# BubleSort

Algorytm sortowania bąbelkowego występuje w kilku wersjach. Na potrzeby laboratorium zaprezentowane zostaną dwie wersje tego algorytmu. Poniżej znajdują się dwie wersje sortowania rosnąco ciągu liczb: 8 4 2 5 1. W pierwszej kolumnie na czerwono zostały zaznaczone aktualnie porównywane liczby. Druga kolumna przedstawia porównywane liczby po ewentualnej zamianie miejscami. Na zielono zostały zaznaczone elementy już posortowanej części tablicy. Linią przerywaną zostały oddzielone kolejne kroki. W kolejnym kroku wykonywana jest jedna operacja mniej w stosunku do kroku poprzedniego. Ideą algorytmu jest porównywanie dwóch sąsiednich liczb między sobą.

## Wersja I

### Złożoność czasowa:

• Optymistyczna: O(n 2 )

• Średnia: O(n 2 )

• Pesymistyczna: O(n 2 )

W każdym kroku jedna liczba jest ustawiona w odpowiedniej kolejności.

## Wersja II

### Złożoność czasowa:

• Optymistyczna: O(n 2 )

• Średnia: O(n 2 )

• Pesymistyczna: O(n 2 )

W każdym kroku minimum jedna liczba jest ustawiona w odpowiedniej kolejności. Jeśli w danym kroku nie wystąpiła żadna zamiana elementów, oznacza to, że zbiór jest już posortowany i nie ma potrzeby wykonywać kolejnego kroku.

# SelectionSort

## Złożoność czasowa:

• Optymistyczna: O(n 2 )

• Średnia: O(n 2 )

• Pesymistyczna: O(n 2 )

# InsertSort

## Złożoność czasowa:

• Optymistyczna: O(n)

• Średnia: O(n 2 )

• Pesymistyczna: O(n 2 )

# QuickSort

Jest to algorytm działający na zasadzie algorytmu „dziel i zwyciężaj”. Z tablicy wybierany jest element rozdzielający, następnie tablica jest dzielona na dwa fragmenty. Pierwsza część tablicy składa się z elementów mniejszych niż element rozdzielający. Natomiast druga część tablicy zawiera elementy większe bądź równe od elementu rozdzielającego dwie części tablicy. Algorytm kończy się, gdy fragment (tablica) uzyskany z podziału zawiera pojedynczy element, jako że jednoelementowa tablica nie wymaga sortowania.

## Złożoność czasowa:

• Optymistyczna: O(nlog(n))

• Średnia: O(nlog(n))

• Pesymistyczna: O(n 2 )

# Algorytm podziału zbioru na dwie części

Algorytm polega na podziale zbioru na dwa rozłączne podzbiory. Pierwszy podzbiór zawiera elementy spełniające warunek podziału, drugi – niespełniające. Algorytm zwraca również indeks pierwszego elementu drugiego podzbioru. W celu wyjaśnienia algorytmu, wprowadźmy następujące oznaczenia:

• A – zbiór pierwotny,

• B – zbiór elementów spełniających warunek 1,

• C – zbiór elementów spełniających warunek 2, tj. warunek1 = ~warunek2

• A = B ∪ C – suma zbiorów daje zbiór pierwotny

• B ∩ C = ∅ - iloczyn zbiorów jest zbiorem pustym stąd wiadomo, że zbiory są rozłączne.

Algorytm polega na poszukiwaniu elementu, który nie spełnia warunku1 z lewej strony tablicy oraz elementu, który nie spełnia warunku2 z prawej strony tablicy. Jeśli takie elementy zostaną znalezione, to następuje zamiana miejscami tych elementów, ponieważ, element, który nie spełniał warunku1 spełnia warunek2, i odwrotnie.

Idea algorytmu podziału na dwa podzbiory jest bardzo podobna do wykonania jednej iteracji algorytmu QuickSort. Różnica polega na tym, że w przypadku sortowania szybkiego jako piwot (element, na podstawie którego zbiór jest dzielony) jest wybierany element o indeksie (lewy+prawy)/2, a w przypadku algorytmu podziału zbioru na dwa podzbiory warunek podziału jest ustalany przez użytkownika/programistę.

Złożoność czasowa: O(n)

# Algorytm podziału zbioru na trzy części

Algorytm polega na podziale zbioru na trzy podzbiory:

• 1 – podzbiór, którego elementy spełniają warunek1,

• 2 – podzbiór, którego elementy spełniają warunek2,

• 3 – podzbiór, którego elementy nie spełniają ani pierwszego, ani drugiego z warunków (spełniają warunek3)

Algorytm polega na poszukiwaniu elementów należących do skrajnych podzbiorów – pierwszego i trzeciego. Zbiór środkowy – drugi utworzy się automatycznie.

Złożoność czasowa: O(n)

# Algorytm naiwny

Jest najprostszym algorytmem wyszukiwania wzorca w tekście. Jego zasada opiera się na porównywaniu odpowiednich znaków w tekście i we wzorcu. Porównywany jest znak po znaku. Jeśli porównywane znaki się zgadzają to następny znak z tekstu i wzorca są porównywane. Jeśli zostaną porównane wszystkie znaki ze wzorca, oznacza to, że wzorzec został znaleziony, algorytm rozpoczyna przeszukiwanie od następnego znaku. Jeśli podczas porównania znak z tekstu i ze wzorca nie zgadzają się, należy rozpocząć algorytm porównania od początku wzorca zaczynając od kolejnego znaku w tekście.

# Algorytm Knutha-Morrisa-Pratta (KMP)

Modyfikuje algorytm naiwny, który przesuwa się zawsze o jeden znak. W algorytmie KMP wykorzystywana jest informacja o tym o ile może przesunąć się algorytm, aby był sens poszukiwań wzorca w tekście. Algorytm korzysta z tablicy częściowych dopasowań, którą należy stworzyć przed rozpoczęciem właściwego wyszukiwania. Wzorzec zawiera w sobie informację pozwalającą określić, gdzie powinna się zacząć kolejna próba dopasowania, pomijając ponowne porównywanie już dopasowanych znaków.

## Tablica dopasowań

Jest wyznaczana na podstawie wzorca. Wartość każdej komórki tablicy dopasowań jest długością najdłuższego właściwego prefiksu tej części, będącego jednocześnie jej najdłuższym właściwym sufiksem. Właściwy prefiks danego słowa, który jest równy jego właściwemu sufiksowi, nazywamy prefikso-sufiksem. Poniżej znajduje się przykładowa tablica dopasowań. Właściwy prefiks/sufiks nie obejmuje całego słowa, tylko n pierwszych/ostatnich znaków.

## Wyszukiwanie wzorca

Wyszukiwanie polega na porównywaniu kolejnych znaków z tekstu oraz ze wzorca. Jeśli zostaną porównane wszystkie znaki ze wzorca, oznacza to, że wzorzec został znaleziony lub jeśli znaki się nie zgadzają, za pomocą tablicy dopasowań wyznaczane jest przesunięcie, na podstawie którego algorytm po raz kolejny zaczyna szukać wzorca. W najgorszym przypadku wyszukiwanie zacznie się ponownie od kolejnego elementu w tekście.

# Algorytm Boyer’a-Moore’a

Porównywanie tekstu ze wzorcem zaczyna się od porównania ostatniego znaku wzorca. Dzięki temu jeśli ostatni znak wzorca nie zgadza się ze znakiem w tekście i znak z tekstu nie występuje we wzorcu, to okno wzorca można od razu przesunąć o tyle pozycji ile znaków ma wzorzec. Jeśli znak w tekście występuje we wzorcu, należy tak przesunąć okno wzorca, aby znak z tekstu i ostatnie wystąpienie tego znaku ze wzorca zrównały się pozycjami. W celu wyznaczenia odpowiedniego przesunięcia algorytm korzysta z tablicy przesunięć.

## Tablica przesunięć

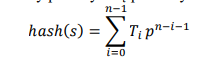
Tablica przyjmuje rozmiar równy liczbie elementów z którego składa się alfabet, czyli liczba wszystkich możliwych znaków, które mogą wystąpić w tekście, jak i we wzorcu. Tworząc tablicę przesunięć należy pamiętać o trzech głównych zasadach: Jeśli znak alfabetu nie występuje we wzorcu, pod indeks odpowiadający znakowi można wpisać -1 lub długość wzorca zamiast liczby. Informacja ta służy jedynie rozróżnieniu w późniejszym etapie, czy dany znak występuje we wzorcu. Jeśli dany znak występuje kilka razy we wzorcu, należy wpisać do tablicy największą odpowiadającą mu wartość indeksu. Można również zastosować podejście, polegające na tym, aby wpisywać najmniejszy indeks, pod którym znajduje się znak licząc od prawej strony. W przykładzie przedstawiony poniżej zostało przedstawione podejście ze wpisywaniem największej wartości indeksu, który odpowiada danemu znakowi. Jeśli znak nie występuje we wzorcu, wpisano -1.

# Algorytm Karpa-Rabina

Algorytm opiera się na wyliczeniu funkcji haszującej. Hasz liczony jest dla wzorca oraz dla znaków w danym oknie. Jeśli hasze są takie same, należy wykonać porównanie wzorca z bieżącym oknem tekstu znak po znaku za pomocą naiwnego algorytmu wyszukiwania wzorca. Możliwe są kolizje, czyli może wystąpić taka sama wartość haszu dla dwóch różnych ciągów znaków. Poniżej zostanie zaprezentowany algorytm wyznaczania haszu. W celu uniknięcia zbyt dużych wartości haszu często stosowana jest arytmetyka modularna – wyniki wszystkich działań są sprowadzane do reszty z dzielenia przez wybrany moduł, który zwykle jest liczbą pierwszą. Zaletą stosowania takiej arytmetyki jest to, że można szukać dłuższych wzorców w tekście Jeśli za podstawę w funkcji haszującej chcemy przyjąć liczebność alfabetu należy wiedzieć z czego składa się alfabet, czyli jakie znaki mogą się pojawić we wzorcu i w tekście, na przykład: duże litery alfabetu albo małe litery alfabetu. Jeśli alfabet nie jest znany należy założyć, że alfabetem jest cała tablica ASCII (256 znaków).

Wyznaczenie haszu

W celu wyznaczenia haszu należy posłużyć się poniższym wzorem:



gdzie: T – liczba, odpowiadająca i-tej literze tekstu T w używanym systemie kodowania, p – podstawa systemu kodowania, n – liczba znaków w tekście, i – numer litery w tekście, licząc od prawej do lewej.

# Wyszukiwanie liniowe

Wyszukiwanie liniowe jest to rodzaj wyszukiwania sekwencyjnego polegający na przeglądaniu kolejnych elementów zbioru. Jeśli przeglądany element jest elementem który jest szukany, to zwracana jest jego pozycja(indeks) w zbiorze i algorytm kończy się. W przeciwnym razie przeglądane są kolejne elementy zbioru aż do momentu przejrzenia wszystkich elementów zbioru. Wynikiem wyszukiwania liniowego jest indeks, pod którym znajduje się szukany element. Rozszerzona wersja algorytmu zwraca indeksy wszystkich elementów zbioru, których wartości są równe szukanej wartości.

## Złożoność czasowa:

• optymistyczna: O(1)

• średnia: O(n)

• pesymistyczna: O(n)

# Wyszukiwanie bisekcyjne (binarne)

Algorytm wyszukiwania bisekcyjnego nazywany jest również wyszukiwaniem binarnym, ponieważ polega na dwukrotnym zawężeniu podzbioru, w którym szukany jest element, w kolejnym kroku. Należy pamiętać, że zbiór musi być posortowany aby zastosować algorytm wyszukiwania bisekcyjnego. Algorytm polega na wyznaczeniu indeksu środkowego zbioru/podzbioru i sprawdzeniu czy element znajdujący się pod tym indeksem jest równy szukanemu. Jeśli tak, algorytm kończy się. W przeciwnym wypadku poszukiwany element jest albo mniejszy od elementu środkowego, albo większy. Elementy mniejsze niż element środkowy będą znajdowały się w lewej części podzbioru, a elementy większe będą znajdowały się w prawej części. W kolejnym kroku zawężamy obszar przeszukiwań do jednego podzbioru.

## Złożoność czasowa:

• Optymistyczna: O(1)

• Średnia: O(log(n))

• Pesymistyczna: O(log(n))

# Stos

Jest liniową strukturą danych, z której odczytujemy elementy w kolejności odwrotnej do ich dodawania. Struktura ta nosi nazwę LIFO ( ang. Last In – First Out – wszedł ostatni, a wyszedł pierwszy ). Dane dokładane są na wierzch stosu i z wierzchołka stosu są pobierane. Zdejmowanie elementów ze stosu odbywa się w odwrotnej kolejności do ich umieszczania.

## Na stosie można wykonać następujące operacje:

• dodawanie elementu na stos(push) – umieszczenie elementu na szczycie stosu,

• usunięcie elementu ze stosu (pop) – zdejmowanie elementu znajdującego się na szczycie stosu,

• sprawdzenie, czy stos jest pusty (isEmpty),

• pobranie ostatniej wartości na stosie(top).

# Kolejka

Kolejka jest strukturą danych, w której nowe dane dopisywane są na końcu, a z początku kolejki usuwane są elementy. Jest to struktura danych typu FIFO (First In, First Out; pierwszy na wejściu, pierwszy na wyjściu). Sposób działania struktury kolejka można porównać do kolejki w sklepie. Najpierw obsługiwani są klienci stojący z przodu obok kasy. Kiedy klient zostanie obsłużony, odchodzi i przychodzi kolejny klient.

## Operacje, które można wykonać na kolejce:

• dodanie elementu na koniec kolejki (push),

• usunięcie elementu z początku kolejki (pop),

• sprawdzenie czy kolejka jest pusta (isEmpty),

• odczytanie elementu z początku kolejki (first).

# Lista

Jest to struktura danych, która umożliwia sekwencyjny dostęp do elementów. Z jednego elementu można przejść do drugiego. Listy stosowane są do reprezentacji w pamięci komputera danych sekwencyjnych na przykład: grafów, kolejek, stosów. Rozróżniane są trzy rodzaje list: lista jednokierunkowa, lista dwukierunkowa oraz lista cykliczna: jednokierunkowa oraz dwukierunkowa.

## Operacje wykonywane na listach:

• dodawanie elementu do listy: o na początek listy(add\_head), o na koniec listy(add\_tail), o na określoną pozycję(add\_position).

• usuwanie elementu z listy: o z początku listy(delete\_head), o z końca listy(delete\_last), o z określonej pozycji(delete\_position).

• sprawdzanie czy lista jest pusta (isEmpty)

• wyświetlenie wszystkich elementach w liście (show),

• odczytanie elementu z początku listy(first),

• odczytanie elementu z końca listy(last).

## Lista dwukierunkowa:

Jedyną różnicą pomiędzy listą jednokierunkową a dwukierunkową jest możliwość wyświetlenia elementów zarówno od początku do końca, jak i od końca do początku (przejście w dwóch kierunkach).

## Lista cykliczna:

Lista cykliczna może być albo listą jednokierunkową albo dwukierunkową. Lista cykliczna jednokierunkowa różni się od listy jednokierunkowej tym, że w liście jednokierunkowej ostatni element przechowuje adres NULL, a w liście jednokierunkowej cyklicznej ostatni element przechowuje adres pierwszego elementu w liście. Natomiast lista cykliczna dwukierunkowa różni się od listy dwukierunkowej tym, że pierwszy element w polu prev przechowuje adres NULL i ostatni element w polu next również przechowuje NULL, a w liście dwukierunkowej cyklicznej, pierwszy element w polu prev przechowuje adres ostatniego elementu, a ostatni element w polu next przechowuje adres pierwszego elementu w liście.

# Drzewo BST:

Drzewo poszukiwań binarnych jest strukturą danych, która zbudowana jest z węzłów, w której każdy element może mieć dwóch potomków: lewego i prawego, nazywanych dziećmi tego elementu. Ten element z kolei jest dla nich przodkiem/rodzicem. Każdy węzeł posiada przodka oraz maksymalnie dwóch potomków (prawego i lewego). Wyjątkiem jest pierwszy węzeł, który nie posiada przodka nazywany jest korzeniem, oraz ostatnie węzły, które nie posiadają potomków nazywane są liśćmi.

# Sortowanie przez kopcowanie

Sortowanie polega na zbudowaniu kopca oraz rozebraniu kopca. W wyniku rozebrania kopca dane zostają posortowane (rosnąco lub malejąco). Sortowanie przez kopcowanie można zrealizować w oparciu o tablicę. W tablicy budowany jest kopiec, a następnie elementy są zdejmowane z kopca – proces rozbierania kopca można uznać za właściwe sortowanie. Poniżej przedstawiony zostanie algorytm tworzenia kopca oraz rozbierania kopca.

Złożoność czasowa algorytmu sortowania wynosi O(nlog2n).

## Kopiec

Jest to drzewo binarne, w którym wszystkie węzły spełniają następujący warunek: klucz węzła nadrzędnego jest większy lub równy (lub mniejszy bądź równy w zależności od kierunku sortowania) kluczom węzłów potomnych. Po dołączeniu do drzewa należy za każdym razem sprawdzać czy warunek kopca jest zachowany. Jeśli warunek nie jest spełniony należy dokonać zmian na ścieżce wiodącej od danego węzła do korzenia. Korzeń jest zawsze elementem największym (lub najmniejszym) w całym kopcu. Cechą charakterystyczną kopca jest to, że jest to drzewo zupełne poza liśćmi. Rozbiór kopca polega na tym, że należy zamienić wartości przechowywane w korzeniu z ostatnim liściem i usunąć liść. Po wykonaniu tego kroku należy sprawdzić czy zachowana została struktura kopca. Jeśli nie, strukturę kopca należy przywrócić idąc od jego korzenia w dół. Kroki należy powtarzać, aż kopiec zostanie pusty - wtedy dane zostały posortowane.