> 给定一个非负整数数组 nums ，你最初位于数组的 第一个下标 。
> 数组中的每个元素代表你在该位置可以跳跃的最大长度。
> 判断你是否能够到达最后一个下标。

分析： 假设，每次都跳最远的长度，判断最后是否能够跳出数组即可。

1. 从0开始，当nums[i] != 0 时，跳跃距离可以为1~nums[i],这样跳过最远距离为i+nums[i], 当遇到nums[i] == 0 时，如果已有的最大值小于等于当前的索引位置i则一定无法跳出。

```
class Solution {
public:
  bool canJump(std::vector<int>& nums) {
    int len = nums.size();
    if (1 >= len) {
      return true;
    }

    int fastest = 0;
    for (int i = 0; i < len; i++) {
      if (fastest < i) {  // 遇到0，无法向前
        return false;
      }
      fastest = std::max(fastest, i + nums[i]);
    }

    return true;
  }
};
```

> 给你一个非负整数数组 nums ，你最初位于数组的第一个位置。
> 数组中的每个元素代表你在该位置可以跳跃的最大长度。
> 你的目标是使用最少的跳跃次数到达数组的最后一个位置。
> 假设你总是可以到达数组的最后一个位置。

分析： 动态规划解法：

1. 状态转移方程是对nums[i]，计算从1~nums[i]递归的最小值。

```
class Solution {
public:
  int jump(std::vector<int>& nums) {
    int              len  = nums.size();
    std::vector<int> memo = std::vector<int>(len + 1, len);
    return dp(nums, len, 0, memo);
  }
```

```
private:
  int dp(std::vector<int>& nums, int len, int t, std::vector<int>& memo) {
    if (t >= len - 1) {
      return 0;
    }

    if (memo[t] != len) {
      return memo[t];
    }

    int steps = nums[t];
    for (int i = 1; i <= steps; i++) {
      int sub = dp(nums, len, t + i, memo);
      memo[t] = std::min(memo[t], sub + 1);
    }

    return memo[t];
  }
};
```

贪心解法：

1. 明确跳出数组，只是调到`nums.size() - 1`的位置；
2. 采用贪心算法，每次都调到最远处，此时需要标记最远处的索引位置，当走到标记的最远处索引位置，表示**已经跳跃一次**，需要更新跳跃到的最远处位置，同时跳跃次数增加一次。

```
class Solution {
public:
  int jump(std::vector<int>& nums) {
    int len = nums.size();
    int end = 0, fastest = 0;
    int jumps = 0;

    for (int i = 0; i < len - 1; i++) {
      fastest = std::max(fastest, i + nums[i]);
      if (end == i) {
        end = fastest;
        jumps++;
      }
    }

    return jumps;
  }
};
```