1 Linear Search – Prosecode

LinearSearch(L, x) returns the position of an element x in a list L or an invalid position if L does not contain x. Therefore the algorithm iterates linearly through L, comparing each iterated element L_i with x. If L_i equals x its position i is returned. If all elements have been iterated but none is equal to x an invalid position Λ is returned.

1. (Initialize) $p \leftarrow \Lambda$
2. (Iterate through L) For each position i in L
   (a) If L_i is equal to x, set $p \leftarrow i$ and stop iterating.
3. (Return) Return p

2 Heapsort – Prosecode

HeapSort(L) sorts the unordered list L. The algorithm splits L into a sorted part S and an unsorted part U. S is iteratively build by removing the maximum element of U using a max-heap H. A max-heap is a binary tree where the value of each node is larger or equal to its children. So the root of H is always the maximum of U. HeapSort iteratively moves the root of H to S, thus building a sorted list. After removing the root, H is updated to maintain its max-heap property. When U is empty, L is completely sorted.

1. (Initialize) $U \leftarrow L, S \leftarrow \emptyset$, build up the max-heap H
2. (Iteratively build S) While H is not empty
   (a) remove root of H and insert it in the end of S
   (b) (siftDown) update H to maintain max-heap property

3 Quicksort – LiterateCode

QuickSort(L) sorts the unordered list L by partitioning L into two sublists S and G and recursively sorting these sublists. The subdivision is done by picking a so called *pivot element* p from L and placing all elements in L smaller than p in S and all elements greater or equal than p into G. Then quicksort is applied recursively to the individual sublists S and G. When each sublist contains only one element, the maximum recursion depth is reached and L is sorted.

The major steps of the algorithm are as follows.

1. Finish if $|L| == 1$
2. Pick a pivot element $p \in L$.
3. Partition L into the sublists S and G
4. Recursively apply step 1, 2 and 3 on the two sublists

We now examine the steps in detail.

1. Finish if $|L| == 1$

   When L consists of one element only, L is naturally sorted and we can finish

2. Pick a pivot element $p \in L$.

   The choice of the pivot element has a crucial effect on the runtime of the algorithm because in the worst case the maximum size of the list to sort in the next recursion step is reduced only by 1. E.g. when the first element of L is chosen and L is already sorted. This leads to a linear recursion depth of $\mathcal{O}(|L|)$. Choosing the median would halve the size of the list in the next recursion step, leading to $\mathcal{O}(log(|L|))$ recursion depth. But since we do not know the median prior to sorting, we choose a random pivot.

3. Partition L into the sublists S and G

   In the partition step every element of L is compared to the pivot and then inserted in the appropriate list, leading to an effort of $\mathcal{O}(|L|)$

4. Recursively apply step 1, 2 and 3 on the two sublists

   The sublists S and G are individually sorted by recursively applying quicksort on them. Thus the overall costs for execution time depend on the recursion depth ($\mathcal{O}(log(|L|))$) multiplied with the cost in each recursion step ($\mathcal{O}(|L|)$). This leads to an overall average cost of $\mathcal{O}(|L|log(|L|))$ and in the worst case, if the pivot is chosen badly, $\mathcal{O}(|L|^2)$.