Scientific Computation
Autumn 2024
Problem sheet 1 solution

---

1. Consider the Python function shown below:

```
def problem1(L):
    n = len(L)
    for i in range(n-1):
        for j in range(i+1,n):
            if L[i]==L[j]:
                return True
    return False
```

   (a) What is the problem that this code is attempting to solve?
   Solution: This code checks the input container `L` for duplicates. It returns `True` if there is at least one value which appears more than once in `L` and returns `False` otherwise.

   (b) Discuss the correctness of the code. Can it return `True` when it should return `False`? Is it possible for it to return `False` when it should return `True`?
   Solution: The code is correct. Say that there are no duplicates in `L`, and the code returned `True`. This would mean that `L[i]==L[j]` evaluated as `True` at some point. Since there are no duplicates, this could only happen if `i=j`, which can't happen since the inner loop ensures that $j > i$.

   Say that there is at least one duplicate where `L[a]=L[b]` and the code returns `False`. This means that there is at least one (a, b) with $a < n$, $b < n$, $a \neq b$ where i and j never take on the values (a, b) or (b, a). We are free to choose (a, b) such that $a < b$. The outer loop ensures that at some point $i = a$ ($a < n - 1$ since $a < b$). The inner loop ensures that if $i = a$, then at some point, $j = b$ and `False` cannot be returned.

   (c) Analyze the worst-case computational cost of the code. Use your analysis to find the asymptotic time complexity of the underlying algorithm (expressed using big-O notation).
   Solution: Prior to the loop there is one length lookup and one assignment (2 operations). There are two increments, two List lookups and one comparison each iteration (5 operations). In the worst-case (no duplicates), there are $n(n-1)/2$ iterations, so the total cost is, $C \approx 5n(n-1)/2 + 2$, and the algorithm is $\mathcal{O}(n^2)$ ($C \leq an^2$ for $n \geq 1$ and $a = 3$).

2. The ith iteration of insertion sort requires the determination of the correct location of the $i + 1$th element in a list. Assume that this location can be found arbitrarily

quickly. What then is the worst-case big-O cost of the insertion sort? (The time complexity of operations using lists and other Python containers can be found here: https://wiki.python.org/moin/TimeComplexity)
Solution: The algorithm remains $\mathcal{O}(n^2)$. In the worst case, during the ith iteration, i elements of the list have to be moved and this requires $\mathcal{O}(i)$ operations (get length-i slice and set length-i slice for implementation in the slides). So there is $\mathcal{O}(n^2)$ cost even if the cost of finding the location of elements is removed.

3. In lecture 3, we defined a family of hash functions for IP addresses: $H(a) = \sum_{j=1}^{4} w_j a_j$ rem p where p is a prime that has been specified, and each weight, w_j, is chosen uniformly at random from $\{0, 1, 2, ..., p-1\}$. Given two distinct IP addresses, how many hash functions in this family will assign the same index to these addresses? Say we have two hash functions, $H^{(1)}$ and $H^{(2)}$, and two IP addresses, x and y. We are interested in cases where $H^{(i)}(x) = H^{(i)}(y)$, but we do not require $H^{(1)}(x) = H^{(2)}(x)$.
Solution: We know from lecture that for a given set of values for $\{w_1, w_2, w_3\}$, there is one value of w_4 for which a hash collision occurs. There are p^3 distinct $\{w_1, w_2, w_3\}$ triplets, so the answer is p^3.