<https://leetcode.com/problems/jump-game-vii/description/>

You are given a **0-indexed** binary string s and two integers minJump and maxJump. In the beginning, you are standing at index 0, which is equal to '0'. You can move from index i to index j if the following conditions are fulfilled:

* i + minJump <= j <= min(i + maxJump, s.length - 1), and
* s[j] == '0'.

Return true *if you can reach index* s.length - 1 *in* s*, or* false *otherwise.*

**Example 1:**

Input: s = "011010", minJump = 2, maxJump = 3

Output: true

Explanation:

In the first step, move from index 0 to index 3.

In the second step, move from index 3 to index 5.

**Example 2:**

Input: s = "01101110", minJump = 2, maxJump = 3

Output: false

**Constraints:**

* 2 <= s.length <= 105
* s[i] is either '0' or '1'.
* s[0] == '0'
* 1 <= minJump <= maxJump < s.length

**Attempt 1: 2023-01-23**

**Solution 1:  Native DFS (30 min, TLE)**

class Solution {

public boolean canReach(String s, int minJump, int maxJump) {

return helper(s, minJump, maxJump, 0);

}

private boolean helper(String s, int minJump, int maxJump, int index) {

if(index == s.length() - 1) {

return true;

}

if(index > s.length() - 1) {

return false;

}

// Don't for loop from 'index' to last position, because you may

// able to choose 'index' doesn't mean you can choose 'index + 1',

// since we may have "s.charAt(index) == '0'" but "s.charAt(i + 1) != '0'",

// the actual process is you can only jump from 'index' to next

// potential position as 'index + i'(i between 'minJump' to 'maxJump'),

// so only recursive on 'index + i' deeper levels

// for(int i = index; i < s.length(); i++) {

// for(int j = minJump; j <= maxJump; j++) {

// if(i + j < s.length() && s.charAt(i + j) == '0'

// && helper(s, minJump, maxJump, i + j)) {

// return true;

// }

// }

// }

for(int i = minJump; i <= maxJump; i++) {

// Don't forget mandatory condition as char at 'index + i' is '0',

// otherwise cannot jump to that position

if (index + i < s.length() && s.charAt(index + i) == '0'

&& helper(s, minJump, maxJump, index + i)) {

return true;

}

}

return false;

}

}

**Solution 2:  Top Down DP (Memoization) (10 min, still TLE)**

class Solution {

public boolean canReach(String s, int minJump, int maxJump) {

Boolean[] memo = new Boolean[s.length()];

return helper(s, minJump, maxJump, 0, memo);

}

private boolean helper(String s, int minJump, int maxJump, int index, Boolean[] memo) {

if(memo[index] != null) {

return memo[index];

}

if(index == s.length() - 1) {

return true;

}

if(index > s.length() - 1) {

return false;

}

for(int i = minJump; i <= maxJump; i++) {

if (index + i < s.length() && s.charAt(index + i) == '0'

&& helper(s, minJump, maxJump, index + i, memo)) {

return memo[index] = true;

}

}

return memo[index] = false;

}

}

**Solution 3:  Bottom Up DP (120 min,  traverse backward, also instead of intuitive initialize as boolean[] dp, we have to use int[] dp instead)**

**Why we have to use int[] dp instead of boolean[] dp ?**

**Because before the final step dp[n - 1], the second last step may reachable by different ways, let's say one example, if we have 2 ways to reach second last step dp[i] based on dp[i - 2],  dp[i - 3], if dp[i - 2] is 'true', then dp[i] will now mark as 'true', but if dp[i - 3] is 'false', then dp[i] will mark as 'false', which override the previous 'true' get from dp[i - 2].**

Test case: s="011010", minJump=2, maxJump=3

class Solution {

public boolean canReach(String s, int minJump, int maxJump) {

int n = s.length();

boolean[] dp = new boolean[n];

dp[0] = true;

System.out.println(s);

System.out.println("===========================");

for(int i = 1; i < n; i++) {

for(int j = minJump; j <= maxJump; j++) {

System.out.println("i=" + i + ", j=" + j);

if(i - j >= 0 && s.charAt(i) == '0') {

System.out.println("Assign dp[i - j]=" + dp[i - j] + " to dp[i]");

dp[i] = dp[i - j];

}

}

System.out.println("---------------------------");

}

return dp[n - 1];

}

public static void main(String[] args) {

Solution s = new Solution();

boolean result = s.canReach("011010", 2, 3);

System.out.println(result);

}

}

=================================================================

011010

===========================

i=1, j=2

i=1, j=3

---------------------------

i=2, j=2

i=2, j=3

---------------------------

i=3, j=2

Assign dp[i - j]=false to dp[i]

i=3, j=3

Assign dp[i - j]=true to dp[i]

---------------------------

i=4, j=2

i=4, j=3

---------------------------

i=5, j=2

Assign dp[i - j]=true to dp[i]

i=5, j=3

Assign dp[i - j]=false to dp[i] --> the wrong override from true to false

---------------------------

false -> expected true

Process finished with exit code 0

**The most intuitive idea to resolve conflict is using HOW to hold the 'true' and avoid 'false' override for a dp[i] if happened once ?**

**Using 'OR' logical connective ? No, still wrong, because if we use 'true' to OR a value, it will always have a chance to wrongly override a dp[i] from 'false' to 'true'**

Test case: s="01101110", minJump=2, maxJump=3

class Solution {

public boolean canReach(String s, int minJump, int maxJump) {

int n = s.length();

boolean[] dp = new boolean[n];

dp[0] = true;

System.out.println(s);

System.out.println("===========================");

for(int i = 1; i < n; i++) {

for(int j = minJump; j <= maxJump; j++) {

System.out.println("i=" + i + ", j=" + j);

if(i - j >= 0 && s.charAt(i) == '0') {

System.out.println("Assign (true || dp[i - j])=" + dp[i - j] + " to dp[i]");

dp[i] = (true || dp[i - j]);

}

}

System.out.println("---------------------------");

}

return dp[n - 1];

}

public static void main(String[] args) {

Solution s = new Solution();

boolean result = s.canReach("01101110", 2, 3);

System.out.println(result);

}

}

=================================================================

01101110

===========================

i=1, j=2

i=1, j=3

---------------------------

i=2, j=2

i=2, j=3

---------------------------

i=3, j=2

Assign dp[i - j]=false ==> (true || dp[i - j])=true to dp[i]

i=3, j=3

Assign dp[i - j]=true ==> (true || dp[i - j])=true to dp[i]

---------------------------

i=4, j=2

i=4, j=3

---------------------------

i=5, j=2

i=5, j=3

---------------------------

i=6, j=2

i=6, j=3

---------------------------

i=7, j=2

Assign dp[i - j]=false ==> (true || dp[i - j])=true to dp[i]

i=7, j=3

Assign dp[i - j]=false ==> (true || dp[i - j])=true to dp[i] -> wrongly override dp[i - j] from false to true consider both j=2 and j=3 both should be false, no chance to reach the last position, let's say even if one of them dp[i - j]=true, then even if a wrongly override won't impact result, but here both are false, override to true means it able to reach the last position, we got a wrong result

---------------------------

true -> expected false

Process finished with exit code 0

**The actual solution comes from change boolean[] dp to int[] dp, if the count of dp[i] > 0, then we will successfully record / hold dp[i] became 'true' at least once**

**Why must dp[i] += dp[i - j] ?**

**A new wrong solution with dp[i] = (dp[i - j] > 0 ? 1 : 0), because only accumulated count on dp[i] is the correct way,  just flip between 0 and 1 of previous step will not help on determine if the {true -> false -> false -> true -> false -> false} => finally is false, but actually should be true, since if maxRange = 3 it will allow index = 3 true able to carry on to the last position also as true**

Test case: s="011010", minJump=2, maxJump=3

class Solution {

public boolean canReach1(String s, int minJump, int maxJump) {

int n = s.length();

System.out.println(s);

System.out.println("===========================");

int[] dp = new int[n];

dp[0] = 1;

for(int i = 1; i < n; i++) {

if(s.charAt(i) == '0') {

for(int j = minJump; j <= maxJump; j++) {

System.out.println("i=" + i + ", j=" + j);

if(i - j >= 0) {

if(dp[i - j] > 0) {

System.out.println("dp[i-j]>0 ==> dp[i]=1");

} else {

System.out.println("dp[i-j]<=0 ==> dp[i]=0");

}

dp[i] = (dp[i - j] > 0 ? 1 : 0);

}

}

} else {

System.out.println("s.charAt(i) == '1' skip");

}

System.out.println("---------------------------");

}

System.out.println("dp[n-1]=" + dp[n - 1]);

return dp[n - 1] > 0;

}

public static void main(String[] args) {

Solution s = new Solution();

boolean result = s.canReach1("011010", 2, 3);

System.out.println(result);

}

}

=============================================================

011010

===========================

s.charAt(i) == '1' skip

---------------------------

s.charAt(i) == '1' skip

---------------------------

i=3, j=2

dp[i-j]<=0 ==> dp[i]=0

i=3, j=3

dp[i-j]>0 ==> dp[i]=1 -> flip dp[3] from 0 to 1

---------------------------

s.charAt(i) == '1' skip

---------------------------

i=5, j=2

dp[i-j]>0 ==> dp[i]=1 -> should stop as dp[5]=1 here

i=5, j=3

dp[i-j]<=0 ==> dp[i]=0 -> wrongly flip dp[5] from 1 to 0

---------------------------

dp[n-1]=0

false

Process finished with exit code 0

**The correct solution is accumulate dp[i] count based on all its previous indexes, which means once dp[i] turn to 1 by any candidate solution, it will be always > 0, then we will successfully record / hold dp[i] became 'true' at least once, but still TLE**

Test case: s="011010", minJump=2, maxJump=3

class Solution {

public boolean canReach(String s, int minJump, int maxJump) {

int n = s.length();

System.out.println(s);

System.out.println("===========================");

int[] dp = new int[n];

dp[0] = 1;

for(int i = 1; i < n; i++) {

if(s.charAt(i) == '0') {

for(int j = minJump; j <= maxJump; j++) {

System.out.println("i=" + i + ", j=" + j);

if(i - j >= 0) {

System.out.println("dp[i]=(dp[i]:" + dp[i] + "+dp[i - j]:" + dp[i-j] + ")=" + (dp[i] + dp[i - j]));

//dp[i] = (dp[i - j] > 0 ? 1 : 0);

dp[i] += dp[i - j];

}

}

} else {

System.out.println("s.charAt(i) == '1' skip");

}

System.out.println("---------------------------");

}

System.out.println("dp[n-1]=" + dp[n - 1]);

return dp[n - 1] > 0;

}

public static void main(String[] args) {

Solution s = new Solution();

boolean result = s.canReach("011010", 2, 3);

System.out.println(result);

}

}

=============================================================

011010

===========================

s.charAt(i) == '1' skip

---------------------------

s.charAt(i) == '1' skip

---------------------------

i=3, j=2

dp[i]=(dp[i]:0+dp[i - j]:0)=0

i=3, j=3

dp[i]=(dp[i]:0+dp[i - j]:1)=1

---------------------------

s.charAt(i) == '1' skip

---------------------------

i=5, j=2

dp[i]=(dp[i]:0+dp[i - j]:1)=1 -> turn on dp[5] as true

i=5, j=3

dp[i]=(dp[i]:1+dp[i - j]:0)=1 -> record/hold dp[5] as true, not turn off even dp[i-j]=0 again

---------------------------

dp[n-1]=1

true

Process finished with exit code 0

**Refer to**

<https://leetcode.com/problems/jump-game-vii/solutions/1230326/java-dp-solutions-bottom-up-backward-forward-and-top-down/>

**(Backward) Quadratic DP (TLE)**

We can start with a "brute force", quadratic time DP, of the standard type: **"counting the number of ways"** in which we can get to the final index. For each index *i* we check if it contains a *0*, if so we look back to the window of indexes *[j, j2]* that a jump to *i* could have originated from, and sum up the values in those indexes. This is short and easy, works for small test cases, but unsurprisingly it TLEs for larger ones.

public boolean canReach(String s, int minJump, int maxJump) {

int n= s.length(), dp[]= new int[n];

dp[0]= 1;

for(int i=1; i<n; i++){

if(s.charAt(i)=='1') continue;

for(int j=Math.max(0, i-maxJump), j2=i-minJump; j<=j2; j++)

dp[i]+= dp[j];

}

return dp[n-1]>0;

}

**To promote the execution time, we have to streamline the logic for turn on dp[i], ff we find previous step has at least once turn on as 1, we can permanently set current index also to 1, then break out to check the next index**

class Solution {

public boolean canReach(String s, int minJump, int maxJump) {

int n = s.length();

int[] dp = new int[n];

dp[0] = 1;

for(int i = 1; i < n; i++) {

if(s.charAt(i) == '0') {

for(int j = minJump; j <= maxJump; j++) {

if(i - j >= 0) {

//dp[i] += dp[i - j];

// If we find previous step has at least once turn on as 1

// we can permanently set current index also to 1, then

// break out to check the next index

if(dp[i - j] == 1) {

dp[i] = 1;

break;

}

}

}

}

}

return dp[n - 1] > 0;

}

}

**Refer to**

<https://leetcode.com/problems/jump-game-vii/solutions/1230326/java-dp-solutions-bottom-up-backward-forward-and-top-down/>

**(Backward) Quadratic DP (2387ms, 5%, Accepted) marking reachable positions**

To avoid the huge numbers problem, rather than representing "ways" to reach the i-th index (which is unnecessary, as we're solving a simpler problem here) we take a step back to the quadratic DP but use the dp[] array just to represent whether we can reach the index (1) or not (0), a **"yes/no decision"** DP problem. This actually gets accepted by Leetcode already (although it's very slow)

public boolean canReach(String s, int minJump, int maxJump) {

int n= s.length(), dp[]= new int[n];

dp[0]= 1;

for(int i=1; i<n; i++){

if(s.charAt(i)=='1') continue;

for(int j=Math.max(0, i-maxJump), j2=i-minJump; j<=j2; j++)

if(dp[j]==1){ dp[i]= 1; break; } // this line is the only difference vs. Approach 1

}

return dp[n-1]>0;

}

**Solution 4:  Bottom Up DP + Sliding Window (120 min)**

**Style 1: int[] dp**

class Solution {

public boolean canReach(String s, int minJump, int maxJump) {

int n = s.length();

int[] dp = new int[n];

dp[0] = 1;

// pre means the number of previous position that we can jump from.

int pre = 0;

for(int i = 1; i < n; i++) {

if(i - minJump >= 0) {

pre += dp[i - minJump];

}

if(i - maxJump - 1 >= 0) {

pre -= dp[i - maxJump - 1];

}

if(s.charAt(i) == '0' && pre > 0) {

dp[i] = 1;

}

}

return dp[n - 1] > 0;

}

}

Time Complexity : O(N)

Space Complexity : O(N)

**Refer to**

<https://leetcode.com/problems/jump-game-vii/solutions/1230326/java-dp-solutions-bottom-up-backward-forward-and-top-down/>

**(Backward) Linear DP (8ms, 92%, Accepted) marking reachable positions with Sliding Window**

We could stop here as Approach 4. is a perfectly acceptable solution with O(N) time and O(N) space. What we're doing next is just improving the "constant" in front of the linear space complexity.

We may notice that the partial sum difference calculated in Approach 4 is essentially a sum of the reachable indexes within a constant-size sliding window; we can save some memory by removing the ps[] array and replace it with a single number.

public boolean canReach(String s, int minJump, int maxJump) {

int n= s.length(), dp[]= new int[n], psDiff= 0;

dp[0]= 1;

for(int i=1; i<n; i++){

int j= i-maxJump-1, j2= i-minJump;

if(j2>-1) psDiff+= dp[j2];

if(j>-1) psDiff-= dp[j];

if(s.charAt(i)=='0')

dp[i]= psDiff>0 ? 1 : 0;

}

return dp[n-1]>0;

}

**Style 2: boolean[] dp**

class Solution {

public boolean canReach(String s, int minJump, int maxJump) {

int n = s.length();

boolean[] dp = new boolean[n];

dp[0] = true;

// pre means the number of previous position that we can jump from.

int pre = 0;

for(int i = 1; i < n; i++) {

if(i - minJump >= 0 && dp[i - minJump]) {

pre++;

}

if(i - maxJump - 1 >= 0 && dp[i - maxJump - 1]) {

pre--;

}

if(s.charAt(i) == '0' && pre > 0) {

dp[i] = true;

}

}

return dp[n - 1];

}

}

Time Complexity : O(k\*N)

Space Complexity : O(N)

**Refer to**

<https://leetcode.com/problems/jump-game-vii/solutions/1224804/java-c-python-one-pass-dp/>

<https://leetcode.com/problems/jump-game-vii/solutions/1224804/java-c-python-one-pass-dp/comments/947586>

# **Explanation**

dp[i] = true if we can reach s[i].pre means the number of previous position that we can jump from.

Here comes the explanation!

1. It's a bottom-up DP implementation. The boolean value represents whether this position is reachable from start. So the first step is to generate the table. Here the table was pre-labeled True or False, thus '1's are already labeled False.
2. **To determine the state of dp[i], one need to check the states in window dp[i-maxJ : i-minJ], because any one of them can reach i if it's labeled True.**
3. **Then you need to check if there is a True in this window. Notice that this is a sliding window problem, so you don't need to calculate it everytime. You only need to remove the effect from dp[i-maxJ-1] and add the dp[i-minJ]. This is done by these two lines of code pre += dp[i - minJ] and pre -= dp[i - maxJ - 1]**
4. The if statements if i >= minJ: and if i > maxJ: are dealing with the initial boundary.

The brilliance of this algorithm is combining the sliding window to DP

public boolean canReach(String s, int minJ, int maxJ) {

int n = s.length(), pre = 0;

boolean[] dp = new boolean[n];

dp[0] = true;

for (int i = 1; i < n; ++i) {

if (i >= minJ && dp[i - minJ])

pre++;

if (i > maxJ && dp[i - maxJ - 1])

pre--;

dp[i] = pre > 0 && s.charAt(i) == '0';

}

return dp[n - 1];

}

# **Complexity**

Time O(n)Space O(n)