Assignment 37

Alex Clemmer
Student number: u0458675

Procedure

Foreward

One thing that I took away from the last timing assignment was a robust technique for accurate timing: you time incrementally as you complete the assignment, adding things that work and subtracting things that do not. I pretty much duplicated my methodology from that assignment here.

I started by just timing how long it took to allocate things to the array. Then I added code that spun the method for a few seconds before. Then I added code to time the loop itself, and subtract it from the total running time. At each point, I stopped and ran the program. If my code brought me closer to what I expected, I continued. If not, I fixed it.

How it works

First, there were a couple of things that I found had no discernible effect on timing. First, it did not obviously matter if the string was completely unique or just randomly generated, or at least, not asymptotically. Because of this, I generated my strings randomly instead.

Whether we spun the allocation for a couple of seconds before actually running the experiment made our results much less erratic. I used python, which has only dynamically-allocated arrays (called lists); in this case, running the method in question repeatedly caused the array to expand, so it was necessary to allocate a new `list` object just before the actual experiment.

Another thing that was important here was to call the garbage collector. Oddly even though *not* calling the garbage collector did not actually cause a memory overflow error, it again mad the results much less erratic. I think this is because the GC tends to operate completely parallel until the compaction step, when it defrags and rearranges all the memory, since this must clearly halt all programs to complete.

The last, and most important thing, was size. Larger sizes were universally erratic and unpredictable. Smaller sizes were the only manageable data.

Also important

To actually time the operations, we stuck the actual allocation operation inside a loop. All the things listed before were important, but this was the actual testing mechanism. In order to do this, we just bookended the timing operations around the loop, and the timed the dry loop and subtracted that from the time.

Results

The case of the first method (`p1`) was pretty clear-cut: it doubles every time.

```
Time (ns): 25 101 204 432 823 1632 3211
Characters: 1   2   3   4   5    6    7
```

As you can see, this quickly becomes prohibitive to compute, especially if you're running the experiment hundreds or thousands of times (like I was). The case of the second method (`p1`)

```
Time (ns): 20 45 65 92 104 131 156
Characters: 1  2  3  4   5   6   7

Time (ns): 20 45 49 83 74 120 137
Characters: 1  2  3  4   5   6   7
```

Analysis

Strangely, this is what we would expect: memoization is a big hit on memory, but it clearly gives us better results overall. Both the second and third algorithms are a clean $O(n^2)$, where the first is certainly exponential.

If you're curious, the code is as follows:

```
def p1(i,j):
  if i == j+1:
    return 0
  elif i == j:
    return 1
  elif A[i] == A[j] and i < j:
    return 2 + p1(i+1,j-1)
  else:
    return max(p1(i+1, j), p1(i, j-1))

def p2(i,j):
  if i == j+1:
    T[(i,j)] = 0
    return 0
  elif i == j:
    T[(i,j)] = 1
    return 1
  elif A[i] == A[j] and i < j:
    ret_val = T[(i,j)]
    if ret_val != -1:
      return ret_val
    else:
      T[(i,j)] = ret_val = 2 + p2(i+1,j-1)
      return ret_val
  else:
    ret_val = T[(i,j)]
    if ret_val != -1:
      return ret_val
    else:
      T[(i,j)] = ret_val = max(p2(i+1, j), p2(i, j-1))
      return ret_val

def p3(i,j):
  if i == j+1:
    return 0
  elif i == j:
    M[i][j] = 1
    return 1
  elif A[i] == A[j] and i < j:
    M[i][j] = 2 + p3(i+1,j-1)
    return M[i][j]
  else:
    M[i][j] = max(p3(i+1, j), p3(i, j-1))
    return M[i][j]
```

Timing was pretty straightfoward