COMP103 Data Structures and Algorithms

Tutorial 15.2: Search Algorithms (Solution)

A. Fundamentals of Search Algorithms

1. What is searching and its goal? [2 marks]

   Answer

   Searching is a process of locating a specific element or item within a collection of data such as arrays, lists, trees, or other structured representations.

   Searching objectives are typically to determine whether the desired element exists within the data, and if so, to identify its precise location or to retrieve it.

2. Why searching is important in programming? [2 marks]

   Answer

   Why searching is important:

   - Efficiency: Efficient searching algorithms improve program performance.
   - Data retrieval: Quickly find and retrieve specific data from large datasets.
   - Database systems: Enables fast querying of databases.
   - Problem solving: Used in a wide range of problem-solving tasks.

3. What are the criteria used for evaluating searching algorithms? [2 marks]

   Answer

   - Target Element: A specific target element or item to find in the data collection i.e. a value, a record, a key, or any other data entity of interest.
   - Search Space: The entire collection of data for finding the target element. Depending on the data structure used, the search space may vary in size and organization.
   - Complexity: Different levels of complexity depending on the data structure and the algorithm used i.e. often measured in terms of time and space requirements.
   - Deterministic vs Non-deterministic: For a comparison, binary search is deterministic i.e. follow a clear and systematic approach. Others, such as linear search, are non-deterministic i.e. need to examine the entire search space in the worst case.

4. Provide some examples of searching algorithms. [2 marks]

   Answer

Some examples of searching algorithms are:

Some examples of popular basic searching algorithms

- Linear search
- Binary search
- Ternary search
- Jump search
- Interpolation search
- Fibonacci search
- Exponential search

B. Programming Search Algorithms

1. Complete the program given below which is about binary search (recursive). [2 marks]

Program:

```
public int indexOf(String value, List<String> data){
return indexOf(value, data, 0, data.size());
}
public int indexOf(String value, List<String> data, int low, int high){
// value in [low .. high)  (if present)
if (low >= high){  return -1;  }             // value not present
```

```
if (comp == 0) {
return mid;
}  // item is present
else if (comp < 0)   {
return indexOf(value, data, low, mid);
}  // item in [low .. mid)
else {
return indexOf(value, data, mid+1, high);
}  // item in [mid+1 .. high)
}
```

Solution

The completed program is as shown below.

Program:

```
public int indexOf(String value, List<String> data){
return indexOf(value, data, 0, data.size());
}
public int indexOf(String value, List<String> data, int low, int high){
// value in [low .. high)  (if present)
if (low >= high){  return -1;  }             // value not present
int mid  =  (low + high) / 2;
int comp =  value.compareTo(data.get(mid));
if (comp == 0) {
return mid;
}  // item is present
else if (comp < 0)   {
return indexOf(value, data, low, mid);
}  // item in [low .. mid)
else {
return indexOf(value, data, mid+1, high);
}  // item in [mid+1 .. high)
}
```

2. Complete the program given below which is about the binary search (iterative). [4 marks]

Program:

```
private int indexOf(String value, List<String> data){
int low = 0;
int high = data.size();
// item in [low .. high)  (if present)
while (low < high){
```

```
if (comp == 0) // item is at mid
     return mid;
if (comp < 0)  // item in [low .. mid)
                    // item in [low .. high)
else           // item in [mid+1 .. high)
                    // item in [low .. high)
```

```
}
return -1;      // item in [low .. high) and low >= high,
}                  // therefore item not present
```

Solution

The completed program is as shown in the figure below.

Program:

```
private int indexOf(String value, List<String> data){
int low = 0;
int high = data.size();
// item in [low .. high)  (if present)
while (low < high){
int mid  =  (low + high) / 2;
int comp =  value.compareTo(data.get(mid));
if (comp == 0) // item is at mid
     return mid;
if (comp < 0)  // item in [low .. mid)
high = mid;           // item in [low .. high)
else           // item in [mid+1 .. high)
low = mid + 1; // item in [low .. high)
}
return -1;      // item in [low .. high) and low >= high,
}                  // therefore item not present
```

3. The following skeleton of a program is the other form of binary search. [3 marks]

Program:

```
/* Return the index of where the item ought to be, whether present or
not.  (!) */
private int findIndex(String value, List<String> data){
// note: correct position might be at end (index =size)
int low = 0;
int high  =  data.size();      // index in [low .. high]

while (low < high){
    if (value.compareTo(data.get(mid)) > 0)  // index in [mid+1 .. high]
                       // index in [low .. high]  low <= high
    else               // index in [low .. mid]
```

```
                                    // index in [low .. high], low<=high
}
return low;                 // index in [low .. high] and low = high
                            // therefore index = low

}
```

Solution

The completed program is as shown in the figure below.

Program

```
/* Return the index of where the item ought to be, whether present or
not.  (!) */

private int findIndex(String value, List<String> data){

// note: correct position might be at end (index =size)
int low = 0;
int high = data.size();      // index in [low .. high]


while (low < high){
    int mid  =  (low + high) / 2;
    if (value.compareTo(data.get(mid)) > 0)  // index in [mid+1 .. high]
          low = mid + 1;      // index in [low .. high]  low <= high
    else                        // index in [low .. mid]
          high = mid;           // index in [low .. high], low<=high
}


return low;                     // index in [low .. high] and low =
high
                                // therefore index = low

}
```

4. Searching item using bisection algorithm. [7 marks]

Program:

```
bisection( -100, 100, (double x)-> {return (3*x*x*x - 4*x*x + 321);} );

bisection( -100, 100, (x)-> (3*x*x*x - 4*x*x + 321) );


public double bisection(double low, Double high, Function<Double,
Double> function){

double fLow = function.apply(low);

double fHigh = function.apply(high);
```

```
return Double.NaN;
}  // same side of axis
while (true) {

if (Math.abs(fMid)<THETA) {return mid;}
else if (Math.signum(fLow) == Math.signum(fMid) ){

}
else {

}
}
}
```

Solution

The completed program is as shown below.

Program:

```
bisection( -100, 100, (double x)-> {return (3*x*x*x - 4*x*x + 321);} );
bisection( -100, 100, (x)-> (3*x*x*x - 4*x*x + 321) );

public double bisection(double low, Double high, Function<Double,
Double> function){
double fLow = function.apply(low);
double fHigh = function.apply(high);
if (Math.abs(fLow)<THETA)  { return fLow; }
if (Math.abs(fHigh)<THETA) { return fHigh; }
if (Math.signum(fLow) == Math.signum(fHigh)) {
return Double.NaN;
}  // same side of axis
while (true) {
double mid = (low+high)/2;
double fMid = function.apply(mid);
if (Math.abs(fMid)<THETA) {return mid;}
else if (Math.signum(fLow) == Math.signum(fMid) ){
low = mid; fLow=fMid;
```

```
}
else {
high = mid;   fHigh=fMid;
}
}
}
```