Exercises

1. .

$D^{(0)}$	1	2	3
1	0	2	1
2	-1	0	∞
3	∞	3	0

$D^{(1)}$	1	2	3
1	0	2	1
2	-1	0	0
3	∞	3	0

$D^{(2)}$	1	2	3
1	0	2	1
2	-1	0	0
3	2	3	0

$D^{(3)}$	1	2	3
1	0	2	1
2	-1	0	0
3	2	3	0

2. (a) $D[k]$ is the LIS ending on k, so if $A[k] < A[i]$, $D[i]$ is at least $D[k] + 1$. If $A[k] < A[i]$ is false for all $k = 0, ..., i - 1$, then $A[i]$ can only be an LIS of length 1.

(b)
```
def LIS(A):
  #A[0]~A[n-1] is the input sequence
  D = [1]
  for i in range(1, len(A)):
    d = 1
    for k in range(0, i):
      if A[k] < A[i] and D[k]+1 > d:
        d = D[k]+1
    D.append(d)
  # find max in D
  m = 0
  for i in range(len(A)):
    if D[i] > m:
      m = D[i]
  return m
```

(c)
```
def LIS_(A):
  #A[0]~A[n-1] is the input sequence
  D = [1]
  prev = [-1]
  for i in range(1, len(A)):
    d = 1
    kbest = -1
```

```
    for k in range(0, i):
      if A[k] < A[i] and D[k]+1 > d:
        d = D[k]+1
        kbest = k
    D.append(d)
    prev.append(kbest)
  # find max in D
  m = 0
  ibest = -1
  for i in range(len(A)):
    if D[i] > m:
      m = D[i]
      ibest = i
  # trace LIS
  out = []
  while ibest != -1:
    out.append(A[ibest])
    ibest = prev[ibest]
  return m, out[::-1]
```

An O(n log n) algorithm by Fredman 1975:

Let S[k] be the smallest element of A that is at the end of an increasing sequence of length k. Let X[k] be the index of S[k] in A. Let p[i] be the parent of i.

```
import bisect
def fredman(A):
  S = []
  X = []
  p = [-1]*len(A)
  for i in range(len(A)):
    x = bisect.bisect_left(S, A[i])  # insertion point
    if x == len(S):
      S.append(A[i])
      X.append(i)
      if len(S)>1:
        p[i] = X[-2]
    else:
      if A[i] < S[x]:
        S[x] = A[i]
        X[x] = i
        if x > 0:
          p[i] = X[x-1]
  # reconstruct
  curr = X[-1]
  LIS = [A[curr]]
  while p[curr] != -1:
    curr = p[curr]
    LIS.append(A[curr])
  return len(S), LIS[::-1]
```

Problems

---

1. (a) We abbreviate minimumElements(n, S) as $f(n)$.
   Base case: (1) $f(0) = 0$. (2) For $0 < n < \min(S)$ or $n < 0$, the algorithm correctly returns None.
   Hypothesis: $f(n)$ is correct for $0 \leq n < k$.
   Inductive step: For $n = k$, $f(k)$ can be either a real number or None. If the former is true, we must have $f(k) = f(k-s) + 1$ for some $s \in S$, because $s > 0$. The algorithm is correct by picking the minimum among all such $f(k-s)+1$. If the latter is true, none of $f(k-s)$ can yield a real number. The algorithm correctly returns None for $f(k)$. This completes the induction.

   (b) Similar to the naive way to calculate Fibonacci numbers, we have $T(n) = T(n-1)+T(n-2)+O(1)$. (In fact, the assertion $2^{\Omega(n)}$ is wrong. The base does affect asymptotic behavior. $T(n)$ grows at least as fast as Fibonacci numbers, so the base is at least the golden ratio.)

   (c) Similar to the given code, with memoization added.
   Running time: If the memo is formed already, we only have $O(|S|)$ calls on the top level. Consider the formation of the memo. Each memo[k] is run at most once. Conceptually aggregate the calculation of the memo from the base cases, and work the way up. Each memo[k] requires at most $O(|S|)$ calls, without further recursion. So to form the memo takes $O(n|S|)$. So overall $O(n|S|)$.

```
def memoization(n, S):
  # initialize memo
  memo = [-1]*(n+1)
  memo[0] = 0
  for i in range(1, n+1):
    if i < min(S):
      memo[i] = None
    else:
      break
  return helper(n, S, memo)

def helper(k, S, memo):
  if memo[k] != -1:
    return memo[k]
  candidates = []
  for s in S:
    if k-s >= 0:
      cand = helper(k-s, S, memo)
    if cand is not None:
      candidates.append(cand+1)
  memo[k] = min(candidates)
  return memo[k]
```

   (d) Crawl the 1D solution array according to the optimal substructure: $sol(n) = \min_{s \in S} sol(n-s)+1$. The array length is $\sim n$. The work for each element is $\sim |S|$. So overall $O(n|S|)$.

```
def dp(n, S):
  # initialize array
  sol = [None]*(n+1)
  sol[0] = 0
  # crawl
```

```
    for k in range(min(S), n+1):
      candidates = []
      for s in S:
        if k-s >= 0 and sol[k-s] is not None:
          candidates.append(sol[k-s]+1)
      sol[k] = min(candidates)
    return sol[n]
```

2. Consider $X[k]$ and whether site k is assigned. The optimal substructure and the base cases are:

$$
\begin{aligned}
X[k] &= \max(Q[k] + X[k-2], X[k-1]) \\
X[0] &= \max(Q[0], 0) \\
X[1] &= \max(Q[1], X[0])
\end{aligned}
$$

The running time and space are $O(n)$, because each step is constant time and there are n steps and 2n storage.

```
def river(Q):
  # initialization
  X = [None]*len(Q) # answer array
  p = [0]   *len(Q) # "picked". for backtrack
  if Q[0] > 0:
    X[0] = Q[0]
    p[0] = 1
  else:
    X[0] = 0
  if Q[1] > X[0]:
    X[1] = Q[1]
    p[1] = 1
  else:
    X[1] = X[0]
  # crawl
  for k in range(2, len(Q)):
    if Q[k]+X[k-2] > X[k-1]:
      X[k] = Q[k]+X[k-2]
      p[k] = 1
    else:
      X[k] = X[k-1]
  # backtrack
  ans = []
  k = len(Q)-1
  while k >= 0:
    if p[k] == 1:
      ans.append(k)
      k -= 2
    else:
      k -= 1
  return X[-1], ans[::-1]
```

3. (a) Let $A[k]$ be the optimal subarray that ends at k. We have $A[k] = \max(B[k], B[k] + A[k-1])$. We iterate k to find all A[k] and record the best one, so $O(n)$.

```
def linear(B):
  Aprev = B[0]
  Amax = Aprev
  kAmax = 0
  s = [1]*len(B) # single-element subarray?
  for k in range(1, len(B)):
    Acurr = B[k]
    if Aprev > 0:
      Acurr += Aprev
      s[k] = 0
    if Amax < Acurr:
      Amax = Acurr
      kAmax = k
    Aprev = Acurr
  kmin = kAmax
  while True:
    if s[kmin] == 1:
      return Amax, kmin, kAmax
      # covers all situations. s[0] is always 1
    kmin -= 1
```

(b) We will use $D_{x,y,i,j}$ as a short for $D[x][y][i][j]$ hereafter.

To reach $O(n^4)$, we can only spend constant time on each $D[x][y][i][j]$. Suppose we iterate through D in the following way:

```
for x in range(n):
  for y in range(n):
    for i in range(x, n):
      for j in range(y, n):
```

which means when we deal with $D[x][y][i][j]$, we have access to $D[x][y][s][t]$ where $s = x, ..., i-1$ and $t = y, ..., j-1$. We therefore write

$$D_{x,y,i,j} = D_{xyi,j-1} + D_{x,y,i-1,j} - D_{x,y,i-1,j-1} + A_{i,j}$$
$$D_{x,y,x,j} = D_{x,y,x,j-1} + A_{x,j}$$
$$D_{x,y,i,y} = D_{x,y,i-1,y} + A_{i,y}$$

to reach $O(n^4)$ time and space as follows:

```
def getD(A, n):
  # initialize 4d
  D=[[[[None for i in range(n)]\
       for j in range(n)]\
      for k in range(n)]\
     for l in range(n)]
  # crawl
  for x in range(n):
    for y in range(n):
      D[x][y][x][y] = A[x][y]
      for j in range(y+1, n):
        D[x][y][x][j] = D[x][y][x][j-1] + A[x][j]
      for i in range(x+1, n):
        D[x][y][i][y] = D[x][y][i-1][y] + A[i][y]
```

```
      for i in range(x+1, n):
        for j in range(y+1, n):
          D[x][y][i][j] = D[x][y][i][j-1] + D[x][y][i-1][j] - D[x][y][i-1][j-1]\
                          + A[i][j]
  return D
```

(c) We do a 4-D search for the max entry of D.

```
def maxD(D, n):
  DD = D[0][0][0][0]
  xx, yy, ii, jj = 0, 0, 0, 0
  for x in range(n):
    for y in range(n):
      for i in range(x, n):
        for j in range(y, n):
          if D[x][y][i][j] > DD:
            DD = D[x][y][i][j]
            xx = x
            yy = y
            ii = i
            jj = j
  return xx, yy, ii, jj
```

(d) Similar to (b), we define

$$E[x][y][i] = \sum_{s=x}^{i} A[s][y]$$

To find E takes $O(n^3)$ time, using $E_{x,i,y} = E_{x,i-1,y} + A_{i,y}$ for each y. Then for each pair (x, i), we apply the algorithm in (a) to E[x][i], which takes $O(n^2 \cdot n)$ time. We record the best answer seen among for all (x, i). Overall $O(n^3)$.

```
def getE(A, n):
  # initialize 3d
  E=[[[None for i in range(n)]\
      for j in range(n)]\
     for k in range(n)]
  # crawl E
  for y in range(n):
    for x in range(n):
      E[x][x][y] = A[x][y]
      for i in range(x+1, n):
        E[x][i][y] = E[x][i-1][y] + A[i][y]
  return E

def n3(E, n):
  # returns best of s, x, a, i, b
  # s: quality seen among all (x, i)
  # a, b: as in (a)
  bestA, bestx, besti, besta, bestb = [None]*5
  for x in range(n):
    for i in range(x, n):
      A, a, b = linear(E[x][i][:])
```

```
    # note if using E[x][y][i] ordering, E[x][:][i] is wrong
    if bestA is None or A > bestA:
      bestA = A
      besta = a
      bestb = b
      bestx = x
      besti = i
return bestA, bestx, besta, besti, bestb
```