Q1-a

(a)Subproblem to Solve:

Define V(i), a boolean value indicates whether or not the subsequence S[1,...,i] is a sequence of valid English words.

Subproblems are determining V(i) for all 1 <= i <= n.

(b)Recursive Formula:

$$V(i) = \begin{cases} \text{True} & \text{if DICT(S[1,...,i]) = True} \\ \text{True} & \text{if } \exists \ 1 <= j < i \text{ s.t } V(j) = \text{True and DICT(S[j,...,i]) = True} \\ \text{False} & \text{otherwise} \end{cases}$$

(c)Algorithm:

```
IS_VALID(S[1..n]):
# initialize a list V to store the result of subproblems
V := list of size n
for i := 1 to n do
        if DICT(S[1..i]) then
                V[i] := True
        else
                j := i - 1
                while  j >= 1 do
                        if DICT([S[j..i]]) and V[j] then
                                V[i] := True
                                break
                        j := j - 1
                V[i] := Flase
return V[n]
```

Proof of Correctness:

(i) The recursive formula in step (b) correctly computes the subproblem in step (a).

By "cut-and-paste" argumentation, a sequence of valid English words **S** = S[1..i] consists of a smaller sequence of valid English words **S'** = S[1,..,j] where j < i concatenate with a valid english word **w** = S[j+1..i]. i.e. **S = S'w**

Case 1: when **S'** is an empty sequence:

⇒ **S = w**

⇒ **S** is a valid sequence of English Words iff **w** is a valid word

Case 2: when **S'** is not empty:

⇒ **S** is valid sequence iff ∃ a **S'** is valid sequence s.t. **w** is valid word

⇒ otherwise, **S** is **not** valid sequence

Therefore we have the recursive formula at above.

(ii) Why the computation in (c) yields a solution to the given problem.

For the computation in (c) we solve the subproblems V(i) for each 1 <= i <= n. And each subproblem depends on some subproblems before it. The original problem is to determine whether or not S[1..n] is a sequence of valid English words. Therefore V[n] in part (c) is the answer for the original problem and get returned as wanted.

Running Time: $\Theta(n^2)$

Q1-b

```
PRINT_IFVALID(S[1..n]):
# initialize a list V to store the result of subproblems
V := list of size n
# initialize a list P to track the last index of a previous valid sequence
P:= list of size n
for i := 1 to n do
        if DICT(S[1..i]) then
                V[i] := True
                P[i] = 1
        else
                j := i - 1
                while  j >= 1 do
                        if DICT([S[j..i]]) and V[j] then
                                V[i] := True
                                P[i] := j
                                break
                        j := j - 1
                V[i] := Flase
# we only print the words when the sequence is valid
if V[n] is True then
        # backtrack to find the "path"
        stack = Stack()
        for i := n to 1 do
                if V[i] then
                        stack.push(P[i])
        # print the words
        start = stack.pop()
        while stack is not empty do
                end = stack.pop()
                print(S[start..end])
                start = end
        print(S[start..n])
```

Q2-a

(a) Subproblem to Solve:

Find $C(i, c_j)$, a **tuple** of two items, the first one $C(i, c_j)[1]$ is the minimum cost of the first i cuttings end at a particular position c_j where $1 <= i, j <= m$; the second one $C(i, c_j)[2]$ is the previous position of c_j which is used to track the path and compute the cost from the $(i-1)^{th}$ cutting to the i^{th} cutting.

$$\text{Define } c_{(j-1)} = \begin{cases} 0 & \text{if there is no such } c_k \\ \max(c_k \mid c_k < c_j \text{ and } c_k \text{ is used before } c_j) & \text{O/W} \end{cases}$$

$$\text{Define } c_{(j+1)} = \begin{cases} n \text{ (length of given string)} & \text{if there is no such } c_k \\ \min(c_k \mid c_k > c_j \text{ and } c_k \text{ is used before } c_j) & \text{O/W} \end{cases}$$

Define $d(c_j)$, a distance function that compute the current cost of cutting a string at position c_j.

$$d(c_j) = \begin{cases} c_{(j+1)} - c_{(j-1)} & \text{if } c_j \text{ is not used previously} \\ \infty & \text{if } c_j \text{ is used previously} \end{cases}$$

(b) Recursive Formula:

$$C(i, c_j) = \begin{cases} (n, \text{nil}) & \text{if } i = 1 \\ (\min\{C(i-1, c_k)[1] + d(c_j) \mid k = 1,...,m\}, c_k) & \text{if } i > 1 \end{cases}$$

(c) Algorithm:

```
MINC([c_1, c_2,...,c_m]):
C := empty m*m matrix
for j := 1 to m do
        C(1, c_j) := (n, nil)
for i := 2 to m do
        for j := 1 to m do
                C(i, c_j) := (∞, nil)
                for k := 1 to m do
                        # backtrack, to get all the info required for d(c_jf)
                        while C(i-1, c_k)[2] is not nil do
                                find c_(j-1), c_(j+1) and whether c_j is used or not
                        if C(i-1, c_k)[1] + d(c_j) < C(i, c_j)[1] then
                                C(i, c_j) := (C(i-1, c_k)[1], c_k)
return min{C(m, c_j)[1] : j = 1,...,m}
```

Proof of Correctness:

(i) The recursive formula in step (b) correctly computes the subproblem in step (a).

Suppose $C(i, c_j)$ contains the minimum cost of the first i cuttings end at a particular position c_j (recall, $C(i, c_j)$ is a **tuple**). Then by "cut-and-paste" argumentation, $\exists$ a $C(i-1, c_k)$ for some k such that $C(i-1, c_k)$ contains the minimum cost of the first i-1 cuttings end at position c_k and $C(i-1, c_k)[1] + d(c_j) = C(i-1, c_j)[1]$ where $d(c_j)$ compute the cost from $C(i-1, c_k)$ to $C(i, c_j)$.

Case(1): when i = 1:

There is no previous position, and the cost of the first cutting is the length of the string.

Therefore $C(1, c_j) = (n, \text{nil})$ for all j = 1,2,..,m

Case(2): when i > 1:

Proof the correctness of the distance function $d(c_j)$:

Note: all the info required for $d(c_j)$ can be found in Matrix C through backtrack

For example, we want to cut a 20-character string at positions 3 and 10.

Let's say we cut at position 3 first, the first cut the cost is the length 20.

By definitions above:

$c_{(j-1)} = 0$	since there is no previous cutting
$c_{(j+1)} = 20$	since there is no previous cutting
$d(3) = 20 - 0$	since we not used 3 in previous cutting
$= 20$	as wanted

Then we cut at position 10, the cost will be the length of S_2 which is 17

By definitions above:

$c_{(j-1)} = 3$	since 3 is the maximum number in previous cutting < 10
$c_{(j+1)} = 20$	since there is no previous cutting
$d(10) = 20 - 3$	since we not used 3 in previous cutting
$= 20$	as wanted

if we try to cut at position 10 again (which is not allowed) we will have:

$d(10) = \infty$	since we used 10 before, we cannot use it again

Then the path contained duplicated positions will never be the answer as a minimum cost path, because the cost is infinite.

Therefore $d(c_j)$ is correct and the info we need can be found through backtrack.

Now we want to use the distance function to do comparison.

We find $C(i, c_j)$ by comparing all $C(i-1, c_k)[1] + d(c_j)$ and get the minimum one

Therefore $C(i, c_j) = (\min\{C(i-1, c_k)[1] + d(c_j) \mid k = 1,...,m\}, c_k)$.

(ii) Why the computation in (c) yields a solution to the given problem.

For the computation in (c) we solve the subproblems $C(i, c_j)$ for each i, j = 1,2,...,m. And each subproblem depends on some subproblems before it ($C(i-1, c_k)$, k = 1,2,...,m and the subproblems we used in backtrack to compute distance). The original problem is to compute the minimum cost of the cutting $c_1 < c_2 < \ldots < c_m$ (sum of the cost of total m cuttings). Therefore $\min\{C(m, j)[1] : j = 1,...,m\}$ is the answer for the original problem and get returned as wanted.

Running Time: Θ(m^4)
Explanation: there are m*m subproblem (result stored in the matrix C), for each subproblem we need to do m comparison (compare with C(n-1, c_k) for k = 1,2,...,m), for each comparison we need O(m) time to do backtrack to gathering the info for distance function d(c_j).

Q2-b

```
MINC([c_1, c_2,...,c_m]):
C := empty m*m matrix
for j := 1 to m do
        C(1, c_j) := (n, nil)
for i := 2 to m do
        for j := 1 to m do
                C(i, c_j) := (∞, nil)
                for k := 1 to m do
                        # backtrack, to get all the info required for d(c_j)
                        while C(i-1, c_k)[2] is not nil do
                                find c(j-1), c(j+1) and whether or not c_j is used
                        if C(i-1, c_k)[1] + d(c_j) < C(i, c_j)[1] then
                                C(i, c_j) := (C(i-1, c_k)[1], c_k)
tracker =  (min{C(m, c_j)[1] : j = 1,...,m}, c_j)
stack = Stack()
while tracker[2] is not nil do
        stack.push(tracker[2])
        tracker = C(tracker[1]-1, tracker[2])
while stack is not empty do
        print(stack.pop())
```

Q3
(a) Subproblem to Solve:
SD(i, k) = boolean function to justify whether the A[1..i] can be partitioned into two subsets whose sums diff by k, where $1 \le i \le n$ and $0 \le k \le \sum_{j=1}^{n} A[j]$

(b)Recursive Formula:

$$SD(i, k) = \begin{cases} (A[1] == k) & \text{if } i = 1 \\ (A[1] - A[2] == k) \text{ or } (A[2] - A[1] == k) \text{ or } (A[1] + A[2] == k) & \text{if } i = 2 \\ SD(i-1, k - A[n]) \text{ or } SD(i - 1, k + A[n]) & \text{if } i > 2 \end{cases}$$

(c)Algorithm:

```
SUMDIFFERENCE(A, k):
total := 0
for i := 1 to n do
        total = total + A[i]
for k := 0 to total do
        if (A[1] = k) then
                SD(1, k) = true
        else
                SD(1, k) = false
for k := 0 to total do
        if (A[1] + A[2] = k) then
                SD(2, k) = true
        else if  (A[1] - A[2] = k) then
                SD(2, k) = true
        else if  (A[2] - A[1] = k) then
                SD(2, k) = true
        else
                SD(2, k) = false
for i := 1 to n do
        for k := 0 to total do
                if (SD(i - 1, k - A[i]) = true) then
                        SD(i, k) = true
                else if (SD(i - 1, k + A[i]) = true) then
                        SD(i, k) = true
                else
                        SD(i, k) = false
return SD(n, k)
```

Proof of Correctness:

(i) The recursive formula in step (b) correctly computes the subproblem in step (a).

By "cut-and-paste" argumentation, an optimal set S can be partitioned into two subsets S_1 and S_2 whose sums differ by k, where $S_2 = S - S_1$.

Without loss of generality, let's assume $S_1 - S_2 = k$.

There are two possible cases considering A[n]:

Case 1: $A[n] \in S_1$

$\Rightarrow$ S - A[n] can be partitioned into two subsets whose sums differ by k - A[n].

Case 2: $A[n] \in S_2$

$\Rightarrow$ S - A[n] can be partitioned into two subsets whose sums differ by k + A[n].

Therefore we have the recursive formula at above.

(ii) Why the computation in (c) yields a solution to the given problem.

For the computation in (c) we solve the subproblems SD(i, k) for each 1 <= i <= n and $0 \le k \le \sum_{j=1}^{n} A[j]$. Each subproblem depends on some subproblems before it. The original problem is to determine whether or not SD[1,..,n] is a set that can be partitioned into two subsets whose sums differ by k. Therefore SUMDIFFERENCE(A, k) where A is a set of size n in part (c) is the answer for the original problem and get returned as wanted.

Running Time: $\Theta(n \sum_{j=1}^{n} A[j])$

Explanation: there are n non-negative numbers in A, and the difference of the two subsets' sums is non-negative, therefore the range of the difference between two subsets' sum is from 0 to $\sum_{j=1}^{n} A[j]$.

When taking one more number into consideration, it would either increase or decrease the difference between the two subsets in the last step. Hence for every number, all the possible values obtained from 0 to $\sum_{j=1}^{n} A[j]$ in the last step need to be taken into consideration. Therefore, the time complexity would be **n** $\sum_{j=1}^{n} A[j]$.

Why the algorithm is pseudo-polynomial?

Answer: The algorithm is pseudo-polynomial since it is polynomial in value of the input instead of polynomial in the size of the input.