Scientific Computation
Autumn 2024
Review exercises (solutions)

---

1. Consider the application of insertion sort to the list, `L = [15,10,16,9]`. How many comparisons will the algorithm require? List the pairs of integers that will be compared in the order that the comparisons take place.
   **Solution:** First iteration: compare 15 and 10 (and swap), second iteration: compare 16 and 15 (do nothing), third iteration: compare 9 with 16, 9 with 15, 9 with 10 (and place 9 at the front shifting the other entries). There are 5 total comparisons.

2. Consider the following list ordered as a binary heap: `L = [0, 4, 3, 14, 10, 11, 9, 17, 16, 16, 13]`. How many comparisons will be required if 3 is inserted into the heap? List the pairs of integer that will be compared in the order that the comparisons take place.
   **Solution:** There are two comparisons. First 3 is compared with 11 and the two are swapped. Then, 3 is compared with 3, nothing happens, and the process terminates.

3. You are given an *adjacency list*, `L`, for a directed graph with nodes numbered from 1 to $N$. Here, `L[i]` is a list, and if there is a link from node $i$ to node $j$, then $j$ will be in `L[i]`. Provide an efficient algorithm for computing the adjacency list for the *reverse* of the directed graph where any link from $i$ to $j$ is replaced with a link from $j$ to $i$. Briefly explain what the big-O cost of your algorithm is.

   **Solution:** First create an empty adjacency list (i.e. a list containing $N$ empty lists). Then iterate through the given adjacency list and add the "reversed edges" as in the code below:

   ```
   for i,l in enumerate(L):
       for j in l:
           L2[j].append(i)
   ```

   This code 'looks' at each link once, and an append to the end of a list has $\mathcal{O}(1)$ cost as does direct access to a list element, so this code snipped has $\mathcal{O}(L)$ cost where $L$ is the total number of links in the graph. Creating the initial list has $O(N)$ cost, so we have a linear time algorithm with cost, $\mathcal{O}(N + L)$. Note: this code assumes nodes are numbered from 0 to $N - 1$ which is how the question should have been written. To adjust the solution to fit the question, use `L2[j-1].append(i+1)`.

4. Consider the application of the Rabin-Karp method to find 4-character patterns in a gene sequence, $S$, which only contain adenine, cytosine, or guanine. Let $S_i$ be the $i$th length-4 sequence in $S$ (containing `S[i:i+4]`). Provide a clear and concise explanation of how to compute the hash of $S_{i+1}$ given the hash of $S_i$. Hashes are

not computed modulo a prime.
**Solution:** Given the hash of $S_i$, $h_i$, we can use a rolling hash function to compute $h_{i+1}$, the next hash:

$$h_{i+1} = 3 * h_i - 3^4 * X_{i,1} + X_{i+1,4}.$$

Here, $X_{i,1}$ is the base-3 representation of `S[i]` ($A : 0$, $C : 1$, $G : 2$), and $X_{i+1,4}$ is the base-3 representation of `S[i+4]`.

5. Consider the following algorithm for finding the shortest path from node $s$ to node $x$ in a weighted directed graph with some negative edge weights: add a large constant to each edge weight so that all edge weights are positive and then apply Dijkstra's algorithm setting $s$ as the source node. Explain if this is a generally valid method. Either provide a sketch of a proof establishing its correctness, or provide a counterexample.
**Solution:** This method is not valid. Say that there are two paths from $s$ to $x$, one with two links (with total length $x$), and one with three (with total length $y$). If the large constant is $c$, then the new lengths will be $x + 2c$ and $y + 3c$. $x + 2c < y + 3c$ does not imply $x < y$ (e.g. take $x = 2$,$y = 1$,$c = 1000$), so shortest paths between a pair of nodes in the original and modified graphs can be different.

6. The code below was generated by claude sonnet 3.5. Provide a clear and concise explanation of the problem the code is solving and the strategy being used to solve the problem.

```
import numpy as np
def code6(A, y0, t0, tf, N):
    """
    Parameters:
    A (numpy.ndarray): 2 x 2 matrix
    y0 (numpy.ndarray): 2-element array
    t0 (float): Initial time
    tf (float): Final time
    N (int): Number of time steps

    """
    dt = (tf - t0) / N
    t = np.linspace(t0, tf, N+1)
    y = np.zeros((N+1, y0.shape[0]))
    y[0] = y0

    for i in range(1, N+1):
        y[i] = np.linalg.solve(np.eye(2) - dt * A, y[i-1])

    return y
```

**Solution:** This function is using the implicit Euler method to compute numerical solutions to a system of 2 linear ODEs: $d\mathbf{y}/dt = \mathbf{A}\mathbf{y}$ where $y \in \mathrm{R}^2$, $A \in \mathrm{R}^{2 \times 2}$, and the initial condition is given by `y0`. The solution advances $N$ time steps with step size = `dt`.

7. Consider the function below:

```
import numpy as np
import matplotlib.pyplot as plt
def rwalk_new(Nt=200,M=100,display=False):

    X = np.zeros((Nt+1,M))
    R = np.random.choice([-1,0,1],size=(Nt,M))

    for i in range(Nt):
        X[i+1,:] = X[i,:] + R[i,:]

    Xave = X.mean(axis=1)
    Xsd = X.std(axis=1)

    if display:
        plt.figure()
        plt.plot(Xave,'k-',label='ave, comp')
        plt.plot(Xsd,'k--',label='sd, comp')
        i = np.arange(Nt+1)
        plt.plot(i,0*i,'b:',label='expected val, theory')
        plt.plot(i,np.sqrt(i),'r:',label='sd, theory')
        plt.legend()
```

Describe the problem that this code is solving. Carefully explain what changes, if any, are needed to the 3rd and 4th calls to `plt.plot`

**Solution:** The code is simulating $M$ $Nt$-step "modified" random walks. The walks are modified as now there are three options that can be selected with equal probability: move +1, move -1, or don't move. The 3rd `plt.plot` does not need to be changed as there is no bias favoring increasing or decreasing $X$. To see this, start with the update equation,

$$X_{i+1} = X_i + \tilde{R}_i \tag{1}$$

where $\tilde{R}_i$ can be -1, 0, or 1 with equal probability. Taking the expectation, and using linearity of expectation gives, $\langle X_{i+1}\rangle = \langle X_i\rangle + \langle \tilde{R}_i\rangle$, and $\langle \tilde{R}_i\rangle = P(\tilde{R}_i = -1)*(1)+P(\tilde{R}_i = 0)*0+P(\tilde{R}_i = 1)*1 = -1/3+0+1/3 = 0$. Then, $\langle X_{i+1}\rangle = \langle X_i\rangle$. Since $X_0$ is always zero, $\langle X_0\rangle = 0$, and $\langle X_i\rangle = 0$ for all $i \geq 0$.

The 4th `plt.plot` does need to change. Squaring (1), taking the expectation, and applying linearity of expectation gives,

$$\langle X_{i+1}^2\rangle = \langle X_i^2\rangle + 2\langle X_i\tilde{R}_i\rangle + \langle \tilde{R}_i^2\rangle.$$

The 2nd term on the RHS is zero for the same reason as for ordinary random walks as explained in lecture 9. The 3rd term on the RHS is different:

$$\langle \tilde{R}_i^2\rangle = (1/3)*1^2 + (1/3)*0^2 + (1/3)*(-1)^2 = 2/3.$$

Consequently, $\langle X_{i+1}^2\rangle = \langle X_i^2\rangle + 2\langle X_i\tilde{R}_i\rangle + 2/3$. Since $X_0 = 0$, $\langle X_0^2\rangle = 0$, and $\langle X_i^2\rangle = 2/3i$. The theoretical standard deviation is $\sqrt{\langle X_i^2\rangle - \langle X_i\rangle^2} = \sqrt{2/3i}$. So, instead of `np.sqrt(i)`, the code should have `np.sqrt(2/3 i)`.

8. In this question, you will analyze *func1* in the code below.

```
def merge(L,R):
    """Merge 2 sorted lists provided as input
    into a single sorted list
    """
    M = [] #Merged list, initially empty
    indL,indR = 0,0 #start indices
    nL,nR = len(L),len(R)

    #Add one element to M per iteration until an entire sublist
    #has been added
    for i in range(nL+nR):
        if L[indL]<R[indR]:
            M.append(L[indL])
            indL = indL + 1
            if indL>=nL:
                M.extend(R[indR:])
                break
        else:
            M.append(R[indR])
            indR = indR + 1
            if indR>=nR:
                M.extend(L[indL:])
                break
    return M


def func1(L_all):
    """Function to be analyzed for question 8
    Input: L_all: A list of N length-M lists. Each element of
    L_all is a list of integers sorted in ascending order.
    Example input: L_all = [[1,3],[2,4],[6,7]]
    """
    if len(L_all)==1:
        return L_all[0]
    else:
        L_all[-1] = merge(L_all[-2],L_all.pop())
        return func1A(L_all)
```

(a) Provide a brief, clear description of the functionality and correctness of `func1` (analysis of `merge` is not required, you may state results from lecture or elsewhere). Include a clear explanation of the function's output and the strategy the function uses to produce the output.

**Solution:** `func1` relies on `merge` which takes two sorted lists of integers as input and returns a single sorted list containing all of the integers in the input lists. The function `func1` receives input (`L_all`) in the form of a list of N length-M sorted sublists. The output is a list containing 1 sorted list which contains all of the integers provided as input. The function is recursive, how-

ever it is more simply viewed as iterative: While `L_all` contains more than one sublist, replace the final two sublists with a single sublist that is the result of merging (and sorting) the final two sublists. Correctness follows from the following observations: 1) merge is correct, 2) all input sublists are sorted, 3) any sublist with length less than $NM$ will be merged with another sublist with length less than $NM$ before termination, 4) any sublist at any time will be sorted, 5) no new integers are introduced and no input integers are removed, 6) after a finite number of iterations, there will be a single length-$NM$ (sorted) sublist, and the function will then terminate.

(b) Analyze the time complexity of `func1` and explain how the cost depends on the input. As part of your analysis, include an estimate of the big-O cost of the function (again, analysis of `merge` is not required, and you may simply state results from lecture).
**Solution:** The worst-case cost of merging lists with lengths $M_1$ and $M_2$ is $C_1(M_1 + M_2) + C_2$. I estimate $C_2 \approx 5$ and $C_1 \approx 8$ (the precise values are unimportant). `func1` requires N-1 merges as described above with $M_1 + M_2 = 2M, 3M, ..., NM$, and the total cost for the merges can be written as $C_1M[N(N+1)/2 - 1] + C_2(N-1)$, which leads to a running time of $O(MN^2)$. The per-iteration cost of modifying `L_all` and checking its length are $O(1)$ and do not effect this estimate.