Information on Sorting

This document gives an overview of different kinds of sorting algorithms relevant to the labs. You can find more information in the recommended textbooks and online. Note that there is a tool on the course website visualising some of these algorithms. There are lots of similar visualisations available online elsewhere.

Note that the last section on *Additional Concepts* is important.

1 Iterative Sorting Algorithms

1.1 Insertion Sort

Insertion sort is an intuitive algorithm that constructs a sorted output list directly. At each iteration, one element from the input is taken and inserted into the right position in the ordered list.

The idea of insertion sort is given as follows:

1. Initialize the output sorted list with the first element;
2. Take the next item, scanning the sorted list from back to front;
3. Find the right place to insert this element;
4. Insert into that position;
5. Repeat steps 2 to 4.

Scanning the sorted list from back to front ensures stability. The sort can also be in-place by dividing the initial list into the sorted and unsorted part.

1.2 Bubble Sort

Bubble sort is a simple sorting algorithm that repeatedly processes the input list, comparing two elements at a time and exchanging them if they are in the wrong order. The sorting is finished if no more swaps are needed.

The name "Bubble sort" comes from the way smaller or bigger elements would "bubble" to the top of the list through swaps.

The idea of bubble sort is given as follows:

1. Compare adjacent elements. Swap those in the wrong order;
2. Repeat step 1 through the given list. After this step, the largest or smallest item would bubble to the top;
3. Repeat above steps each time with one less item, until there's no other pair of items that needs to be compared.

1.3 Bucket Sort

Bucket sort works by partitioning a the data into a finite number of buckets. The data needs to be easily bucketed e.g. numeric data can be bucketed using modular arithmetic; any data can be bucketed using bit information. Elements within each bucket are sorted individually, then merged together in order. It is a Distribution sort. The computational complexity estimates involve the number of buckets.

The idea of bucket sort with n buckets is given as follows:

1. Set up a fixed array of n buckets;
2. Go through the given list, putting each element into corresponding bucket;
3. Sort each non-empty bucket (e.g. using insertion sort). You don't need to sort buckets if they only contain one element;
4. Put elements from non-empty buckets back to the original list.

Pseudocode for bucket-sort is provided in Figure 1.

```
function bucket-sort(array, length, n){
  buckets ← new array of n initially empty lists
  for each item i in array do
    // The key must be between 0 and n-1
    let k be the key of item i
    insert i into buckets[k]
  for i = 0 to n - 1 do
    // Assuming an in-place sort
    sort(buckets[i])
  concatenate buckets[0], ..., buckets[n-1]
}
```

Figure 1: Algorithm for Bucket Sort

1.4 Radix Sort

Radix sort is a non-comparative integer sorting algorithm that processes integer along individual digits, either starting from the least significant digit (LSD, i.e., the rightmost digit) or from the most significant digit (MSD, i.e., the leftmost digit).

Processing each digit is usually done using bucket sort, which is efficient since there are usually only a small number of digits.

The idea of **least significant digit radix sort** is given as follows:

1. Take the least significant digit of each key.
2. Sort the list of elements based on that digit, grouping elements with the same digit into one bucket.
3. Repeat the grouping process with each more significant digit.
4. Concatenate the buckets together in order.

The sequence in which digits are processed by a most significant digit (MSD) radix sort is the opposite of the sequence in which digits are processed by an LSD radix sort.

Pseudocode is provided in Figure 2.

```
function radix-sort(array, length) {
  MAX_NUM ← the maximum number in array
  DIGIT_NUM ← number of digits needed to represent MAX_NUM
  buckets ← new array of BASE(e.g. 10) initially empty lists
  for each i in DIGIT_NUM
    for each item in array
      d ← i^th least significant digit of item
      insert item into bucket[d]
    for j = 0 to BASE - 1 do
      sort(buckets[j])
  concatenate buckets[0], ..., buckets[BASE-1]
}
```

Figure 2: Algorithm for LSD Radix Sort

1.5 Selection Sort

Selection sort is a straightforward method that each time picks the minimum element from the list remaining to be sorted and moves to the end of an ordered list. For example, given an array

8 3 2 1 7 4 6 5

Assume the left part is the sorted sublist (currently empty) while the rest is an unsorted one. The algorithm progresses step by step as follows:

(i) 1 3 2 8 7 4 6 5 // swap(8, 1)

(ii) 1 2 3 8 7 4 6 5 // swap(3, 2)

(iii) 1 2 3 4 7 8 6 5 // swap(8, 4)

(iv) 1 2 3 4 5 8 6 7 // swap(7, 5)

(v) 1 2 3 4 5 6 8 7 // swap(8, 6)

(vi) 1 2 3 4 5 6 7 8 // swap(8, 7)

1.6 Shell Sort

Shell sort works by dividing the whole list into subsets, each of which is then sorted by applying insertion sort to it. This step is repeated until the size of subsets decreases to 1.

The special part of shell sort is how sublists are chosen. Instead of contiguous items, each subset contains elements that are i elements apart, while i is called **increment** or **gap**, and **shell sort** is also called **diminishing increment sort**. The running time of Shell Sort depends heavily on the choice of gap sequences.

Shell sort can be seen as an advanced version of insertion sort. As it works on short sublists first; when lists get longer they are almost sorted in order. Insertion sort works efficiently in both situations.
For example, given an array

13 49 38 65 97 26 13 27 49 55 41 94 60 27 73 25 39 10

Starting from setting gap as 5, we can rewrite the array in a table of 5 columns to better present the algorithm. Now each **column** is a subset containing every 5th number in array.

13 49 38 65 97
26 13 27 49 55
41 94 60 27 73
25 39 10

Then we sort each **column**

13 13 10 27 55
25 39 27 49 73
26 49 38 65 97
41 94 60

that gives the whole array as

13 13 10 27 55 25 39 27 49 73 26 49 38 65 97 41 94 60

The next gap would be set as 3.

13 13 10
27 55 25
39 27 49
73 26 49
38 65 97
41 94 60

After sorting each column, it turns to be

13 13 10
27 27 25
38 27 49
39 55 49
41 65 60
73 94 97

And finally, the gap would be 1, which means a simple insertion sort to the whole list. **An already optimized version of pseudocode is provided in Figure 1.** Think about how it performs insertion sort on each sublist.

```
function shell-sort(array, length){
  gap ← half of length
  while gap > 0 do
        for i = gap .. length - 1 do:
            temp ← a[i]
            j ← i
            while j ≥ gap and a[j - gap] > temp
            do:
                a[j] ← a[j - gap]
                j ← j - gap
            a[j] ← temp
        gap ←  half of gap
}
```

Figure 3: Algorithm for Shell Sort

It is also worth noticing that we are choosing gap starting from length / 2, halving each time until decreasing to 1. There are more efficient choices. Check here if you are interested in it.

2 Recursive Sorting Algorithms

2.1 Merge Sort

Merge sort is an efficient, divide and conquer sorting algorithm based on the merge operation.

The idea of merge sort is given as follows:

1. Recursively dividing the input list into equal-sized sublists until the sublists are trivially sorted;
2. Merge the sublists while returning up the call chain.

Figure 4 gives pseudocode for merge sort.

```
// One could implement this recursively but then the length of
    lists
// that could be merged is limited by the size of the call stack
merge(L_1, L_2){
    if one of the lists is empty return the other list
    Let Merged be big enough to contain both lists
    l1_index, l2_index, m_index = 0
    while each of the indexes is still valid
      if L_1[l1_index] > L_2[l2_index] then
          Merged[m_index++] = L_1[l1_index++]
      else
          Merged[m_index++] = L_2[l2_index++]
    if one of the lists still contains elements copy these into
       Merged
    return Merged
}

mergeSort(L){
    if |L| ≤ 1
      return L
    Split L into two roughly equal halves L = L_l + L_r
    return merge(mergeSort(L_l), mergeSort(L_r))
}
```

Figure 4: Algorithm for Merge Sort

2.2 Quick Sort

Quick sort is an efficient, divide and conquer sorting algorithm based on partitioning.

The idea of quick sort is given as follows:

1. Pick one element called "pivot" from the list.
2. Reorder the list so that all elements smaller than pivot come before the pivot; while those with values greater than pivot come after it. After this partitioning, the pivot is in its final position.
3. Recursively apply step 1-2 to the two sublists divided by pivot.

Figure 5 gives pseudocode for quick sort. Note that this pseudocode is not 'in-place' i.e. it assumes that we create new lists rather than editing the existing lists. What are the memory requirements of doing it like this? How can it be made in-place?

Note that the choice of pivot can be important for deciding the complexity of the algorithm – notice what happens if we make the 'worst' choice of pivot on each step.

```
function quicksort(L){
    if length of L ≤ 1
      return L
    remove an element x from L to use as a pivot
    L≤ := elements of L less than or equal to x
    L> := elements of L greater than x
    Ll := quicksort(L≤)
    Lr := quicksort(L>)
    return Ll + [x] + Lr
}
```

Figure 5: Algorithm for Quick Sort

3 Additional Concepts

3.1 Stability

A sorting algorithm is said to be stable if any two records with equivalent keys retain their relative order in the sorted output. You met this concept previously in Lab 4 when inserting people into the linked list in sorted order. In that lab, if two people had the same age or name then the person who was inserted first should have appeared first.

3.2 Criteria for choosing a sorting algorithm

If you are dazzled by all the sorting algorithms and wonder which one to choose when it comes to real life problems, consider the following properties:

- Time complexity - It is usually $O(n^2)$ for most simple algorithms. If there are huge data sets, you might want to choose a more sophisticated one that has $\mathrm{O}(n \log n)$ time complexity on average cases.
- Space complexity - How much auxiliary memory is allowed here?
- Stability - Do we need to preserve the relative order of equivalent keys in output?
- Complexity of algorithm itself - For example, insertion sort, bubble sort are elementary algorithms; while quick sort, merge sort, bucket sort and radix sort are sophisticated ones.

For more information, you might find this helpful.

There is always a trade-off in between evaluation criteria. It is hard to find an algorithm with low time complexity requiring no additional memory at the same time. Or maybe you could be the first person who invents a perfect solution?

3.3 Binary Search

You should have met this general idea before. If a list is sorted and you pick a random element of the list then you can always tell if another element should appear before or after

that element. This gives us a way of quickly searching the list by recursively halving the list. You should be used to this as a typical example where we get logarithmic complexity – do you remember why?

3.4 Amortization

I'm sure the following scenario is familiar to you. You can manually edit some data or write a script to do it for you. The script will take 5-10 minutes to write whilst manually editing will take under 5 minutes. So you manually edit the data. But then you have to repeat this 4 or 5 times; it would have been quicker to write the script in the first place.

This general idea is known as amortization. Fans of xkcd may already be familiar with this concept and you will certainly meet it again when looking at advanced data structures.

Once learning about these ideas (binary search and amortization) it is tempting to apply it everywhere. But sometimes it is not worth the cost. If we have a large *unordered* list and we need to find a single item in it then the cost of first sorting the list and then using binary search will definitely be more expensive than a simple linear search of the list.