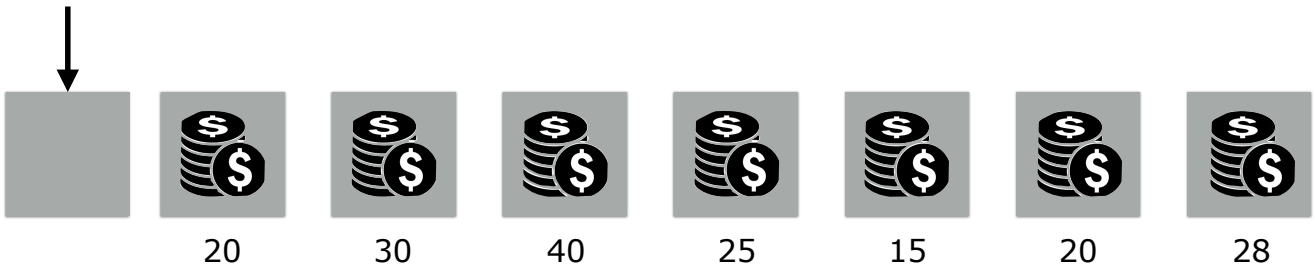
Exercise

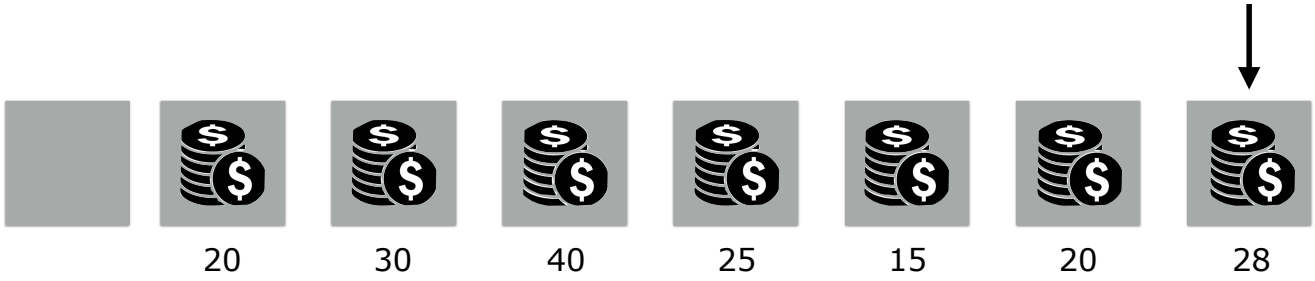
1. There are N big cubes which are arranged sequentially at equal distances. You are standing on the first cube indexed 0. You need to jump over the cubes to reach to the end. In one jump you can jump over maximum of X stones. Whenever you jump on a stone you need to pay certain coins. You need to figure out what is the minimum coins that you need to pay to reach the last stone. The coins that need to be paid at each stone is given in an array C.

X = 3



1. There are N big cubes which are arranged sequentially at equal distances. You are standing on the first cube indexed 0. You need to jump over the cubes to reach to the end. In one jump you can jump over maximum of X stones. Whenever you jump on a stone you need to pay certain coins. You need to figure out what is the minimum coins that you need to pay to reach the last stone. The coins that need to be paid at each stone is given in an array C.

X = 3



1. Implement a Dynamic programming solution
2. Analyze the time and space complexity
3. Modify the solution to print out the solution. i.e the optimal path.

Hint : Start from the last and work your way backwards.

1. State

1. State

Parameters

i - current index.

Cost function

minCost(i,C,X) - Minimum number of coins required to reach the stone at index i

2. Transitions

2. Transitions

Base case

i=0, return 0

Choices

minCost(i,C,X)

If we are at index i. We must have reached at i by jumping anywhere between 1 and X blocks. So we must haved jumped from

i-1 or i-2 or i-X. Now we can reduce the problem to smaller problems

minCost(i-1), minCost(i-2), minCost(i-3) … minCost(i-X)

Optimal choice

Because we are want path which costs least amount we pick the minimum

minCost(i,C,X) = MIN(minCost(i-j)) for j=1,2,3…X and i-j >=0

X = 3

minCost(5) = MIN(minCost(4),minCost(3),minCost(2))+C[5]

minCost(4) = MIN(minCost(3),minCost(2),minCost(1))+C[4]

minCost(3) = MIN(minCost(2),minCost(1),minCost(0))+C[3]

minCost(2) = MIN(minCost(1),minCost(0))+C[2]

minCost(1) = MIN(minCost(0))+C[1]

minCost(0) = 0

3. Recursive solution

3. Recursive solution

Using this recurrence relation we can implement a recursive solution

Pseudo code

```
minCost(i,C,X)
   if i == 0
      return 0
  min = INFINITE
  for j=1;j<=X and (i-j)>=0; i++
      min = MIN(min,minCost(i-j,C,X)+C[i])
  return min
```

Java

```java
public static int minCost(int[] C,int X){
    return minCost(C.length-1,C,X);
}
public static int minCost(int i,int[] C,int X){
    if(i == 0){
        return 0;
    }
    int min = Integer.MAX_VALUE;
    for(int j=1;j<=Math.min(i,X);j++){
        min = Math.min(min,C[i]+minCost(i-j,C,X));
    }
    return min;
}
```

Python

```
def min_cost_path(i, C, X):
    if i == 0:
        return 0
    min_cost = sys.maxsize
    for j in range(1, min(X, i) + 1):
        min_cost = min(min_cost, C[i] + min_cost_path(i - j, C,
X))
    return min_cost
```

X = 3

minCost(5) = MIN(minCost(4),minCost(3),minCost(2))+C[5]

minCost(4) = MIN(minCost(3),minCost(2),minCost(1))+C[4]

minCost(3) = MIN(minCost(2),minCost(1),minCost(0))+C[3]

minCost(2) = MIN(minCost(1),minCost(0))+C[2]

minCost(1) = MIN(minCost(0))+C[1]

minCost(0) = 0

4. Memoize

4. Memoization

We need to cache the results instead of solving it again and again

We can use an array

Key -> Index i

Value -> Minimum cost to reach i

Java

```java
public static int minCostMemo(int[] C,int X){
    int[] cache = new int[C.length];
    return minCostMemo(C.length-1,C,X,cache);
}
public static int minCostMemo(int i,int[] C,int X,int[] cache){
    if(i == 0){
        return 0;
    }
    if(cache[i] != 0){
        return cache[i];
    }
    int min = Integer.MAX_VALUE;
    for(int j=1;j<=Math.min(i,X);j++){
        min = Math.min(min,C[i]+minCostMemo(i-j,C,X,cache));
    }
    cache[i] = min;
    return min;
}
```

Python

```
def min_cost_path_memo(i, C, X, cache):
    if i == 0:
        return 0
    if cache[i] != 0:
        return cache[i]
    min_cost = sys.maxsize
    for j in range(1, min(X, i) + 1):
        min_cost = min(min_cost, C[i] + min_cost_path(i - j, C,
X))
    cache[i] = min_cost
    return min_cost
```

5. Bottom up approach

5. Bottom up approach

Recurrence relation

minCost(i,C,X) = MIN(minCost(i-j)) for j=1,2,3…X and i-j >=0

Bottom up equation

dp[i] = MIN(dp[i-j]+C[i]) for j=1,2,….X AND (i-j)>=0

dp[0] = 0

We solve for all i=1,2,3….N

Java

```java
public static int minCostDP(int[] C,int X){
    int N = C.length;
    int[] dp = new int[N];
    for(int i=1;i<N;i++){
        dp[i] = Integer.MAX_VALUE;
        for(int j=1;j<=Math.min(X,i);j++){
            dp[i] = Math.min(dp[i],dp[i-j]+C[i]);
        }
    }
    return dp[N-1];
}
```

Python

```python
def min_cost_dp(C, X):
    N = len(C)
    dp = [sys.maxsize for _ in range(0, N)]
    dp[0] = 0
    for i in range(0, N):
        for j in range(1, min(X, i) + 1):
            dp[i] = min(dp[i], dp[i - j] + C[i])
    return dp[N - 1]
```

6. Reconstruct the path

We also need to record the optimal choice in an array ‘jump’

For every index i, we record index y such that jumping from y to i results in the minimum cost.

To reconstrcut the path , we start at the last index N-1. We check the ‘jump’ array and get the index from which we jumped to i. We print y.

Then we assign i=jump[i], and then repeat the process, until we hit i=0

Java

```java
public static int minCostDPReconstruct(int[] C,int X){
    int N = C.length;
    int[] dp = new int[N];
    int[] jump = new int[N];
    dp[0]=0;
    for(int i=1;i<N;i++){
        dp[i] = Integer.MAX_VALUE;
        for(int j=1;j<=Math.min(X,i);j++){
            if(dp[i-j]+C[i] < dp[i]){
                jump[i] = i-j;
                dp[i] = dp[i-j]+C[i];
            }
        }
    }
```

Java

```
 int i=N-1;
    System.out.print(i+" -> ");
    while(i>0){
        i = jump[i];
        System.out.print(i+" -> ");
    }
    return dp[N-1];
}
```

Python

```python
def min_cost_dp_reconstruct(C, X):
    N = len(C)
    dp = [sys.maxsize for _ in range(0, N)]
    jump = [0 for _ in range(0, N)]
    dp[0] = 0
    for i in range(0, N):
        for j in range(1, min(X, i) + 1):
            if dp[i - j] + C[i] < dp[i]:
                dp[i] = dp[i - j] + C[i]
                jump[i] = i - j
    i = N - 1
    print(str(i) + " -> ", end='')
    while i > 0:
        i = jump[i]
        print(str(i) + " -> ", end='')
    return dp[N - 1]
```

Time and space complexity

Time and space complexity analysis

Recursive solution

If we draw recursion tree

Every node has upto X children.

Total height of the tree is N

Number of nodes in the tree is X^N

Time complexity is $O(X^N)$, Exponential

We do not use any extra memory, space complexity is O(1)

Dynamic programming

We use two for loops

$$\sum_{i=0}^{N} \sum_{j=1}^{MIN(X,i)} 1 = \sum_{i=0}^{N} X = \text{X+X+X....X} = \text{NX}$$

Time complexity is O(NX)

Space complexity is O(N)

2. Word break problem. Given a string S containing only lowercase letters and no space, and a dictionary W which contains set of words made of only lowercase letters, compute how many ways the given sentence can be broken into words which are all valid words from the dictionary.

S="pineapplepenapple"

D = ["apple", "pen", "applepen", "pine", “pineapple"]

Output = 3,

S can be broken as

“pine” “apple” “pen” “apple”

“pine” “applepen" “apple”

“pineapple” “pen” “apple”

Hint : Start from the last and figure out how to break the problem into subproblems

1. State

1. State

Parameters

i - index of the last character inside S, the substring that we are considering is from 0 to i

Cost function

countWays(i,S,D) - Number of ways the substring of string S ending at index i can be broken into valid words.

2. Transitions

2. Transitions

Base case

i = -1 , return 1 , “” - Empty string , this means that all the words have been sucessfully broken and we are left with only empty string.

Choices

countWays(i,S,D)

Start from last. The given input is from 0 to i.

Word break

S = “pineapplepenapple”
D = ["apple", "pen", "applepen", "pine", “pineapple"]

0 i=11
pineapplepen

j = 11, i=11 , S[j:i] = n

Word break

S = “pineapplepenapple”
D = ["apple", "pen", "applepen", "pine", “pineapple"]

0 i=11
pineapplepen j = 10, i=11 , S[j:i] = en

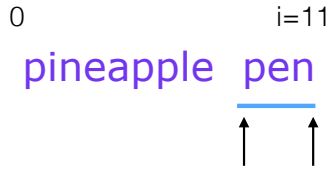
Word break

S = “pineapplepenapple”
D = ["apple", "pen", "applepen", "pine", “pineapple"]

j =9, i=11 , S[j:i] = pen

Word break

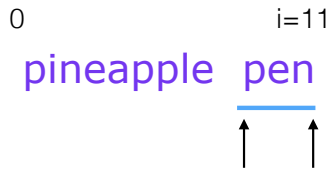
S = “pineapplepenapple”
D = ["apple", "pen", "applepen", "pine", “pineapple"]



j =9, i=11 , S[j:i] = pen

Word break

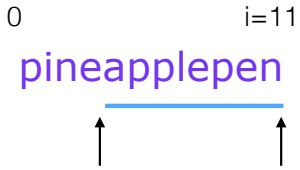
S = “pineapplepenapple”
D = ["apple", "pen", "applepen", "pine", “pineapple"]



j =9, i=11 , S[j:i] = pen

Word break

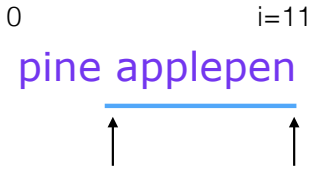
S = “pineapplepenapple”
D = ["apple", "pen", "applepen", "pine", “pineapple"]



j =4, i=11 , S[j:i] = pen

Word break

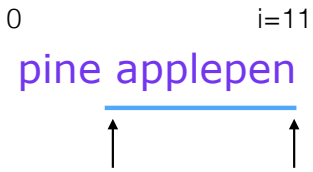
S = “pineapplepenapple”
D = ["apple", "pen", "applepen", "pine", “pineapple"]



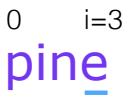
j =4, i=11 , S[j:i] = pen

Word break

S = “pineapplepenapple”
D = ["apple", "pen", "applepen", "pine", “pineapple"]



j =4, i=11 , S[j:i] = pen



2. Transitions

Base case

i = -1 , Return 1 , (Empty string) this means that all the words have been sucessfully broken and we are left with only empty string.

Choices

countWays(i,S,D)

Start from last. The given input is from 0 to i. How can we problem into subproblems.

countWays(11,S,D)

countWays(8,S,D)

countWays(3,S,D)

Optimal choice

We need to add all the solutions

countWays(i,S,D) = SUM(countWays(j-1,S,D)) for all j, S[j:i] is in D

3. Recursive solution

3. Recursive solution

Pseudo code

```
countWays(i,S,D)
    if i == -1
        return 1
    count = 0
    for j=i;j>=0;j—
        if S[j:i] in D
            count=count+countWays(j-1,S,D)
    return count
```

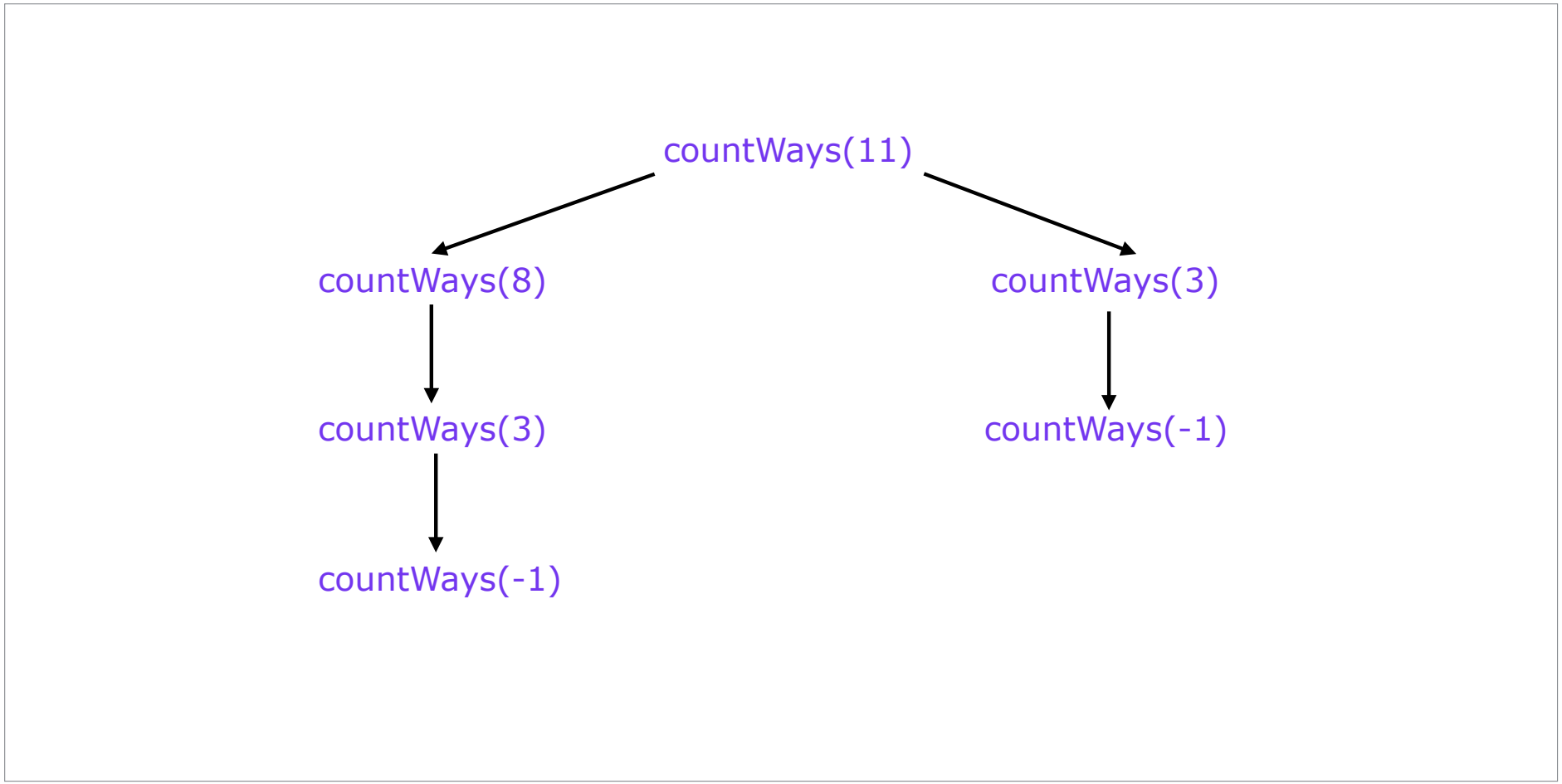
Java

```java
public static int countWays(String S, Set<String> dictionary){
    return countWays(S.length()-1,S,dictionary);
}
public static int countWays(int i,String S,Set<String> dictionary){
    if(i == -1){
        return 1;
    }
    int count = 0;
    for(int j=i;j>=0;j--){
        if(dictionary.contains(S.substring(j,i+1))){
            count+=countWays(j-1,S,dictionary);
        }
    }
    return count;
}
```

Python

```python
def count_ways(i, S, D):
    if i == -1:
        return 1
    count = 0
    for j in range(i, -1, -1):
        if S[j:i+1] in D:
            count += count_ways(j - 1, S, D)
    return count
```

Word break

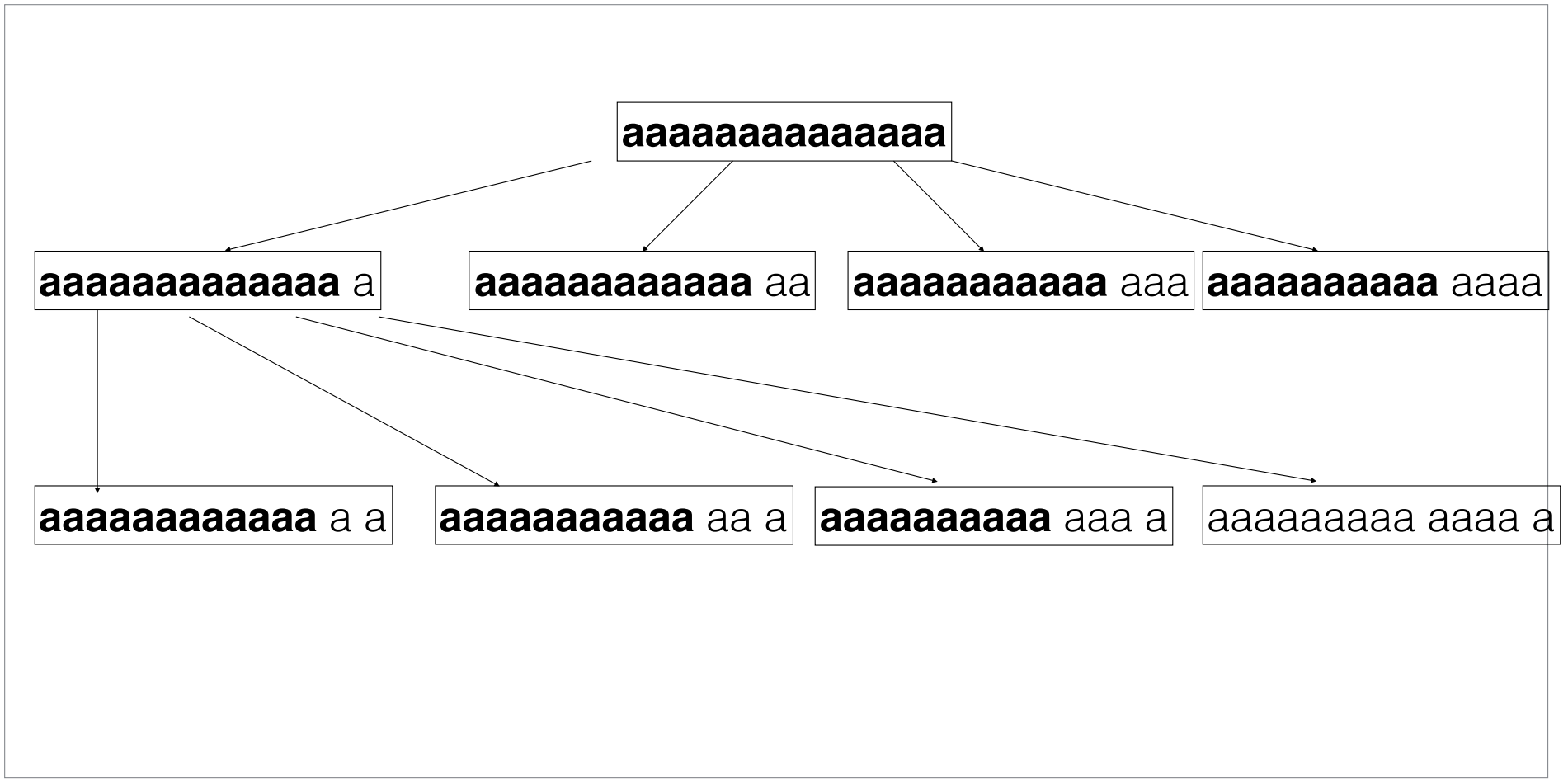

countWays(11)
countWays(8)
countWays(3)
countWays(3)
countWays(-1)
countWays(-1)

Word break

S = “aaaaaaaaaaaaaa”

D ={“a” , “aa” , “aaa” , “aaaa”}

Word break


aaaaaaaaaaaaaa
aaaaaaaaaaaaa a
aaaaaaaaaaaaa aa
aaaaaaaaaaaaa aaa
aaaaaaaaaa aaaa
aaaaaaaaaaaaa a a
aaaaaaaaaaaa aa a
aaaaaaaaaa aaa a
aaaaaaaaa aaaa a

4. Memoize

4. Memoize

Cache the results

We can use an array to cache the result.

Key -> i , Ending index of the string

Value -> Number of ways string 0…i can be broken into valid words.

Default value ? -1

Java

```java
public static int countWaysMemo(int i, String S, Set<String> dictionary, int[] cache) {
    if (i == -1) {
        return 1;
    }
    if (cache[i] != -1) {
        return cache[i];
    }
    int count = 0;
    for (int j = i; j >= 0; j--) {
        if (dictionary.contains(S.substring(j, i + 1))) {
            count += countWaysMemo(j - 1, S, dictionary, cache);
        }
    }
    cache[i] = count;
    return count;
}
```

Python

```python
def count_ways_memo(i, S, D, cache):
    if i == -1:
        return 1
    if cache[i] != -1:
        return cache[i]
    count = 0
    for j in range(i, -1, -1):
        if S[j:i + 1] in D:
            count += count_ways_memo(j - 1, S, D,cache)
    cache[i] = count
    return count
```

5. Bottom up approach

5. Bottom up approach

Flip it upside down and get rid of recursion.

countWays(i,S,D) = SUM(countWays(j-1,S,D)) for all j, S[j:i] is in D

dp[i] = SUM(dp[j-1]) for all j=i,i-1,…0, S[j:i] is in D

dp[0] - Is always 1, it represents empty string.

dp[i] - Stores the result for the string 0…i-1

Java

```java
public static int countWaysDP(String S,Set<String> dictionary){
    int N = S.length();
    int[] dp = new int[N+1];
    dp[0] = 1;
    for(int i=1;i<=N;i++){
        for(int j=i;j>=1;j--){
            if(dictionary.contains(S.substring(j-1,i))){
                dp[i]+=dp[j-1];
            }
        }
    }
    return dp[N];
}
```

Python

```python
def count_ways_dp(i, S, D):
    N = len(S)
    dp = [0 for _ in range(0, N + 1)]
    dp[0] = 1
    for i in range(1, N + 1):
        for j in range(i, 0, -1):
            if S[j-1:i] in D:
                dp[i] += dp[j-1]
    return dp[N]
```

Time and space complexity

Time and space complexity

Recursive implementation

Recurrence equation

$$T(n) = 1 + \sum_{i=0}^{n-1} T(i) = 2^N$$

$$T(0) = 1$$

Time complexity is $O(2^N)$, Exponential

Space complexity is O(1)

Time and space complexity

Dynamic programming implementations

There are two for loops

Outer for loop goes from i=0,1,….N

Inner loop goes from j=i,i-1,i-2,…0

Time complexity , $O(N^2)$

Space complexity, we use an array of length N , O(N)