=[2,3,1,5,4]
print("before sort ", a)
insertionSort(a)
print("after sort ", a)
```

- Sekwencja wywołań:
  - ('before sort ', [2, 3, 1, 5, 4])
  - [2, 1, 3, 5, 4]
  - [1, 2, 3, 5, 4]
  - [1, 2, 3, 4, 5]
  - ('after sort ', [1, 2, 3, 4, 5])

Implementacja (C++)

- void InsertSort(int tab[]) {
-
- for(int i=1; i<n; i++)
- {
- int j=i; // 0..i-1 jest juz posortowane
- int temp=tab[j];
- while ((j>0) && (tab[j-1]>temp))
- {
- tab[j]=tab[j-1];
- j--;
- }
- tab[j]=temp;
- }
- }

Złożoność obliczeniowa

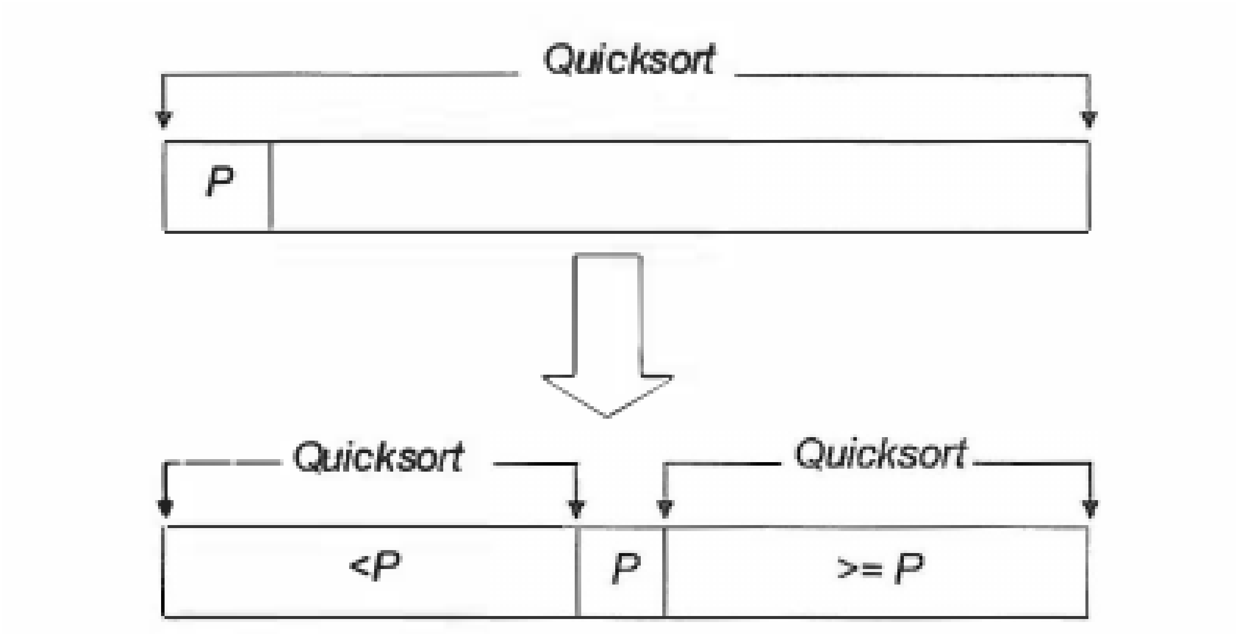
- W najgorszym przypadku odwrotnie posortowanej tablicy złożoność obliczeniowa jest dokładnie taka sama jak dla sortowania bąbelkowego.
- Zatem $T(n)$ jest klasy $O(n^2)$.

Quicksort
Sortowanie szybkie

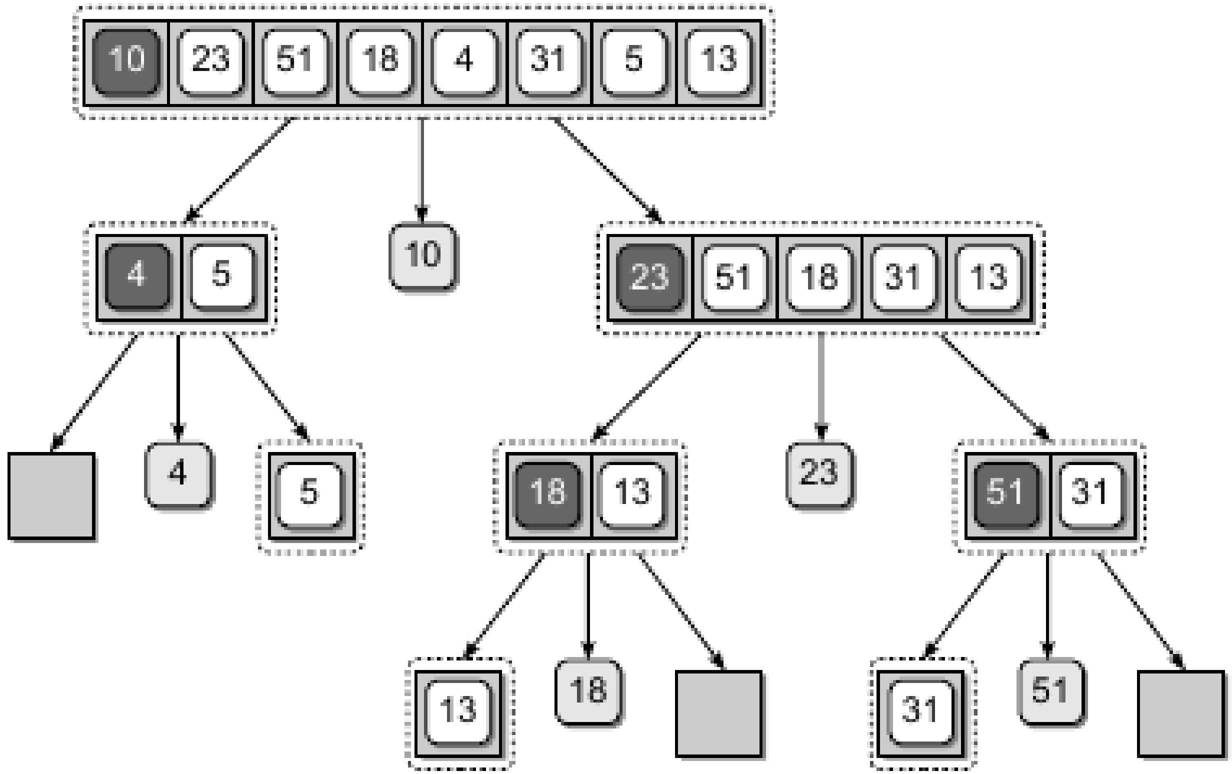
Opis

- Wybieramy element osiowy (pivot) p – może to być pierwszy element tablicy.
- Sortujemy tablicę tak, aby po lewej stronie były elementy mniejsze od p, a po prawej większe.
- Przykład działania:
  - https://www.youtube.com/watch?v=ywWBy6J5gz8

Działanie



Działanie



Najprostsze ale nieefektywne podejście (Python)

```python
def quicksort(xs):
    if xs:
        below = [i for i in xs[1:] if i < xs[0]]
        above = [i for i in xs[1:] if i >= xs[0]]
        return quicksort(below) + [xs[0]] + quicksort(above)
    else:
        return xs
```

- Zauważ, że algorytm jest nieefektywny, bo iterujemy po tablicy xs **dwa razy** tworząc tablice
  - below – elementy mniejsze od pivota xs[0]
  - above – elementy większe lub równe pivotowi xs[0]
- Tworzenie takiego podziału możemy zrobić efektywniej używając algorytmu partycjonowania...

Partycjonowanie (Python)

- def partition(array, begin, end):
- pivot = begin
- **print("before partitioning", array[begin:end+1])**
- for i in range(begin+1, end+1):
- if array[i] <= array[begin]:
- pivot += 1
- array[i], array[pivot] = array[pivot], array[i]
- **print("partitioning", array[begin:end+1])**
- array[pivot], array[begin] = array[begin], array[pivot]
- **print("after partitioning", array[begin:end+1])**
- return pivot

- Partycjonowanie ma na celu podzielenie tablicy na elementy większe i mniejsze od pivota – tutaj pierwszego elementu tablicy.
- Schemat działania partycjonowania:
  - ('before partitioning', [**5**, 2, 8, 9, 1, 3])
  - ('partitioning', [**5, 2**, 8, 9, 1, 3])
  - ('partitioning', [5, **2**, **8**, 9, 1, 3])
  - ('partitioning', [5, 2, **8**, **9**, 1, 3])
  - ('partitioning', [5, 2, **1**, 9, **8**, 3])
  - ('partitioning', [5, 2, 1, **3**, 8, **9**])
  - ('after partitioning', [**3**, 2, 1, **5**, 8, 9])

Quicksort (Python)

- def _**quicksort**(array, begin, end):
-     if begin >= end:
-         return
-     #print("array=", array)
-     **pivot = partition(array, begin, end)**
-     print("pivot index=", pivot, "array=", array)
-     _**quicksort**(array, begin, pivot-1)
-     _**quicksort**(array, pivot+1, end)
-
- #interfejs
- def quicksort(array, begin=0, end=None):
-     if end is None:
-         end = len(array) - 1
-     return _**quicksort**(array, begin, end)
-
- a=[5,2,8,9,1,3]
- print("before sort",a)
- quicksort(a)
- print("after sort",a)

Quicksort (C++)

- // Swap two elements - Utility function
- void swap(int* a, int* b)
- { int t = *a;
- *a = *b;
- *b = t;}

- // partition the array using last element as pivot
- int **partition** (int arr[], int low, int high)
- { int pivot = arr[high]; // pivot
- int i = (low - 1);
- for (int j = low; j <= high- 1; j++)
- { //if current element is smaller than pivot, increment the low element
- //swap elements at i and j
- if (arr[j] <= pivot)
- { i++; // increment index of smaller element
- swap(&arr[i], &arr[j]); } }
- swap(&arr[i + 1], &arr[high]);
- return (i + 1); }

- //quicksort algorithm
- void **quickSort**(int arr[], int low, int high)
- { if (low < high)
- { //partition the array
- int **pivot = partition**(arr, low, high);
- //sort the sub arrays independently
- **quickSort**(arr, low, pivot - 1);
- **quickSort**(arr, pivot + 1, high); } }

- void **displayArray(int arr[], int size)**
- { int i;
- for (i=0; i < size; i++)
- cout<<arr[i]<<"\t"; }

- int **main()**
- { int arr[] = {12,23,3,43,51,35,19,45};
- int n = sizeof(arr)/sizeof(arr[0]);
- cout<<"Input array"<<endl;
- displayArray(arr,n);
- cout<<endl;
- quickSort(arr, 0, n-1);
- cout<<"Array sorted with quick sort"<<endl;
- displayArray(arr,n);
- return 0; }

Złożoność obliczeniowa (najlepszy przypadek)

- Najlepszy przypadek zachodzi, gdy wybierzemy pivot tak, aby po sortowaniu był on elementem środkowym posortowanej tablicy.
- Partycjonowanie wymaga jednej iteracji po tablicy o n elementach, więc jest klasy O(n)
- Następnie rekurencyjnie wywołujemy quicksort po dwóch połowach tablicy:
  - $2T(n/2)$
- Równanie rekurencyjne dla quicksort w najlepszym przypadku to:
  - $T(n) = 2T(n/2) + O(n) = O(n*\log(n))$
- Zatem quicksort w idealnym przypadku ma złożoność klasy O(n*log(n))

Złożoność obliczeniowa (najgorszy przypadek)

- Najgorszy wypadek jest wówczas, gdy pivot okaże się najmniejszym lub największym elementem po posortowaniu.
- Wówczas prawa lub lewa tablica względem pivota jest jednoelementowa, a druga ma (n-1) elementów.
- Partycjonowanie ciągle jest klasy O(n).
- Mamy więc równanie rekurencyjne:
  - $T(n)=T(1)+T(n-1)+O(n)$
- Daje to rozwiązanie (rozwiązać!)
  - $T(n)$ jest klasy $O(n^2)$

Quicksort - Podsumowanie

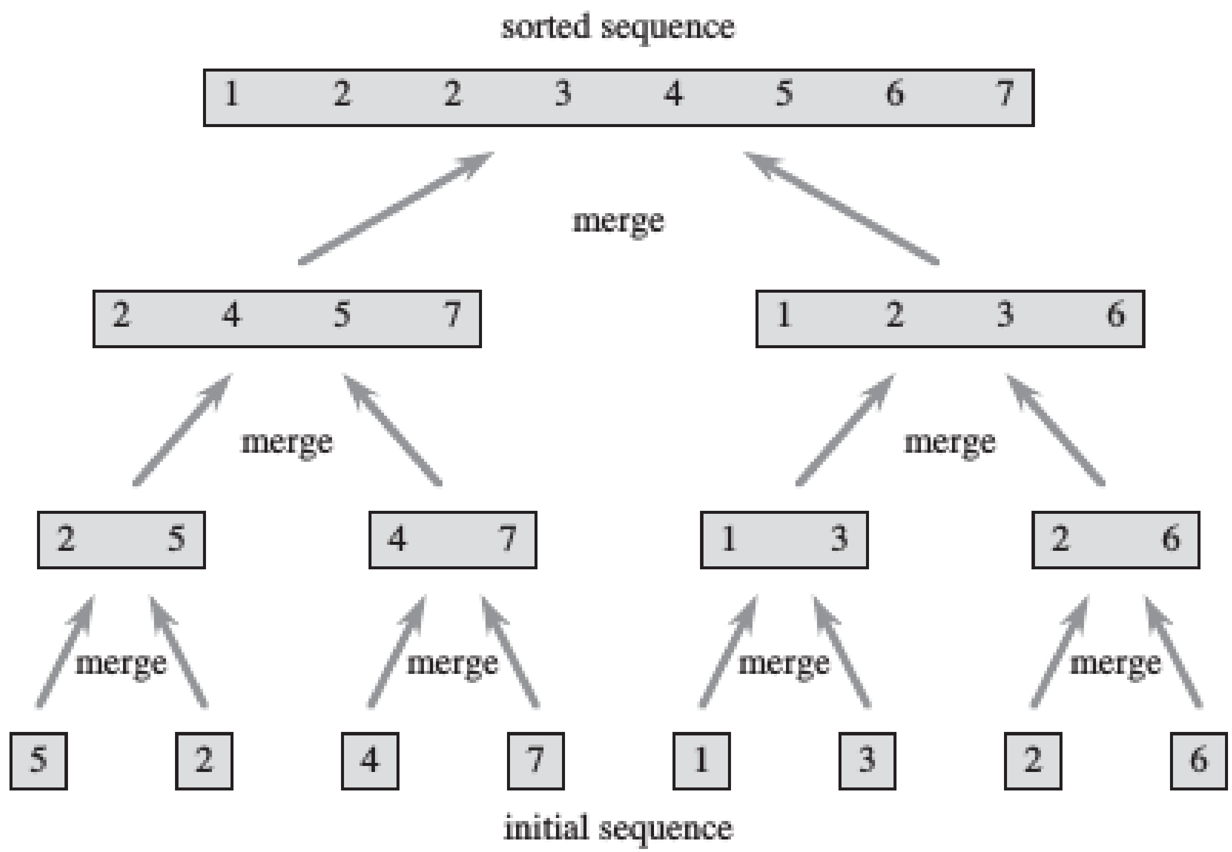
- W najlepszym przypadku algorytm jest klasy O(n*log(n)).
- W najgorszym przypadku może on być klasy O(n^2).

Sortowanie przez scalanie
Mergesort

Opis

- Dzielimy rekurencyjnie tablicę na pojedyncze elementy, a następnie scalamy w odpowiedniej kolejności.
- Przykład działania:
  - https://www.youtube.com/watch?v=XaqR3G_NVoo

Mergesort


sorted sequence
1 2 2 3 4 5 6 7
merge
2 4 5 7
1 2 3 6
merge
merge
2 5
4 7
1 3
2 6
merge
merge
merge
merge
5
2
4
7
1
3
2
6
initial sequence

Algorytm

- Merge-Sort(A,p,r)
  - If p<r
    - q=floor((p+r)/2)   #indeks środka
    - Merge-Sort(A,p,q)   #sortowanie lewej części
    - Merge-Sort(A,q+1,r) #sortowanie prawej części
    - Merge(A,p,q,r)   #łączenie części z uwzględnieniem uporządkowania

Merge (Python)

```
def merge(a, b):
    out = []
    while (len(a) > 0 and len(b) > 0):
        if (a[0] <= b[0]):
            out.append(a[0])
            del a[0]
        else:
            out.append(b[0])
            del b[0]
    while (len(a) > 0):
        out.append(a[0])
        del a[0]
    while (len(b) > 0):
        out.append(b[0])
        del b[0]
    return out

a=[1,2,3]
b=[4,5]
print(merge(a,b))
```

Merge Sort (Python)

- def **half**(arr):
- mid = len(arr) / 2
- return arr[:mid], arr[mid:]
-
- def mergesort(arr):
- if (len(arr) <= 1):
- return arr
- left, right = **half**(arr)
- L = **mergesort**(left)
- R = **mergesort**(right)
- return **merge**(L, R)
-
- a =[3,2,5,4]
- print(mergesort(a))

Merge (C++)

- const int N = 10;
- int T1[N] = {4, 6, 4, 12, -3, 6, -6, 1, 8, 50};
- int T2[N]; // Tablica pomocnicza

- void **merge**(int left, int mid, int right) {
-   int i,j,k;
-   for (i=left; i<=right; i++)
-     T2[i]=T1[i];
-   i=left; j=mid+1; k=left;
-   while (i<=mid && j<=right) {
-     if (T2[i]<T2[j])
-         T1[k++]=T2[i++];
-     else
-         T1[k++]=T2[j++];
-   }
-   while (i<=mid) T1[k++]=T2[i++];
- }

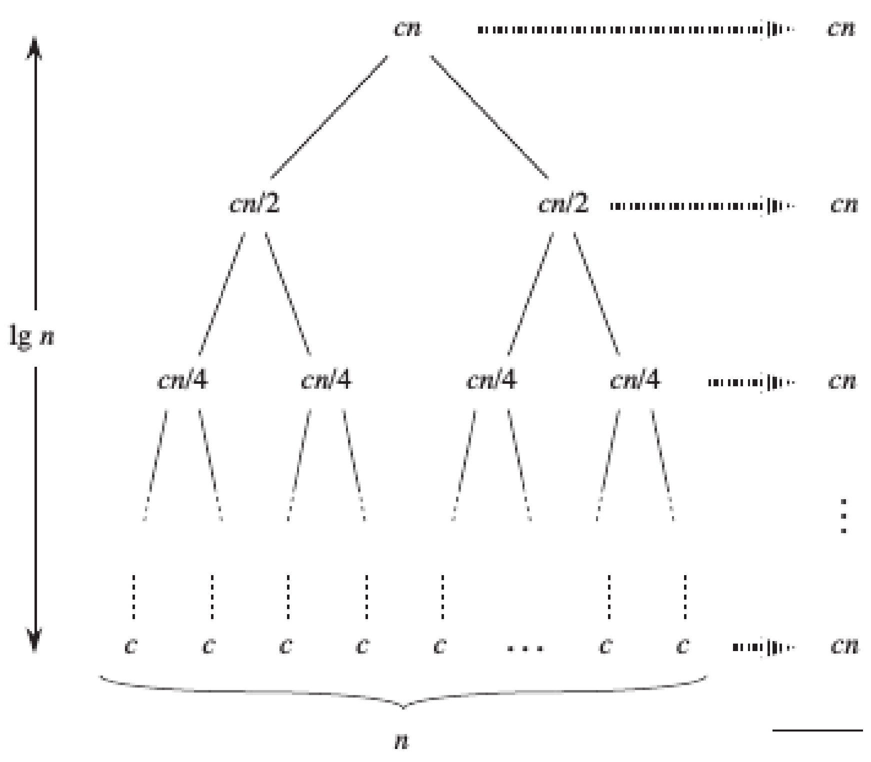
Merge Sort (C++)

- const int N = 10;
- int T1[N] = {4, 6, 4, 12, -3, 6, -6, 1, 8, 50};
- int T2[N]; // Tablica pomocnicza

- void **MergeSort**(int left, int right) {
- int mid;
- if (left<right) {
-   mid=(left+right)/2;
-   **MergeSort(left, mid);**
-   **MergeSort(mid+1, right);**
-   **Merge(left, mid, right); }}**

- int main() {
- int i, x;
- cout << "Przed sortowaniem:\n";
- for(i=0; i<N; i++) cout << T1[i] << " "; cout << endl;
- MergeSort(0, N-1);
- cout << "Po sortowaniu:\n";
- for(i=0; i<N; i++) cout << T1[i] << " "; cout << endl;
- Return 0; }

Złożoność obliczeniowa



- Równanie rekurencyjne:
  - T(n)=2T(n/2)+O(n)
- T(n) jest klasy O(n*log(n)).

Sortowanie przez kopcowanie
Heap Sort

Opis

- Sortowanie przez kopcowanie wykorzystuje własność kopca (sterty).
- Złożoność obliczeniowa jest klasy O(n*log(n)).
- Dokładniej omówimy to sortowanie, gdy omówimy stertę/kopiec (heap).
- Przykład działania:
  - https://www.youtube.com/watch?v=Xw2D9aJRBY4

Podsumowanie

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

Literatura dodatkowa

- Rozdział 4 - Algorytmy. Struktury danych i techniki programowania.
- Rozdziały 6-9 – T. Cormen, ‘Wprowadzenie do algorytmów’, PWN
- Rozdział 5 - Data Structures and Algorithms using Python.

Koniec