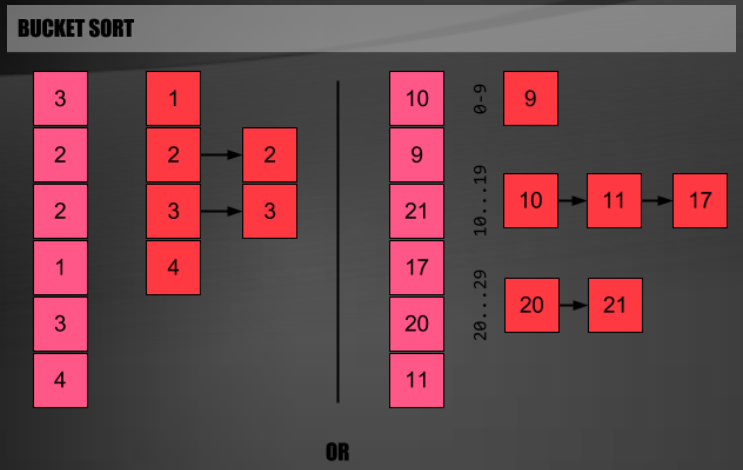
Telerik Academy



Sorting and Searching Algorithms

Data Structures and Algorithms

Telerik Software Academy

http://academy.telerik.com

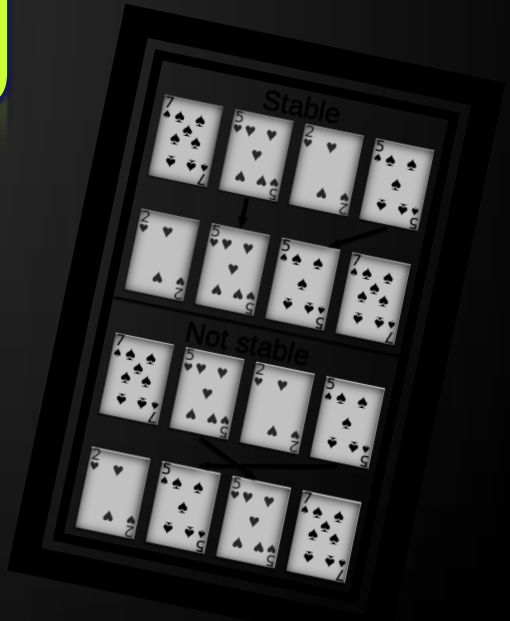
Telerik Academy

Table of Contents

- Sorting
  - Sorting and classification
  - Review of the most popular sorting algorithms
- Searching
  - Linear search
  - Binary search
  - Interpolation search
- Shuffling

Telerik Academy

Sorting



What is a Sorting Algorithm?

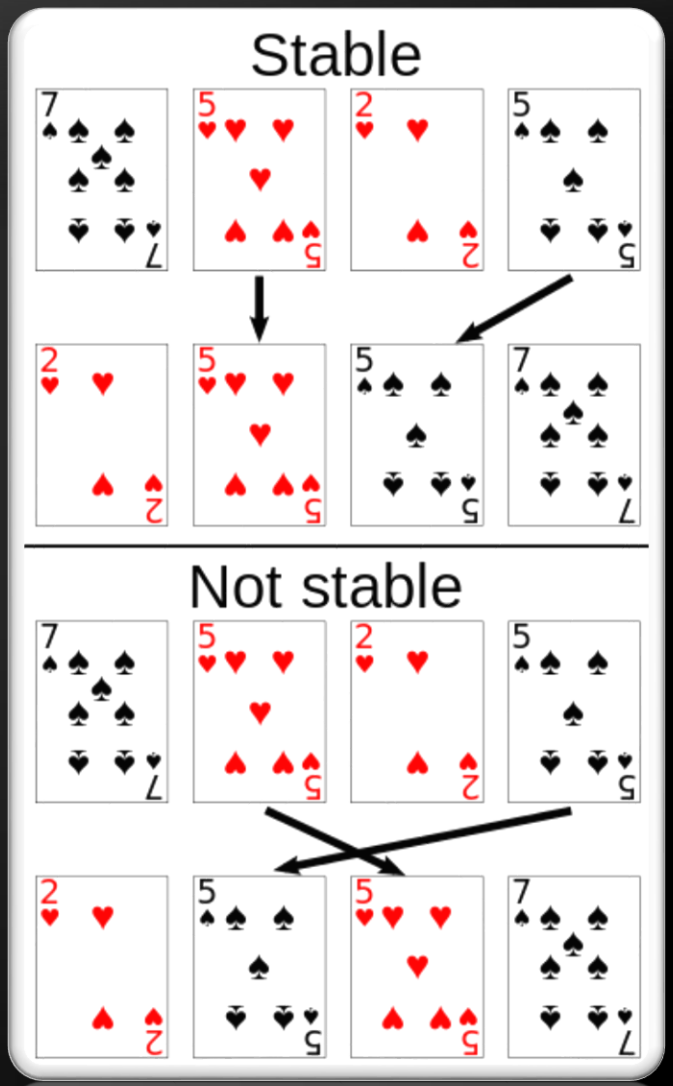
- Sorting algorithm
  - An algorithm that puts elements of a list in a certain order (most common lexicographically)
- More formally:
  - The output is in some (non-decreasing) order
  - The output is a permutation of the input
- Efficient sorting is important for
  - Producing human-readable output
  - Canonicalizing data
  - Optimizing the use of other algorithms
- Sorting presents many important techniques

Classification

- Sorting algorithms are often classified by
  - Computational complexity
    - worst, average and best behavior
  - Memory usage
  - Recursive or non-recursive
  - Stability
  - Whether or not they are a comparison sort
  - General method
    - insertion, exchange (bubble sort and quicksort), selection (heapsort), merging, serial or parallel…

Stability of Sorting

- Stable sorting algorithms
  - Maintain the relative order of records with equal values
- If two items compare as equal, then their relative order will be preserved
  - When sorting only part of the data is examined when determining the sort order



Selection sort

- Very simple and very inefficient algorithm
  - Best, worst and average case: n^2
  - Memory: 1 (constant, only for the min element)
  - Stable: No
  - Method: Selection

```
for (j = 0; j < n-1; j++) {
    /* find the min element in the unsorted a[j .. n-1] */
    iMin = j;
    for ( i = j+1; i < n; i++) {
        if (a[i] < a[iMin]) iMin = i;
    }
    if (iMin != j) swap(a[j], a[iMin]);
}
```

- http://en.wikipedia.org/wiki/Selection_sort

Bubble sort

- Repeatedly stepping through the list, comparing each pair of adjacent items and swap them if they are in the wrong order
  - Best case: n, worst and average case: n^2
  - Memory: 1, Stable: Yes, Method: Exchanging

```
procedure bubbleSort( A : list of sortable items )
   repeat
     swapped = false
     for i = 1 to length(A) - 1 inclusive do:
       /* if this pair is out of order */
       if A[i-1] > A[i] then
         /* swap them and remember something changed */
         swap( A[i-1], A[i] )
         swapped = true
       end if
     end for
   until not swapped
end procedure
```

- http://en.wikipedia.org/wiki/Bubble_sort

Insertion sort

- Builds the final sorted array one item at a time
  - Best case: n, worst and average case: n^2
  - Memory: 1, Stable: Yes, Method: Insertion

```
for i ← 1 to i ← length(A)-1
{
  valueToInsert ← A[i]
  holePos ← i
  while holePos > 0 and valueToInsert < A[holePos - 1]
  {
    A[holePos] ← A[holePos - 1] // shift the larger value up
    holePos ← holePos - 1       // move the hole position down
  }
  A[holePos] ← valueToInsert
}
```

- http://en.wikipedia.org/wiki/Insertion_sort

Quicksort

- First divides a large list into two smaller sub-lists then recursively sort the sub-lists
  - Best and average case: n*log(n), worst: n^2
  - Memory: log(n) stack space
  - Stable: Depends
  - Method: Partitioning

Stable implementation

```
function quicksort('array')
    if length('array') ≤ 1
        return 'array'  // an array of zero or one elements is already sorted
    select and remove a pivot value 'pivot' from 'array'
    create empty lists 'less' and 'greater'
    for each 'x' in 'array'
        if 'x' ≤ 'pivot' then append 'x' to 'less'
        else append 'x' to 'greater'
    return concatenate(quicksort('less'), 'pivot', quicksort('greater'))
```

- http://en.wikipedia.org/wiki/Quicksort

Merge Sort

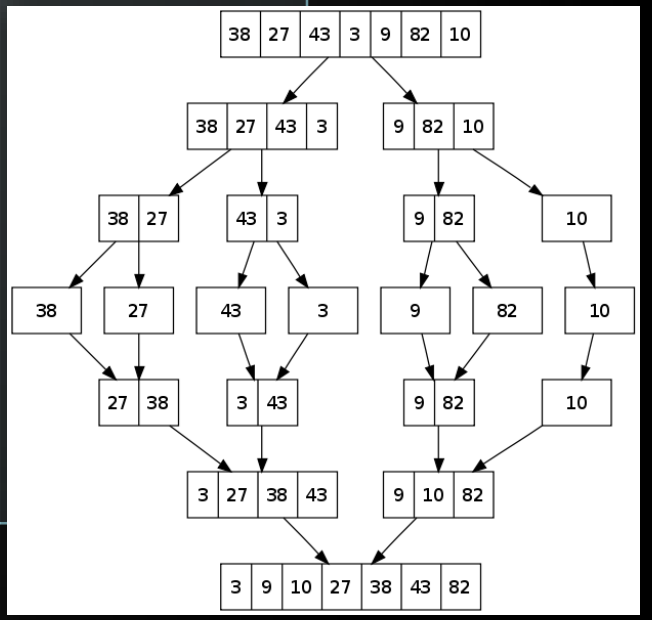
- Conceptually, a merge sort works as follows
  - Divide the unsorted list into n sublists, each containing 1 element (list of 1 element is sorted)
  - Repeatedly merge sublists to produce new sublists until there is only 1 sublist remaining
- Best, average and worst case: n*log(n)
- Memory: Depends; worst case is n
- Stable: Yes; Method: Merging
- Highly parallelizable (up to O(log(n)) using the Three Hungarian's Algorithm
- http://en.wikipedia.org/wiki/Merge_sort

Merge Sort Pseudocode

```
function merge_sort(list m)
    // if list size is 0 (empty) or 1, consider it sorted
    // (using less than or equal prevents infinite recursion for a
zero length m)
    if length(m) <= 1
        return m
    // else list size is > 1, so split the list into two sublists
    var list left, right
    var integer middle = length(m) / 2
    for each x in m before middle
         add x to left
    for each x in m after or equal middle
         add x to right
    // recursively call merge_sort() to further split each sublist
    // until sublist size is 1
    left = merge_sort(left)
    right = merge_sort(right)
    // merge the sublists returned from prior calls to merge_sort()
    // and return the resulting merged sublist
    return merge(left, right)
```

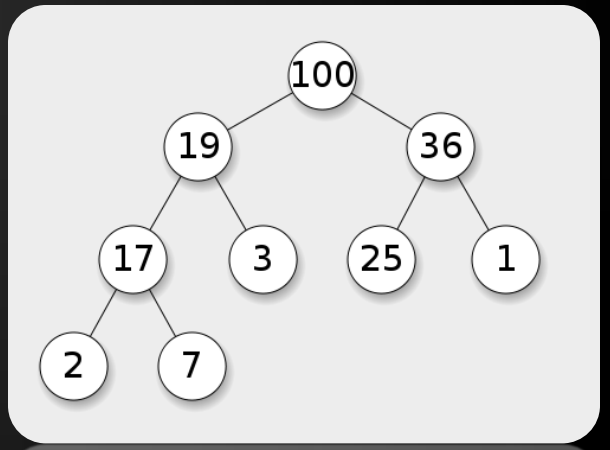
Merge Sort Pseudocode (2)

```
function merge(left, right)
    var list result
    while length(left) > 0 or length(right) > 0
        if length(left) > 0 and length(right) > 0
            if first(left) <= first(right)
                append first(left) to result
                left = rest(left)
            else
                append first(right) to result
                right = rest(right)
        else if length(left) > 0
            append first(left) to result
            left = rest(left)
        else if length(right) > 0
            append first(right) to result
            right = rest(right)
    end while
    return result
```



Heap

- Specialized tree-based data structure that satisfies the heap property:
  - Parent nodes are always greater (less) than or equal to the children
    - No implied ordering between siblings or cousins



find-min	Θ(1)
delete-min	Θ(log n)
insert	Θ(log n)
decrease-key	Θ(log n)
merge	Θ(n)

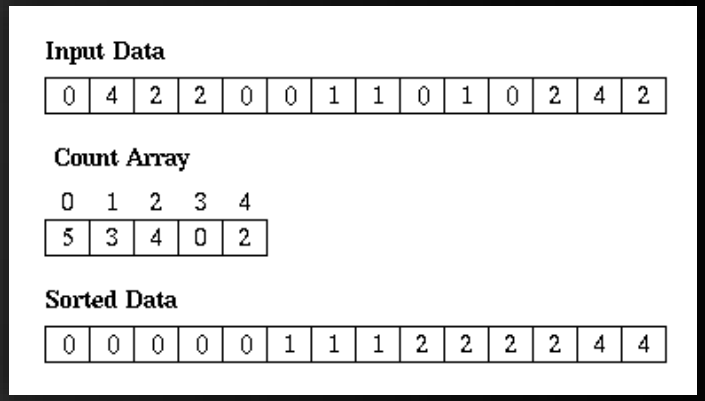
- en.wikipedia.org/wiki/Heap_(data_structure)

Heapsort

- Can be divided into two parts
  - In the first step, a heap is built out of the data
  - A sorted array is created by repeatedly removing the largest element from the heap
- Best, average and worst case: `n*log(n)`
- Memory: Constant - `O(1)`
- Stable: No
- Method: Selection
- http://en.wikipedia.org/wiki/Heapsort

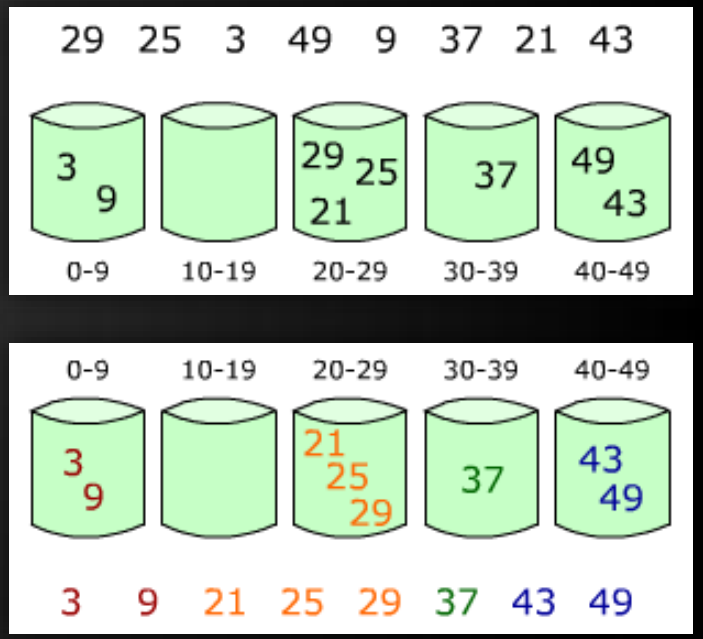
Counting sort

- Algorithm for sorting a collection of objects according to keys that are small integers
- Not a comparison sort
- Average case: `n + r`
- Worst case: `n + r`
  - `r` is the range of numbers to be sorted
- Stable: Yes
- Memory: `n + r`
- http://en.wikipedia.org/wiki/Counting_sort



Bucket sort

- Partitioning an array into a number of buckets
  - Each bucket is then sorted individually
- Not a comparison sort
- Average case: n + k
  - k = the number of buckets
- Worst case: n^2 * k
- Stable: Yes
- Memory: n * k
- http://en.wikipedia.org/wiki/Bucket_sort



Telerik Academy

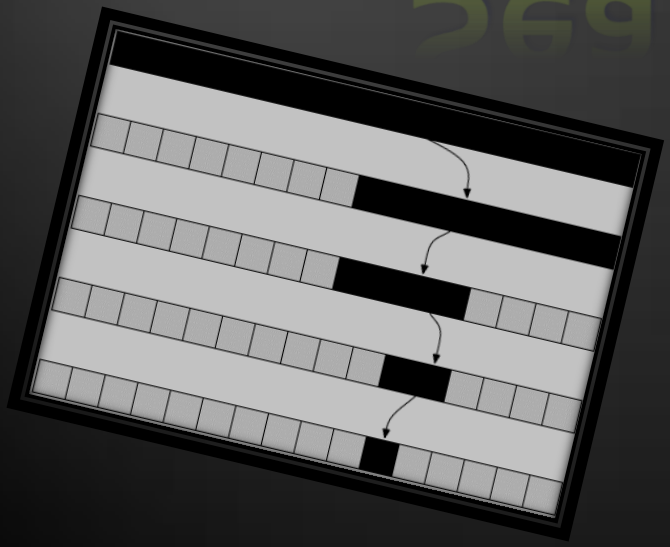
Comparison of Sorting Algorithms

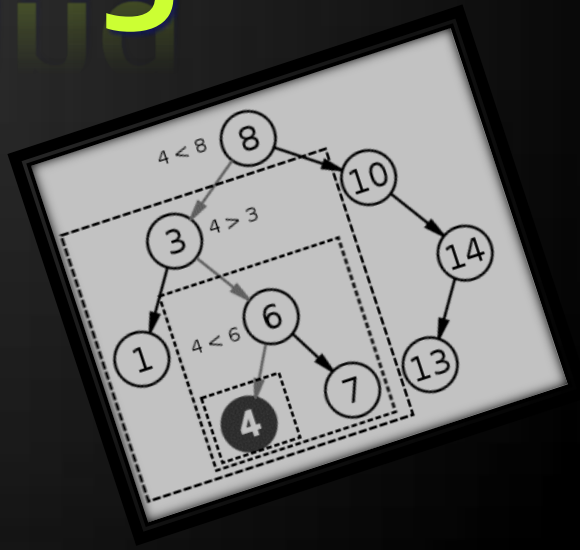
- There are hundreds of sorting algorithms

Name	Best	Average	Worst	Memory	Stable	Method
Selection sort	n^2	n^2	n^2	1	No	Selection
Bubble sort	n	n^2	n^2	1	Yes	Exchanging
Insertion sort	n	n^2	n^2	1	Yes	Insertion
Quicksort	n*log(n)	n*log(n)	n^2	log(n)	Depends	Partitioning
Merge sort	n*log(n)	n*log(n)	n*log(n)	Depends worst case is n	Yes	Merging
Heapsort	n*log(n)	n*log(n)	n*log(n)	1	No	Selection
Bogosort	n	n*n!	n*n!	1	No	Luck
…	…	…	…	…	…	…

Telerik Academy

Searching





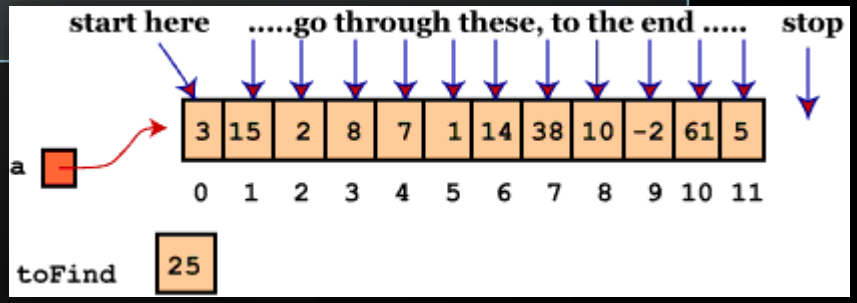
Search Algorithm

- An algorithm for finding an item with specified properties among a collection of items
- Different types of searching algorithms
  - For virtual search spaces
    - satisfy specific mathematical equations
    - try to exploit partial knowledge about structure
  - For sub-structures of a given structure
    - graph, a string, a finite group
  - Search for the max (min) of a function
  - etc.

Linear Search

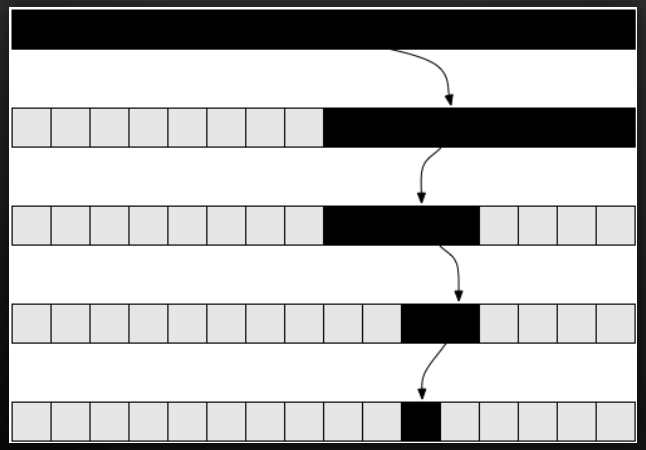
- Method for finding a particular value in a list
  - Checking every one of the elements
  - One at a time in sequence
  - Until the desired one is found
- Worst and average performance: O(n)

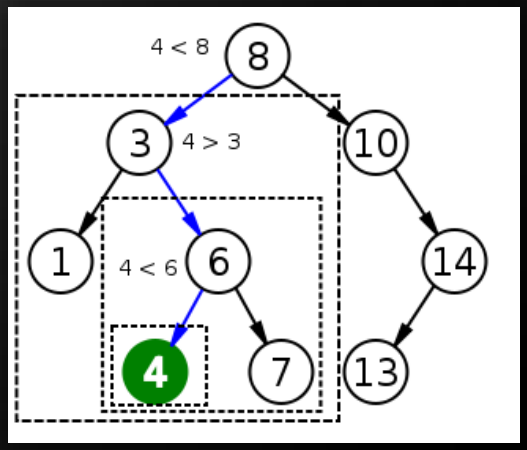
```
for each item in the list:
    if that item has the desired value,
        stop the search and return the item's location.
return nothing.
```



Binary Search

- Finds the position of a specified value within a `sorted` data structure
- In each step, compare the input with the middle
  - The algorithm repeats its action to the left or right sub-structure
- Average performance: `O(log(n))`





Recursive Binary Search

```
int binary_search(int A[], int key, int imin, int imax)
{
  if (imax < imin)
     // set is empty, so return value showing not found
     return KEY_NOT_FOUND;
  else
  {
     // calculate midpoint to cut set in half
     int imid = midpoint(imin, imax);
     if (A[imid] > key)
        // key is in lower subset
        return binary_search(A, key, imin, imid-1);
     else if (A[imid] < key)
        // key is in upper subset
        return binary_search(A, key, imid+1, imax);
     else
        // key has been found
        return imid;
  }
}
```

Iterative Binary Search

```
int binary_search(int A[], int key, int imin, int imax)
{
   // continue searching while [imin,imax] is not empty
   while (imax >= imin)
   {
      /* calculate the midpoint for roughly equal partition */
      int imid = midpoint(imin, imax);
      // determine which subarray to search
      if (A[imid] < key)
         // change min index to search upper subarray
         imin = imid + 1;
      else if (A[imid] > key)
         // change max index to search lower subarray
         imax = imid - 1;
      else
         // key found at index imid
         return imid;
   }
   return KEY_NOT_FOUND;
}
```

Interpolation Search

- An algorithm for searching for a given key value in an indexed array that has been ordered by the values of the key
  - Parallels how humans search through a telephone book
  - Calculates where in the remaining search space the sought item might be
    - Binary search always chooses the middle element
- Average case: `log(log(n))`, Worst case: `O(n)`
- http://youtube.com/watch?v=l1ed_bTv7Hw

Interpolation Search Sample Implementation

```
public int interpolationSearch(int[] sortedArray, int toFind){
  // Returns index of toFind in sortedArray, or -1 if not found
  int low = 0;
  int high = sortedArray.length - 1;
  int mid;
  while(sortedArray[low] <= toFind && sortedArray[high] >= toFind) {
    mid = low + ((toFind - sortedArray[low]) * (high - low)) /
                (sortedArray[high] - sortedArray[low]);
                // out of range is possible here
    if (sortedArray[mid] < toFind)
      low = mid + 1;
    else if (sortedArray[mid] > toFind)
      high = mid - 1;
    else
      return mid;
  }
  if (sortedArray[low] == toFind) return low;
  else return -1; // Not found
}
```

Shuffling

```
// Shuffle using Fisher-Yates (Knuth) algorithm
for (int i = len; i > 0; i--)          - Count DOWN
{
    int j = (int)(Math.random() * i);  - Index from 0..i

    int temp = lotteryBalls[j];
    lotteryBalls[j] = lotteryBalls[i];
    lotteryBalls[i] = temp;
}
```

Shuffling

- A procedure used to randomize the order of items in a collection
  - Generating random permutation

- http://en.wikipedia.org/wiki/Shuffling

Telerik Academy

Fisher–Yates shuffle algorithm

```
public static IEnumerable<T> Shuffle<T>(this IEnumerable<T> source)
{
    var array = source.ToArray();
    var n = array.Length;
    for (var i = 0; i < n; i++)
    {
        // Exchange a[i] with random element in a[i..n-1]
        int r = i + RandomProvider.Instance.Next(0, n - i);
        var temp = array[i];
        array[i] = array[r];
        array[r] = temp;
    }
    return array;
}

public static class RandomProvider
{
    private static Random Instance = new Random();
}
```

Telerik Academy

Sorting and Searching Algorithms

http://academy.telerik.com

Homework

- Open `Sorting-and-Searching-Algorithms-Homework.zip` and:
  1. Implement `SelectionSorter.Sort()` method using selection sort algorithm
  2. Implement `Quicksorter.Sort()` method using quicksort algorithm
  3. Implement `MergeSorter.Sort()` method using merge sort algorithm
  4. Implement `SortableCollection.LinearSearch()` method using linear search
     - Don't use built-in search methods. Write your own.

Homework (2)

5. Implement `SortableCollection.BinarySearch()` method using binary search algorithm
6. Implement `SortableCollection.Shuffle()` method using shuffle algorithm of your choice
   - Document what is the complexity of the algorithm
7. * Unit test sorting algorithms
   - `SelectionSorter.Sort()`
   - `Quicksorter.Sort()`
   - `MergeSorter.Sort()`
8. * Unit test searching algorithms
   - `SortableCollection.LinearSearch()`
   - `SortableCollection.BinarySearch()`

Free Trainings @ Telerik Academy

- C# Programming @ Telerik Academy
  - csharpfundamentals.telerik.com

- Telerik Software Academy
  - academy.telerik.com

- Telerik Academy @ Facebook
  - facebook.com/TelerikAcademy

- Telerik Software Academy Forums
  - forums.academy.telerik.com