Sorting

Note that they're indexing from 1

1 Insertion Sort

```
InsertionSort (a_1, ..., a_n ∈ ℝ, n ⩾ 2)
for j=2 to n do
    x = a_j
    i=j-1
    while i>0 and a_i > x do
        a_{i+1} = a_i
        i=i-1
    end while
    a_{i+1} = x
end for
```

We know:

- When j has a certain value, it insets the jth element into already sorted sequence $a_1, ..., a_{j-1}$
- Can be proved correct by using invariant "after jth iteration first j+1 elements are in order" (not necessarily in the correct position for at the end of the sort)
- Running time between n-1 and $\frac{n(n-1)}{2}$ - worst case $O(n^2)$

Method:

- After the first n cycles, the first n+1 numbers are in order (not necessarily wrt the rest of the list)
- If there are repeats as the algorithm looks for strictly greater, suppose the list started with 2 twos, the algorithm would look at it as sorted. The order of repeated elements will not be changed. We call an algorithm that works like this **stable**

2 Selection sort

```
SelectionSort (a_1, ..., a_n ∈ ℝ, n ⩾ 2)
for i=1 to n-1 do
    elem = a_i
    pos = i
    for j=i+1 to n do
        if a_j<elem then
            elem=a_j
            pos=j
        end if
    end for
    swap a_i and a_pos
end for
```

How does it work?

- Invariant: after ith iteration positions 1,...,i contain the overall i many smallest elements in order
- Not necessarily the first i elements
- In ith iteration of outer loop, we search the ith smallest element in remainder (positions $i + 1, ..., n$) of input and swap it into position i
  - elem keeps track of the current idea of value ith smallest element

- pos keeps track of current idea of position of the ith smallest element

Time Complexity:

$$\begin{aligned}\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1 &= \sum_{i=1}^{n-1}(n-i)\\ &= \left(\sum_{i=1}^{n-1} n\right) - \left(\sum_{i=1}^{n-1} i\right)\\ &= \frac{n(n-1)}{2}\\ &= O\left(n^2\right)\end{aligned}$$

Always do the same number of comparisons, no best or worst case, always takes the same number of comparisons for a list of any given length

This algorithm is unstable, the order of repeated elements will not be preserved

3 Bubble sort

```
BubbleSort-1 (a_1, ..., a_n ∈ ℝ, n ⩾ 2)
for i=1 to n-2 do
    for j=1 to n-1 do
        if a_j > a_{j+1} then
            swap a_j and a_{j+1}
        end if
    end for
end for
```

This can be improved by keeping track of whether or not an element was swapped

```
BubbleSort-1 (a_1, ..., a_n ∈ ℝ, n ⩾ 2)
for i=1 to n-1 do
    swaps=0
    for j=1 to n-1 do
        if a_j > a_{j+1} then
            swap a_j and a_{j+1}
            swaps=swaps+1
        end if
    end for
    if swaps ==0 then
        break
    end if
end for
```

This algorithm is stable

3.1 Correctness

A sequence $a_1, ...a_n$ is sorted if for every adjacent pair a_i, a_{i+1} we have $a_i \leqslant a_{i+1}$
Bubble sort achieves just that
Good way to think about it: the ith iteration of the outer loop bubbles the ith largest element to where it belongs

3.2 Time

$$\begin{aligned}\sum_{i=1}^{n-1}\sum_{j=1}^{n-1} 1 &= \sum_{i=1}^{n-1}(n-1)\\ &= (n-1)^2 = O\left(n^2\right)\end{aligned}$$

4 MergeSort

The basic idea is simple

- If given sequence of no more than one element then we're done
- Otherwise (Length> 1) split sequence into two shorter sequences of equal lenth, sort them recursively and merge the two resulting sequences

Assumption:

- Length of top level input sequence is a power of two
- This allows for nice splitting into equally sized sub problems as they can all reduce to 2×1

```
list MergeSort (list m)
if length(m) ⩽ 1 then
    return m
end if
int middle = length(m) / 2
list left, right, leftsorted, rightsorted

left = m[1..middle]
right = m[middle+1..length(m)]

leftsorted = MergeSort(left)
rightsorted = MergeSort(right)

return Merge(leftsorted, rightsorted)
```

This depth first approach is the easiest way to program it

- There can be issues here with recursion depth as all the data is stored in memory, so some computers/programming languages will be unable to do it
- Note that this keeps a copy of the original data

4.1 How to merge two sorted sequences

- Also simple
- Suppose two sorted sequences given as arguments, say left and right
- Start with initially empty result sequence
- If both left and right aren't empty look at the leftmost element from each, say l_1 and r_1
- If $l_1 < r_1$ then append l_1 to the result (and remove it from left) otherwise append r_1 to result (and remove it from right)
- If either left or right is empty, append the entire other one to the result
- Repeat until both empty

```
list MergeSort (list left, list right)

list result
while length(left)>0 or length(right)>0 do
    if length(left)>0 and length(right)>0 then
        if first(left) ⩽ first(right) then
            append first(left) to result
            left = rest(left)
        else
        #Keeping extra copies of the data in the result array
            append first(right) to result
            right = rest(right)
        end if
    else if length(left)>0 then
        append left to result
        left = empty list
    else    # Length(right) > 0
        append right to result
        right = empty list
    end if
end while
return result
```

- Best case is for merging two sublists length n is n, where the 1st list are all smaller than the 2nd list
- In worst case the lists will be empty upon reaching the last element of both
  - Alternating inputs in sublists (1,3,5,7) and (2,4,6,8) for example

Mergesort:

- is probably the simplest recursive sorting algorithm
- It's bad cases are a lot less bad than those of some other sorting algorithms
- It's good cases, however may be worse than some algorithms
- More technically, MergeSort always requires $n\log(n)$ steps to sort any n numbers
- Some of the above can get away with $\approx n$ in particularly nice, but **will** require $\approx n^2$ for others
- n^2 is a lot worse than $n\log(n)$

5 QuickSort

Note that this algorithm is all in place

This is an example of divide and conquer

- Does the hard work at the start, unlike merge sort
- Split input into two parts
- Recursively sort then, one after the other
- Concatenate the resulting, sorted subsequences

Method:

- At the beginning of each recursive call, QuickSort picks one element from the current sequence, the **pivot**
- The partitioning will be done wrt to the pivot

  - Each element smaller than the pivot goes into the left part
  - Each element bigger than the pivot goes into the right part
  - parts may have very different sizes

- In some sense,
  - MergeSort does the complicated part after the sorted subsequences come back from recursive calls
  - QuickSort does its difficult bit prior to recursing. This means that simple concatenation afterwards is OK

The basic flow is as follows:

```
QuickSort (int A[1...n], int left, int right)

 1: if (left < right) then
 2:     // rearrange/partition in place
 3:     // return value "pivot" is index of pivot element
 4:     // in A[] after partitioning
 5:     pivot = Partition (A, left, right)

 6:     // Now:
 7:     // everything in A[left...pivot-1] is smaller than pivot
 8:     // everything in A[pivot+1...right] is bigger than pivot
 9:     QuickSort (A, left, pivot-1)
10:     QuickSort (A, pivot+1, right)
11: end if
```

This starts with left=1, right=n

5.1 The partitioning function

If the partition selected is the largest value, this is inefficient as one of the created sublists is empty

The simplest method is to pick a fixed position in the current sequence, for example the right most, however this is not most efficient

Partition moves everything smaller than the pivot, and everything bigger to the right. It does not sort

This is how the partition procedure can be implemented

```
int Partition (A[1...n], int left, int right)

 1: int x = A[right]
 2: int i = left-1

 3: for j=left to right-1 do
 4:     if A[j] < x then
 5:         i = i+1
 6:         swap A[i] and A[j]
 7:     end if
 8: end for

 9: swap A[i+1] and A[right]
10: return i+1    // pivot position after partitioning
```

This will swap values so that the values on the left are smaller than the pivot
It moves the pivot to just above the elements that have been swapped because they are less than 2

5.2 Worst case

Remember that if you are calling quicksort on two numbers, you will still have to make a quicksort on an empty space, due to the recursive nature of the algorithm. However, it will fail the if statement in the quick sort algorithm.

The worst case will depend on how you choose your pivot.

The worst case is where partition is called the maximum number of times. So the occasion where there are lists of length 0 and 1 should be delayed as much as possible.

On choosing the right most element as the pivot, an already sorted list will be highly inefficient. This is because a right sublist would never be made

The more evenly sized the sublists, the better the algorithm will perform

This has a lower recursion depth than merge sort and does not keep copies of data. Slow memory fast CPU - quicksort. Fast memory slow CPU - mergesort

5.3 Below length 4

For example in quicksort, when the lengths of the sublists get less than 4, use a different algorithm.