1. *Exercise 19 on page 329*

[Algorithm]
Suppose the length of s is n.
Define x' be the repetition of x with length n
Define y' be the repetition of y with length n
Let s_i denotes i-th character of s.
Let x'_i denotes i-th character of x'.
Let y'_i denotes i-th character of y'.
Let $dp[i][j]$ denote whether or not $s_1 s_2 ... s_{i+j}$ is interleaving of $x'_1 x'_2 ... x'_i$ and $y'_1 y'_2 ... y'_j$
$dp[i][j] = 1$ if it is, otherwise $dp[i][j] = 0$
Then we have the recurrence
$dp[i][j] = 1$ *if* $(dp[i][j-1] = 1$ *and* $s_{i+j} = y'_j)$ *or* $(dp[i-1][j] = 1$ *and* $s_{i+j} = x'_i)$
Then s is interleaving of x and y if there exists i, j such that $dp[i][j] = 1$ and $i + j = n$
Thus the algorithm is the following:

```
1 dp[0][0] = 1
2 For i = 0 to n:
3     For j = 0 to n:
4         If (dp[i][j-1] = 1 and s_{i+j} = y'_j) or (dp[i-1][j] = 1 and s_{i+j} = x'_i):
5             dp[i][j] = 1
6             If i + j = n: Return True
7         Else:
8             dp[i][j] = 0
9 Return False
```

[Proof of correctness]
The algorithm performs an exhaustive search with dynamic programming, so it suffices to prove the recurrence $dp[i][j] = 1$ *if* $(dp[i][j-1] = 1$ *and* $s_{i+j} = y'_j)$ *or* $(dp[i-1][j] = 1$ *and* $s_{i+j} = x'_i)$ by induction.
(Base Case)
$dp[i][j] = 1$ because we can just choose nothing from x' and y'
(Inductive Case)
Suppose $dp[k][j]$ holds for all $0 \le k \le n-1,\ 0 \le j \le n$ and $dp[i][l]$ holds for all $0 \le l \le j-1$.
We want to show $dp[i][j]$ holds. Note we already know $dp[i][j-1]$ (i.e. whether or not $s_1 ... s_{i+j-1}$ is interleaving of $x'_1 ... x'_i$ and $y'_1 ... y'_{j-1}$). If it is, then for $s_1 ... s_{i+j}$ to be interleaving of $x'_1 ... x'_i$ and $y'_1 ... y'_j$, it must be the case where $s_{i+j} = y'_j$. Similarly, we know $dp[i-1][j]$ (i.e. whether or not $s_1 ... s_{i+j-1}$ is interleaving of $x'_1 ... x'_{i-1}$ and $y'_1 ... y'_j$). If it is, then for $s_1 ... s_{i+j}$ to be interleaving of $x'_1 ... x'_i$ and $y'_1 ... y'_j$, it must be the case where $s_{i+j} = x'_i$. Thus, we have $dp[i][j] = 1$ *if* $(dp[i][j-1] = 1$ *and* $s_{i+j} = y'_j)$ *or* $(dp[i-1][j] = 1$ *and* $s_{i+j} = x'_i)$.

[Time complexity analysis]
We have two nest loops at line 2 and 3, which loops $O(n^2)$ times. During each iteration, we perform line 4 to 8, which takes constant time. Thus, the time complexity of this algorithm is $O(n^2)$.

2. *Exercise 22 on page 330*

[Algorithm]
We want to find the number of shortest $v - w$ paths in G, that has edges with negative cost, but without negative cycle.

Suppose we have V vertices.
Suppose we have E edges.
Then the longest simple path (a path without cycle) has length $n - 1$.
(Because if we have more than $n - 1$ edges, then we are repeating some nodes. Thus a cycle. Since we are assuming no negative cycle, then having a cycle can only increase the distance.)
Let $dp[n][l]$ denotes the shortest distance from v to n of length at least l.
Then we have the recurrence $dp[n][l] = min(dp[k][l-1] + w_k)$ *for every* $(k,\ n)$ *edge with weight* w_k.

The algorithm is the following:

```
1  dp[*][*] = ∞ (Initialize everything to ∞)
2  dp[v][0] = 0
3  count = 0
4  minDistance = ∞
5  For i = 1 to V − 1 :
6     For j = 0 to E − 1 :
7        Let x be the source and y be the target of j -th edge
8        Let weight be the weight of j -th edge
9        If dp[x][i − 1] ≠ ∞ :
10           dp[y][i] = min(dp[y][i], dp[x][i − 1] + weight)
11           If y = w :
12              If minDistance = dp[y][i] :
13                 count = count + 1
14              Else if minDistance > dp[y][i] :
15                 minDistance = dp[y][i]
16                 count = 1
17 Return count
```

[Proof of correctness]
The algorithm performs an exhaustive search to find minimum distance using dynamic programming. Then every time it compute the distance to w, we keep track of the minimum distance and how many time that distance appeared throughout the algorithm. Finally, we return the count of the minimum distance. Thus, it suffices to show why the recurrence $dp[n][l] = min(dp[k][l-1] + w_k)$ *for every* $(k,\ n)$ *edge with weight* w_k makes sense.

We will prove this by induction.
(Base Case)
$dp[v][0] = 0 \; and \; dp[k][0] = \infty \; for \; all \; k \neq v$ because if we can use no edge, then we can't go anywhere.
(Inductive Case)
Suppose $dp[n][l-1]$ is given for all n. We want to show
$dp[n][l] = min(dp[k][l-1] + w_k) \; for \; every \; (k, \; n) \; edge \; with \; weight \; w_k$.

Note to each node n with length l, it must reach neighbors of n with length $l-1$. Since we are interested in finding the minimum distance/weight, we have
$dp[n][l] = min(dp[k][l-1] + w_k) \; for \; every \; (k, \; n) \; edge \; with \; weight \; w_k$.

[Time complexity analysis]
We have nested for loop at line 5 and 6, which is $O(VE)$. During each iteration, we perform line 7 to 16, which can be done in constant time. So the time complexity of this algorithm is $O(VE)$.

3. *Exercise 24 on page 331*

[Algorithm]
Suppose there are n precincts
Suppose there are m votes every precinct
Denote the number of votes for party A in i-th precinct be a_i
Denote the number of votes for party B in i-th precinct be b_i
Define $f_i = a_i - b_i$
Then if $\sum_{i=1}^{n} f_i = 0$, then there's no way to split the precincts such that both districts have majority.
If $\sum_{i=1}^{n} f_i > 0$, then we want to know whether it's possible to split the precincts such that both districts have majority votes for party A.
If $\sum_{i=1}^{n} f_i < 0$, then we want to know whether it's possible to split the precincts such that both districts have majority votes for party B.
Without loss of generality, assume $\sum_{i=1}^{n} f_i > 0$ (if $\sum_{i=1}^{n} f_i < 0$, just switch A and B).
So we want to know whether or not we can separate the precincts into two districts such that party A is the majority in both of them.
The problem can be translated as whether or not we can separate f_i into two groups such that sum of f_i in each group is positive.
Let $dp[i][p][q][k]$ denote whether or not we can put k precincts out of the first i precincts to district 1, rest to district 2, and have both districts' value positive given that the value in district 1 is p and the value in district 2 is q.
If it is $dp[i][p][q][k] = true$. Otherwise, $dp[i][p][q][k] = false$
Then we have the recurrence $dp[i][p][q][k] = true$ iff $(dp[i-1][p+f_i][q][k-1] \text{ and } k \geq 1)$ or $(dp[i-1][p][q+f_i][k] \text{ and } i \geq 1)$
Note $dp[i][p][q][k] = true$ iff $p, q > 0$, and we want to know $dp[n][0][0][n/2]$.
Also note the most p and q can get is sum of all positive f_i. Denote this sum as U
The least p and q can get is sum of all negative f_i. Denote this sum as L

The algorithm is the following:

```
1 Suppose dp[i][p][q][k] with 0 ≤ i ≤ n, L ≤ p,q ≤ U, 0 ≤ k ≤ i
2 dp[0][p][q][0] = true for all p,q > 0
3 For i = 1 to n:
4    For p = L to U:
5       For q = L to U:
6          For k = 0 to i:
```

```
7            If (dp[i − 1][p + f_i][q][k − 1] and k ≥ 1) or (dp[i − 1][p][q + f_i][k] and i ≥ 1):
8                dp[i][p][q][k] = true
9 Return dp[n][0][0][n/2]
```

[Proof of correctness]

The algorithm performs an exhaustive search using dynamic programming. Thus, it suffices to show why the recurrence $dp[i][p][q][k] = true$ iff $(dp[i-1][p+f_i][q][k-1] \text{ and } k \geq 1)$ or $(dp[i-1][p][q+f_i][k] \text{ and } i \geq 1)$ is correct.

We will prove this by induction.

(Base Case)

$dp[0][p][q][0] = true$ for all $p, q > 0$ is trivial because District 1 and District 2 already have positive values and we don't have any precinct to assign.

(Inductive Case)

Suppose we know $dp[i-1][p][q][k]$ for all possible $p, q, \text{ and } k$. We want to show $dp[i][p][q][k] = true$ iff $(dp[i-1][p+f_i][q][k-1] \text{ and } k \geq 1)$ or $(dp[i-1][p][q+f_i][k] \text{ and } i \geq 1)$.

So either the i-th precinct belongs to district 1 or it belongs to district 2.

If the i-th precinct belongs to district 1, then if $dp[i-1][p+f_i][q][k-1]$ is true, then this means the value in district 1 is still positive even with i-th precinct in district 1.

If the i-th precinct belongs to district 2, then if $dp[i-1][p][q+f_i][k]$ is true, then this means the value in district 2 is still positive even with i-th precinct in district 2.

So $dp[i][p][q][k] = true$ iff $(dp[i-1][p+f_i][q][k-1] \text{ and } k \geq 1)$ or $(dp[i-1][p][q+f_i][k] \text{ and } i \geq 1)$

[Time complexity analysis]

Denote $S = U - L$

We have four nested loops. Line 3 and 6 takes $O(n^2)$. Line 4 and 5 takes $O(S^2)$. During each iteration, we perform line 7 and 8, which takes constant time. Thus, the time complexity of this algorithm is $O(n^2 S^2)$.

4. *Exercise 7 on page 417*

[Algorithm]

Suppose we have n clients and k base stations.

The problem can be translated into Ford-Fulkerson algorithm (i.e. flow network problem).

Let v_i be a node that represents client i.

Let w_j be a node that represents base station j.

If client i is within the distance r from base station j, then we create an edge (v_i, w_j) with capacity 1.

Let s be the source node that connects to every v_i with capacity 1.

Let t be the sink node that connects to every w_j with capacity L.

Then if there is a $s-t$ flow with value n, then every client can be connected to a base station subject to the range and load conditions.

[Proof of algorithm]

We want to prove the statement "there is a $s-t$ flow with value n ⇔ every client can be connected to a base station subject to the range and load conditions"

($\Leftarrow$)

Suppose every client can be connected to a base station subject to the range and load conditions. From source node s, we can send flow of value 1 to each client v_i (total flow is n because there are n clients). Then from each client v_i, we can send a flow to a base station w_j without sending a flow over L to a base station (we know it satisfies load condition by assumption). Thus, there is a $s-t$ flow with value n.

($\Rightarrow$)

Suppose there is a $s-t$ flow with value n. Then, source node s must have sent a flow of value 1 to every v_i. Then, every v_i must have sent a flow to a base station it can reach without overloading any base station because of the capacity constraint L. Thus every client can be connected to a base station subject to the range and load conditions.

[Time complexity analysis]

We have $O(n+k)$ nodes and $O(nk)$ edges. Note the max flow from source s is n. Since the time complexity of Ford-Fulkerson algorithm is $O(mC)$, where m is number of edges and C is the maximum flow, time complexity of this algorithm is $O(n^2k)$.

5. *Exercise 9 on page 419*

[Algorithm]

This problem is pretty much the same as previous problem.

Suppose we have n patients and k hospitals.

The problem can be translated into Ford-Fulkerson algorithm (i.e. flow network problem).

Let v_i be a node that represents patient i.

Let w_j be a node that represents hospital j.

If patient i is within half-hour's driving time from hospital j, then we create an edge (v_i, w_j) with capacity 1.

Let s be the source node that connects to every v_i with capacity 1.

Let t be the sink node that connects to every w_j with capacity n/k.

Then if there is a $s-t$ flow with value n, then every patient can be sent to a hospital subject to the driving time and load constraints.

[Proof of algorithm]

We want to prove the statement "there is a $s-t$ flow with value n ⇔ every patient can be sent to a hospital subject to the driving time and load constraints"

($\Leftarrow$)

Suppose every patient can be sent to a hospital subject to the driving time and load constraints. From source node s, we can send flow of value 1 to each patient v_i (total flow is n because there are n patients). Then from each patient v_i, we can send a flow to a hospital w_j without sending a flow over n/k to a base station (we know it satisfies load condition by assumption). Thus, there is a $s-t$ flow with value n.

($\Rightarrow$)

Suppose there is a $s-t$ flow with value n. Then, source node s must have sent a flow of value 1 to every v_i. Then, every v_i must have sent a flow to a hospital it can reach without overloading any hospital because of the capacity constraint n/k. Thus every patient can be sent to a hospital subject to the driving time and load constraints.

[Time complexity analysis]

We have $O(n+k)$ nodes and $O(nk)$ edges. Note the max flow from source s is n. Since the time complexity of Ford-Fulkerson algorithm is $O(mC)$, where m is number of edges and C is the maximum flow, time complexity of this algorithm is $O(n^2k)$.

6. *Given a sequence of numbers find a sub-sequence of alternating order, where the sub-sequence is as long as possible. (that is, find a longest sub-sequence with alternate low and high elements). As always, prove the correctness of your algorithm and analyze its time complexity.*

Example Input: 8, 9, 6, 4, 5, 7, 3, 2, 4
Output: 8,9,6,7,3,4 (of length 6) because 8 < 9 > 6 < 7 > 3 < 4

[Algorithm]
Let $a[i]$ be i-th element of the input
Let $dp_S[i]$ be length of longest alternating subsequence ending at index i and last element is smaller than the previous element
Let $dp_L[i]$ be length of longest alternating subsequence ending at index i and last element is greater than the previous element
Then we want to find $max(dp_S[i],\ dp_L[i])$ for all i
Note we have the recurrence $dp_S[i] = max(dp_L[k] + 1)\ for\ all\ k < i\ \ and\ a[k] > a[i]$ and $dp_L[i] = max(dp_S[k] + 1)\ for\ all\ k < i\ and\ a[k] < a[i]$.
Thus the algorithm is the following:

```
1  dp_S[i] = dp_L[i] = 1 for all i
2  res = 1
3  For i = 1 to n:
4     For k = 1 to i − 1:
5        If a[k] > a[i]:
6           dp_S[i] = max(dp_S[i], dp_L[k] + 1)
7        If a[k] < a[i]:
8           dp_L[i] = max(dp_L[i], dp_S[k] + 1)
9     res = max(dp_S[i], dp_L[i])
10 Return res
```

[Proof of correctness]
Our algorithm performs an exhaustive search using dynamic programming. Thus it suffices to show the recurrence $dp_S[i] = max(dp_L[k] + 1)\ for\ all\ k < i\ \ and\ a[k] > a[i]$ and $dp_L[i] = max(dp_S[k] + 1)\ for\ all\ k < i\ and\ a[k] < a[i]$ is correct.
We will prove this by induction.
(Base Case)
$dp_S[1] = 1,\ dp_L[1] = 1$ are obvious because if we only have one number, then the alternating subsequence is that one number itself.

(Inductive Case)
Suppose we have $dp_S[i],\ dp_L[i]$ for all $1 \le i \le n-1$.
We want to show we can get $dp_S[n],\ dp_L[n]$ by the recurrence we propose.
Since we already know $dp_L[i]$ for all $1 \le i \le n-1$, we can find $dp_S[n]$ by trying every possible subsequence that ends at index $i\ (1 \le i \le n-1)$ and add $a[n]$ to that sequence and see whether it is an alternating sequence. Thus, we have the recurrence
$dp_S[i] = max(dp_L[k]+1)\ for\ all\ k < i\ \ and\ a[k] > a[i]$. $a[k] > a[i]$ is there to make sure it is indeed an alternating subsequence. By the same logic, we have
$dp_L[i] = max(dp_S[k]+1)\ for\ all\ k < i\ and\ a[k] < a[i]$

[Time complexity analysis]
There are two nested loops at line 3 and 4, which takes $O(n^2)$. During each iteration, we execute line 5 to 9, which takes constant time. Thus the time complexity of this algorithm is $O(n^2)$.