1. Solution:

We want the value of overall chance of the failure, $1 - \prod_{i=1}^{n}(1 - (1 - r_i)^{b_i})$, to be as small as possible. Which is equivalent to make $\prod_{i=1}^{n}(1 - (1 - r_i)^{b_i})$ as large as possible. So, this question is to find the optimal combination of the number of backup subsystems for every subsystem, and make their cost is less or equal to B. Let $dp[0..B]$ is an array, $dp[i]$ denote that the maximal value of $\prod_{i=1}^{n}(1 - (1 - r_i)^{b_i})$ when our budget is i.

Then we can get the following recurrence formula

(we go through n subsystems and select one subsystem from the n subsystems, say k, with cost c_k, to maximize dp[i], which also means, minimize the failure rate):

$dp[i] = (dp\left[i - c_k\right]\frac{1-(1-r_k)^{b_k+1}}{1-(1-r_k)^{b_k}})$ When $i - c_k \geq 0$

In order to compute this value, we need to store the selection method of every budget i. That is, for every $dp[i]$, we will have a table $T_i[1..n]$ to store the number of every subsystem. That's where we can get our b_k on the formula above ($b_k = T_i[k]$). Obviously, for $i = 0$, $T_0[1..n] = [0, 0, ..., 0]$ and $dp[0] = \prod_{i=1}^{n}(1 - (1 - r_i)^{0}) = 0$. So, we have initialized our base problem. We can then compute $dp[1]$, $dp[2]$, …, $dp[B]$. Of course, we must also keep track of the value of $T_i[1..n]$. After we compute the $dp[B]$. The $T_B[1..n]$ will record the number of every subsystem that will get a maximal value of $\prod_{i=1}^{n}(1 - (1 - r_i)^{b_i})$ (the probability of success). Total running time is $O(Bn)$, because for every i, we need $O(n)$ time to compute the optimal combination and the optimal value $dp[i]$.

2. Solution:

```
Alg:uniquePaths(self, m: int, n: int) -> int:
    d = [[1...1]...[1...1]] # n * m
    for col->1 to m:
        for row->1 to n:
            d[col][row] = d[col - 1][row] + d[col][row - 1]

    return d[m - 1][n - 1]
```

3. Solution:

Greedy Algorithm can be implemented to solve this problem:

1.Sort the items by the weight (increasing)
2.Take one item each time.

Proof:
It can be proven that the algorithm satisfies the greedy-choice property.

Greedy-choice property:
Suppose that exists an optimal solution that the item j is taken and the item i is not taken, while we have Wj > Wi abd Vi < Vj. Then, we can take item j out of the knapsack and put item i in the knapsack to get a higher value solution. That is contradiction to the original solution being optimal.

Optimal substructure property:
If item j is removed from an optimal packing, the remaining packing is an optimal packing with weight at most W-wj that can be taken from the n-1 items other than j.

A solution not using greedy can also get full marks if it is right. However, the proof is necessary.

4.
Many kinds of solutions. Here some examples:
Dp:

```java
enum Index {
    GOOD, BAD, UNKNOWN
}

public class Solution {
    Index[] memo;

    public boolean canJumpFromPosition(int position, int[] nums) {
        if (memo[position] != Index.UNKNOWN) {
            return memo[position] == Index.GOOD ? true : false;
        }

        int furthestJump = Math.min(position + nums[position], nums.length - 1);
        for (int nextPosition = position + 1; nextPosition <= furthestJump; nextPosition++) {
            if (canJumpFromPosition(nextPosition, nums)) {
                memo[position] = Index.GOOD;
                return true;
            }
        }

        memo[position] = Index.BAD;
        return false;
    }

    public boolean canJump(int[] nums) {
        memo = new Index[nums.length];
        for (int i = 0; i < memo.length; i++) {
            memo[i] = Index.UNKNOWN;
        }
        memo[memo.length - 1] = Index.GOOD;
        return canJumpFromPosition(0, nums);
    }
}
```

Dp-bottom-up

```java
enum Index {
    GOOD, BAD, UNKNOWN
}

public class Solution {
    public boolean canJump(int[] nums) {
        Index[] memo = new Index[nums.length];
        for (int i = 0; i < memo.length; i++) {
            memo[i] = Index.UNKNOWN;
        }
        memo[memo.length - 1] = Index.GOOD;

        for (int i = nums.length - 2; i >= 0; i--) {
            int furthestJump = Math.min(i + nums[i], nums.length - 1);
            for (int j = i + 1; j <= furthestJump; j++) {
                if (memo[j] == Index.GOOD) {
                    memo[i] = Index.GOOD;
                    break;
                }
            }
        }

        return memo[0] == Index.GOOD;
    }
}
```

Greedy:

```java
public class Solution {
    public boolean canJump(int[] nums) {
        int lastPos = nums.length - 1;
        for (int i = nums.length - 1; i >= 0; i--) {
            if (i + nums[i] >= lastPos) {
                lastPos = i;
            }
        }
        return lastPos == 0;
    }
}
```