Coursework 2

Pierre Eugene Valassakis , pev115, 00644379
580- Algorithms

March 3, 2016

1 Question 1

(a) We have a sequence $A = [A_1, ...A_N]$ of integers.One possible version of the DISTINCT algorithm that has the required properties is:

Algorithm 1 DISTINCT(Input : sequence $A = [A_1, ...A_N]$)

```
1: procedure DISTINCT(A = [A_1, ...A_N]):
2: for each element A_i ∈ A do
3:    for each element A_j ∈ A where j > i  do
4:       if A_i == A_j  then
5:          return  false  and HALT
6:       end if
7:    end for
8: end for
9: return  true  and HALT
```

This algorithm does not allocate any memory besides the initial memory required to store the sequence, and hence has space complexity of O(1).
On the other hand, it goes through each one of the N elements of the sequence and for each element i it does a comparison with the remaining $N - i$ elements of the sequence.
As a result, we get that the algorithm completes in time $T(N)$ where:

$$\begin{aligned} T(N) &= N - 1 + N - 2 + ... + 2 + 1 \\ &= \frac{N^2 - N}{2} \end{aligned} \tag{1}$$

Hence we get that this procedure uses $O(N^2)$ time as required.
Variations on this algorithm could be:

- In line 2: to just require $i \neq j$ rather than $i < j$. This would still be $O(N^2)$ but slightly less efficient as it would run in $T(N) = N^2 - N$ time.

- We could make the algorithm slightly more efficient by having i span from 1 to N-1 in line 1: which would save us one iteration.

An alternative procedure which would satisfy the same requirements would consist of sorting the list in place (so we conserve the space complexity) and then checking if any adjacent elements are equal.

Algorithm 2 DISTINCT(Input : sequence $A = [A_1, ...A_N]$)

1: **procedure** DISTINCT($A = [A_1, ...A_N]$):
2: Call **insertion_sort**(A)
3: **for** i from 1 to $N - 1$ **do**
4: **if** $A_i == A_{i+1}$ **then**
5: **return false** and HALT
6: **end if**
7: **end for**
8: **return true** and HALT

Here **insertion_sort** is the standard insertion sort algorithm seen in lectures. As such, sorts the elements in A in $O(N^2)$ time and $O(1)$ space.
Following this, the algorithm just goes through the sequence once more, and hence the overall time complexity of this procedure is $O(N^2)$.

(b) In order to satisfy the new requirements, we now assume that we have an implementation of a hash table v, with a hash function $h(k)$ such as:

- The key k can be an integer.
- Uses chaining (a linked list) to handle collisions.
- Any two identical integer keys would map to the same position in the table.
- Uses simple uniform hashing.

We can then have the following algorithm:

Algorithm 3 DISTINCT(Input : sequence $A = [A_1, ...A_N]$)

```
1: Initialize a N-size hash table v to Null
2: for each element A_i ∈ A do
3:    Apply the hash function h_i = h(A_i) to the element A_i
4:    if v(h_i) == Null then
5:       v(h_i) == A_i                    \* store A_i in v(h_i) *\
6:    else
7:       for each element v_j in the chain at v(h_i) do
8:          if v_j == A_i then
9:             return false and HALT
10:         else
11:            Add A_i at the beginning of the list at v(h_i)
12:         end if
13:      end for
14:   end if
15: end for
16: return true and HALT
```

Naming conventions:

- A_i is the i^{th} element of A
- $h_i = h(A_i)$ is the hash of A_i
- $v(h_i)$ is the the entry of v with key h_i
- v_j is the j^{th} element of the linked list at the entry $v(h_i)$

This solution creates a hash table with N elements and therefore requires $O(N)$ space.

It also iterates through each element of A, and for each it makes a search through the hash table. The search time through a hash table is on average O(N/m) where m is the size of the table. Here m=N so the average search time in our algorithm is O(1). Additionally, adding an element in the hash table (as we do in lines 5 and 11) is also O(1).

So in every iteration through the elements of A we are only performing O(1) operations. Hence overall we get a time complexity of $O(N)$ which is what we want.

2 Question 2

In order to satisfy the requirements we will use dynamic programming. We follow a bottom up approach, where we iteratively consider more and more elements of our set while recording the length of longest sequence ending with each element. We also use this information in order to find the length of the biggest sequence possible after each iteration. As a result at the end we are able to return the length of the longest sequence within A.
We get the following algorithm:

Algorithm 4 LONGEST(Input : sequence $A = [A_1, ...A_N]$)

```
1: procedure LONGEST(A = [A_1, ...A_N]):
2: Initialize an array v = [v_1, ...v_N] of N elements where v[1] = 1
3: Initialize Max = v[1] = 1            \*Keep track of the longest length *\

4: for i from 2 to N do
5:    j=1
      \* Find the first value in A that is smaller than A_i *\
6:    while A_i < A_{i-j} and j < i  do
7:       increment j
8:    end while

      \*Update v with the length of the biggest sequence ending with A_i *\
9:    if j==i then
10:      v[i] = 1
11:   else
12:      v[i] = v[i - j] + 1
13:   end if

      \* Update the maximum length as appropriate *\
14:   if v[i] > Max then
15:      Max = v[i]
16:   end if
17: end for

18: return Max and HALT
```

Notes:
The algorithms logic is as follows.
In **lines 6-13** we are storing the longest path ending with element A_i in table v.
In **lines 5-8** we find the first element $A_k \in A$ such that $A_k < A_i$.
In **lines 9-3** we have the following logic: If $j == i$ that means that A_i is the smallest value so far so the length of the longest sequence ending with A_i is 1. Else we have that $j < i$ and $A_{i-j} < A_i$ hence the longest sequence finishing with A_i is the longest sequence ending with A_{i-j} plus A_i. From that follows that the length is $v[i] = v[i - j] + 1$.
In **lines 14-16** we are updating the maximum length found so far by comparing it to the newly calculated value.

In terms of time complexity:
Take the worst case scenario where all the values in A are strictly decreasing. In this case we get that for each element $A_i \in A$ we need to check all elements $A_k \in A$ where $k < i$.This is visible from lines 4-8 and will run in $T(N) = \frac{N^2-2N}{2}$, similarly to question 2. All other operations in lines 9-18 run in constant time and will only be executed once for every element in A. We can therefore conclude that this algorithm will run in $O(N^2)$ time as required.