# Algorytmy i Struktury Danych

## Sortowanie 2

(c) Marcin Sydow

# Zawartość wykładu:



- Własność stabilności algorytmów sortujących
- algorytm sortowania szybkiego (QuickSort)
- czy można sortować szybciej niż ze złożonością $\Theta(n \cdot log(n))$ ?
- algorytm sortowania przez zliczanie (CountSort)
- (meta) algorytm RadixSort

# Własność stabilności



Algorytm sortujący nazywamy stabilnym wtedy i tylko wtedy, gdy zachowuje pierwotny względny porządek elementów o tej samej wartości.
np. jeśli ciąg wejściowy zawiera kilka elementów o wartości '2', to wszytkie te elementy w ciągu posortowanym będą w takiej samej względnej kolejności:
4, $2_a$, 3, 1, $2_b$, 5, $2_c$ -> 1, $2_a$, $2_b$, $2_c$, 3 ,4 ,5

Stabilność jest cechą, która ma ważne zastosowania praktyczne, np. sortowanie wielo-atrybutowych rekordów. Dzięki stabilności można posortować takie rekordy sortując kolejno po pojedynczych atrybutach (np. lista pracowników po atrybutach wiek, pensja, etc.) czyli sortując po kolejnym atrybucie nie niszczy się wyników sortowania po poprzednim, etc. Zauważmy, że np. w bazie danych równość wartości jednego atrybutu dla różnych rekordów nie oznacza równości całych rekordów.

# Podsumowanie poprzedniego wykładu:



Na poprzednim wykładzie zostały omówione 3 algorytmy:

- selectionSort
- insertionSort
- mergeSort

Ostatni algorytm ma niższy rząd złożoności czasowej niż 2 pierwsze, ale ma wyższą złożoność pamięciową.

# Algorytm Sortowania Szybkiego (QuickSort)



Algorytm QuickSort zaprojektowany jest na zasadzie "dziel i zwyciężaj" ("divide and conquer"). Jest to algorytm rekurencyjny i korzysta z pomocniczej funkcji `Partition` (wspomnianej przy okazji algorytmu wyszukiwania elementu k-tego co do wielkości)

Przypomnienie funkcji `Partition`:
Przypomnijmy, że funkcja ta reorganizuje wejściowy ciąg tak, że wybiera pewien (w tym wykładzie umawiamy się, że pierwszy) element p z ciągu (nazwijmy go "oś", ang. "pivot) i przestawia wszystkie elementy tak, że po przestawieniu p stoi na pewnej pozycji, wszystkie elementy na lewo od p są niewiększe niż p, a wszystkie elementy na prawo od p są niemniejsze niż p.
Algorytm zwraca indeks elementu p w zreorganizowanym ciągu.
przykład: 5,2,1,7,2,6,1,3,4,8,6,0 -> 3,2,1,0,2,4,1,5,6,8,6,7 (zwraca pozycję 7)

# Idea działania algorytmu QuickSort



Pomysł algorytmu QuickSort:

1. jeśli aktualny ciąg do posortowania ma długość niewiększą niż 1, zakończ rekursję (nie ma już nic do zrobienia)
2. jeśli ciąg jest dłuższy, to wykonaj funkcję `Partition` na ciągu i następnie **rekurencyjnie** wykonaj algorytm QuickSort zarówno na podciągu elementów na lewo od elementu p (ustawionego przez Partition) i na prawo od tego elementu

uwaga: po wykonaniu funkcji partition element p może trafić na jeden z końców ciągu - wtedy ciąg "na lewo" lub "na prawo" od p jest pusty i nie wykonuje się tam wywołania rekurencyjnego

Pomysł algorytmu (jak i funkcja `Partition`) pochodzi od C.A.R.Hoare.

# Analiza



Algorytm QuickSort na każdym poziomie rekursji używa szybkiej funkcji partition (która ma **liniową** złożoność czasową), dzięki czemu cały algorytm jest szybki. Ponadto, funkcja partition, reorganizując tablicę pracuje **w miejscu**, czyli nie potrzebuje **żadnej dodatkowej pamięci** poza tablicą przechowującą ciąg (w przeciwieństwie np. do algorytmu MergeSort)

Dzięki temu algorytm QuickSort jest efektywny czasowo i pamięciowo.

# Funkcja Partition - pseudokod



`partition(S, l, r)` **input**: a - tablica liczb całkowitych; l,r skrajne lewy i prawy indeksy aktualnego podciągu
**output**: liczba naturalna będąca indeksem elementu p ("oś") po zreorganizowaniu tablicy a (na lewo od p elementy niewiększe, na prawo - niemniejsze)

```
partition(a, l, r){

  i = l + 1;
  j = r;
  p = a[l]; //"pivot"
  temp;

  do{
    while((i < r) && (a[i] <= p)) i++;
    while((j > i) && (a[j] >= p)) j--;
    if(i < j) {temp = a[i]; a[i] = a[j]; a[j] = temp;}
  }while(i < j);
  // when (i==r):
  if(a[i] > p) {a[l] = a[i - 1]; a[i - 1] = p; return i - 1;}
  else {a[l] = a[i]; a[i] = p; return i;}
}
```

# Analiza funkcji Partition



- **operacja dominująca:** porównanie 2 elementów ciągu
- **rozmiar danych**: długość bieżącego ciągu $n = (r - l + 1)$

Analiza: indeks i przesuwa się w prawo dopóki nie napotka liczby większej od p a indeks j w lewo dopóki nie napotka liczby mniejszej od p, w takim wypadku liczby są zamieniane miejscami. Indeksy przesuwają się dopóki się nie spotkają, wtedy ostatecznie element p zamieniany jest miejscami z ostatnim elementem "lewego" ciągu, czyli miejscem spotkania indeksów i oraz j. Zauważmy, że porównanie wykonywane jest jednokrotnie dla każdej pozycji indeksu i oraz indeksu j, a więc łączna liczba porównań jest równa liczbie elementów w ciągu (minus jeden, bo bez elementu p).

Złożoność czasowa: $W(n) = A(n) = \Theta(n)$

Złożoność pamięciowa: $S(n) = O(1)$

# Algorytm QuickSort - pseudokod



QuickSort(a, l, r)
**input:** a - tablica liczb całkowitych; l,r - lewy i prawy skrajny indeks sortowanego podciągu (liczby naturalne)

(żadna wartość nie jest zwracana)

```
quicksort(a, l, r){

    if(l >= r) return;
    k = partition(a, l, r);
    quicksort(a, l, k - 1);
    quicksort(a, k + 1, r);
}
```

# QuickSort - analiza



Rozmiar danych: długość ciągu wejściowego (oznaczmy przez n)

Operacja dominująca: porównanie 2 elementów w ciągu

Zauważmy, że na każdym poziomie rekursji całkowita liczba porównań (w wywołaniach partition) wynosi $\Theta(n)$

Całkowita złożoność czasowa zależy więc od głębokości (liczby poziomów) rekursji.

# QuickSort - analiza



Algorytm QuickSort po każdym wywołaniu partition wywołuje siebie rekurencyjnie na obu częściach zreorganizowanego ciągu (o ile oba podciągi mają długość większą niż 1)

Zauważmy, że element p może trafić na dowolną pozycję reorganizowanego ciągu (zależy to od zawartości ciągu).

W szczególności, jeśli element p trafiałby zawsze idealnie w połowę ciągu, to otrzymalibyśmy idealnie zbalansowane drzewo rekursji (podziałów ciągu) czyli schemat analogiczny do algorytmu MergeSort - głębokość rekursji wynosiłaby $\Theta(log_2(n))$ (za każdym razem ciągi są ok. 2 razy krótsze).

Ponieważ na każdym poziomie rekursji łączna liczba porównań jest $\Theta(n)$ (liniowa) to w takim "idealnym" przypadku otrzymalibyśmy rząd złożoności czasowej QuickSort $\Theta(n \cdot log(n))$.

# QuickSort - pesymistyczna złożoność czasowa



Z kolei w "pesymistycznym" przypadku element p trafiałby zawsze na jeden z końców z aktualnego ciągu.

Zauważmy, że powoduje to wyjątkowo "niezbalansowany" kształt drzewa rekursji, gdyż ciąg za każdym razem zamiast dzielić się na 2 części, staje się **tylko o 1 krótszy**. W efekcie daje to liniową głębokość rekursji $\Theta(n)$, czyli w efekcie **kwadratową** złożoność pesymistyczną algorytmu.

$$W(n) = \Theta(n^2)$$

Skrajnym przypadkiem powodującym takie zachowanie algorytmu jest posortowanie lub odwrotne posortowanie ciągu wejściowego.

# QuickSort - przeciętna złożoność czasowa



Załóżmy następujący model losowy danych: każda permutacja elementów wejściowych jest jednakowo prawdopodobna.

Można udowodnić, że w takim przypadku oczekiwana (przeciętna) głębokość rekursji jest rzędu $\Theta(log(n))$, a więc przeciętna złożoność czasowa QuickSort wynosi:

$A(n) = \Theta(n \cdot log(n))$ (jest liniowo-logarytmiczna)

Co więcej, można pokazać, że w takim modelu losowości danych stała multiplikatywna przy dominującym czynniku wynosi około 1.44.

Algorytm QuickSort jest przeciętnie szybszy niż algorytm MergeSort i wogóle uchodzi za jeden z najszybszych algorytmów sortujących przez porównania.

# Quick Sort - ulepszenia podstawowej wersji



Algorytm jest szybki w przypadku przeciętnym, ale w wersji podstawowej niestety ma kwadratową (b.wysoką) złożoność czasową w przypadku pesymistycznym.

Powstały jednak bardziej skomplikowane wersje tego algorytmu, które gwarantują liniowo-logarytmiczną złożoność pesymistyczną.

Ponadto, podstawowa (zaprezentowana na wykładzie) wersja algorytmu nie jest stabilna. Jest jednak możliwe ulepszenie implementacji zapewniające stabilność bez pogarszania rzędu złożoności algorytmu.

# Czy można sortować szybciej niż $O(n \cdot log(n))$?



Zarówno MergeSort jak i QuickSort mają **liniowo-logarytmiczną** złożoność czasową, przy czym algorytm QuickSort jest dla przeciętnych danych szybszy niż MergeSort.

Powstaje więc naturalne pytanie:

# Czy można sortować szybciej niż $O(n \cdot log(n))$?



Zarówno MergeSort jak i QuickSort mają **liniowo-logarytmiczną** złożoność czasową, przy czym algorytm QuickSort jest dla przeciętnych danych szybszy niż MergeSort.

Powstaje więc naturalne pytanie:

Czy istnieje algorytm sortujący oparty na porównaniach elementów, który ma istotnie niższy rząd złożoności czasowej niż liniowo-logarytmiczna?

# Czy można sortować szybciej niż $O(n \cdot log(n))$?



Zarówno MergeSort jak i QuickSort mają **liniowo-logarytmiczną** złożoność czasową, przy czym algorytm QuickSort jest dla przeciętnych danych szybszy niż MergeSort.

Powstaje więc naturalne pytanie:

Czy istnieje algorytm sortujący oparty na porównaniach elementów, który ma istotnie niższy rząd złożoności czasowej niż liniowo-logarytmiczna?

Niestety można matematycznie wykazać, że nie istnieje taki algorytm, tj. najlepszą możliwą do uzyskania przeciętną złożonością czasową algorytmów sortujących przez porównania jest liniowo-logarytmiczna.

# Granica $O(n \log(n))$ - uzasadnienie



Na problem sortowania n elementów przez porównania można patrzeć w nieco inny, następujący sposób.
Zadaniem jest znalezienie sposobu przestawienia n elementów (permutacji tych elementów) tak, by były uporządkowane. Każde porównanie "$a < b$" jest pytaniem, na które odpowiedź jest "tak" albo "nie".
Na algorytm można patrzeć tak, że zadaje on takie "pytania" i w zależności od poszczególnych odpowiedzi wykonuje odpowiednie akcje.
Na algorytm sortujący przez porównania można więc patrzeć jak na binarne drzewo decyzyjne, gdzie korzeniem jest wejściowy ciąg, a każdy węzeł to porównanie 2 elementów i mamy na najniższym poziomie wszystkie możliwe przypadki. Jest ich $n!$, gdyż tyle jest różnych możliwych permutacji n elementowego ciągu.
Wysokość najlepiej zbalansowanego drzewa binarnego o $n!$ liściach (czyli minimalna wysokość takiego drzewa) jest rzędu $log_2(n!)$. Można wykazać, że jest to równe $\Theta(n \cdot log(n))$. Zauważmy, że ta wysokość to dokładnie liczba porównań jakie algorytm musi wykonać, aby odkryć właściwą permutację.

Wynika z tego, że nie można zaprojektować algorytmu sortującego przez porównania, który miałby niższy rząd przeciętnej i pesymistycznej złożoności czasowej niż $\Theta(n \cdot log(n))$.

# Szybsze sortowanie?



Postawmy więc pytanie:

czy jest możliwe, aby sortować z niższą niż liniowo-logarytmiczna złożonościa czasową?

# Szybsze sortowanie?



Postawmy więc pytanie:

czy jest możliwe, aby sortować z niższą niż liniowo-logarytmiczna złożonościa czasową?

**tak**

# Szybsze sortowanie?



Postawmy więc pytanie:

czy jest możliwe, aby sortować z niższą niż liniowo-logarytmiczna złożonościa czasową?

**tak**

jak to możliwe?

# Szybsze sortowanie?



Postawmy więc pytanie:

czy jest możliwe, aby sortować z niższą niż liniowo-logarytmiczna złożonościa czasową?

**tak**
jak to możliwe?

Nie używając porównań. Pokazaliśmy istnienie granicznego rzędu złożoności tylko dla algorytmów używających porównań.

Oczywiście wszystko ma swoją cenę - za większą szybkość trzeba będzie zapłacić zużyciem pamięci.
**Jest to typowy "deal" w algorytmice: czas vs pamięć**

# Algorytm CountSort (sortowanie przez zliczanie)



Jak sortować nie używając porównań? Można używać tzw. **adresowania bezpośredniego**, czyli umieszczać elementy ciągu wejściowego bezpośrednio w wynikowej tablicy na podstawie ich wartości, nie porównując z innymi elementami.

Aby taki pomysł mógł być zrealizowany efektywnie (szybko), dane muszą się mieścić w **pamięci o dostępie swobodnym** (ang. RAM - Random Access Memory). W pamięci takiej można wartość przypisać w czasie stałym do dowolnego adresu pamięci.

Przedstawiony zostanie podstawowy algorytm używający w/w pomysłu: **CountSort**

# CountSort - założenia i pomocnicze tablice



Zakładamy, że elementy do posortowania są liczbami **naturalnymi** (przy czym można nietrudno przerobić algorytm tak, aby pracował na liczbach całkowitych).

Algorytm tworzy aż 2 pomocnicze tablice:

- counts: do przechowywania liczników wystąpień poszczególnych wartości w ciągu wejściowym. Długość tej tablicy odpowiada maksymalnej liczbie w ciągu wejściowym + 1
- result: do umieszczenia wynikowego posortowanego ciągu

# CountSort - idea



Po zainicjalizowaniu w/w tablic, algorytm działa w 3 fazach:

1. przejście tablicy wejściowej, aby umieścić w tablicy counts liczby wystąpień poszczególnych elementów sortowanego ciągu
2. przyrostowe posumowanie elementów tablicy counts od lewej do prawej, aby było wiadomo ile dokładnie elementów w ciągu jest nie większych od danego elementu
3. ponowne przejście tablicy wejściowej, aby wysłać każdy napotkany element do tablicy results pod adresem odczytanym z tablicy counts (tutaj za każdym razem zmniejszamy o 1 licznik, żeby w przypadku późniejszego napotkania kolejnego elementu o takiej samej wartości wysłać go pod inny adres). Aby zachować stabilność, w tej fazie przechodzimy tablicę wejściową "od tyłu" (od prawej do lewej).

# CountSort - przykład



ciąg wejściowy:
(3,2,5,1,2,6,8,1,2,4)

maksymalna liczba w ciągu wejściowym: 8 (długość tablicy counts będzie o 1 większa odpowiadająca liczbom 0-8)

tablica counts po pierwszej fazie:
0,2,3,1,1,1,1,0,1 (zero nie wystąpiło, '1' 2 razy, '2' 3 razy, etc.)

tablica counts po drugiej fazie:
0,2,5,6,7,8,9,9,10
(uwaga: nie jest to jeszcze końcowy stan tej tablicy, gdyż wykonywana będzie jeszcze trzecia faza)

# CountSort - kod



**input:** a - tablica liczb naturalnych; l - długość ciągu

```
countSort(a, l){

    max = maxValue(a, l);
    l1 = max + 1;
    counts[l1];
    result[l];
    for(i = 0; i < l1; i++) counts[i] = 0;

    for(i = 0; i < l; i++) counts[a[i]]++;
    for(i = 1; i < l1; i++) counts[i] += counts[i - 1];
    for(i = l - 1; i >= 0; i--)
       result[--counts[a[i]]] = a[i];
}
```

# CountSort - analiza



**operacja dominująca:** przypisanie wartości w tablicy
**rozmiar danych** (2 argumenty):
długość ciągu: n, maksymalna wartość w ciągu: m

Algorytm tylko 2-krotnie sekwencyjnie przechodzi każdą z tablic. Wobec tego jego złożoność czasowa jest liniowa (!):

$$A(n,m) = W(n,m) = 2n + 2m = \Theta(n+m)$$

Niestety, ceną za to jest również liniowa (wysoka) złożoność pamięciowa:

$$S(n,m) = n + m = \Theta(n+m)$$

(Uwaga: Obie tablice muszą mieścić się w pamięci RAM)

Zauważmy też, że algorytm nie jest dobrym rozwiązaniem, gdy maksymalna liczba jest dużo wyższa niż liczba elementów w ciągu wejściowym. (np. dla posortowania następującego 2-elementowego ciągu: $(10^9,1)$, algorytm potrzebuje aż miliard kroków, i tablicy pomocniczej o długości $10^9$ !)

# Schemat sortowania RadixSort



Schemat RadixSort służy do sortowania wielo-elementowych ciągów o stałej długości (np. łańcuchów znaków lub wielocyfrowych liczb, etc.) za pomocą innego pomocniczego algorytmu sortowania. Pomocniczy algorytm musi być **stabilny**.

Schemat ten sortuje wielo-elementowe obiekty najpierw po ostatniej pozycji, potem po przedostatniej aż do pierwszej. Dzięki stabilności, sortowanie k-tej pozycji nie niszczy wyników sortowania poprzednio posortowanych pozycji.

W przypadku, gdy zbiór możliwych wartości (np. cyfry 0-9, litery a-z, etc.) jest niewielki, jako algorytm pomocniczy bardzo dobry jest CountSort.

# RadixSort - przykład



ciąg wejściowy:
(212, 305, 115, 202, 131)

po (stabilnym) sortowaniu po ostatniej pozycji:
(131, 212, 202, 305, 115)

po (stabilnym) sortowaniu po przedostatniej pozycji:
(202, 305, 212, 115, 131)

po (stabilnym) sortowaniu po pierwszej pozycji:
(115, 131, 202, 212, 305) (ciąg posortowany)

# Przykładowe problemy/pytania:



- Podaj definicję stabilności i do czego jest przydatna?
- Dla każdego z 5 omawianych algorytmów sortujących sprawdź, czy jest stabilny. Które miejsce w kodzie każdego algorytmu decyduje o jego (nie)stabilności?
- Podaj specyfikację funkcji Partition i pokaż końcową zawartość danej tablicy po wykonaniu na niej funkcji partition. Dokonaj analizy złożoności (czas, pamięć) tej funkcji.
- Napisz pseudokod QuickSort, dokonaj jego analizy złożoności (czas, pamięć) i omów możliwe ulepszenia.
- Jaka jest dolna granica rzędu złożoności sortowania przez porównania?
- Wykonaj CountSort na podanym ciągu. Pokaż końcową zawartość pomocniczej tablicy (counts) po zatrzymaniu algorytmu. Dokonaj analizy złożoności (czas, pamięć) tego algorytmu.
- Porównaj parami wszystkie omówione 5 algorytmów sortowania podając zalety, wady i ewentualne założenia techniczne.
- Omów rozszerzenie algorytmu CountSort tak, aby mógł sortować liczby całkowite (zarówno dodatnie jak i ujemne).
- Wykonaj RadixSort na podanym ciągu liczb 3-cyfrowych. Dokonaj analizy złożoności czasowej tego algorytmu dla liczb d-cyfrowych.

Dziękuję za uwagę!