FIT1045/FIT1053 Algorithmic Problem Solving – Tutorial 06.
Solutions

Objectives

After this tutorial, you should be able to:

- Describe computational complexity and Big-O.
- Identify the computational complexity of an algorithm.
- Describe the complexity of different algorithms from this unit.
- Introduce binary search.

Prepared Question

1. Using selection sort:
   $3, 1, 4, 5, 2$
   $1, 3, 4, 5, 2$
   $1, 2, 4, 5, 3$
   $1, 2, 3, 5, 4$
   $1, 2, 3, 4, 5$
2. Using insertion sort:
   $3, 2, 1, 5, 4$
   $2, 3, 1, 5, 4$
   $1, 2, 3, 5, 4$
   $1, 2, 3, 5, 4$
   $1, 2, 3, 4, 5$
3. The elements have to be in a non-decreasing order in order for the algorithm to be $O(n)$ because insertion sort checks each element and sees makes a swap if the element to it's left is larger itself. If the elements are in non-decreasing order the algorithm will only make one pass through without any swaps.
4. This is because selection sort will always make a pass through the entire list looking for the smallest index and inserting it at the first element of the list. Then it will look for the next smallest index by passing through the entire list again minus 1 and enter it at the second element of the list. It continues this process for every single element.

Warm-up

Do not solve these questions by running in Python.

Polynomial Bounds: For each of the following concrete time complexity give the tightest bound in terms of a simple polynomial function n^c using big-Oh notation. That is, determine the smallest c such that $T(n) \in O(n^c)$.

- $T(n) = n^2 - 10n - 1$
- $T(n) = n \ mod \ 100$

Complexity and Big-O: Count the number of steps taken by the following programs and then determine it's upper bound complexity in Big-O.

- ```
  def program1(n):
      count = 0
      i=0
      while i<n:
          count += i
          i+=1
      return count
  ```

- ```
  def program2(n):
      count =0
      i=0
      while i<n:
          j=i+1
          while(j<i+3):
              count+=1
              j+=1
          i+=1
      return count
  ```

Python Complexity, Best and Worst Case: Below are two separate algorithms that searches for an element in a sequence. For each algorithm determine the best and worst case time complexities for a given non-empty list `lst` and any integer `n`. Explain how you came to your conclusion.

- ```
  def find(x, lst):
      n = len(lst)
      for i in range(n):
          if lst[i] == x:
              return i
  ```

- ```
  def find(x, lst):
      n = len(lst)
      for i in list(range(n)):
          if lst[i] == x:
              return i
  ```

Euclid's Algorithm: Earlier in the semester we learnt Euclid's algorithm. The algorithm is an example of reduce and conquer. Given the following values of `m` and `n`, iterate through the algorithm determining each new value of `m` and `n` until the GCD is computed.

- gcd(147, 12)

- gcd(13, 21)

```
def gcd(m, n):
    """
    Input: integers m and n such that not m==n==0
    Output: the greatest common divisor of m and n
    """
    while n != 0:
        r = m % n
        m = n
        n = r
    return m
```

Complexity and Big-O

- Program 1 takes $0 + (n + 1) + 2n + 0 = 3n + 1$. Therefore, $O(n)$. (Warm-up Question Solution)
- Program 2 takes $0 + n * (2 + 2 * 3 + 2) + 1 = 10n + 1$. Therefore, $O(n)$. (Warm-up Question Solution)
- Program 3 takes $0 + 0 + (n + 1) + 0 + (n * (n + 1)) + 2n^2 + n + 0 = 3n^2 + 3n + 1$. Therefore, $O(n^2)$.
- Program 4 takes $0 + (\lfloor \log_2 n \rfloor + 2) + 2(\lfloor \log_2 n \rfloor + 1) + 1 = 3\lfloor \log_2 n \rfloor + 5$. Therefore, $O(log(n))$.
- Program 5 takes:

$$\begin{aligned} C &= 2 + (n + 1) + n + (\frac{n}{2})(n + 3) + n(n + 1) + n + 1 \\ &= 2 + n + 1 + n + \frac{n^2}{2} + \frac{3n}{2} + n^2 + n + n + 1 \\ &= \frac{3}{2}n^2 + \frac{11}{2}n + 4 \end{aligned}$$

  Therefore, $O(n^2)$.

We can calculate the complexity for Program 5 by recognising that `j<n` will take:

- $n + 1$ checks when `i=0`
- n checks when `i=1`
- $n - 1$ checks when `i=2`
- $n - 2$ checks when `i=3`
- ...
- 3 checks when `i=n-2`
- 2 checks when `i=n-1`

We know that `i` never will equal `n` because the first while loop condition is `while i<n`. We also know that `j` will be set to `i` exactly `n` times. We can recognise that we can 'pair' the checks such that every pair equals $n + 3$ - you can see this is the case by adding the first group of checks to the last group of checks ($(n+1)+(2) = n+3$), and the second group of checks to the second last group of checks ($(n) + (3) = n + 3$). We know that there are half as many pairs of groups of checks as there are groups of checks. Because we know there are n groups of checks (we know this because `j` will be set to `i` exactly `n` times), then there must be $n/2$ pairs of checks. Thus, the check in the line `while j<n:` must occur $(n/2)(n + 3)$ times, and counts for $(n/2)(n + 3)$ elementary steps.

Similar reasoning applies to the code inside the second while loop. We know that we will enter the second while loop:

- n times when `i=0`
- $n - 1$ times when `i=1`
- $n - 2$ times when `i=2`
- ...
- 2 times when `i=n-2`
- 1 time when `i=n-1`

We can recognise that we can 'pair' the while loop entry such that every pair equals $n + 1$. We know that there are n groups of entries, and thus $n/2$ pairs of groups. Thus we know that there will be $(n/2)(n + 1)$ entries into the while loop. We know there are 2 elementary steps inside the while loop, so every entry has 2 elementary steps associated with it. Thus, the inner part of the second while loop has $2(n/2)(n + 1) = n(n + 1)$ elementary steps.

Note that if we plan on stating the complexity of Program 5 in Big-O notation, we only need to recognise that there are as many as n^2 elementary steps, making the rest of our calculations redundant.

Python Complexity

In class:

- `x in a_string` has a best case of $O(1)$, which occurs when `x` is the first element in `a_string`, and a worst case of $O(n)$ (where n is the length of `a_string`), which occurs when `x` is not in `a_string`. The algorithm must check every element in `a_string`, hence it being linear time.
- `a_string == b_string` has a best case of $O(1)$, which occurs when the first elements of `a_string` and `b_string` do not match, and a worst case of $O(n)$ (where n is the length of either string), which occurs when the two strings either match completely, or match up to the final checked element. The algorithm must check every element in both strings, hence it being linear time.
- `lst[i]` has a best case and a worst case of $O(1)$. Fetching an element from a given index occurs in constant time, regardless of the length of the list or the indexed being accessed. (You will learn why this is the case in FIT1008.)
- `lst[i:j]` has a best and a worst case of $O(j-i)$ (where i is the starting index of the slice, and j is the ending index of the slice). Slicing in Python creates a copy of the elements in the original list. The time taken is proportional to the number of elements that need to be copied, hence it being linear time.
- `lst.pop()` has a best and a worst case of $O(1)$. The original list must update the reference at one index (the final index), and the method returns that reference. Both of these actions have constant time, hence it being constant time.
- `lst.append(x)` has a best and a worst case of $O(1)$. The original list must update the reference at one index (the final index) which takes constant time, hence it being constant time.
- `lst.pop(0)` has a best and a worst case of $O(n)$ (where n is the length of `lst`). The original list must return the reference of the item in the first index, and then update the references of every index after the first index by moving each item up by one. The algorithm must change the reference of every element in `lst`, hence it being linear time.

Practice:

- `all(lst)`: Best case of $O(1)$, worst case of $O(n)$.
- `max(lst)`: Best and worst case of $O(n)$.
- `reversed(lst)`: Best and worst case of $O(n)$.
- `set(lst)`: Assuming `set` is implemented naively, best case of $O(n)$ (original list contains repeats of the same element), worst case of $O(n^2)$ (original list only contains unique elements). However, Python's `set` likely uses the data structure *hash tables*, which you will learn about in FIT1008. This means the average complexity of `set` is likely much better than $O(n^2)$. (Not examinable.)
- `lst.remove(x)`: Best and worst case of $O(n)$ (either item is in first index, thus whole list must be updated; or item not in list, thus whole list must be checked).
- `lst.insert(i,x)`: Best case of $O(1)$, worst case of $O(n)$.

Decrease and Conquer

A decrease-and-conquer algorithm is an algorithm that exploits a relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem.
Some examples of problems faced in real life where a decrease and conquer algorithm is used include:

- finding Terry Pratchett's *Monstrous Regiment* at the library - once you were in fiction you would find the P's, then you would find the Pr's, then the Pratchett's, then the correct book.

Binary search and insertion sort are both decrease and conquer algorithms. Binary search decreases the problem by half until it gets a solution, and insertion sort decreases the problem by one until it gets a solution.

Binary Search

The maximum number of guesses is 7 to guess a number in between 1 to 100, by using the binary search.

This can be done by choosing a number in the middle of the list, that is $\lfloor \frac{1+100}{2} \rfloor = 50$. Now, the list is cut into two parts, where the first half consists of numbers from 1 to 49, and the second half consists of numbers from 51 too 100. If 50 is the desired number, then we are done. If it is too small (resp. too large), pick the second half (resp. first half) of the list that consists of numbers from 51 to 100 (resp. 1 to 49).

Choose a number in the middle of the second half of the list, that is $\lfloor \frac{51+100}{2} \rfloor = 75$. Observe that the list is again cut into two parts, with numbers 51 to 74 in the first half and 76 to 100 in the second half. If 75 is the desired number, we are done. Else, select the relevant part (which depends if 75 is too small or too large) and continue the same process until the desired number is obtained.

An algorithm for the binary search is as follows:

```
def binary_search(lst, target):
    """
    INPUT: 'lst', a sorted list of integers from 0 to n
    OUTPUT: 'target', value to find inside 'lst' if no target is found
            then the function will return None.
    """
    lower = 0
    upper = len(lst)-1

    ret_val = None
    while lower <= upper:
        mid = lower + (upper - lower) // 2

        if lst[mid] == target:
            ret_val = mid
            break
        elif lst[mid] < target:
            lower = mid + 1
        else:
            upper = mid - 1
    return ret_val
```

Checkpoint

1. What is the tightest bound complexity for $T(n) = n/2$?
2. Given the insertion sort algorithm below. Determine the number of steps and state its time complexity (best and worst case).

```
def insert(i, lst):
    j = i
    while j > 0 and lst[j-1] > lst[j]:
        lst[j-1], lst[j] = lst[j], lst[j-1]
        j -= 1

def insertion_sort(lst):
    for i in range(1, len(lst)):
        insert(i, lst)
```

3. State the best and worse case time complexity for finding an element in a table (list of lists) `table` that has the outer and inner list with size `n`.
4. Explain how a binary search is an example of a reduce and conquer algorithm.

Solutions

1. $O(n)$
2.
```
def insert(i, lst):
    j = i #O(1)
    while j > 0 and lst[j-1] > lst[j]:  # Best case O(1), worst case O(n)
        lst[j-1], lst[j] = lst[j], lst[j-1] # Best case O(0), worst case O(n)
        j -= 1  # Best case O(0), worst case O(n)

def insertion_sort(lst):
    for i in range(1, len(lst)): # O(n)
        insert(i, lst)   # best case is O(n), worst case is O(n^2)
```

   Overall the algorithm runs at an $O(n)$ time complexity at its best case and $O(n^2)$ its worst case. For its best case the for loop will be $O(n)$ as it runs $'n'$ times exactly where as every initialization will be $O(1)$. The while loop check will also be considered as $O(1)$ due to the best cast complexity will be when the list is already sorted meaning that the element that is left of the current element is always going to be smaller than the current element so it will skip the body of the while loop. Hence making the next two lines afterwards $O(0)$.

   The worst case complexity is $O(n^2)$ due to having the list in reversed sorted order. Everything remains the same except the while loop check will be $O(n)$ since the current element is now forced to shuffle down to its correct position n times, while also being nested inside the original for loop that runs n times.

3. Best case would be $O(1)$ if the item in the list was the very first element of the outer list and the first element of the inner list. The worst case would be $O(n^2)$ with the case the item you want to find is not in the list because you would still have to traverse all the rows and all the columns via two nested for loops in order to reach the last element.
4. A binary search is a reduce and conquer algorithm because we are reducing our problem into a single problem that is smaller than the original. This is done by reducing the search space of the element we want to find by reducing the problem by $\log(n)$ each time.