**BINARY SEARCH**

Finding a particular value in a linear array, by ruling out half of the data at each step. finds the median, makes a comparison to determine whether the desired value comes before or after it, and then searches the remaining half in the same manner. A binary search is an example of a divide and conquer algorithm   
 **ITERATIVE  
  
def** binary\_search**(**l, value**)**:  
 low = **0**  
 high = **len(**l**)**-**1**  
 **while** low <= high:   
 mid = **(**low+high**)**//**2**  
 **if** l**[**mid**]** > value: high = mid-**1**  
 **elif** l**[**mid**]** < value: low = mid+**1**  
 **else**: **return** mid  
 **return** -**1  
  
RECURSIVE**

**def** binary\_search**(**l, value, low = **0**, high = -**1)**:  
 **if** **not** l: **return** -**1**  
 **if(**high == -**1)**: high = **len(**l**)**-**1**  
 **if** low >= high:  
 **if** l**[**low**]** == value: **return** low  
 **else**: **return** -**1**  
 mid = **(**low+high**)**//**2**  
 **if** l**[**mid**]** > value: **return** binary\_search**(**l, value, low, mid-**1)**  
 **elif** l**[**mid**]** < value: **return** binary\_search**(**l, value, mid+**1**, high**)**  
 **else**: **return** mid  
  
**PSEUDOCODE**BIN-TREE-FIND(t,target)   
 IF t.value = target OR t = 0   
 RETURN TRUE   
 ELSE IF target < t.value   
 RETURN BIN-TREE-FIND(t.left, target)   
 ELSE RETURN BIN-TREE-FIND(t.right, target)   
 RETURN FALSE  
  
PREORDER(tree)

PRINT tree.value

IF tree.left ≠ 0

PREORDER (tree.left)

IF tree.right ≠ 0

PREORDER (tree.right)

INORDER(tree)

IF tree.left ≠ 0

PREORDER (tree.left)

PRINT tree.value

IF tree.right ≠ 0

PREORDER (tree.right)

POSTORDER(tree)

IF tree.left ≠ 0

PREORDER (tree.left)  
 IF tree.right ≠ 0

PREORDER (tree.right)

PRINT tree.value

**MERGESORT**

Mergesort is also called divide and conquer algorithm, because it divides the original data into smaller pieces of data to solve the problem. Merge sort works in the following way:

* Divide into 2 collections. Mergesort will take the middle index in the collection and split it into the left and right parts based on this middle index.
* Resulting collections are again recursively splited and sorted
* Once the sorting of the two collections is finished, the results are merged
* Now Mergesort it picks the item which is smaller and inserts this item into the new collection.
* Then selects the next elements and sorts the smaller element from both collections

**QUICKSORT**

Quick sort is better than merge sort from a memory usage comparison. Because quick sort doesn’t require additional storage to work. It only uses a small auxiliary stack.

Skiena discovered via experimentation that a properly implemented quicksort is typically 2-3 times faster than mergesort or heapsort

* If the array contains only 1 element or 0 elements then the array is sorted.
* If the array contains more than 1 element:
* Select randomly an element from the array. This is the "pivot element".
* Split into 2 arrays based on pivot element: smaller elements than pivot go to the first array, the ones above the pivot go into the second array
* Sort both arrays by recursively applying the Quicksort algorithm.
* Merge the arrays

**BINARY SEARCH**

Binary search is also easy to implement and easy to understand. We actually make use of binary search without knowing when searching in a dictionary for a specific word or in a phone book. Technically it’s not the same but it is a very similar process.

We open a dictionary more or less by the middle if the word we are searching for starts in a letter over the middle, we discard the first half of the dictionary and now focus more or less by the middle of the second half. If the word is below the middle of the second half, we discard the top half and keep applying this same technique until we find our word

* Receive a sorted array of n elements
* Compute middle point
* If index toSearch is less than array[mid] then highestIndex = mid -1 (This changes mid)
* If index toSearch is greater than array[mid] then lowestIndex = mid +1
* else if index isn't greater nor less than array[mid], this means it's equal so return 0
* If not greater, less nor equal, then it's not in the array so return -1

**SELECTION SORT**

It is called selection sort because it repeatedly selects the smallest remaining item:

1. Find the smallest element. Swap it with the first element.
2. Find the second smallest element. Swap it with the second element
3. Find the third smallest element. Swap it with the third element
4. Repeat finding the smallest element and swapping in the correct position until the list is sorted

## Array Sorting Algorithms

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Time Complexity | | | Space Complexity |
|  | Best | Average | Worst | Worst |
| [Quicksort](http://en.wikipedia.org/wiki/Quicksort) | Ω(n log(n)) | Θ(n log(n)) | O(n^2) | O(log(n)) |
| [Mergesort](http://en.wikipedia.org/wiki/Merge_sort) | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(n) |
| [Heapsort](http://en.wikipedia.org/wiki/Heapsort) | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(1) |
| [Bubble Sort](http://en.wikipedia.org/wiki/Bubble_sort) | Ω(n) | Θ(n^2) | O(n^2) | O(1) |
| [Insertion Sort](http://en.wikipedia.org/wiki/Insertion_sort) | Ω(n) | Θ(n^2) | O(n^2) | O(1) |
| [Selection Sort](http://en.wikipedia.org/wiki/Selection_sort) | Ω(n^2) | Θ(n^2) | O(n^2) | O(1) |
| [Tree Sort](https://en.wikipedia.org/wiki/Tree_sort) | Ω(n log(n)) | Θ(n log(n)) | O(n^2) | O(n) |

|  |
| --- |
| BELLMANFORD(v, e, s)  distance[v] ← inf  predecessor[v] ← null  distance[source] ← 0  FOR i FROM 1 TO size(vertices)-1  FOR each edge (u, v) with weight w in edges:  IF distance[u] + w < distance[v]:  distance[v] ← distance[u] + w  predecessor[v] ← u  FOR each edge (u, v) with weight w in edges:  IF distance[u] + w < distance[v]:  ERROR “Negative-weight cycle"  RETURN distance, predecessor |

DEPTH-FIRST-SEARCH (G, v)

S←new Stack()

visited←[]

S.push(v)

WHILE S is not empty

u←S.pop()

IF u is not in visited

visited.append(u)

## FOR all edges, e, from u, S.push(e.to) RETURN visited **За SAP в България**

Българският филиал на световния софтуерен лидер SAP SE присъства на българския пазар вече повече от 15 години. През декември 1999г., SAP SE регистрира дъщерното си дружество **SAP България**, а през 2002г. инвестира и в развоен център – **SAP Labs България.**  
На българския пазар SAP България предлага цялото си портфолио от продукти в 5 иновационни направления: Applications, Analytics, Mobility, Database&Technology и Cloud, както и силен фокус върху създаването на солидна екосистема от партньори чрез програмата SAP PartnerEdge.

Така наречените "Лабове" (laboratories) в SAP са центровете за разработка на софтуер, които създават иновативни софтуерни продукти, привличайки таланти и използвайки достиженията на технологиите. **SAP Labs България** е създадена през 2002 г. и в момента в компанията работят повече от 600 професионалиста, които създават водещи Cloud и In-Memory технологични решения, допълнени от качествени услуги по поддръжка и технологично консултиране.SAP Labs България е един от пионерите и лидерите в прилагането на Lean философия и гъвкави (agile) методологии при създаването на софтуер в България. Компанията е активен член на българското бизнес общество и организира и подкрепя бизнес събития с фокус върху информационните технологии и създаването на софтуер.

SAP products being used in the German Football Team to analyse each game to find out what would be the best choice to make every minute.