Problem 1:

1.解題說明:

圖中函式**A**以遞迴的方式執行:

t為計數

m=0:A(m,n)=n+1

m>0, **n=0**:A(m,n)=A(m-1,1)

m>0, **n>0**:A(m,n)=A(m-1,A(m,n-1))

```
int t = 0;
int A(int m, int n) {

    if (m == 0) {
        t++;
        return n + 1;
    }
    else if (m > 0 && n == 0) {
        t++;
        return A(m - 1, 1);
    }
    else if (m > 0 && n > 0) {
        t++;
        return A(m - 1, A(m, n - 1));
    }
}
```

為改善程式執行效能, 我對程式做了更改:

新增二維陣列**arrack**, 並將其陣列內容全改為**-1**。

於函式**A**中, 在遞迴前會先偵測:若**arrack[m][n]!=1**, 則得知該陣列索引位置已計算過, 則不再繼續計算。

下面遞迴從**return** 改為 **arrack[m][n]=...**, 以將運算結果填入陣列中。

```
const int maxn = 1000;
long long arrack[maxm][maxn];
int t=0;
long long A(int m, int n) {
    if (arrack[m][n] != -1) {
        return arrack[m][n];
    }
    if (m == 0) {
        arrack[m][n] = n + 1;
        t++;
    }
    else if (m > 0 && n == 0) {
        arrack[m][n] = A(m - 1, 1);
        t++;
    }
    else if (m > 0 && n > 0) {
        arrack[m][n] = A(m - 1, A(m, n - 1));
        t++;
    }
    return arrack[m][n];
}
```

```cpp
int main() {
    int m, n;
    long long d;
    cout << "輸入m,n：";
    cin >> m >> n;
    for (int i = 0; i < maxm; i++) {
        for (int j = 0; j < maxn; j++) {
            arrack[i][j] = -1;
        }
    }
    d = A(m, n);
    cout << d << endl;
    cout << t << endl;
    return 0;
}
```

```cpp
int main() {
    int m, n;
    int d;
    cin >> m >> n;
    d = A(m, n);
    cout << d << endl;
    cout << t << endl;
    return 0;
}
```

主程式則是宣告**m,n**，**d**負責存函式回傳值，修改後程式則多了迴圈把**arrack**陣列成員皆改為**-1**。

2.效能分析：

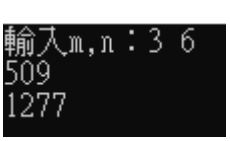
以原本的遞迴函式執行：

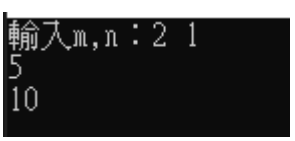
以修改後的遞迴函式執行：

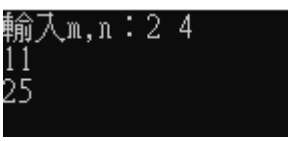
以**m=2**，**n=3**為例，修改後的函式執行**20**次，修改前則**44**次。

3.測試與驗證：

以下為三組測資的測試結果：







4.申論與心得:

在設計這個程式時，我先從題目的數學定義開始理解阿克曼函式的結構。遞迴的特性讓這個函式特別適合以遞迴方式實現，因為它的運算依賴於先前的值，而這正是遞迴算法的強項。撰寫程式過程中讓我感受到遞迴可以很簡潔地解決問題，但也因為阿克曼函式的急速增長特性，當輸入值較大時，遞迴層數會迅速增加，可能導致效能問題。

Problem 2:

1.解題說明:

```
void powerset(char S[], int n, char subset[], int subsetSize, int index) {
    if (index == n) {
        printSubset(subset, subsetSize);
        return;
    }
    powerset(S, n, subset, subsetSize, index + 1);
    subset[subsetSize] = S[index];
    powerset(S, n, subset, subsetSize + 1, index + 1);
}
```

這個函式負責遞迴:

S[ ]:原集合。

n:集合的大小。

subset[ ]:目前正在構造的子集。

subsetSize:目前子集的大小。

index:追蹤目前處理到集合中的哪個元素。

index=n時, 則呼叫printSubset

```
void printSubset(char subset[], int size) {
    cout << "{ ";
    for (int i = 0; i < size; i++) {
        cout << subset[i] << " ";
    }
    cout << "}" << endl;
}
```

這個函式負責輸出。

```
int main() {
    char* S;
    int n;
    cout << "請輸入集合大小：" << endl;
    cin >> n;
    cin.ignore();
    S = new char[n];
    cout << "請輸入集合成員：" << endl;
    for (int i = 0; i < n; i++) {
        cin >> S[i];
    }
    char* subset;
    subset= new char[n];
    cout << "Powerset: " << endl;
    powerset(S, n, subset, 0, 0);
    delete[] S;
    delete[] subset;
    return 0;
}
```

此為主程式。

n為集合大小

cin.ignore();負責抓上一個**cin**的**\n**值

由動態陣列宣告**S**的大小，再由使用者輸入**S[i]**值

subset為子集

2.效能分析:

```
int t = 0;
```

```
void powerset(char S[], int n, char subset[], int subsetSize, int index) {
    t++;
    if (index == n) {
        printSubset(subset, subsetSize);
        return;
    }
    powerset(S, n, subset, subsetSize, index + 1);
    subset[subsetSize] = S[index];
    powerset(S, n, subset, subsetSize + 1, index + 1);
}
```

```
cout << t << endl ;
```

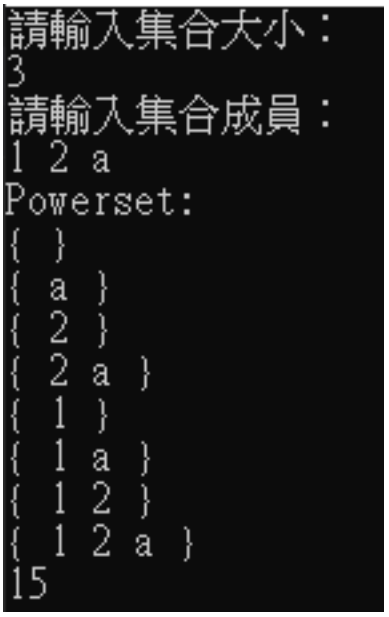
如第一題，宣告**t**計數，在函式中**t++**以計數，在主程式中輸出以查看遞迴執行次數

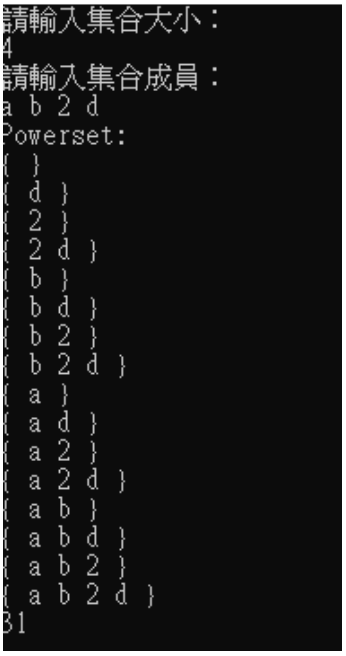
```
請輸入集合大小：
3
請輸入集合成員：
a b c
Powerset:
{ }
{ c }
{ b }
{ b c }
{ a }
{ a c }
{ a b }
{ a b c }
15
```

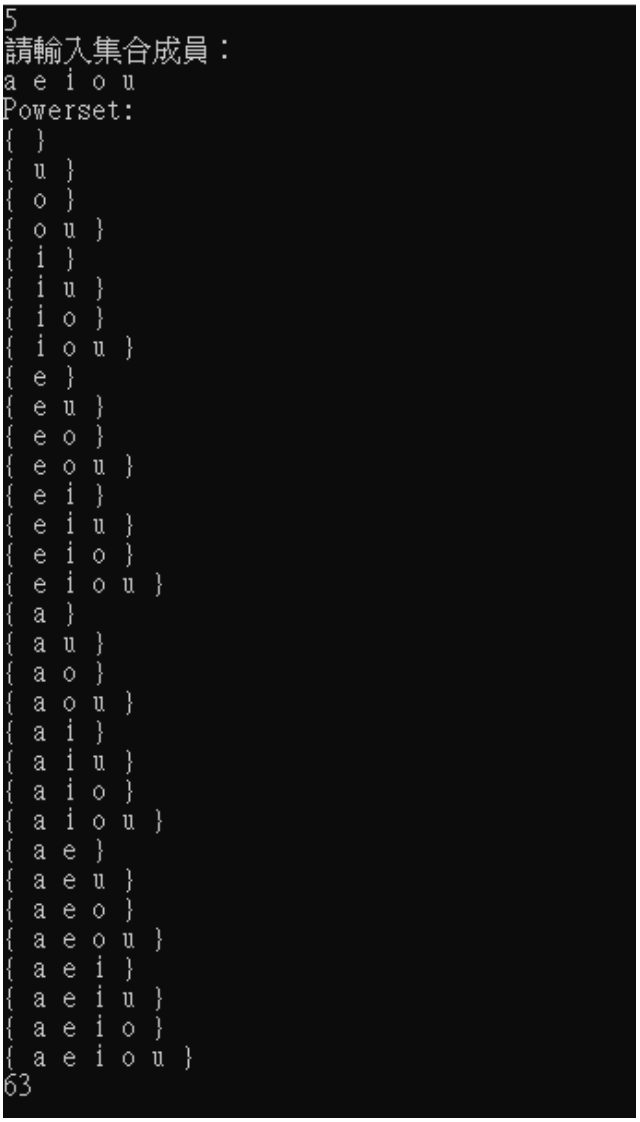
此為集合大小為**3**的遞迴次數

3.測試與驗證:

以下為三組測資的測試結果:







4.申論與心得:

當我拿到這題時，我知道要用遞迴，因為冪集的生成過程本質上就是一個分治問題:每個元素都有「選擇」或「不選擇」的兩種狀態，這樣會自然地產生樹狀結構。遞迴是處理這類問題的理想方法，因為它可以輕鬆遍歷所有可能的分支。基於這個概念，我設計了程式來遞迴處理每個元素，並在遞迴到底時輸出子集。