1. (a) $a = 4$, $b = 2$, $c = \log b(a) = 2$, $f(n) = n^2 \log(n)$. It falls into case 2.
   $T(n) = \Theta(n^2) * (\log(n)^2)$

   (b) $a = 8$, $b = 6$, $c = \log b(a) = \log 6(8)$, $f(n) = n\log n$. It falls into case 1, $T(n) = \Theta(n^c) = \Theta(n^{(\log 6(8))})$

   (c) $a = \mathrm{sqrt}(6006)$, $b = 2$, $c = \log b(a) = \log 2(\mathrm{sqrt}(6006))$, $f(n) = n^{\mathrm{sqrt}(6006)}$. It falls into case 3. $T(n) = \Theta(n^{\mathrm{sqrt}(6006)})$

   $\log 2\mathrm{sqrt}(6006) + e = \mathrm{sqrt}(6006)$
   $E = \mathrm{sqrt}(6006) - \log 2(\mathrm{sqrt}(6006))$

   (d) $a = 10$, $b = 2$, $c = \log 2(10)$ $f(n) = 2n$, $f(n) = O(n^{(\log 2(10) - e)})$. It falls into case 1.
   $T(n) = \Theta(n^{(\log 10)})$

   (e) set $S(m) = T(2^n)$. $S(m) = T(2^n) = 2T(2^{(n/2)}) + n = 2S(m/2) + n$
   $a = 2$, $b = 2$, $c= 1$, $f(m) = n$, it falls into case 2. $T(m) = m\log m$, $T(n) = f(\log m) = \log(n) \log\log(n)$

- $a =1$, $b = 2$, $c = \log 2(1) = 0$, $f(n) = 10 - n$, $T(n) = \Theta(\log n)$
- $a = 2^n$, $b = 2$, $c = n$, $f(n) = n$ $T(n) = \Theta(n^n)$
- $A = 2$, $b = 4$, $c = 0.5$, $f(n) = n^{(0.51)}$, $T(n) = \Theta(n^{(0.51)})$
- $A = 0.5$, $b = 2$, $c = \log 2(0.5) = - 1$, $f(n) = 1/n$, $T(n) = \Theta(n^{(-1)} \log n)$
- $A = 16$, $b = 4$, $c = 2$, $f(n) = n!$, $T(n) = \Theta(n!)$

2. (a) prove by contradiction: assume such local minimum for A does not exist. Then array A has to be strictly increasing or decreasing, but according to the problem, A1 >=A2 and An >= An-1, it cannot be strictly increasing or decreasing. Therefore, it has to have a local minimum.
   (b)
    1. if n == 3, we only need to return A2
    2. Else if n > 3, we take the number at i = n/2,
    3. If A[i-1] >= A[i] <= A[i+1] we have found local minimum,
    4. If A[i] > A[i-1], we continue searching for n = A[0:i], else, we search for n = A[i:A.length]

   Proof: if A[i] > A[i-1] and also according to the problem A[1] >= A[2], by induction we can be sure that there must be a local minimum between 0 and i. Similarly, if not A[i] > A[i-1], we can induce that there must be a local minimum between i and n because of A[n] >= A[n-1]

Recurrence equation: T(n) = T(n/2) + T(1)

a = 1, b = 2, c = 0 T(n) = Θ(logn)

3. Subproblem: when at ith city, we need to decide in terms of the following 4 cases:
   Case1: Let Marco visit ith city
   Case 2: Let Polo visit ith city
   Case 3: Let Marco skip the last city in his current list and visit ith city instead, and let Polo visit the city that Marco skips.
   Case 4: Let Polo skip the last city in his current list and visit ith city instead, and let Marco visit the city that Polo skips.
   Choose the case that minimizes the total travel time at ith city.

   Recurrence relation: total_travel_time = min(Marco_travel_ith_city_time + Polo_travel_time, Marco_travel_time + polo_travel_ith_city, Marco_travel_ith_city_instead_his_last_city_time + Polo_travel_Marco_last_city_time, Marco_travel_Polo_city_time + Polo_travel_ith_city_instead_his_last_city_time)
   Conner Case: if n ==2, then Marco visits city 0, and Polo visits city1

```
def min_travel_time (cities, travel_time):
    marco = [0] # give marco 0 city as start
    polo = [1] # give polo 1 city as start
    marco_travel_time = 0
    polo_travel_time = 0
    for i in range(2, len(cities)):
     # 1. marco travels to ith city
     if len(marco) == 0:
        marco_travel_plus_ith_city  = 0
     else:
        marco_travel_plus_ith_city = marco_travel_time + travel_time[marco[-1]][i]
     total_travel_time1 = marco_travel_plus_ith_city + polo_travel_time

     # 2. polo travels to ith city
     if len(polo) == 0:
        polo_travel_plus_ith_city = 0
     else:
        polo_travel_plus_ith_city = polo_travel_time + travel_time[polo[-1]][i]
     total_travel_time2 = polo_travel_plus_ith_city + marco_travel_time

     # 3. marco travels to ith city instead of marco[-1] city
     if len(marco) == 0:
        total_travel_time3 = math.inf
        # if marco only has one city to visit, this cannot happen, set the total travel time to +inf
```

```
    elif len(marco) == 1:
        last_city1 = marco[0]
        insert1 = len(polo)
        marco_travel_ith_city_instead = 0
        for j in range(len(polo)):
            if last_city1 < polo[j]:
                insert1 = j
                break
        if insert1 == 0:
            polo_travel_marco_city = travel_time[last_city1][polo[insert1]] + polo_travel_time
            #total_travel_time3 = polo_travel_marco_city + marco_travel_ith_city_instead
        elif insert1 == len(polo):
            polo_travel_marco_city = polo_travel_time + travel_time[polo[-1]][last_city1]
            #total_travel_time3 = polo_travel_marco_city + marco_travel_ith_city_instead
        else:
            polo_travel_marco_city = polo_travel_time - travel_time[polo[insert1 - 1]][polo[insert1]]
+ travel_time[polo[insert1-1]][last_city1]+ travel_time[last_city1][polo[insert1]]
            #total_travel_time3 = polo_travel_marco_city + marco_travel_ith_city_instead
    else:
        last_city1 = marco[-1]
        marco_travel_ith_city_instead = marco_travel_time - travel_time[marco[-2]][marco[-1]] +
travel_time[marco[-2]][i]
        insert1 = len(polo)
        for j in range(len(polo)):
            if last_city1 < polo[j]:
                insert1 = j
                break
        if insert1 == 0:
            polo_travel_marco_city = travel_time[last_city1][polo[insert1]] + polo_travel_time
            #total_travel_time3 = polo_travel_marco_city + marco_travel_ith_city_instead
        elif insert1 == len(polo):
            polo_travel_marco_city = polo_travel_time + travel_time[polo[-1]][last_city1]
            #total_travel_time3 = polo_travel_marco_city + marco_travel_ith_city_instead
        else:
            polo_travel_marco_city = polo_travel_time - travel_time[polo[insert1-1]][polo[insert1]] +
travel_time[polo[insert1-1]][last_city1] + travel_time[last_city1][polo[insert1]]

    total_travel_time3 = marco_travel_ith_city_instead + polo_travel_marco_city

     # 4. polo travels to ith city instead of polo[-1] city
    if len(polo) == 0:
        total_travel_time4 = math.inf
    elif len(polo) == 1:
        polo_travel_ith_city_instead = 0
```

```
            last_city2 = polo[0]
            insert2 = len(marco)
            for j in range(len(marco)):
                if last_city2 < marco[j]:
                    insert2 = j
                    break
            if insert2 == 0:
                marco_travel_polo_city = travel_time[last_city2][marco[insert2]] + marco_travel_time
                #total_travel_time4 = marco_travel_polo_city + polo_travel_ith_city_instead
            elif insert2 == len(marco):
                marco_travel_polo_city = marco_travel_time + travel_time[marco[-1]][last_city2]
                #total_travel_time4 = marco_travel_polo_city + polo_travel_ith_city_instead
            else:
                marco_travel_polo_city = marco_travel_time -
travel_time[marco[insert2-1]][marco[insert2]] + travel_time[marco[insert2-1]][last_city2] +
travel_time[last_city2][marco[insert2]]
                #total_travel_time4 = marco_travel_polo_city + polo_travel_ith_city_instead
        else:
            last_city2 = polo[-1]
            polo_travel_ith_city_instead = polo_travel_time - travel_time[polo[-2]][polo[-1]] +
travel_time[polo[-2]][i]
            insert2 = len(marco)
            for j in range(len(marco)):
                if last_city2 < marco[j]:
                    insert2 = j
                    break
            if insert2 == 0:
                marco_travel_polo_city = travel_time[last_city2][marco[insert2]] + marco_travel_time
                #total_travel_time4 = polo_travel_ith_city_instead + marco_travel_polo_city
            elif insert2 == len(marco):
                marco_travel_polo_city = marco_travel_time + travel_time[marco[-1]][last_city2]
                #total_travel_time4 = marco_travel_polo_city + polo_travel_ith_city_instead

            else:
                marco_travel_polo_city = marco_travel_time -
travel_time[marco[insert2-1]][marco[insert2]] + travel_time[marco[insert2-1]][last_city2] +
travel_time[last_city2][marco[insert2]]
        total_travel_time4 = polo_travel_ith_city_instead + marco_travel_polo_city

        min_time = min(total_travel_time1, total_travel_time2, total_travel_time3,
total_travel_time4)

        if total_travel_time1 == min_time:
            marco.append(i)
```

```
            marco_travel_time = marco_travel_plus_ith_city
            polo_travel_time = polo_travel_time

        elif total_travel_time2 == min_time:
            polo.append(i)
            marco_travel_time = marco_travel_time
            polo_travel_time = polo_travel_plus_ith_city
        elif total_travel_time3 == min_time:
            marco = marco[:len(marco)-1] + [i]
            polo = polo[:insert1] + [last_city1] + polo[insert1:]
            marco_travel_time = marco_travel_ith_city_instead
            polo_travel_time = polo_travel_marco_city
        elif total_travel_time4 == min_time:
            marco = marco[:insert2] + [last_city2] + marco[insert2:]
            polo = polo[:len(polo)-1] + [i]
            marco_travel_time = marco_travel_polo_city
            polo_travel_time = polo_travel_ith_city_instead

    return marco, polo # return marco, polo city lists
```

Runtime complexity: the first loop iterates n times, the inner loop that calculates where to insert the city iterates at most n times. So the total runtime complexity is O(n^2)

Proof of its correctness: Firstly as prompted, n>=2, we do not worry about n=1. When there are n cities, we iterate through from i=2 to i=n. At each step, we make sure that the total travel time is the minimum we can make. So when i=n, the plan we make ensures the minimum total travel time.

4. (a) Subproblem: when Erica is making the plan for ith day, the decision is whether she should change i-1th day's schedule so that on both (i-1th + i-2th day) and (i-1th + ith day) have at least K lectures, or she can just change ith day's schedule so that (i-1th + ith day) have at least K lectures. If the penalty at i-2th day + the maximum in between (K - (i-1th day) - (i-2th day)), (K - (ith day) - (i-1th day)), and 0 is smaller than the penalty at i-1th day + the maximum in between 0, and (K - ith day - (i-1)th day), then she should change the schedule on i-1th day. Otherwise, she should change the schedule on ith day. Once the penalty list is generated, she can schedule her study plan by adding the penalty difference from each day and a[i] for i >= 1, for day 0, she can just study a[0] lectures

(b) Recurrence equation: dp[i] = min(dp[i-2] + max(K - A[i-1] - A[i-2], K - A[i] - A[i-1], 0), dp[i-1] + max(K - A[i] - A[i-1], 0))

Conner cases: if n == 1: total_penalty = max(K - A[0], 0)
If n == 2: total_penality = max(K - A[0] - A[1], 0)
If n >= 3: dp[0] = 0, dp[1] = max(K - A[0] - A[1], 0)

(c)
```
def helper(A,K):
        dp = [0] * len(A)
        plan = [0] * len(A)
        if len(A) == 1:
                plan[0] = K
                dp[0] = max(K-A[0], 0)
                return plan, dp[0]
        if len(A) == 2:
                plan[0] = A[0]
                plan[1] = max(K - A[0], A[1])
                 dp[0] = A[0]
                dp[1] = max(K - A[0] - A[1], 0)
                return plan, dp[1]
        dp[0] = 0
        dp[1] = max(K - A[0] - A[1], 0)
        for i in range(2, len(A)):
                dp[i] = min(dp[i-2] + max(K - A[i-1] - A[i-2], K - A[i] - A[i-1], 0), dp[i-1] +
max(K - A[i] - A[i-1], 0))

        plan[0] = A[0]
        for i in range(1, len(dp)):
                plan[i] = A[i] + dp[i] - dp[i-1]

        return plan, dp[len(A) - 1]
```

Return value plan is the new plan, dp[len(A) -1] is the total penalty

(d) the algorithm iterates the list in linear time, so the time complexity is O(n)

5. (a) Top down approach. Start from the end of the array to the start (from n-1 to 0, 0 indexed). If the current number can be reached, any number after it can also be reached. The subproblem is: at index i, choose the current number i + the sum at i + A[i] or choose the sum at i+1.

   (b) Recurrence equation: dp[i] = max(a[i] + dp[i+a[i]], dp[i+1])

Conner cases: if i + a[i] >= n and i + 1 >= n, then dp[i] = a[i].
If i + a[i] >= n, then dp[i] = max(a[i], dp[i+1])

(c)
```
n = len(a)
dp = [-1] * len(a)
for i in range(len(a) -1, -1, -1):
    if i + a[i] >= n and i + 1 >= n:
        dp[i] = a[i]
    elif i + a[i] >= n:
        dp[i] = max(a[i], dp[i+1])
    else:
        dp[i] = max(a[i] + dp[i+a[i]], dp[i+1])

return dp[0]
```
(d) The algorithm iterates the list only once, so Time complexity is O(n) where n is the length of a.

6. Proof only strings having the same length can be J-similar:
Assume there are two strings that are not the same length and J-similar. String a has length a1, and string b has length a1 + n. It will not satisfy the first condition a == b. In order for a and b to potentially satisfy the second condition. String a and b have to be even lengths. len(a) = a1, len(b) = a1 + 2n. Base case is len(a) = 2, len(b) = 4. When a and b enter the second call stack of J-similar function len(a) = 1, len(b) =2. It will not satisfy condition1 and condition 2. Induction Step: for any positive n, String a and b will not be J_similar. Induction Step: since original a and b do not have the same length, multiple J-similar functions must be called, and in each call, len(a) and len(b) will be divided by 2. Case 1: both a and b can be divided by 2. It will have a difference by 2n/ # of times the J-similar function gets called. Eventually, len(a) and len(b) will be 1 and 2. Therefore, they cannot be J-similar. Case 2: len(a) or len(b) will not be able to be divided by 2 during one of the function calls. If a or b cannot be further divided and a != b, they cannot be J-similar.
Therefore, only a and b with the same length can be J-similar.

Algorithm:
(1) If len(a) == 1, check if a is equal to b. (because a and b must be the same length, so len(a) == 1 means len(b) == 1 as well)
(2) Else If a == b, if true return true
(3) Check if len(a) != len(b) if their lengths are different, return false
(4) Check if a and b can be further divided by 2, if not , return false
(5) A1 = first half of a A2 = second half of a, b1 = first half of b, b2 = second half of b
(6) Check (J(a1, b1) and J(a2, b2)) or (J(a1, b2) and J(a2, b1)) is true, if one of them is true, return true

Recurrence equation: $T(n) = 2T(n/2) + n$

$A = 2$, $b = 2$, $c = 1$. $f(n) = n$, $T(n) = \Theta(n\log n)$

7. Original p = [0, 1, 1, 0] reversed q = [1, 0, 0, 1]
First inversion: pq = [0, 1, 1, 0, 1, 0, 0, 1]
Second inversion: = [0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0] = pqqp
Third inversion = pqqpqppq
…

Every 4 elements contain two original order arrays and two reversed order arrays.
So we can divide the infinite long array into 3 pieces:

1. From a counting forward, find the first index i that makes the equation i % (4*n) == 0
   Iterate the array to find the number of 1s in between a and i, mark as I1
2. From b counting backward, find the first index j that makes the equation j %(4*n) ==0. Iterate the array to count the number of 1s in between j and b, mark as I2
3. The number of 1s in between i and j can be found using equation: (j-i)/(4n) number of 1s in length 4n.

Step 1 and 2 takes linear time, step 3 takes constant time. The total runtime complexity is O(n)