# Scroll down to see the answer

# Question 1 - Search

Given that you have natural numbers from 1,2,3,…29,30, list all the possible combinations the number summary equals to 50, no duplicated number in the summary.

For example: 30 + 20;

30+19+1 ;

30+18+2;

….

20+19+11;

20+19+10+1;

…

# Question 2 - Finding the best plan

This is a problem we had to solve recently. Let’s say a cloud service offers different plans. For example, Office 365 has [different plans](http://www.microsoft.com/en-us/office365/compare-plans.aspx) called E1, P2, etc. Each plan has different features such as voice, email, archiving, etc. Now let say the user selects a set of features he/she wants. The goal is to write code (in any language) that finds the combination of plans that offers all selected features at the lowest price. Note that in some cases, it will be just one plan, but in other cases you will need 2+ plans to get all the features you want.

To get you started:

class Feature { public string Name; }  
class Plan { public string Name; public double Cost; public Feature[] Features; }

// The list of plans available instantiated as per the above  
Plan[] allPlans;

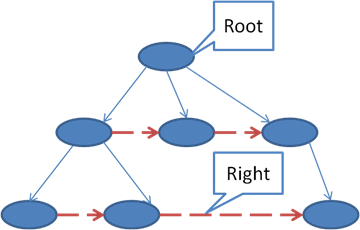
// The list of features the user wants  
Feature[] selectedFeatures;   
  
// Goal: find combinations of 1-N plans that fulfill selected features -> select the cheapest combination(s)

# Question 3 - Linking nodes in a tree

Each Node represents an element of a tree and specifies a list of immediate children. The 'Children' property lists all children (in order) but the 'Right' property is set to null. Suppose you are given the root of a fully populated tree (i.e. a Node instance called rootNode). Write the code to set the 'Right' property so that each node is linked to right siblings using memory efficient algorithm. Specifically, the algorithm should consume *O(1)* of memory. Make sure to test your code with the sample tree below.

To get you started:

class Node  
{  
 public Node[] Children;  
 public Node Right;  
}  
  
Node rootNode;



**Question 1 Answer:**

**A backtracking (recursive) method can be used in this scenario.**

**Number: 1,2,3,… 28,29,30**

**Possible Combinations: 30 + 20 =50**

**30 + 19 + 1 =50**

**30 + 18 + 2 = 50**

**30 + 17 + 3 = 50**

**……**

class Program

{

/// <summary>

/// A simple example backtracking algorithem

/// Give you the nubmer from 1,2,3... to 30.

/// List all the combination the nubmer sum equals to 50

/// For example: 30, 20

/// 30, 19,1

/// ...

///

/// </summary>

static bool [] selected = new bool[31]; //hard code array to track the number selected for sum, set to true if selected, otherwise restore to false.

static void Main(string[] args)

{

for (int i = 0; i < 31; i++)

selected[i] = false;

ListAll(30, 50, 0);

Console.Read();

return;

}

static string ListAll(int endNum, int targetSum, int curSum)

{

for (int i = endNum; i >= 1; i--)

{

selected[i] = true;

int sum = curSum + i;

if (sum < targetSum)

{

ListAll(i - 1, targetSum, sum);

selected[i] = false;

}

else if (sum == targetSum)

{

for (int j = 30; j >=1; j--)

{

if (selected[j] == true)

Console.Write(j + " ");

}

selected[i] = false;

// Console.Read();

Console.WriteLine();

}

else if (sum > targetSum)

selected[i] = false;

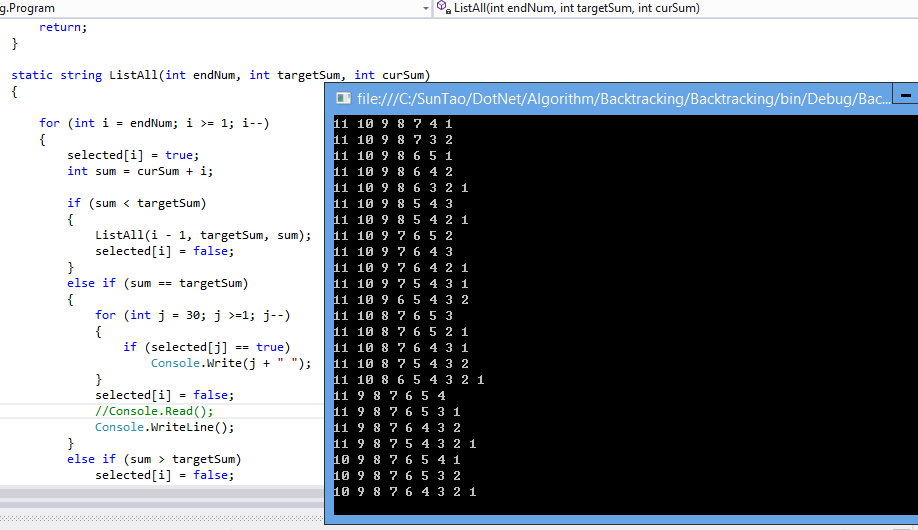
}

return "Done";

}

}

**The result printed on my machine is as follows:**



**Question 2: Finding the best plan**

PlanSearch**. GetLowCostPlan () return the results.**

namespace Backtracking

{

public class Feature

{

public string Name;

public Feature(string name)

{

Name = name;

}

}

public class Plan

{

public string Name;

public double Cost;

public List<Feature> Features;

public Plan(string name, double cost, List<Feature> lstFea)

{

Name = name;

Cost = cost;

Features = new List<Feature>();

foreach (var f in lstFea)

Features.Add(f);

}

}

public class PlanSearch

{

private List<Plan> AllPlans; // all plans

private List<Feature> SelectedFeatures; // all features user selected

private List<Plan> SelectedPlans; // trace the plan which has user's feature

private List<Plan> FinalBestPlans; // store the lowest cost plans

//constructor

public PlanSearch(List<Plan> allPlans, List<Feature> selectedFeatures)

{

AllPlans = new List<Plan>();

AllPlans = allPlans;

SelectedFeatures = new List<Feature>();

SelectedFeatures = selectedFeatures;

SelectedPlans = new List<Plan>();

FinalBestPlans = new List<Plan>();

}

public List<Plan> **GetLowCostPlan**()

{

return(GetLowCostPlan(AllPlans, SelectedFeatures));

}

private List<Plan> GetLowCostPlan(List<Plan> allPlans, List<Feature> selectedFeatures)

{

List<Plan> usedPlans = new List<Plan>();

foreach (var p in allPlans)

{

IEnumerable<Feature> intersectFeatures = p.Features.Intersect(selectedFeatures); //search plan

if (intersectFeatures != null)

{

if (intersectFeatures.Any())

{

SelectedPlans.Add(p); //current plan has user's features

}

}

List<Plan> remainPlans = new List<Plan>();

List<Feature> remainFeatures = new List<Feature>();

foreach (var f in (selectedFeatures).Except(p.Features)) //get remained features

remainFeatures.Add(f);

usedPlans.Add(p);

foreach (var rp in allPlans) // get remain plans

{

if (!usedPlans.Contains(rp))

remainPlans.Add(rp);

}

if (remainFeatures.Count == 0) //find a combination, update the result if cost is less

{

PrintPlans(SelectedPlans); //for troubleshooting, print out every details

SetBestPlans(SelectedPlans);

SelectedPlans.Remove(p);

}

else if (remainFeatures.Count > 0 && remainPlans.Count > 0) //continue to search in remained plans,

{

GetLowCostPlan(remainPlans, remainFeatures); //recursive

SelectedPlans.Remove(p);

}

else

SelectedPlans.Remove(p);

}

return FinalBestPlans;

}

private void SetBestPlans(List<Plan> plans)

{

double curCost = 0.0;

double targetCost;

foreach (var p in plans)

curCost = curCost + p.Cost;

if (FinalBestPlans.Count > 0)

{

targetCost = 0.0;

foreach (var p in FinalBestPlans)

targetCost = targetCost + p.Cost;

}

else

{

targetCost = double.MaxValue;

}

if (curCost <= targetCost)

{

FinalBestPlans.Clear();

foreach (var p in plans)

FinalBestPlans.Add(p);

}

}

public void PrintBestPlan()

{

double price = 0.0;

foreach (var p in FinalBestPlans)

{

price = price + p.Cost;

Console.Write(p.Name + " ");

}

Console.Write(" The best plans price is: " + price.ToString());

Console.WriteLine();

}

private void PrintPlans(List<Plan> plans)

{

double price = 0.0;

foreach(var p in plans)

{

price = price + p.Cost;

Console.Write( p.Name + " ");

}

Console.Write(" Qualified plans price is: " + price.ToString());

Console.WriteLine();

}

}

}

**Test cases:**

**…**

//Construct one test case:

Feature [] features = new Feature[11];

features[0] =new Feature("Email");

features[1] =new Feature("Sharepoint");

features[2] =new Feature("OneDrive");

features[3] =new Feature("Lync");

features[4] =new Feature("Skype");

features[5] =new Feature("Word");

features[6] =new Feature("Excel");

features[7] =new Feature("Powerpoint");

features[8] =new Feature("Access");

features[9] =new Feature("Support");

features[10] = new Feature("Warranty");

List<Plan> plans = new List<Plan>();

var tmp = new List<Feature>();

tmp.Add(features[1]); tmp.Add(features[2]);tmp.Add(features[3]);

plans.Add(new Plan("Plan0", 400, tmp));

tmp.Clear(); tmp.Add(features[1]);

plans.Add(new Plan("Plan1", 100, tmp));

tmp.Clear(); tmp.Add(features[3]);

plans.Add(new Plan("Plan2", 100, tmp));

tmp.Clear(); tmp.Add(features[3]); tmp.Add(features[9]);

plans.Add(new Plan("Plan3", 100, tmp));

tmp.Clear(); tmp.Add(features[2]); tmp.Add(features[8]);

plans.Add(new Plan("Plan4", 100,tmp));

tmp.Clear(); tmp.Add(features[2]); tmp.Add(features[3]); tmp.Add(features[7]);

plans.Add(new Plan("Plan5", 800, tmp));

tmp.Clear(); tmp.Add(features[1]); tmp.Add(features[3]); tmp.Add(features[7]);

plans.Add(new Plan("Plan6", 1800, tmp));

List<Feature> selectedFeatures = new List<Feature>();

selectedFeatures.Add(features[1]);

selectedFeatures.Add(features[2]);

selectedFeatures.Add(features[3]);

var p = new PlanSearch(plans, selectedFeatures); //plan research

var ret = p.GetLowCostPlan();

double cost = 0.0;

Console.WriteLine();

foreach (var el in ret)

{

cost = cost + el.Cost;

Console.Write(el.Name + " ");

}

Console.Write(" The best plans price: " + cost.ToString());

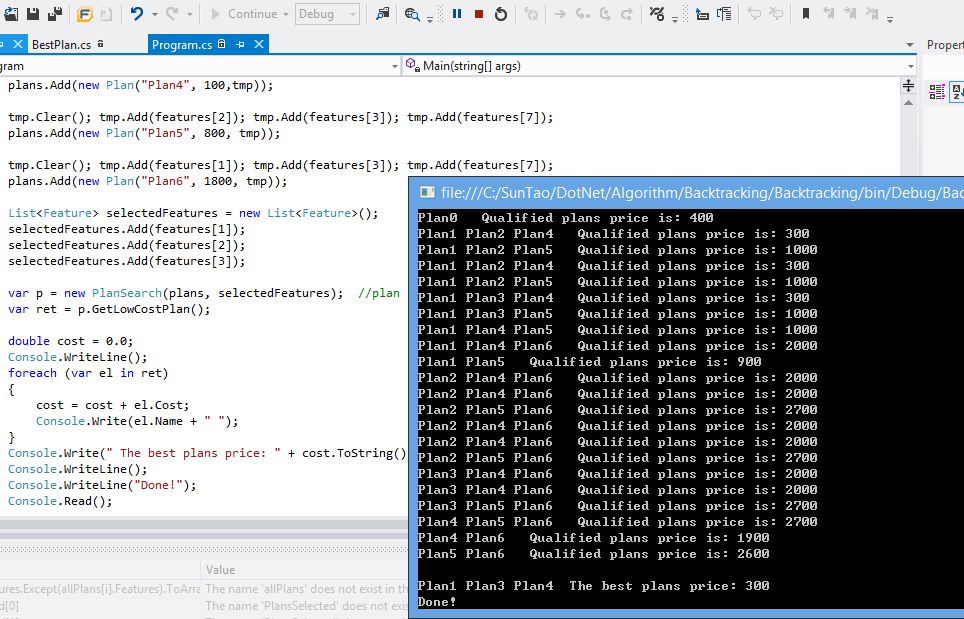
Console.WriteLine();

Console.WriteLine("Done!");

Console.Read();

…

**Test results:**



**Question 3 - Linking nodes in a tree**

**Removed the previous binary tree sibling search code and multiple tree prototype. The attached code** **snippet tested.**

**Construct a test case:**

MultipleTree tree = new MultipleTree();

int data = 10;

int[] ary = new int[6]{1,10,20,30,50,60};

tree.InsertNode(data, ary);

data = 5;

ary = new int[3] { 7, 8, 9 };

tree.InsertNode(data, ary);

data = 55;

ary = new int[4] { 100, 200, 300,400 };

tree.InsertNode(data, ary);

tree.SetRightSiblings(tree.Root);

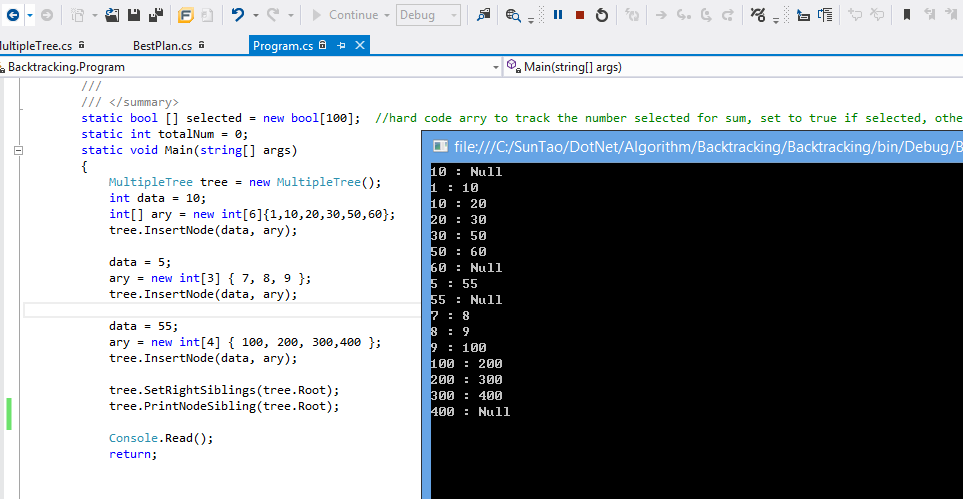
tree.PrintNodeSibling (tree.Root);

Console.Read();

**Above test code constructed a tree as below:**



**Tree traversal results:**



**The whole MultipleTree class implementation are as follows:**

namespace Backtracking

{

public class Node

{

public int Data { get; set; }

public Node[] Children { get; set; }

public Node Right { get; set; }

public Node(int data)

{

this.Data = data;

this.Children = null;

this.Right = null;

}

}

public class MultipleTree

{

public Node Root { get; set; }

public MultipleTree()

{

}

//Set node's right value

public void SetRightSiblings(Node node)

{

Queue<Node> q = new Queue<Node>();

q.Enqueue(node);

Node previous = null;

Node current = null;

while (q.Count > 0)

{

current = q.Dequeue();

if (current.Children != null)

{

for (int i = 0; i < current.Children.Length; i++)

{

q.Enqueue(current.Children[i]);

}

}

if (previous != null && previous.Data < current.Data)

{

previous.Right = current;

//Console.WriteLine(previous.Data + " : " + previous.Right.Data);

}

previous = current;

}

}

//Queue tree traversal

public void PrintNodeSibling(Node node)

{

Queue<Node> q = new Queue<Node>();

q.Enqueue(node);

Node current = null;

while (q.Count > 0)

{

current = q.Dequeue();

if (current.Children != null)

{

for (int i = 0; i < current.Children.Length; i++)

{

q.Enqueue(current.Children[i]);

}

}

if (current != null && current.Right != null)

Console.WriteLine(current.Data + " : " + current.Right.Data);

else

Console.WriteLine(current.Data + " : Null");

}

}

// Recursively tree traversal

public void PrintNodeSibling(Node node, bool recursive)

{

if(node != null)

{

if (node.Right != null)

Console.WriteLine(node.Data + ":" + node.Right.Data);

else

Console.WriteLine(node.Data + ": null");

if(node.Children != null)

{

for (int i = 0; i < node.Children.Length; i++)

PrintNodeSibling(node.Children[i]);

}

}

}

/// <summary>

/// used to contruct a multiple tree

/// </summary>

/// <param name="data"></param>

/// <param name="children"></param>

public void InsertNode(int data, int[] children)

{

Node node = CreateNode(data, children);

if (Root == null)

{

Root = node;

}

else

{

Node current = Root;

while (current.Children != null)

{

Node[] aryNodes = current.Children;

for (int i = 0; i < aryNodes.Length - 1; i++)

{

if (data <= aryNodes[0].Data)

{

current = aryNodes[0];

break;

}

if (data >= aryNodes[aryNodes.Length-1].Data)

{

current = aryNodes[aryNodes.Length-1];

break;

}

if (aryNodes[i].Data <= data && data < aryNodes[i + 1].Data)

{

current = aryNodes[i];

break;

}

}

}

current.Children = new Node[1];

current.Children[0] = node;

}

}

private Node CreateNode(int data, int[] children)

{

Node node = new Node(data);

Array.Sort(children);

if (children != null && children.Length > 0)

{

node.Children = new Node[children.Length];

for (int i = 0; i < children.Length; i++)

{

node.Children[i] = new Node(children[i]);

}

}

return node;

}

}

}