Технически университет - София

филиал Пловдив

Курсова работа по дисциплина

Съвременни Java технологии

Тема: Сравнителна характеристика на алгоритъм за линейно търсене и алгоритъм за двоично търсене в масив, SelectionSort – алгоритъм за сортиране на масив

Изготвил: Тодор Пейчинов

Фак. №: 378950

Група: 41А

I. Кратка теория:

Търсенето е процес, в който трябва да открием конкретен елемент от масив – например можем да изследваме дали даден резултат е част от списък с резултати. Търсенето е една основна задача в компютърното програмиране, която се налага ежедневно да извършваме в процеса на разработка на какъвто и да е софтуер. Съществуват редица алгоритми решаващи този проблем – като тема на текущото разглеждане ще бъдат едни от най-често използваните в практиката алгоритми - за линейно търсене и двоично търсене в масив.

II. Линейно търсене

При линейното търсене сравняваме ключовият елемент (елементът, който търсим) с всеки един елемент, който е част от масива. Търсенето продължава докато не намерим елементът, който търсим, или докато не претърсим целия масив и не стигнем до заключението, че търсеният елемент не е част от масива. Ако елементът, който търсим съвпадне с някой от елементите в масива, линейното търсене връща индекса на елемента, който съвпада с търсения елемент. Ако елементът, който търсим не е част от масива, то линейното търсене връща -1, за да ни покаже, че търсеният елемент не е намерен.

Методът на линейното търсене сравнява ключовият елемент с всеки един елемент от масива. Елементите в масива могат да бъдат подредени в какъвто и да е ред. Средно този алгоритъм трябва да претърси половината от елементите в масива преди да намери търсеният, ако той изобщо съществува в масива. Тъй като времето на изпълнение на методът за линейно търсене нараства линейно с нарастването на елементите на масива, който претърсваме, линейното търсене е неефективно при много големи масиви.

Линейното търсене има свои предимства и недостатъци, които се отнасят към различни аспекти на алгоритъма:

Предимства на линейното търсене:

1. **Простота на реализация**: Линейното търсене е много лесно за разбиране и имплементиране. Не изисква сложна логика или структури данни.

2. **Приложимост**: Може да се прилага върху всякакви списъци или масиви, без значение от техния размер или структура.

3. **Ефективност при малки размери на данните**: За малки списъци или масиви линейното търсене може да бъде достатъчно ефективно и бързо.

Недостатъци на линейното търсене:

1. **Ниска ефективност при големи размери на данните**: При големи списъци или масиви, линейното търсене може да бъде много бавно, особено ако целевият елемент се намира към края на списъка или масива.

2. **Линейна сложност**: В най-лошия случай, когато целевият елемент се намира в края на списъка или масива, времето за търсене ще бъде линейно пропорционално на размера на списъка или масива. Това означава, че алгоритъмът има сложност O(n), където n е броят на елементите в списъка.

3. **Неефективност при сортиране**: Ако списъкът или масивът не е сортиран, линейното търсене изисква да се проверят всички елементи, за да се намери търсеният. Това може да бъде неефективно, особено ако елементът се намира към края на списъка или масива.

В заключение, линейното търсене е подходящо за определени сценарии, особено при работа с малки размери на данните или когато сложността на алгоритъма не е от критично значение. За по-големи и организирани структури от данни обаче, ефективността му може да бъде недостатъчна.

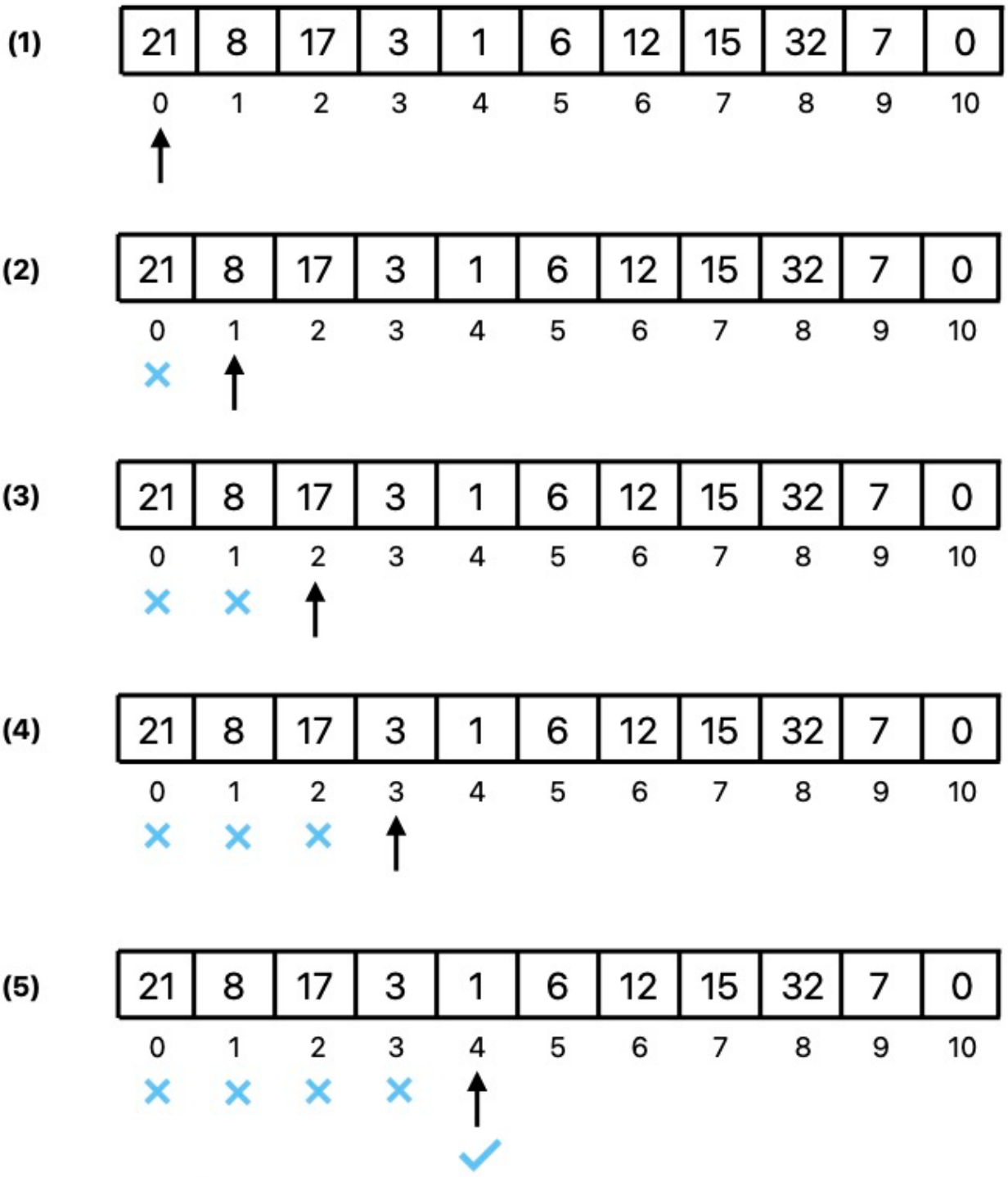
Сложността на линейното търсене е O(n), където n е броят на елементите в списъка или масива, върху който се извършва търсенето.

Това означава, че времето за търсене ще нараства линейно с увеличаване на броя на елементите в списъка. По силата на това, в най-лошия случай, когато целевият елемент се намира към края на списъка или масива, линейното търсене ще изисква проверка на всеки елемент, което води до сложност O(n).

Програмна реализация:

```
public class LinearSearch  {
	public static int linearSearch (int[] list, int key) {
		for (int i = 0; i < list.lenght; i++) {
			if (key == list[i])
				return i;
		}
		return -1;
	}
}
```

LinearSearch(1)



return 4;

III. Двоично търсене

Двоичното търсене е друг много често използван метод за търсене на елемент в списък с елементи. За да е възможно използването на метода за двоично търсене е необходимо масивът да бъде сортиран преди да започне търсенето в него. Сортирането трябва да бъде направено във възходящ ред. Двоичното търсене първо сравнява ключовият елемент (елементът, който търсим) с елемента, който се намира в средата на масива, който претърсваме. След това имаме три възможности, чрез които да продължим търсенето:

1. Ако ключът е по-малък от елемента, който се намира в средата на масива, трябва да продължим търсенето само в първата половина от масива.

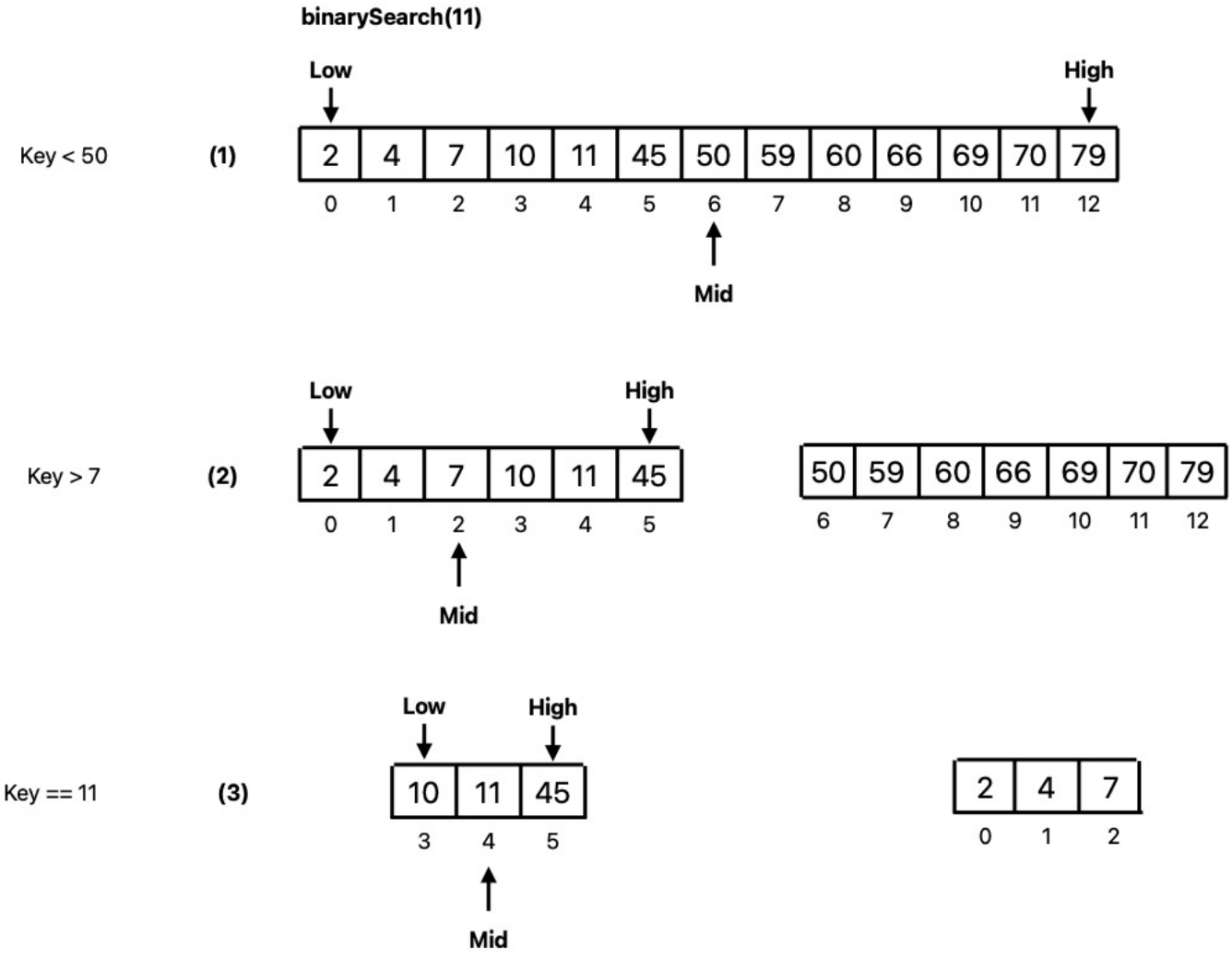
2. Ако ключът е равен на елементът, който се намира в средата на масива, то тогава сме намерили търсеният елемент.

3. Ако ключът е по-голям от елемента, намиращ се в средата на масива, трябва да продължим търсенето на ключа само във втората половина от масива.

На практика методът на двоичното търсене елиминира най-малко половината от масива след всяко сравнение. Ако предположим, че масивът има n – на брой елемента, то още след първото сравнение, елементите, които ще останат да претърсим са n / 2. След второто сравнение елементите, които остават за проверка са (n / 2) / 2. След k – тата проверка, елементите, които остават да бъдат проверени са $n / 2^k$. Когато $k = \log_2 n$, само един елемент е останал в масива, съответно остава само едно сравнение. Нещо повече, в най-лошият случай, когато използваме алгоритъмът за двоично търсене, ще трябва да направим $\log_2 n+1$ проверки, за да открием един елемент в сортирания масив. В най-лошият случай за списък състоящ се от 1024 (2^{10}) елемента, двоичното търсене ще трябва да

направи само 11 проверки, докато за сравнение, с алгоритъма за линейно търсене ще трябва да извършим 1024 проверки в най-лошият случай.

Частта данни от масива, който претърсваме намаля наполовина след всяка една проверка. Нека отбележим с low и high първият и последният индекс на масива, който ще претърсваме. Low ще бъде равно на 0, а high ще бъде размера на масива – 1. С mid ще отбележим индекса на елемента, намиращ се в средата на масива, така че mid ще бъде равно на (low + high) / 2. Следващата фигура показва как се намира ключ със стойност 11 в масив от елементите {2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79} посредством алгоритъма за двоично търсене.



Програмна реализация

Програмната реализация на алгоритъма за двоично търсене на Java, започва с първата итерация. Преди това за най-малкият елемент от масива дефинираме int low = 0, за най-големият елемент, който е равен на размера на масива – 1 int high = list.length – 1 и за елементът, който се намира в средата на масива (low + high) / 2.

Следващата стъпка от алгоритъма е да бъде сравнен ключовият елемент, с елемента, намиращ се в средата на масива. Ако елементът е по – голям от ключовият елемент, присвояваме на индекса посочващ най-големият елемент от масива индекса на средния елемент - 1, high = mid – 1. Ако елементът е по-малък от ключовият елемент, присвояваме на индекса показващ най-малкият елемент от масива индекса на средният елемент + 1, low = mid + 1. Ако елемента, който сравняваме е равен на ключовият елемент, то ние сме намерили индекса на търсения елемент.

Когато ключовият елемент не е намерен позицията на low е позицията, където може да бъде поставен ключовият елемент, така че да не разваляме редът на сортиране на масива. Много по-добрият подход при програмната реализация на конкретния алгоритъм е да се връща като стойност, индекса където можем да поставим търсеният елемент, когато той не е намерен, в сравнение с това да връщаме -1. Но друг подход, който може да бъде използван е алгоритъма да връща -1, когато ключовият елемент не е част от претърсвания масив. А също друг много добър вариант за реализация в случаите, когато елементът не бъде открит е методът да връща – low – 1. Така методът ще връща не само стойност, която показва, че елементът не е открит, но и позицията, където той може да бъде поставен без да се наруши редът на масива.

Вариант 1:

```
public class BinarySearch {
  public static int binarySearch(int[] list, int key) {
    int low = 0;
    int high = list.length - 1;

    while (high >= low) {
      int mid = (low + high) / 2;
      if (key < list[mid])
        high = mid - 1;
      else if (key == list[mid])
        return mid;
      else
        low = mid + 1;
    }

    return - 1; //
  }

  public static void main(String[] args) {
    int[] list = {-3, 1, 2, 4, 9, 23};
    System.out.println(binarySearch(list, 2));
  }
}
```

Вариант 2:

```
public class BinarySearch {
  public static int binarySearch(int[] list, int key) {
    int low = 0;
    int high = list.length - 1;

    while (high >= low) {
      int mid = (low + high) / 2;
      if (key < list[mid])
        high = mid - 1;
      else if (key == list[mid])
        return mid;
      else
        low = mid + 1;
    }

    return -low; //
  }

  public static void main(String[] args) {
    int[] list = {-3, 1, 2, 4, 9, 23};
    System.out.println(binarySearch(list, 2));
  }
}
```

Вариант 3:

```
public class BinarySearch {
  public static int binarySearch(int[] list, int key) {
    int low = 0;
    int high = list.length - 1;

    while (high >= low) {
      int mid = (low + high) / 2;
      if (key < list[mid])
        high = mid - 1;
      else if (key == list[mid])
        return mid;
      else
        low = mid + 1;
    }

    return -low - 1; //
  }

  public static void main(String[] args) {
    int[] list = {-3, 1, 2, 4, 9, 23};
    System.out.println(binarySearch(list, 2));
  }
}
```

Двоичното търсене връща индекса на търсения елемент, ако той е част от списъка с елементи. В противен случай връща -low – 1. А какво би станало ако вместо (high >= low) напишем (high > low)? Търсенето би могло да пропусне даден елемент, който търсим.

А също друг въпрос, който би изникнал е дали методът ще работи, ако има елементи, които се повтарят в масива, и отговорът е, че той ще работи, стига масивът да е подреден във възходящ ред. Алгоритъма ще върне индекса на един от повтарящите се елементи, който търсим.

Специфичното при двоичното търсене е това, че масивът от елементи трябва да бъде сортиран, преди да използваме търсенето. Друго отличително за метода е това, че когато намери търсеният елемент връща индекса на елемента в масива, а когато елементът не е част от масива, то резултат на търсенето е отрицателната стойност на индекса, където може да бъде поставен, така че да не бъде нарушена сортировката на масива – 1.

Двоичното търсене е ефективен алгоритъм за намиране на елемент в сортиран списък или масив. Той работи, като сравнява целевия елемент с елемента в средата на списъка и решава в кой от двата подсписъка да продължи да търси, като използва свойствата на сортирания списък.

Ето предимствата и недостатъците на двоичното търсене:

Предимства на двоичното търсене:

1. **Логаритмична сложност**: Двоичното търсене има времева сложност O(log n), където n е броят на елементите в списъка. Това означава, че при всяко стъпало в търсенето, броят на елементите за проверка се намалява до половината.

2. **Ефективност върху големи данни**: За големи списъци или масиви двоичното търсене е много по-ефективно от линейното търсене, тъй като се извършват значително по-малко проверки.

3. **Простота на имплементацията**: Въпреки че може да изисква малко по-сложен код от линейното търсене, двоичното търсене е все пак доста прости и лесно разбираеми.

Недостатъци на двоичното търсене:

1. **Изисква сортиран списък**: Един от най-големите недостатъци на двоичното търсене е, че списъкът трябва да бъде предварително сортиран, което може да изисква допълнително време и памет.

2. **Неефективност при чести промени в данните**: Ако данните в списъка се променят често (добавяне, изтриване, промяна на стойности), двоичното търсене може да се окаже неефективно, тъй като сортираният ред на данните трябва да се поддържа.

3. **Необходимост от памет за рекурсия или итерация**: Реализацията на двоичното търсене може да използва допълнителна памет за стека при рекурсивно изпълнение или за структури като опашка при итеративно изпълнение.

В общи линии, двоичното търсене е мощен алгоритъм за търсене в сортирани списъци или масиви, особено при големи данни, но изисква сортиран списък и може да бъде неефективен при чести промени в данните.

Сложността на двоичното търсене е O(log n), където "n" е броят на елементите в сортирания списък или масив, върху който се извършва търсенето. Това означава, че времето за търсене ще нараства логаритмично с увеличаване на броя на елементите в списъка.

По силата на това, двоичното търсене е много по-ефективно от линейното търсене, особено при големи списъци или масиви, тъй като за всяко стъпало в търсенето, броят на елементите за проверка се намалява до половината.

Като обобщение на сравнителни характеристики между двоичното и линейното търсене:

1. **Сложност на времето**:

   - Линейно търсене: O(n) - времето за търсене нараства линейно с броя на елементите в списъка или масива.

   - Двоично търсене: O(log n) - времето за търсене нараства логаритмично с броя на елементите в сортирания списък или масив.

2. **Изисквания за сортираност**:

   - Линейно търсене: Не изисква сортиран списък.

   - Двоично търсене: Изисква предварително сортиран списък.

3. **Ефективност**:

   - Линейно търсене: Подходящо за малки списъци или масиви, но неефективно за големи данни.

   - Двоично търсене: Особено ефективно за големи данни и сортирани списъци, като гарантира логаритмична сложност.

4. **Промени в данните**:

   - Линейно търсене: По-подходящо при чести промени в данните, тъй като не изисква сортираност.

   - Двоично търсене: По-ефективно при търсене в статичен или рядко променящ се списък.

5. **Простота на имплементацията**:

   - Линейно търсене: По-лесно за разбиране и имплементиране.

   - Двоично търсене: Малко по-сложно за разбиране, но все пак доста прости и лесно разбираеми.

Изборът между двоичното и линейното търсене зависи от конкретните изисквания на проблема и характеристиките на данните. Двоичното търсене е предпочитано за големи и сортирани данни, докато линейното търсене може да бъде по-подходящо за малки данни или при чести промени в данните.

IV. Selection Sort

Тъй като двоичното търсене изисква масивът от стойности да бъде сортиран, преди да е възможно да търсим данни в него, то един от най-често използваните методи е този със Selection Sort алгоритъма за сортиране.

Selection sort е прост алгоритъм за сортиране, който работи като разделя входния списък на две части: сортирана и несортирана. Постепенно алгоритъмът избира най-малкия (или най-големия) елемент от несортираната част и го разменя с първия елемент на несортираната част. След това сортираната част нараства с един елемент, а несортираната се намалява с един елемент.

Ето по-подробно как работи Selection sort:

1. Начало: Нека имаме списък от елементи, който трябва да бъде сортиран.

2. Инициализация: За начало, целият списък се счита за несортиран. Първоначално сортираната част е празна, а целият списък е несортиран.

3. Намиране на най-малкия елемент: Алгоритъмът започва като избира първия елемент на несортираната част на списъка и го счита за най-малкия. Този елемент се запазва в паметта като текущия минимум.

4. Обхождане на несортираната част: След това алгоритъмът обхожда останалите елементи от несортираната част на списъка и сравнява всеки елемент с текущия минимум. Ако се намери по-малък елемент, той става новият текущ минимум.

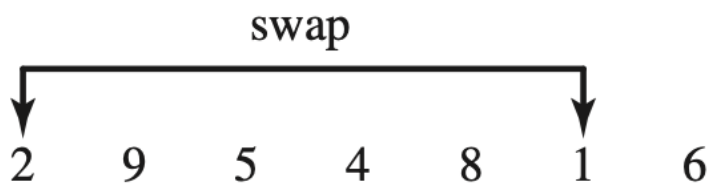
5. Размяна на елементи: Накрая, когато алгоритъмът премине през всички елементи на несортираната част, текущият минимум се разменя с първия елемент на несортираната част. Така първият елемент на несортираната част става част от сортираната част, а новата несортирана част се намалява с един елемент.

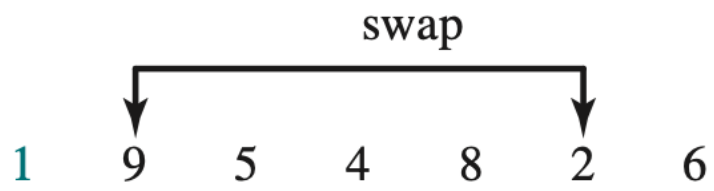
6. Продължаване на процеса: Процесът се повтаря, като алгоритъмът избира следващия най-малък елемент от оставащата несортирана част и го разменя с първия елемент на несортираната част.

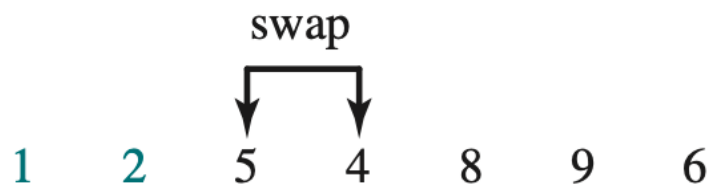
7. Повторение: Алгоритъмът продължава да извършва тези стъпки, докато целият списък стане подреден.

8. Край: Когато несортираната част стане празна, списъкът е напълно подреден.

Selection sort има сложност на времето O(n^2), където "n" е броят на елементите в списъка. Той е по-малко ефективен от по-сложни алгоритми за сортиране като Quick sort или Merge sort, но все пак е добър избор за сравнително малки списъци или когато се изискват прости решения.







1 2 4 5 8 9 6

1 2 4 5 6 8 9

Програмна реализация

```
public class SelectionSort {
  public static void selectionSort(double[] list) {
    for (int i = 0; i < list.length - 1; i++) {
      double currentMin = list[i];
      int currentMinIndex = i;

      for (int j = i + 1; j < list.length; j++) {
        if (currentMin > list[j]) {
          currentMin = list[j];
          currentMinIndex = j;
        }
      }


      if (currentMinIndex != i) {
        list[currentMinIndex] = list[i];
        list[i] = currentMin;
      }
    }
  }

  public static void main(String[] args) {
    double[] list = {-2, 4.5, 5, 1, 2, -3.3};
    selectionSort(list);
    for (int i = 0; i < list.length; i++)
      System.out.print(list[i] + " ");
  }
}
```