# Safe Fruit

**Date: 2021-4-23**

# Chapter 1: Introduction

According to the requirement, **this project aims to select those fruits that can be eaten together.** We can acquire several pairs of fruits that must not be eaten together, namely tips, from the input. Additionally, the number of fruits ***N***, tips ***M***, and the price of each fruit are contained in the input.

Besides, the distinct number ***N*** and ***M*** will be no more than 100, and there is no duplicated tips or fruits. To make it simple, each fruit is indexed by a 3-digit number. Every price is a positive integer which is no more than 1000.

For simplicity, the index starts from 001 and the indices are consecutive. The number ***N*** and ***M*** are assumed to be positive.

To find the solution to the problem, the key algorithm, written in C, uses techniques, including backtracking, DFS(Deep-First Search), and pruning.

# Chapter 2: Data Structure / Algorithm Specification

**Backtracking** is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree). In backtracking, we use recursion to explore all the possibilities until we get the best result for the problem.

**Depth-first search** is an algorithm for traversing or searching a tree or other graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. The basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then it backtracks and checks for other unmarked nodes and traverse them. Finally it prints the nodes in the path.

**Pruning** is a technique associated with decision trees. Pruning reduces the size of decision trees by removing parts of the tree that do not provide the power to classify instances. In this project, we use a simple pruning to discard the branch in which **the number** of fruits in the basket is impossible to exceed the known maximum number in the further search, or **the total price** of the fruits, those not judged and those in the current basket, exceeds the known minimum price. In another word, we abandon the branch in which **we cannot find a better solution** that has **more fruits** in the basket or have **lower total price**. The key code of pruning as follow:

```
if((potential_num + cur_num < maxnum) ||
(potential_num +
cur_num == maxnum && potential_price >= minprice))   // an easy pruning
  return -1
```

In the project, we use the **array** to store all indices, prices of fruits, and fruits that can be eaten together, namely *basket*. We use linked lists to store the fruit pairs that cannot be eaten together and construct the following structure to store the information of all the fruits:

```
// use linked lists to represent unsafe fruits
struct fruit
{
    int opposite; // the fruit cannot be eaten together
    struct fruit *next;
}fruits[SIZE];
typedef struct fruit fruit;
```

The pseudo-code for Backtracking used in the program is as follow:

```
 bool FoundSolution( int i, other params) // using backtracking
{   Found = false;
    if ( no fruit can be selected )
        return true; /* solved with (x1, …, xN) */

    for ( each fruit ) { // using DFS
        /* check if satisfies the restriction R */
        OK = Check((x1, …, xi) , R ); /* pruning */
        if ( OK ) {
            Count xi in;
            Found = FoundSolution( i+1 );
            if ( !Found )
                Undo( i ); /* recover to (x1, …, xi-1) */
        }
        if ( Found ) break;
    }
    return Found;
}
```

# Chapter 3: Test

## 3.1 Correctness of Code

The implementation has many constraints, and the implementation needs input to ensure the index of fruits to be consecutive and every fruit has a price. We give three easy samples to test the code.

| Cases/Inputs | Results | purpose |
|---|---|---|
| **16 20<br>001 002<br>003 004<br>004 005<br>005 006<br>006 007<br>007 008<br>008 003<br>009 010<br>009 011<br>009 012<br>009 013<br>010 014<br>011 015<br>012 016<br>012 017<br>013 018<br>020 99<br>019 99<br>018 4<br>017 2<br>016 3<br>015 6<br>014 5<br>013 1<br>012 1<br>011 1<br>010 1<br>009 10<br>008 1<br>007 2<br>006 5<br>005 3<br>004 4<br>003 6** | 12<br>002 004 006 008 009 014 015 016 017 018 019 020<br>239 | the sample given by the requirement |

| | | |
|---|---|---|
| **002 1**<br>**001 2** | | |
| **50 50**<br>**43 50**<br>**22 7**<br>**21 23**<br>**19 12**<br>**9 20**<br>**28 12**<br>**21 47**<br>**4 11**<br>**37 30**<br>**17 44**<br>**7 35**<br>**28 28**<br>**6 46**<br>**25 42**<br>**5 46**<br>**37 24**<br>**17 21**<br>**19 37**<br>**27 36**<br>**10 39**<br>**32 48**<br>**17 27**<br>**30 50**<br>**24 50**<br>**26 45**<br>**43 19**<br>**21 21**<br>**50 30**<br>**10 9**<br>**4 14**<br>**24 14**<br>**42 38**<br>**3 15**<br>**38 32**<br>**13 26**<br>**34 5**<br>**26 19**<br>**1 51**<br>**33 2**<br>**1 11**<br>**45 24** | 30<br>001 004 008 010 012 015 016 018 020 022 023 024 025 026 029 030 031 033 034 035 036 038 040 041 043 044 046 047 048 049<br>14749 | a sample randomly generated |

| | | |
|---|---|---|
| **2 20**<br>**17 35**<br>**2 37**<br>**29 50**<br>**27 9**<br>**48 3**<br>**19 11**<br>**6 1**<br>**37 30**<br>**1 384**<br>**2 174**<br>**3 332**<br>**4 736**<br>**5 633**<br>**6 403**<br>**7 990**<br>**8 825**<br>**9 943**<br>**10 419**<br>**11 872**<br>**12 432**<br>**13 869**<br>**14 567**<br>**15 895**<br>**16 327**<br>**17 182**<br>**18 927**<br>**19 922**<br>**20 279**<br>**21 53**<br>**22 298**<br>**23 395**<br>**24 20**<br>**25 317**<br>**26 350**<br>**27 257**<br>**28 924**<br>**29 24**<br>**30 986**<br>**31 536**<br>**32 903**<br>**33 872**<br>**34 893**<br>**35 78** | | |

| | | |
|---|---|---|
| **36 234**<br>**37 115**<br>**38 854**<br>**39 930**<br>**40 642**<br>**41 419**<br>**42 768**<br>**43 25**<br>**44 852**<br>**45 818**<br>**46 849**<br>**47 634**<br>**48 89**<br>**49 158**<br>**50 419** | | |
| **15 30**<br>**1 2**<br>**3 4**<br>**5 6**<br>**7 8**<br>**9 10**<br>**11 12**<br>**13 14**<br>**15 16**<br>**17 18**<br>**19 20**<br>**21 22**<br>**23 24**<br>**25 26**<br>**27 28**<br>**29 30**<br>**1 1**<br>**2 2**<br>**3 1**<br>**4 2**<br>**5 1**<br>**6 2**<br>**7 1**<br>**8 2**<br>**9 1**<br>**10 2**<br>**11 1**<br>**12 2**<br>**13 1** | 15<br>001 003 005 007 009 011 013 015 017 019 021 023 025 027 029<br>15 | construct a special input:<br><br>**n 2n**<br>**1,2**<br>**3,4**<br>**…**<br>**2n-1,2n**<br>**1,1**<br>**2,2**<br>**3,1,**<br>**…**<br>**2n-1,1**<br>**2n,2**<br>Then we make n=15<br>To check whether the program outputs the solution with the lowest total price |

| | | |
|---|---|---|
| **14 2** | | |
| **15 1** | | |
| **16 2** | | |
| **17 1** | | |
| **18 2** | | |
| **19 1** | | |
| **20 2** | | |
| **21 1** | | |
| **22 2** | | |
| **23 1** | | |
| **24 2** | | |
| **25 1** | | |
| **26 2** | | |
| **27 1** | | |
| **28 2** | | |
| **29 1** | | |
| **30 2** | | |

# Chapter 4: Analysis

In short, the space complexity is quite acceptable comparing to using a matrix, since the matrix will be sparse with high probability. To improve time complexity, a more accurate and more efficient pruning is needed.

## 4.1 Space Complexity

Since we use arrays and the linked list to store the information of fruits, the space complexity of the algorithm is only related to the number of fruit given in the input file. Therefore, the space complexity of the whole algorithm is ***O(N)***. In contrast to using a matrix, linked lists and arrays take significantly less space.

## 4.2 Time Complexity

To explore the solution to the problem, we need to construct a searching tree. In every level of the searching tree, we will decide whether to add the distinct fruit or not. It is apparent that the searching tree without any pruning will have $2^N$ leaf nodes, so the time complexity is $O(2^N)$. Using pruning in the algorithm can improve the searching efficiency to some extent since we need not explore all branches in the searching tree. However, the time complexity of the algorithm with pruning is somehow too complex to analyze here.

In practice, the program runs for approximately 30 seconds when dealing with sample 2, which may take $2^{50}$ recursions without pruning. It's also hard to prove how many recursions are saved by the simple pruning we used in the program.

We searched the Internet for a more efficient algorithm and we found *Bron–Kerbosch*, which solves the problem as a graphical problem. The algorithm takes each fruit as a vertex and each tip as an edge. It uses an array to store the number of vertices in the maximum clique in the graph formed by the last several vertices, which provides a more accurate pruning. The pseudo-code we found online is as follow (please see References):

```
const int SIZE = 105;
int mat[SIZE][SIZE];
int dp[SIZE];
int mx;
int stack[SIZE][SIZE];
void dfs(int N,int num,int step){
    if(num==0){
        if(step > mx){
            mx=step;
        }
        return ;
    }

    for(int i=0;i<num;i++){
        int k = stack[step][i];
        if(step+N-k<=mx) return ;
        if(step+dp[k]<=mx) return ;
        int cnt = 0;
        for(int j=i+1;j<num;j++)
            if(mat[k][stack[step][j]]){
                stack[step+1][cnt++]=stack[step][j];
            }
        dfs(N,cnt,step+1);
    }
}

void run(int N){
    mx =0;
    for(int i=N-1;i>=0;i--){
        int sz =0;
        for(int j=i+1;j<N;j++)
            if(mat[i][j]) stack[1][sz++]=j;
        dfs(N,sz,1);
        dp[i]=mx;
    }
}
```

# Chapter 5: Conclusion

The project uses a backtracking algorithm to find the maximum number of fruits that can be eaten together and makes the total price as low as possible. The basic idea to solve the problem is to use DFS to search the whole search tree and backtracking algorithm. However, we find out that our program is quite slow to find out the solution, which indicates that our algorithm is not so efficient (since the pruning we use is not that accurate), other algorithms, such as *Bron–Kerbosch*, may have better performance when implemented in this problem.

# Appendix: Source Code

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define SIZE 1005

// use linked lists to represent unsafe fruits
struct fruit
{
    int opposite; // the fruit cannot be eaten together
    struct fruit *next;
}fruits[SIZE];
typedef struct fruit fruit;

int maxnum, minprice, basket[SIZE];  // for the final solution, basket[] stores the exact solution
int total[SIZE], amount, tips, bptr, curprice;      // for the process
int price[SIZE];    // the price of each fruit

// insert an unsafe fruit into the list
void inserttips(fruit *f, int opp)
{
    fruit *tmp = (fruit *)malloc(sizeof(struct fruit));
    tmp->opposite = opp;
    tmp->next = f->next;
    f->next = tmp;
    return ;
}
```

```c

// check if a fruit is unsafe to the basket
int findx(int x)
{
    if(price[x] == 0)
        return -1;      // no such fruit
    for(int i=0; i<bptr; i++)
    {
        fruit *ptr = fruits[total[i]].next;
        while (ptr)
        {
            if(ptr->opposite == x)
                return 1;           // a conflict is found
            else
                ptr = ptr->next;
        }
    }
    return 0;           // the fruit can be added to the basket
}

// for a rough pruning
// count the total cost of those not in the basket
int countprice(int x)
{
    int ct = 0;
    for(int i=x; i<=amount; i++)
    {
        ct += price[i];
    }
    return ct + curprice;
}

// judge whether a fruit can be added to the basket
// update the information if necessary
int judge(int x)
{
    if(!findx(x))
    {
        int available = amount - x + 1, a_price = countprice(x);
        if((available+bptr < maxnum) || (available+bptr == maxnum && a_price
  >= minprice))   // an easy pruning
        {
            return -1;
        }
        else    // goes on and update if necessary
        {
            // more fruits or lower cost
            if((bptr+1 > maxnum) || (bptr+1 == maxnum && curprice + price[x] < minprice))
            {
                maxnum = bptr + 1;
                for(int i=0; i<bptr; i++)
                {
                    basket[i] = total[i];
                }
                basket[bptr] = x;
                minprice = curprice + price[x];
            }
            return 1;
        }
    }
    else            // conflict
    {
        return 0;
    }
}

// just like a dfs function
void findsolution(int level)
{
    if(level == amount +1)  // the search has ended
        return ;
    int i = level;
    int exam = judge(i);
    if(exam == 1)   // the recursion goes on
    {
        total[bptr++] = i;
        curprice += price[i];
        findsolution(level +1);     // find solutions when current fruit is in the basket
        curprice -= price[i];
        total[--bptr] = 0;
        findsolution(level +1);     // find solutions when current fruit is not in the basket
    }
    else if(exam == -1) // prune
    {
        return ;
    }
    else            // conflict
    {
        findsolution(level +1);
    }
}

int main()
{
    // initialization
    for(int i=0; i<SIZE; i++)
    {
        fruits[i] = {-1, NULL};
        price [i] = 0;
    }
    scanf("%d %d", &tips, &amount);
    for(int i=0; i<tips; i++)
    {
        int a, b;
        scanf("%d %d", &a, &b);
        inserttips(&fruits[a], b);
        inserttips(&fruits[b], a);
    }
    for(int i=0; i<amount; i++)
    {
        int index, p;
        scanf("%d %d", &index, &p);
        price[index] = p;
    }
    maxnum = bptr = curprice = 0;
    minprice = 0x7FFFFFFF;

    // time with easy pruning: 0.000300 ?
    // clock_t start = clock();
    findsolution(0);
    // clock_t end = clock();
    // printf("time with easy pruning: %f\n", (double)(end-start)/CLK_TCK);

    printf("%d\n", maxnum);
    for(int i=0; i<maxnum; i++)
    {
        if(i == 0)
            printf("%03d", basket[i]);
        else
```

```
            printf(" %03d", basket[i]);
    }
    printf("\n%d", minprice);
    return 0;
}
```

## References

[1] Mark Allen Weiss, “Data Structures and Algorithm Analysis in C, Second Edition”, *机械工业出版社*, 2019.11
[2] https://blog.csdn.net/qq_41562704/article/details/103377879

## Author List

## Declaration

***We hereby declare that all the work done in this project titled***

***"Safe Fruit" is of our independent effort as a group.***

## Signatures

Each author must sign his/her name here.