中途相遇/折半搜索 Meet in the Middle

中途相遇是一种将搜索空间分成大致相等的两部分的技巧。对这两部分分别进行独立的搜索，最后将搜索结果合并。

如果存在一种高效的方式来合并搜索结果，就可以使用这种技巧。在这种情况下，两次搜索可能比一次大规模搜索更省时间。通常，利用中间相遇技巧，我们可以将复杂度从 2^n 降低到 $2^{n/2}$。因此，中途相遇也称为**折半搜索**。

例如，考虑一个问题：给定一个包含 n 个数字的列表和一个数字 x，我们需要判断是否能从列表中选出若干数，使它们的和为 x。例如，给定列表 $[2,4,5,9]$ 和 $x=15$，我们可以选择数字 $[2,4,9]$ 来得到 $2+4+9=15$。然而，如果对于同一个列表 $x=10$，则无法得到这样的和。

一个简单的算法是遍历所有可能的子集，并检查是否有子集的和等于 x。这种算法的时间复杂度为 $O(2^n)$，因为共有 2^n 个子集。然而，利用中间相遇技巧，我们可以实现更高效的 $O(2^{n/2})$ 时间算法。需要注意的是，$O(2^n)$ 和 $O(2^{n/2})$ 是不同的复杂度，因为 $2^{n/2}$ 等于 $\sqrt{2^n}$。

其核心思想是将列表分成两个子列表 A 和 B，使它们各包含大约一半的数字。 第一次搜索生成 A 的所有子集，并将其和存入列表 S_A；相应地，第二次搜索从 B 生成列表 S_B。之后，只需检查是否能从 S_A 和 S_B 中分别选取一个元素，使它们的和等于 x。**这等价于判断是否能用原列表中的数字组合出 x。**

例如，假设列表为 $[2,4,5,9]$，且 $x=15$。首先，我们将列表分成 $A=[2,4]$ 和 $B=[5,9]$。然后，生成列表 $S_A=[0,2,4,6]$（对应子集 $\emptyset,\{2\},\{4\},\{2,4\}$）和 $S_B=[0,5,9,14]$（对应子集 $\emptyset,\{5\},\{9\},\{5,9\}$）。此时，可以组合出 $x=15$，因为 S_A 中有 6，S_B 中有 9，而 $6+9=15$，对应解 $[2,4,9]$。

该算法的时间复杂度可优化至 $O(2^{n/2})$。 首先生成排序后的列表 S_A 和 S_B，这可以通过类似归并的方式在 $O(2^{n/2})$ 时间内完成。之后，由于列表已排序，我们可以在 $O(2^{n/2})$ 时间内检查是否能从 S_A 和 S_B 中组合出 x。

例1 CSES1628 中间相遇法

时间限制： 1.00 秒
内存限制： 512 MB

给定一个包含 n 个数字的数组，求有多少种方式可以选择一个子集，使其元素之和等于 x。

输入

第一行输入包含两个数字n 和x，分别表示数组的大小和目标总和。
第二行包含n 个整数$t_1,t_2,\ldots,t_n$，表示数组中的数字。

输出

输出能够组成和x 的方式数目。

约束条件

- $1\le n\le 40$
- $1\le x\le 10^9$
- $1\le t_i\le 10^9$

分析

我们可以将给定的数组分成两个独立的部分。设 `left` 数组包含下标从 0 到 $\frac{n}{2}-1$ 的元素，`right` 数组包含下标从 $\frac{n}{2}$ 到 $n-1$ 的元素。这两个数组最多各包含 20 个元素，因此我们最多只需要 2^{21} 次操作即可遍历它们的所有子集，这在时间上是完全可行的。

在计算出这两个数组的所有子集和后，我们需要将它们重新组合以寻找答案。对于 `left` 数组中的每一个子集和 `sum`，我们只需在 `right` 数组中查找等于 $x-$ `sum` 的元素个数。这一步可以通过简单的二分查找来实现。

时间复杂度： $O(N\cdot 2^{N/2})$

代码

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
const int N = 40 + 5;
int a[N];
ll lis[(1 << 20) + 5];
int main() {
    int n, ncnt = 0; ll x;
    cin >> n >> x;
    for (int i = 0; i < n; i++) cin >> a[i];
    for (int i = 0; i < (1 << (n / 2)); i++) {
        ll sum = 0;
        for (int j = 0; j < n / 2; j++) {
            if (i >> j & 1) {
                sum += a[j];
            }
        }
        lis[++ncnt] = sum;
    }
    sort(lis + 1, lis + 1 + ncnt);
    ll ans = 0;
    for (int i = 0; i < (1 << (n - n / 2)); i++) {
        ll sum = 0;
        for (int j = 0; j < n - n / 2; j++) {
            if (i >> j & 1) {
                sum += a[j + n / 2];
            }
        }
        auto d = lower_bound(lis + 1, lis + 1 + ncnt, x - sum);
        auto u = upper_bound(lis + 1, lis + 1 + ncnt, x - sum);
        ans += u - d;
    }
    cout << ans;
    return 0;
}
```

例2 严格递增三元组

问题描述

给定一个长度为 n（$n \leq 2000$）的序列 $a_1, a_2, \ldots, a_n$，统计其中严格递增的三元组子序列（即 $i < j < k$ 且 $a_i < a_j < a_k$）的数量。

分析

虽然 **折半搜索** 通常用于优化指数级问题（如子集和问题），但在这里我们可以稍作变通，利用类似的思想：

1. **固定中间元素** a_j，然后统计左边比 a_j 小的元素数量（记为 `left`），以及右边比 a_j 大的元素数量（记为 `right`）。
2. 对每个 j，贡献的三元组数量为 `left * right`。
3. 总时间复杂度为 $O(n^2)$，可以通过预处理或动态优化进一步优化。

C++ 代码实现

```cpp
#include <iostream>
#include <vector>
using namespace std;

long long solve(const vector<int>& a) {
  int n = a.size();
  long long res = 0;

  for (int j = 1; j < n - 1; ++j) {
    int left = 0;  // 统计左边比 a[j] 小的数的个数
    for (int i = 0; i < j; ++i) {
      if (a[i] < a[j]) {
        left++;
      }
    }

    int right = 0; // 统计右边比 a[j] 大的数的个数
    for (int k = j + 1; k < n; ++k) {
      if (a[j] < a[k]) {
        right++;
      }
    }

    res += left * right; // 组合数贡献
  }

  return res;
}

int main() {
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; ++i) {
```

```
    cin >> a[i];
  }

  cout << solve(a) << endl;
  return 0;
}
```

复杂度分析

- **时间复杂度**：$O(n^2)$，因为对每个 j，需要遍历左边和右边的所有元素。
- **空间复杂度**：$O(1)$，仅用常数额外空间。

优化思路

如果需要更优的解法（如 $O(n \log n)$），可以使用 **树状数组（Fenwick Tree）或线段树** 来高效统计左边比 a_j 小和右边比 a_j 大的元素数量。但 $n \leq 2000$ 时，$O(n^2)$ 已经足够高效。

如果问题规模更大（如 $n \leq 10^5$），则需要用数据结构优化。

例题3 CSES1642 和为 x 的 4 个元素

问题描述

给定一个包含 `n` 个整数的数组，找出四个不同位置的数，使得它们的和等于给定的 `x`。

分析

折半搜索的核心思想是将问题分成两部分，分别解决后再合并结果。对于这个问题将“四个数的和”转化为“两个数的和 + 两个数的和 = x”。即 `a + b + c + d = x` 可以看作 `(a + b) = sum1` 和 `(c + d) = sum2`，其中 `sum1 + sum2 = x`。

使用双重循环遍历数组，计算所有可能的两数之和 `sum = a[i] + a[j]` （`i < j`），并记录这两个数的索引 `(i, j)`。用HASH表（如 `unordered_map`）存储两数之和到对应索引对的映射。键是两数之和，值是该和对应的所有索引对列表。

对于哈希表中的每一个 `sum1`，计算其互补和 `sum2 = x - sum1`。然后检查 `sum2` 是否存在于哈希表中：

- 如果存在，遍历 `sum1` 和 `sum2` 对应的所有索引对，确保四个索引互不相同。
- 如果找到满足条件的四个索引，返回对应的四个数。
- 在检查索引对时，需要确保四个索引互不重叠，即 `(i, j)` 和 `(k, l)` 中的 `i, j, k, l` 都是不同的。

以下是 C++ 实现代码：

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
unordered_map<ll, pair<int, int>> mp;
const int N = 1000 + 5;
ll a[N];
int main() {
```

```
  int n; ll x;
  cin >> n >> x;
  for (int i = 1; i <= n; i++) cin >> a[i];
  for (int i = 1; i <= n; i++) {
    for (int j = i + 1; j <= n; j++) {
      ll cur = a[i] + a[j];
      if (mp.find(cur) == mp.end()) {
        mp[cur] = {i, j};
      }
      else {
        if (mp[cur].second > j) {
          mp[cur] = {i, j};
        }
      }
    }
  }



  for (int i = 1; i <= n; i++) {
    for (int j = i + 1; j <= n; j++) {
      ll sum = a[i] + a[j], cur = x - sum;
      if (mp.find(cur) != mp.end()) {
        if (mp[cur].second < i) {
          cout << mp[cur].first << ' ' << mp[cur].second << ' ' << i << ' ' << j;
          return 0;
        }
      }
    }
  }
  cout << "IMPOSSIBLE";
  return 0;
}
```

例题4 CF888E 最大子序列

时间限制：每测试1秒
内存限制：每测试256兆字节

题目大意

给你一个由 n 个整数组成的数组a，以及一个整数 m。你需要选择一个索引序列 $b_1, b_2, \ldots, b_k$ $(1 \le b_1 < b_2 < \ldots < b_k \le n)$，使得$(\sum_{i=1}^{k} a_{b_i}) \bmod m$的值最大化。选择的序列可以为空。

输出$(\sum_{i=1}^{k} a_{b_i}) \bmod m$的最大可能值。

$1 \le n \le 35, 1 \le m \le 10^9$，$1 \le a_i \le 10^9$

分析

对于这个问题，我们给定 $n \leq 35$ 个数，需要找到一个组合，使得该组合中所有元素的和模 m 的结果最大化。朴素的方法，即尝试所有 2^{35} 种可能的组合，是不可行的。然而，我们可以使用折半搜索将时间复杂度降低到 $2^{\frac{n}{2}}$，这足以满足我们的时间限制。

我们首先将给定的数组分成两个长度相等的半部分。如果 n 是奇数，我们只需在第一半多放一个元素。然后，我们通过使用位掩码生成第一半中所有数字的组合，并将结果模 m 保存在一个有序集合中。对于第二半，我们也生成所有的组合。对于这些组合中的每一个，可以得到的最大和模 m 的余数，是这个组合中所有元素的和模 m 的余数，记为 sum，加上第一半组合中 **小于** $m - sum$ 的最大和 $left\_sum$。选择大于 $m - sum$ 的值永远不是最优的，因为结果将是 $left\_sum + sum - m$，这严格小于 sum。

```
#include <bits/stdc++.h>
using namespace std;

int unmask(int mask, const vector<int> &a,
           int mod) { // 求mask表示的子集和模m的余数
  int sum = 0;
  for (int i = 0; i < a.size(); i++) {
    if ((mask >> i & 1) == 1) {
      sum += a[i];
      sum %= mod;
    }
  }
  return sum;
}
int main() {
  int n, m;
  cin >> n >> m;
  // 将输入数组分成两部分
  vector<int> left_a((n + 1) / 2);
  vector<int> right_a(n / 2);
  for (int &i : left_a) {
    cin >> i;
  }
  for (int &i : right_a) {
    cin >> i;
  }

  // 存储 left_a 中所有组合的和模 m 的结果
  set<int> left_sums;
  for (int mask = 0; mask < (1 << left_a.size()); mask++) { // 枚举子集
    left_sums.insert(unmask(mask, left_a, m));
  }

  int ans = *left_sums.rbegin();   //先以左半数组的最大值为答案
  for (int mask = 0; mask < (1 << right_a.size()); mask++) {
    int sum = unmask(mask, right_a, m);
    ans = max(ans, *prev(left_sums.lower_bound(m - sum)) + sum);  //*prev()获得小
于m-sum的最大值
  }

  cout << ans << endl;
```

```
}
```

例题5 洛谷P3067 Balanced Cow Subsets G

题目描述

我们定义一个奶牛集合 S 是平衡的，当且仅当满足以下两个条件：

- S 非空。
- S 可以被**划分**成两个集合 A, B，满足 A 里的奶牛产奶量之和等于 B 里的奶牛产奶量之和。划分的含义是，$A \cup B = S$ 且 $A \cap B = \varnothing$。

现在给定大小为 n 的奶牛集合 S，询问它有多少个子集是平衡的。请注意，奶牛之间是互不相同的，但是它们的产奶量可能出现相同。

$1 \leq n \leq 20$，$1 \leq a_i \leq 10^8$

分析

如果 a_i上限不是非常大，有一个相对简单的动态规划解法。由于n 较小，尝试考虑指数时间的算法。

每个数有"不选"，"放进 A 组"，"放进 B 组"三种情况， 而3^{20}的枚举量会超时，因此考虑折半搜索。

考虑将 n头奶牛分为前后两半，每组分别暴力求解，时间复杂度$O(3^{\frac{n}{2}})$可以通过。

假设在前一半中，在 A 组中放的数的和为a，在 B 组中放的数为b。

假设在后一半中，在 A 组中放的数的和为c，在 B 组中放的数为d。

那么就要满足等式：$a+c=b+d$

由于我们要对每一半分开处理，所以考虑将同一半的数放在一起处理，即移项得$a-b=d-c$。这个式子说明前一半中的两组的和的差值与后一半中的两组的和的差值相同。

沿用前几个例题的折半搜索的思路，对于前一半的每个子集，计算所有可能的差值并存储它们；同样处理后一半。

然后对于每个可能的差值，将所有具有该值的前一半子集与所有具有该值的后一半子集配对，形成平衡的完整集合。

用DFS实现每个数的三种选择。由于维护的是划分到A, B两个组的数之和的差值，设这个差值为 sum，有一个很巧妙的做法：

- 对于选入 A 组 的数，在 dfs 传参时传一个 $sum + a[i]$
- 对于选入 B 线的数，在 dfs 传参时传一个 $sum - a[i]$

在dfs1 到达搜索边界后，在 map 中记录当前差值对应的子集状态 msk，它表示了某一半中哪些数构成了当前的子集。

在 dfs2 到达搜索边界时，需要合并答案，把map中记录的在dfs1中当前的差值的相反数$-sum$ 对应的msk 并到dfs2中的子集状态中，即合并成一个整数 tot，表示前后两半选出tot中表示那些数，可以划分成两个和相等的子集，因此将$vis[tot]=1$ 。最后枚举整个 n 的子集，把 $vis[i]=1$ 的数全部加起来，就是答案。

```
```

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 20 + 5;
using ll = long long;
ll a[N];
map<ll, vector<int>> mp;
int n;
void dfs(int i, ll sum, int msk) {  //搜索前一半
    if (i > n / 2) {
        mp[sum].push_back(msk);          //记录当前差值的子集
        return;
    }
    dfs(i + 1, sum, msk);                          // 不选a[i]
    dfs(i + 1, sum + a[i], msk | (1 << (i - 1)));       //a[i]加入A
    dfs(i + 1, sum - a[i], msk | (1 << (i - 1)));       //a[i]加入B
}
bool vis[(1 << 20) + 5];
void dfs2(int i, ll sum, int msk) {      //搜索后一半
    if (i > n) {
        for (auto x : mp[-sum]) {        //合并差值相同的两半的子集
            vis[msk | x] = 1;
        }
        return;
    }
    dfs2(i + 1, sum, msk);
    dfs2(i + 1, sum + a[i], msk | (1 << (i - 1)));
    dfs2(i + 1, sum - a[i], msk | (1 << (i - 1)));
}
int main() {
    cin >> n;
    for (int i = 1; i <= n; i++) cin >> a[i];
    dfs(1, 0, 0);
    dfs2(n / 2 + 1, 0, 0);
    int ans = 0;
    for (int i = 0; i < (1 << n); i++) ans += vis[i];
    cout << ans - 1;
    return 0;
}
```

参考链接

1. Meet in the Middle (Topic Stream) - Codeforces
2. Meet in the Middle | Tutorial & Problems

附录

生日悖论（Birthday Paradox）

生日悖论是概率论中一个非常有趣且反直觉的现象，它描述了在随机选择的群体中，两个人拥有相同生日的概率比大多数人想象的要高得多。具体来说：

> **生日悖论**：在一个房间里，如果有23个人，那么至少有两个人生日相同的概率超过50%；如果有57个人，这一概率超过99%。

1. 直观理解

大多数人会认为，一年有365天，因此需要大约183人（365的一半）才能使两人生日相同的概率达到50%。然而，实际情况是只需要23人，概率就超过了50%。这是因为我们忽略了**“配对”**的数量。

2. 数学推导

计算“至少两人生日相同”的概率，可以转化为计算“所有人生日都不同”的概率，然后用1减去它。

- **假设**：
    - 一年有365天（忽略闰年）。
    - 每个人的生日是独立且均匀分布的。
- **计算“所有人生日都不同”的概率**：
    - 第1个人：可以是任意一天（365/365）。
    - 第2个人：必须与第1个人不同（364/365）。
    - 第3个人：必须与前两个人不同（363/365）。
    - ...
    - 第n个人：必须与前n-1个人不同（(365 - n + 1)/365）。

  因此，所有人生日都不同的概率为：

$$P(\text{all different}) = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \cdots \times \frac{365 - n + 1}{365}$$

  可以写成：

$$P(\text{all different}) = \frac{365!}{(365 - n)! \times 365^n}$$

- **计算“至少两人生日相同”的概率**：

$$P(\text{at least one pair}) = 1 - P(\text{all different})$$

- **近似计算**：
  当n较小时，可以使用泰勒展开近似：

$$P(\text{all different}) \approx e^{-\frac{n(n-1)}{2 \times 365}}$$

  因此：

$$P(\text{at least one pair}) \approx 1 - e^{-\frac{n(n-1)}{730}}$$

3. 具体数值

人数（n）	至少两人生日相同的概率
10	~11.7%
20	~41.1%
23	**~50.7%**
30	~70.6%
40	~89.1%
50	~97.0%
57	**~99.0%**

4. 为什么是“悖论”?

- **直觉与现实的差距**：人们通常会低估“配对”的数量。在23人中，可能的配对数是 $C(23,2)=253$，远多于直觉认为的“23 vs 365”。
- **指数增长**：随着人数增加，配对数呈平方增长（$C(n,2)=\frac{n(n-1)}{2}$），因此碰撞概率迅速上升。

5. 应用场景

生日悖论不仅仅是一个数学趣题，它在计算机科学和密码学中有重要应用：

1. **哈希碰撞**：
    - 在哈希表中，如果哈希函数的值域是 N，则大约 $\sqrt{N}$ 次操作后，发生碰撞的概率就很高。
    - 例如，如果哈希输出是64位（2^{64} 种可能），大约 2^{32} 次操作后就会发生碰撞。
2. **密码学攻击**：
    - 生日攻击（Birthday Attack）利用生日悖论，减少暴力破解的复杂度。例如，破解哈希函数时，可以利用碰撞概率加速攻击。
3. **随机数生成**：
    - 在生成随机数或密钥时，需要确保足够的长度，以避免生日悖论导致的意外碰撞。