中山大学数据科学与计算机学院本科生实验报告

课程名称：**算法设计与分析**　　任课教师：**张子臻**

年级	**2018**	专业（方向）	**软件工程**
学号	**18342075**	姓名	**米家龙**
电话	**18566916535**	Email	**781131011@qq.com**
开始日期	**2020-05-26**	完成日期	**2020-05-26**

1. 实验题目： soj 1176 Two Ends

2. 实验目的

* 熟悉并掌握动态规划的算法

3.程序设计

思路：

使用递归实现；当数组中只剩下两个元素的时候，递归停止，否则继续；数组一次减少两个元素，分别计算两次，第一次计算先手选取数组的头，后手贪心，对剩下的数组进行递归；第二次则是先手选取数组的尾，后手贪心，对剩下的数组进行递归

相关函数如下图：

```
int twoEnd(int begin, int end, int arr[])
{
  // 终止递归
```

```
  if (begin == end - 1)
    return arr[begin] > arr[end] ? arr[begin] - arr[end] : arr[end] - arr[begin];

  // 先手选前
  bool nextChooseFront = arr[begin + 1] >= arr[end];
  int frontChoose = arr[begin] - arr[nextChooseFront ? begin + 1 : end] + twoEnd(n
extChooseFront ? begin + 2 : begin + 1, nextChooseFront ? end : end - 1, arr);

  // 先手选后
  nextChooseFront = arr[begin] >= arr[end - 1];
  int endChoose = arr[end] - arr[nextChooseFront ? begin : end - 1] + twoEnd(nextC
hooseFront ? begin + 1 : begin, nextChooseFront ? end - 1 : end - 2, arr);

  return frontChoose > endChoose ? frontChoose : endChoose;
}
```

在使用样例测试的过程中，能够得到正确的答案，但是在 OJ 进行提交的时候，会出现超时的 TLE 错误，发现有些部分会计算多次，最终导致时间消耗过大，因此尝试使用一个二维数组来储存对应的结果，另一个二维数组来储存上一个二维数组对应的位置是否储存有对应的结果，并在每个数组的输入之前进行重置。

对于上述的两个二维数组，使用一个自己定义类来将 bool 和 int 变量放在一起，类定义如下：

```
class unit
{
private:
  int res;
  bool hasCal;

public:
  unit()
  {
    this->res = 0;
    this->hasCal = false;
  }

  unit(int res, bool hasCal)
  {
    this->res = res;
    this->hasCal = hasCal;
```

```
  }

  bool getCal()
  {
    return this->hasCal;
  }

  void setHasCal(bool newHasCal)
  {
    this->hasCal = newHasCal;
  }

  int getRes()
  {
    return this->res;
  }

  void setRes(int newRes)
  {
    this->res = newRes;
  }

  void reset()
  {
    this->res = 0;
    this->hasCal = false;
  }
};
```

优化 twoEnd() 函数后变为：

```
int twoEnd(int begin, int end, int arr[])
{
  // 终止递归
  if (begin == end - 1)
  {
    matrix[begin][end].setHasCal(true);
    matrix[begin][end].setRes(arr[begin] > arr[end] ? arr[begin] - arr[end] : arr[
end] - arr[begin]);
    return matrix[begin][end].getRes();
  }

  // 如果已经存在结果
  if (matrix[begin][end].getCal())
    return matrix[begin][end].getRes();
```

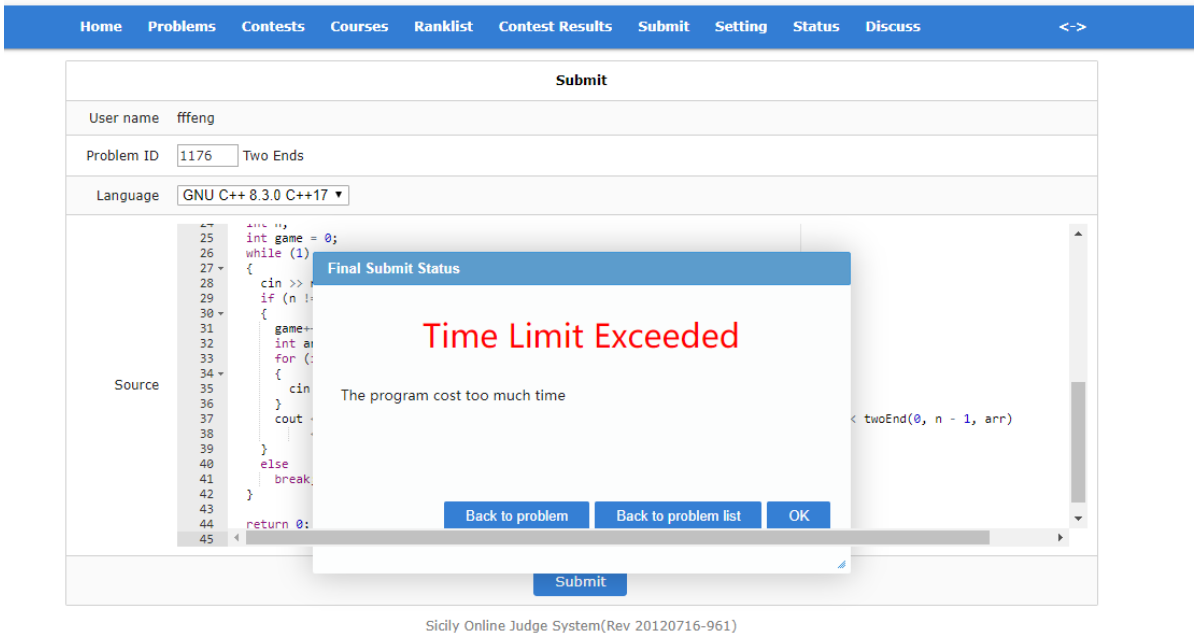
```
  // 先手选前
  bool nextChooseFront = arr[begin + 1] >= arr[end];
  int frontChoose = arr[begin] - arr[nextChooseFront ? begin + 1 : end] + twoEnd(n
extChooseFront ? begin + 2 : begin + 1, nextChooseFront ? end : end - 1, arr);

  // 先手选后
  nextChooseFront = arr[begin] >= arr[end - 1];
  int endChoose = arr[end] - arr[nextChooseFront ? begin : end - 1] + twoEnd(nextC
hooseFront ? begin + 1 : begin, nextChooseFront ? end - 1 : end - 2, arr);

  matrix[begin][end].setHasCal(true);
  matrix[begin][end].setRes(frontChoose > endChoose ? frontChoose : endChoose);
  return matrix[begin][end].getRes();
}
```

4. 程序运行与测试

未使用储存进行优化：

```
root@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法# g++ 1176.cpp
root@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法# ./a.out
4 3 2 10 4
In game 1 , the greedy strategy might lose by as many as 7 points
8 1 2 3 4 5 6 7 8
In game 2 , the greedy strategy might lose by as many as 4 points
8 2 2 1 5 3 8 7 3
In game 3 , the greedy strategy might lose by as many as 5 points
0
```

```
root@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法# ./a.out
10
1 1 1 1 1 1 5 6 4 7 8 9
In game 1, the greedy strategy might lose by as many as 4 points
```



使用储存进行优化之后：

```
root@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法# ./a.out
4 3 2 10 4
In game 1, the greedy strategy might lose by as many as 7 points
process time = 0.015625
8 1 2 3 4 5 6 7 8
In game 2, the greedy strategy might lose by as many as 4 points
process time = 0
8 2 2 1 5 3 8 7 3
In game 3, the greedy strategy might lose by as many as 5 points
process time = 0
0
```

5. 实验总结与心得

题目本身比较简单，思路比较清晰，比较方便使用动态编程的思维来解题，而困难的其实是如何考虑并且优化重复计算的部分。

附录、提交文件清单

源代码：

```
#include <iostream>

using namespace std;

class unit
```

```
{
private:
  int res;
  bool hasCal;

public:
  unit()
  {
    this->res = 0;
    this->hasCal = false;
  }

  unit(int res, bool hasCal)
  {
    this->res = res;
    this->hasCal = hasCal;
  }

  bool getCal()
  {
    return this->hasCal;
  }

  void setHasCal(bool newHasCal)
  {
    this->hasCal = newHasCal;
  }

  int getRes()
  {
    return this->res;
  }

  void setRes(int newRes)
  {
    this->res = newRes;
  }

  void reset()
  {
    this->res = 0;
    this->hasCal = false;
  }
};

const int MAX = 1000;
```

```
unit matrix[MAX][MAX];

int twoEnd(int begin, int end, int arr[])
{
  // 终止递归
  if (begin == end - 1)
  {
    matrix[begin][end].setHasCal(true);
    matrix[begin][end].setRes(arr[begin] > arr[end] ? arr[begin] - arr[end] : arr[
end] - arr[begin]);
    return matrix[begin][end].getRes();
  }

  // 如果已经存在结果
  if (matrix[begin][end].getCal())
    return matrix[begin][end].getRes();

  // 先手选前
  bool nextChooseFront = arr[begin + 1] >= arr[end];
  int frontChoose = arr[begin] - arr[nextChooseFront ? begin + 1 : end] + twoEnd(n
extChooseFront ? begin + 2 : begin + 1, nextChooseFront ? end : end - 1, arr);

  // 先手选后
  nextChooseFront = arr[begin] >= arr[end - 1];
  int endChoose = arr[end] - arr[nextChooseFront ? begin : end - 1] + twoEnd(nextC
hooseFront ? begin + 1 : begin, nextChooseFront ? end - 1 : end - 2, arr);

  matrix[begin][end].setHasCal(true);
  matrix[begin][end].setRes(frontChoose > endChoose ? frontChoose : endChoose);
  return matrix[begin][end].getRes();
}

int main()
{
  int n = INT32_MIN;
  int game = 0;
  while (1)
  {
    cin >> n;
    if (n != 0)
    {
      game++; // 表示第几个游戏
      int arr[n] = {0};
      for (int i = 0; i < n; i++)
      {
        cin >> arr[i];
```

```
      }

      // 重置矩阵
      for (int i = 0; i < MAX; i++)
      {
        for (int j = 0; j < MAX; j++)
        {
          matrix[i][j].reset();
        }
      }

      cout << "In game " << game << ", the greedy strategy might lose by as many a
s " << twoEnd(0, n - 1, arr)
           << " points." << endl;
    }
    else
      break;
  }

  return 0;
}
```

1. 实验题目： soj 1011 Lenny’s Lucky Lotto

2. 实验目的

熟悉并掌握 动态规划 的算法

3. 程序设计

思路：

一开始决定使用递归的方式来编写函数，并且得到了对于长度为 i ，最后一个数为 j 的序列数的通式为 $f[i][j] = \sum_{k=1}^{j/2} f[i-1][k]$ 此时发现使用循环会更加方便，并且在 N 和 M 较大的时候减少递归的次数，递归函数如下：

```
int lotto(int i, int j, int **arr)
{
  if (i == 1)
    return arr[i][j];
  int res = 0;
  for (int e = 1; e <= (j / 2); e++)
  {
    res += lotto(i - 1, e, arr);
  }
  arr[i][j] = res;
  return arr[i][j];
}
```

修改成循环函数之后代码如下：

```
void lotto_loop(int N, int M, int **arr)
{
  if (N == 1)
    return;

  // 从长度为 2 开始
  for (int i = 2; i <= N; i++)
  {
    for (int j = 1; j <= M; j++)
```

```
    {
      for (int k = 1; k <= j / 2; k++)
        arr[i][j] += arr[i - 1][k];
    }
  }
}
```

发现对于 N＝10 ， M＝2000 的情况下，会出现数据溢出的情况，因此将数组类型改为 long，修改后代码如下图

```
void lotto_loop(int N, int M, long **arr)
{
  if (N == 1)
    return;

  // 从长度为 2 开始
  for (int i = 2; i <= N; i++)
  {
    for (int j = 1; j <= M; j++)
    {
      for (int k = 1; k <= j / 2; k++)
        arr[i][j] += arr[i - 1][k];
    }
  }
}
```

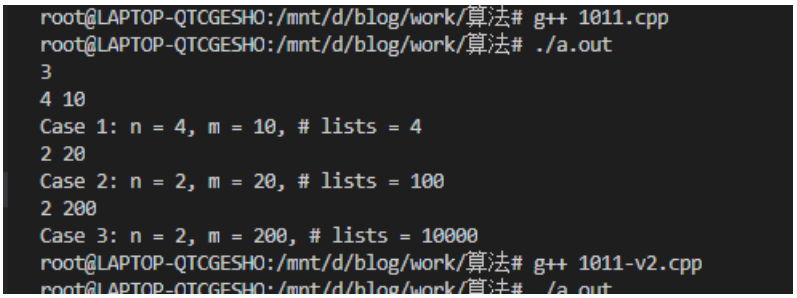
由于该数组中存放的是最后一个数必定为 J 的时候的结果，因此如果需要求出题目要求的结果，还需要将数组第 N 行的数据累加起来，最终形成了三重循环，如果将这个数组改为储存长度为 i ，最后一个数小于等于 j 的情况下的结果，能够得到通式$s[i][j] = s[i][j-1] + f[i][j]$，同时我们可以知道$f[i][j] = s[i-1][\frac{j}{2}]$因此能够得到$s[i][j] = s[i][j-1] + s[i-1][\frac{j}{2}]$，修改后的代码如下：

```
long lotto(int i, int j, long **arr)
{
  if (i == 1)
    return arr[i][j];
  for (int k = 2; k <= i; k++)
  {
    for (int e = 1; e <= j; e++)
      arr[k][e] = arr[k][e - 1] + arr[k - 1][e / 2];
  }
```

```
  return arr[i][j];
}
```

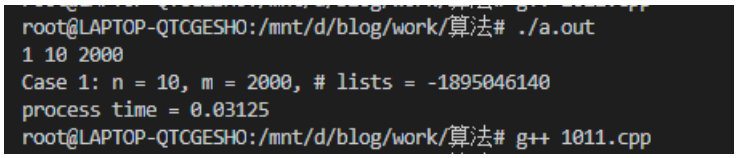
4. 程序运行与测试

使用递归：

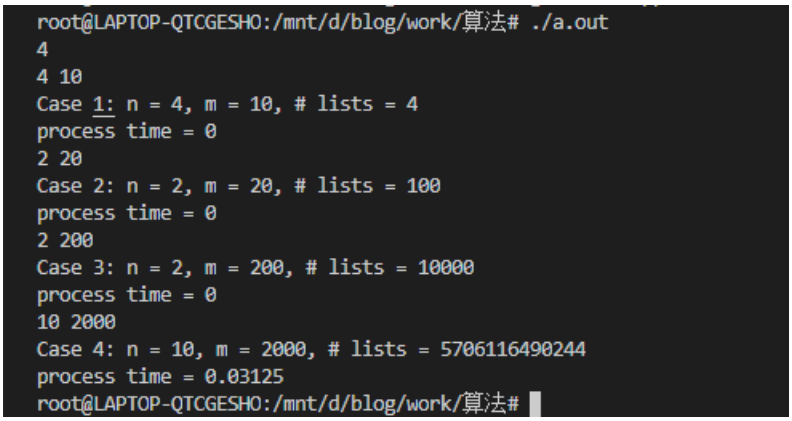


使用循环：

发现会出现数据溢出的情况：



将数组从 int 修改为 long 之后：



将数组改为储存最后一位小于或等于 j 的结果后：

```
root@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法# ./1011.out
1 10 2000
Case 1: n = 10, m = 2000, # lists = 5706116490244
process time = 0.03125
root@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法# ./a.out
1 10 2000
Case 1: n = 10, m = 2000, # lists = 5706116490244
process time = 0
root@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法# ./a.out
1 10 3000
Case 1: n = 10, m = 3000, # lists = 398823992503604
process time = 0
root@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法# ./1011.out
1 10 3000
Case 1: n = 10, m = 3000, # lists = 398823992503604
process time = 0.0625
```

其中 1011.out 是未修改数组储存结果的代码生成的文件，而 a.out 则是修改后的代码生成的文件

扩大数据到 N = 10，M = 10000：

```
root@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法# ./a.out
1 10 10000
Case 1: n = 10, m = 10000, # lists = 35142403166634253OO
process time = 0
root@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法# ./1011.out
1 10 10000
Case 1: n = 10, m = 10000, # lists = 35142403166634253OO
process time = 0.65625
```

扩大数据到 M = 10,0000：

```
root@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法# ./1011.out
1 10 100000
Case 1: n = 10, m = 100000, # lists = 7685704360626058600
process time = 68.125
root@LAPTOP-QTCGESHO:/mnt/d/blog/work/算法# ./a.out
1 10 100000
Case 1: n = 10, m = 100000, # lists = 7685704360626058600
process time = 0
```

5. 实验总结与心得、

在做这题的过程中，发现动态规划并不只能够通过递归来实现，还可以使用循环，虽然看上去并不像，但实际上确实是动态规划的思路；并且由于这道题偏向于数学，在没有得到通式之前比较难理清思路，因此数学归纳也是一个重要的一环。

这次题目中，随着手动扩大数据量，会发现一个较好的算法能够节省大量的

事件，这在 M＝10,0000 的时候充分体现了出来，因此，对于现在的算法题尽量乃至实际应用，应当尽可能地优化算法，算法的力量确实很强大。

附录、提交文件清单

1011.cpp (对应后续对比时的 1011.out)：

```cpp
#include <iostream>
#include <time.h>

using namespace std;

// 这函数没用了
int lotto(int i, int j, int **arr)
{
  if (i == 1)
    return arr[i][j];
  int res = 0;
  for (int e = 1; e <= (j / 2); e++)
  {
    res += lotto(i - 1, e, arr);
  }
  arr[i][j] = res;
  return arr[i][j];
}

void lotto_loop(int N, int M, long **arr)
{
  if (N == 1)
    return;

  // 从长度为 2 开始
  for (int i = 2; i <= N; i++)
  {
    for (int j = 1; j <= M; j++)
    {
      for (int k = 1; k <= j / 2; k++)
        arr[i][j] += arr[i - 1][k];
    }
  }
}
```

```
int main()
{
  int n = 0, caseNum = 0;
  cin >> n;
  while (n--)
  {
    caseNum++;
    int N, M;
    cin >> N >> M;
    int startTime = clock();
    // 算法代码
    {
      // 初始化二维数组
      long **arr = new long *[N + 1];
      for (int i = 0; i < N + 1; i++)
      {
        arr[i] = new long[M + 1];
        for (int j = 0; j < M + 1; j++)
        {
          arr[i][j] = i == 1 ? 1 : 0;
          if (i == 0 || j == 0)
            arr[i][j] = i == 0 ? j : i;
        }
      }
      // 需要进行累加
      long res = 0;
      lotto_loop(N, M, arr);
      for (int i = 1; i < M + 1; i++)
        res += arr[N][i];
      cout << "Case " << caseNum << ": n = " << N << ", m = " << M << ", # lists =
 "
           << res << endl;

      // for (int i = 0; i < N + 1; i++)
      // {
      //   for (int j = 0; j < M + 1; j++)
      //     cout << arr[i][j] << '\t';
      //   cout << endl;
      // }
    }
    int endTime = clock();
    cout << "process time = " << (double)(endTime - startTime) / CLOCKS_PER_SEC <<
 endl;
  }
  return 0;
}
```

1011-v2.cpp (对应后面对比时的 a.out) ：

```cpp
#include <iostream>
#include <time.h>

using namespace std;

long lotto(int i, int j, long **arr)
{
  if (i == 1)
    return arr[i][j];
  for (int k = 2; k <= i; k++)
  {
    for (int e = 1; e <= j; e++)
      arr[k][e] = arr[k][e - 1] + arr[k - 1][e / 2];
  }
  return arr[i][j];
}

int main()
{
  int n = 0, caseNum = 0;
  cin >> n;
  while (n--)
  {
    caseNum++;
    int N, M;
    cin >> N >> M;
    int startTime = clock();
    // 算法代码
    {
      // 初始化二维数组
      long **arr = new long *[N + 1];
      for (int i = 0; i < N + 1; i++)
      {
        arr[i] = new long[M + 1];
        for (int j = 0; j < M + 1; j++)
        {
          arr[i][j] = i == 1 ? j : 0;
        }
      }

      cout << "Case " << caseNum << ": n = " << N << ", m = " << M << ", # lists =
 "
           << lotto(N, M, arr) << endl;
```

```
      // for (int i = 0; i < N + 1; i++)
      // {
      //   for (int j = 0; j < M + 1; j++)
      //     cout << arr[i][j] << '\t';
      //   cout << endl;
      // }
    }
    int endTime = clock();
    cout << "process time = " << (double)(endTime - startTime) / CLOCKS_PER_SEC <<
 endl;
  }
  return 0;
}
```