Bài 1: Two City Scheduling (medium)

https://leetcode.com/problems/two-city-scheduling/

Một công ty dự định phỏng vấn N người .Cho mảng H có N phần tử (2 <= N <= 100, N chẵn), mỗi phần tử gồm 2 số nguyên H[i] = [ACost, BCost] tương ứng với số tiền bỏ ra để đưa người thứ i đến thành phố A hoặc B. Yêu cầu tìm số tiền bỏ ra tối thiểu để đưa một nửa số người đến thành phố A và một nửa đến thành phố B.

Ví dụ: H = [[10,20],[30,200],[400,50],[30,20]]

Output: 110. Đưa người đầu tiên và thứ 2 đến địa điểm A, người thứ 3 và 4 đến địa điểm B sẽ cho chi phí thấp nhất.

Problem analysis using brute-force:

- How to generate candidate and how to check candidate: Mỗi người có thể đến thành phố A hoặc B, trong phương pháp brute-force tạo ra các khả năng chia người sao cho số người đến 2 thánh phố là bằng nhau, cuối cùng chọn ra cấu hình cho chi phí thấp nhất.
- Memory space, execution time and data structure:
    - Memory space: O(2^N)
    - Execution time: O(N)
    - Data structure: Array

Brute-force code:

```
vector<vector<int>> a{{259,770},{448,54},{926,667},{184,139},{840,118},{577,469}};
int ret = INT_MAX;
void dfs(int index, int x, int y, int temp) {
	if (index == a.size()) {
		ret = min(ret, temp);
	}
	if (x < a.size()/2) {
		dfs(index+1, x+1, y, temp+a[index][0]);
	}
	if (y < a.size()/2) {
		dfs(index+1, x, y+1, temp+a[index][1]);
	}
}

int main(){
```

```
        dfs(0, 0, 0, 0);
        cout<<ret<<endl;
        return 0;
}
```

Improvement analysis by using gready algorithm:

- Theoretical basis to apply greedy: Để ý thấy ta có thể sắp xếp chuỗi N tăng dần theo giá trị X[0] – X[1], trong đó X[0] là chi phí để người X đến thành phố A và X[1] là chi phí để người X đến thành phố B. Sau đó một nửa số người đầu dãy sẽ sắp xếp cho đến thành phố A, nửa còn lại đến thành phố B.

Chứng minh: Giả sử người X có chi phí đến A là X1, đến B là X2, và người Y có chi phí là Y1, Y2, Và X1 – X2 < Y1 – Y2

Ta có: X1 – X2 < Y1 – Y2 <=> X1 + Y2 < X2 + Y1. Điều này có nghĩa chi phí để người X đến A và người Y đến B sẽ nhỏ hơn chi phí người X đến B và người Y đến A

- Memory space, execution time and data structure.
    - Memory space: O(N)
    - Execution time: O(NlogN)
    - Data structure: Array
- Code

```
 int Solve(vector<vector<int>>& costs) {
        int size = costs.size();
        int ret = 0;
        vector<vector<int>> store(size, vector<int>(2,0));
        for (int i=0; i<size; ++i){
                store[i][0] = costs[i][0] - costs[i][1];
                store[i][1] = i;
         }
        sort(store.begin(), store.end());
        for (int i=0; i<size/2; ++i){
         ret+=costs[store[i][1]][0];
        }
        for (int i=size/2; i<size; ++i){
                ret+=costs[store[i][1]][1];
```

```
        }
        return ret;
    }
```

Bài 2: Remove Covered Intervals (easy)

https://leetcode.com/problems/remove-covered-intervals/

Cho một danh sách N các khoảng (a, b). Khoảng (a, b) bị bao bởi khoảng (c, d) khi c <= a và b <= d. Yêu cầu sau khi loại bỏ các khoảng bị bao, trả về số lượng các khoảng còn lại của danh sách ban đầu.

Ví dụ: A = [[1,4],[3,6],[2,8]].

Output: 2. Loại bỏ khoảng [3, 6] vì bị bao bởi khoảng [2, 8], khi đó số lượng khoảng còn lại là 2.

Giới hạn: 0 <= N <= 1000

Problem analysis using brute-force:

- How to generate candidate and how to check candidate: Sử dụng brute-force so sánh từng khoảng với tất cả các khoảng còn lại, kiểm tra xem khoảng đó có bị bao bởi khoảng khác hay không.
- Memory space, execution time and data structure:
    - Memory space: O(N)
    - Execution time: O(N^2)
    - Data structure: Array

Brute-force code:

```
int Solve(vector<vector<int>>& a) {
    vector<bool> check(a.size(), true);
    for (int i = 0; i<a.size(); ++i) {
        for (int j = 0; j<a.size(); ++j) {
            if (i == j) continue;
            if (a[j][0] <= a[i][0] && a[i][1] <= a[j][1]) {
                check[i] = false;
                break;
            }
```

```
        }
    }
    for (int i=0; i<a,size(); ++i) {
        if (check) ret++;
    }
    return ret;
}
```

Improvement analysis by using greedy algorithm:

- Theoretical basis to apply greedy:

Để kiểm tra xem một khoảng có bị bao bởi khoảng khác hay không, ta có thể sắp xếp mảng ban đầu theo phần tử bắt đầu của khoảng đó tăng dần, sau đó ta chỉ cần so sánh phần tử cuối của khoảng đó để biết khoảng đó có bị bao hay không.

- Memory space, execution time and data structure.
    - Memory space: O(N)
    - Execution time: O(NlogN)
    - Data structure: Array
- Code

```
static bool com(vector<int> &a, vector<int> &b){
    if (a[0] == b[0]) return a[1] > b[1];
    return a[0] < b[0];
}
int Solve(vector<vector<int>>& intervals) {
    int max = -1;
    int count = 0;
    //Sắp xếp mảng tăng dần theo giá trị bắt đầu của khoảng đó, nếu bằng nhau thì khoảng nào có phần tử kết thúc lớn hơn thì xếp lên trước.
    sort(intervals.begin(), intervals.end(), com);
    for (int i=0; i<intervals.size(); ++i){
        if (intervals[i][1] > max)
        {
            count++;
            max = intervals[i][1];
        }
```

```
        }
        return count;
    }
```

Bài 3: Assign Cookies (easy)

https://leetcode.com/problems/assign-cookies/

Chia bánh cho n đứa trẻ. Mỗi chiếc bánh có kích thước s[i]. Đứa trẻ j sẽ vui nếu được chia bánh có kích thước tối thiểu là g[j] (s[i] >= g[j]). Xác định tối đa bao nhiêu đứa trẻ sẽ vui.

Ví dụ: g = [1,2,3], s = [1,1]

Chỉ có thể chia bánh cho đứa trẻ thứ nhất vui, output là 1.

Điều kiện:

1 <= g.length <= 3 * 10^4

0 <= s.length <= 3 * 10^4

1 <= g[i], s[j] <= 2^31 – 1

Greedy:

Sắp xếp cả 2 mảng theo thứ tự tăng dần.

Duyệt đồng thời cả 2 mảng, ứng với mỗi đứa trẻ, tìm bánh nhỏ nhất mà có thể làm cho đứa trẻ vui. Nếu tìm được tăng biến đếm lên.

Dừng duyệt nếu ko còn đứa trẻ nào cần chia bánh (duyệt đến cuối mảng 1) hoặc chiếc bánh to nhất (duyệt đến cuối mảng 2) cũng không thỏa mãn làm đứa trẻ hiện tại vui.

Time complexity: O(NlogN)

Code:

```
int findContentChildren(vector<int>& g, vector<int>& s) {
    sort(g.begin(), g.end());
    sort(s.begin(), s.end());
    int i=0, j=0;
    int ans=0;
    while(i<g.size() && j<s.size()) {
        if(g[i] <= s[j]) {
            ++ans;
```

```
            ++i;
          }
          ++j;
        }
        return ans;
    }
```

Bài 4: Jump Game (medium)

https://leetcode.com/problems/jump-game/

mCho 1 mảng nums gồm N số nguyên không âm. Ban đầu bạn đứng ở phần tử đầu tiên. Từ mỗi phần tử, bạn có thể nhảy đến bất cứ phần tử nào sau nó miễn là độ dài bước nhảy không vượt quá giá trị số nguyên tại nơi bạn đang đứng.
Xác định xem liệu bạn có thể nhảy đến phần tử cuối cùng hay không.
Ví dụ: nums = [2,3,1,1,4]
Đứng ở phần tử đầu tiên, bạn có thể nhảy đến phần tử thứ 2 hoặc thứ 3.
Đứng ở phần tử thứ 2, bạn có thể nhảy đến phần tử thứ 3, 4 hoặc 5.

mGiới hạn:
1 <= N <= 10000
0 <= nums[i] <= 100000

Brute force approach

Starting with i = 0, if nums[i] > 0 then create a subarray containing all the elements from nums[i+1] to nums[i + nums[i]]. Recursively do this for each element in the resulting subarray until one of your subarray contains the last element or until you cannot create any more subarray

Worst-case time complexity: O(N!)
Explanation: Assume nums = [N-2, N-3, N-4, . . . , 2, 1, 0, 1]
First index is 0, last index is N-1.
From nums[0], there are N-2 possible way to jump to the next element, but we cannot reach nums[N-1].
So we have to check all N-2 elements after nums[0]
Repeat for all elements from nums[0] to nums[N-2], we found that there is no way to reach nums[N-1]
The number of path we have to check is (N-2)*(N-3)*. . .*2*1 = (N-2)!
So time complexity is O(N!)

Greedy approach:

- Define a value max which is the furthest index that all elements from 0 to max can jump to. Set max to 0 initially.
- For each nums[i], find the furthest element you can jump to which is i + nums[i]. If it is greater than max, then set it as max.
- Repeat until i == max or i == N-1
- Time complexity in worst case: O(N)
- Explanation:
    - For each element i, the furthest element you can jump to from i is nums[i+nums[i]]
    - Assume for all nums[k] with i <= k <= i + nums[i], k + nums[k] <= i + nums[i] then regardless of how we jump, we can never go further than i + nums[i]. We set max = i + nums[i]
    - If there exist k such that k + nums[k] > i + nums[i], then the furthest we can go now is k + nums[k] by choosing to jump from i to k and then to k + nums[k]. We update max = k + nums[k]
    - After we have checked all elements from nums[0] to nums[max], if max < N-1 then we can conclude it is not possible to reach the last element based on above logic. Otherwise it is possible.

Code:

```
class Solution
{
public:
   bool canJump(vector<int>& nums)
   {
                    bool retval = false;
      int N = nums.size();
      int maxIdx = 0; //Highest index that we can jump to. Set it to 0 initially
      for(int i = 0; i < N; i++)
      {
                              //If max is already equal to or greater than the last index, then return true
         if(maxIdx >= N - 1)
                              {
                                       retval = true;
            break;
                              }
```

```
                        //If i > max then we have already checked all elements from i to max.
                        //Return false because max is less than N - 1
        if(i > maxIdx)
                        {
                                retval = false;
          break;
                        }
                        //Update value of max
        maxIdx = (i + nums[i]) > maxIdx ? i + nums[i] : maxIdx;
      }
      return retval;
    }
};
```

Bài 5: Jump Game II (medium)

https://leetcode.com/problems/jump-game-ii/

Cho 1 mảng số không âm nums, bạn đang đứng ở đầu mảng. Mỗi phần tử của mảng biểu thị cho độ dài tối đa có thể nhảy được. Xác định số bước nhảy tối thiểu để có thể đến được cuối mảng. Giả sử rằng luôn luôn tồn tại cách nhảy để có thể đến được vị trí cuối cùng.

Ví dụ: nums = [2, 3, 1, 1, 4];

Số bước nhảy tối thiểu là 2. Bước nhảy đầu tiên nhảy từ phần tử số 1 sang phần tử số 2. Bước thứ 2 nhảy từ phần tử số 2 sang phần tử cuối cùng của mảng.

Điều kiện:

1 <= nums.length <= 1000

0 <= nums[i] <= 10^5

Brute-force:

Dùng back-tracking tìm tất cả các cách nhảy.

Nếu tìm được cách có tổng số bước nhảy nhỏ hơn, lưu lại số này

Time complexity: O(N!)

Code:

```
int jump(vector<int>& nums) {
    return minJump(nums, 0);
  }
  int minJump(vector<int>& nums, int start) {
    if(start == nums.size()-1)
      return 0;
    if(nums[start]==0)
      return -1;
    int minStep=1500;
    int farthestJump = min((int)nums.size()-1, start+nums[start]);
    for(int i=start+1; i<=farthestJump; ++i) {
      int tmp=minJump(nums, i);
      if(tmp != -1 && tmp+1 < minStep)
        minStep = tmp+1;
    }
    return minStep;
  }
```

Improvement by using greedy:

Để có thể tìm được cách có tổng số bước nhảy nhỏ nhất, cần phải nhảy xa nhất có thể ở mỗi bước nhảy.

Không phải lúc nào nhảy tối đa cũng sẽ cho kết quả tốt nhất. Ví dụ nums = [2, 3, 1, 1, 4], nếu luôn nhảy tối đa, cần 3 bước để đến được vị trí cuối mảng. Vị trí 1 -> 3 -> 4 -> 5.

Thay vì phải kiểm tra từng vị trí ở mỗi bước nhảy như brute-force, chỉ cần xác định vị trí xa nhất có thể nhảy được. Tại mỗi lần nhảy chỉ cần update lại giá trị này thay vì phải duyệt tất cả các vị trí có thể nhảy được.

Ví du: nums = [3, 5, 2, 4, 2, 1, 1, 2, 1]

Index:	1	2	3	4	5	6	7	8	9
Nums:	3	5	2	4	2	1	1	2	1
Steps:	0		1			2			3

Bắt đầu: Khởi tạo step=0, đang đứng ở vị trí đầu tiên

Lần 1: vị trí 1 nhảy xa nhất đến vị trí 1+3=4, đánh dấu vị trí xa nhất có thể nhảy được trong 1 bước là 4, farthestJump = 4. -> vị trí của lần nhảy tiếp theo là {2, 3, 4}. Tăng biến đếm step thêm 1 -> step=1

Lần 2: Từ vị trí {2, 3, 4}, xác định vị trí xa nhất

Vị trí 2: 2+nums[2] = 2+5 = 7

Vị trí 3: 3+nums[3] = 3+2 = 5

Vị trí 4: 4+nums[4] = 4+4 = 8

Vậy vị trí xa nhất có thể đạt được là vị trí 8, farthestJump=8. -> các vị trí có thể nhảy tới được trong lần nhảy kế tiếp: {5, 6, 7, 8}. Tăng step thêm 1 -> step=2

Note: Mặc dù từ vị trí 2 có thể nhảy đến vị trí 3, 4, ta không cho 3 và 4 vào tập vị trí cho lần nhảy kế tiếp do các vị trí này có thể nhảy tới được trong lần nhảy trước. Ta chỉ cần update những vị trí chưa được nhảy tới. Như vậy đảm bảo được tập các vị trí của lần nhảy thứ *ith* phải cần tối thiểu *i* lần nhảy.

Lần 3: Khoảng vị trí cần xét cho lần nhảy tiếp {5, 6, 7, 8}

Vị trí 5: 5+nums[5] = 5+2 = 7

Vị trí 6: 6+nums[6] = 6+1 = 7

Vị trí 7: 7+nums[7] = 7+1 = 8

Vị trí 8: 8+nums[8] = 8+2 = 10

Vậy vị trí xa nhất có thể đạt được trong lần nhảy này là {9, 10} và cũng là qua điểm cuối cùng. Tăng biến đếm step thêm 1 và trả về giá trị này -> step=3

Code:

```
int jump(vector<int>& nums) {
    int step=0;
// biến l, r dùng để lưu khoảng vị trí của mỗi lần nhảy
// sau mỗi lần duyệt khoảng vị trí từ l đến r, tăng biến step thêm 1
    for(int l=0,r=0; r<nums.size()-1; ++step) {
      int farthestJump=0;
      for(int i=l; i<=r; ++i)
        farthestJump = max(farthestJump, i+nums[i]);
// sau khi xác định đc farthestJump, update lại tập vị trí cho lần nhảy kế tiêp
```

```
// gán vị trí tối thiểu l=r+1 để chắc chắn vị trí cho lần nhảy kế tiếp là chưa được duyệt
// update vị trí tối đa r=farthestJump
            l=r+1;
            r=farthestJump;
        }
        return step;
    }
```

Time complexity: O(N)