数据结构与算法
DATA STRUCTURE

第二十四讲 贪婪算法和最小生成树

胡浩栋

信息管理与工程学院

2018 - 2019 第一学期

课堂内容

- 作业回顾
- 贪婪算法
- 最小生成树

1. 用queue实现stack

```cpp
#include <queue>
using namespace std;

class IntStack
{
public:
    IntStack() {};

    inline bool IsEmpty() {return _data.size() <= 0;}

    int Peek()
    {
        int top = Pop();
        Push(top);
        return top;
    }

    void Push(int i)
    {
        _data.push(i);
    }

    int Pop();

 private:
     std::queue<int> _data;
};
```

```cpp
int IntStack::Pop()
{
    int sz = _data.size();
    assert(sz > 0);

    while (sz-- > 1)
    {
        _data.push(_data.front());
        _data.pop();
    }

    int top = _data.front();
    _data.pop();
    return top;
}
```

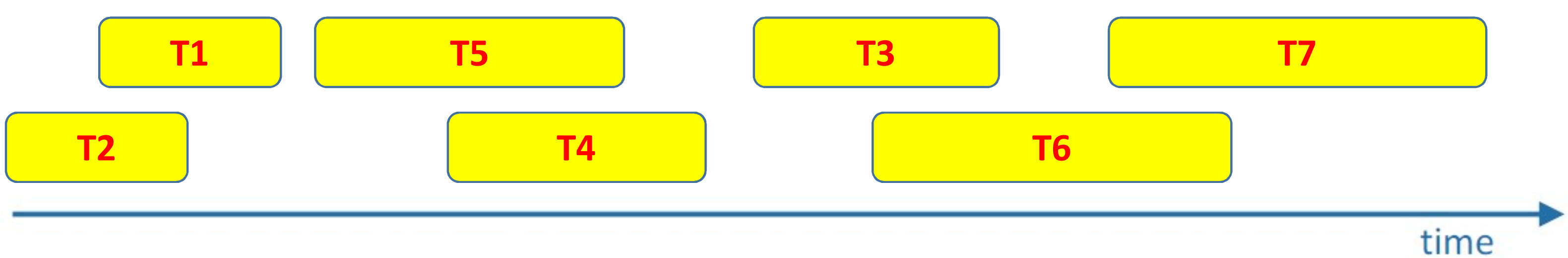
贪婪算法
Greedy Algorithm

任务安排问题

- 一天内有一批任务，每个任务都有起始/结束时间
- 假如同一时间只能做一个任务
- 如何安排这些任务，使得完成的任务数量最多

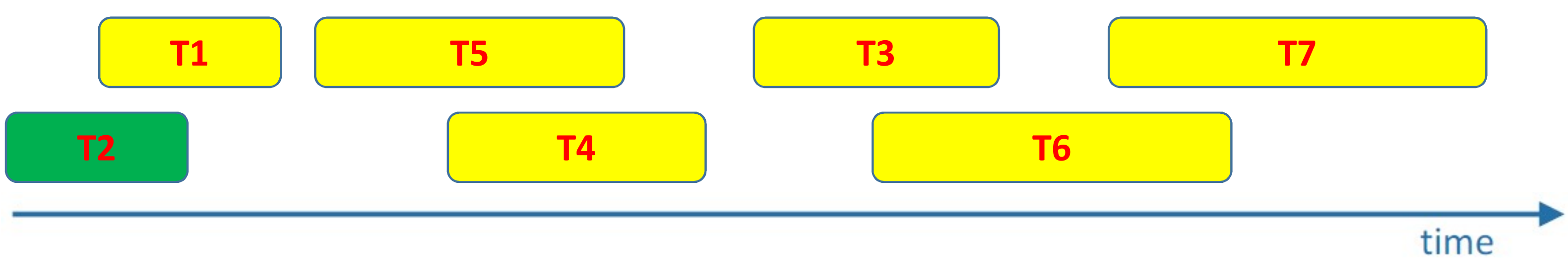
贪婪算法

- 对任务按结束时间排序
- 挑一个结束时间最早的任务
  - 和之前选好的任务没有冲突
- 重复上一步骤



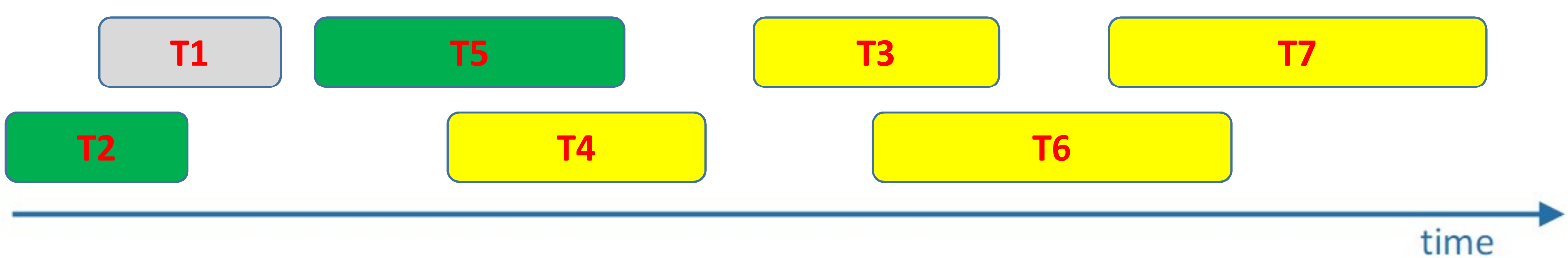
贪婪算法

- 对任务按结束时间排序
- 挑一个结束时间最早的任务
  - 和之前任务没有重叠
- 重复上一步骤



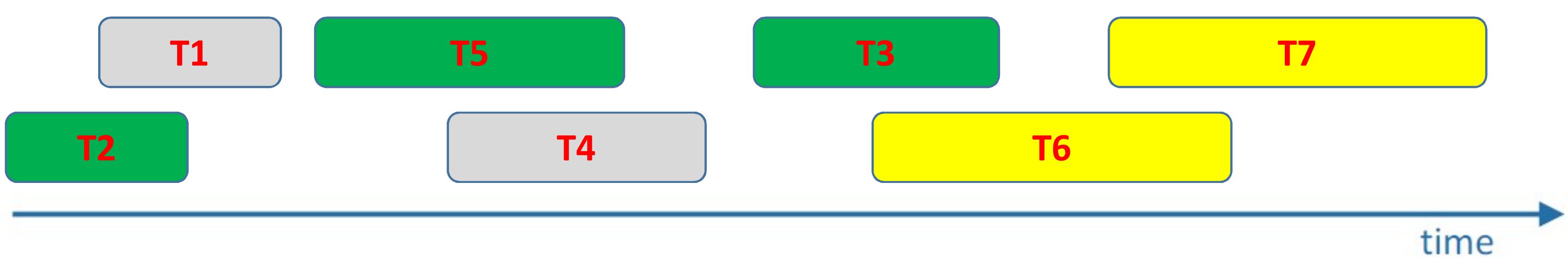
贪婪算法

- 对任务按结束时间排序
- 挑一个结束时间最早的任务
  - 和之前任务没有重叠
- 重复上面步骤



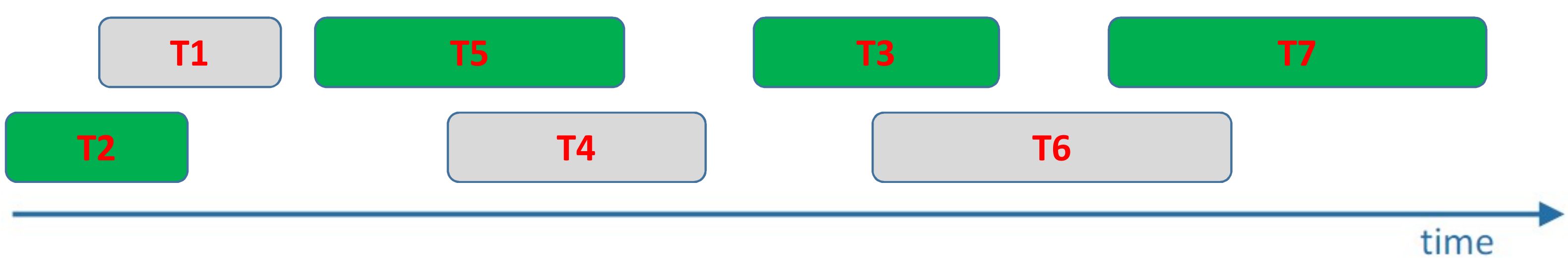
贪婪算法

- 对任务按结束时间排序
- 挑一个结束时间最早的任务
  - 和之前任务没有重叠
- 重复上面步骤



贪婪算法

- 对任务按结束时间排序
- 挑一个结束时间最早的任务
  - 和之前任务没有重叠
- 重复上面步骤



任务安排问题

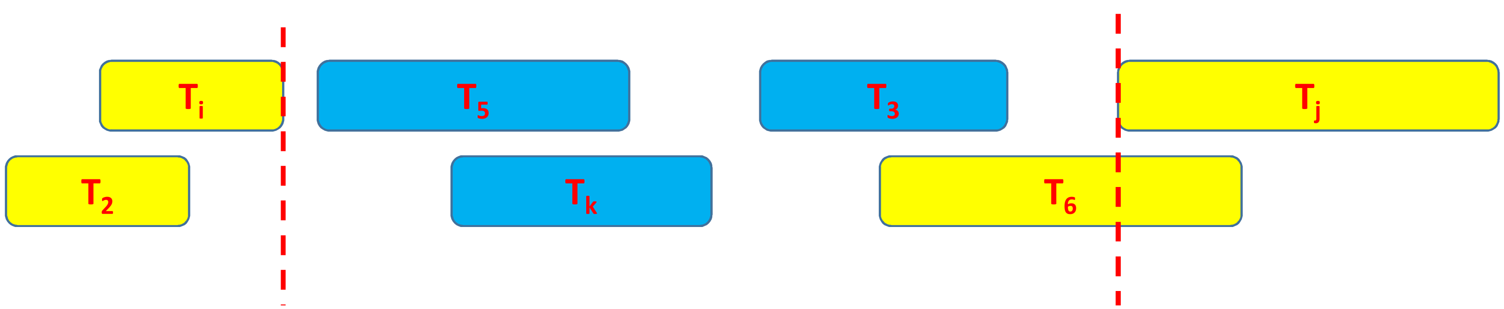
- 贪婪算法的时间复杂度
  - 排序需要$O(nlogn)$
  - 挑选只需要$O(n)$

- 能否证明这个贪婪算法是最优？
  - 即没有别的挑法能选出更多的任务来
  - 或者证明存在某个最优解，使得当前的选择是其子集

- 比较DP方法

用DP方式分解原问题

- 子问题：令$C[i,j]$是任务T_i结束后，并且任务T_j开始前，最多能安排的任务数
- 那么递归表达式是$C[i,j] = \max_k \{ C[i,k] + 1 + C[k,j] \}$
- 这里$\boldsymbol{T_k}$是所有介于T_i和T_j之间的任务



贪婪算法分解原问题

- 每一步，我们都选出了一个任务（结束时间最小）
- 最多n步后，能得到最终解
- 贪婪算法更优（如果结果是最优解）

贪婪算法的思路

- 把原问题分解成子问题
- 原问题只依赖于一个子问题

- 证明贪婪算法就是最优解，一般比较复杂
    - 需要证明一个子问题的最优解可以推出原问题的最优解
    - 或者证明每一步选择，都存在一个最优解，包含之前的选择

贪婪算法特点

- 每一步，都做了当前最“好”的选择，局部最优
- 最终结果不一定全局最优，比如
  - 旅行商问题(近似算法)

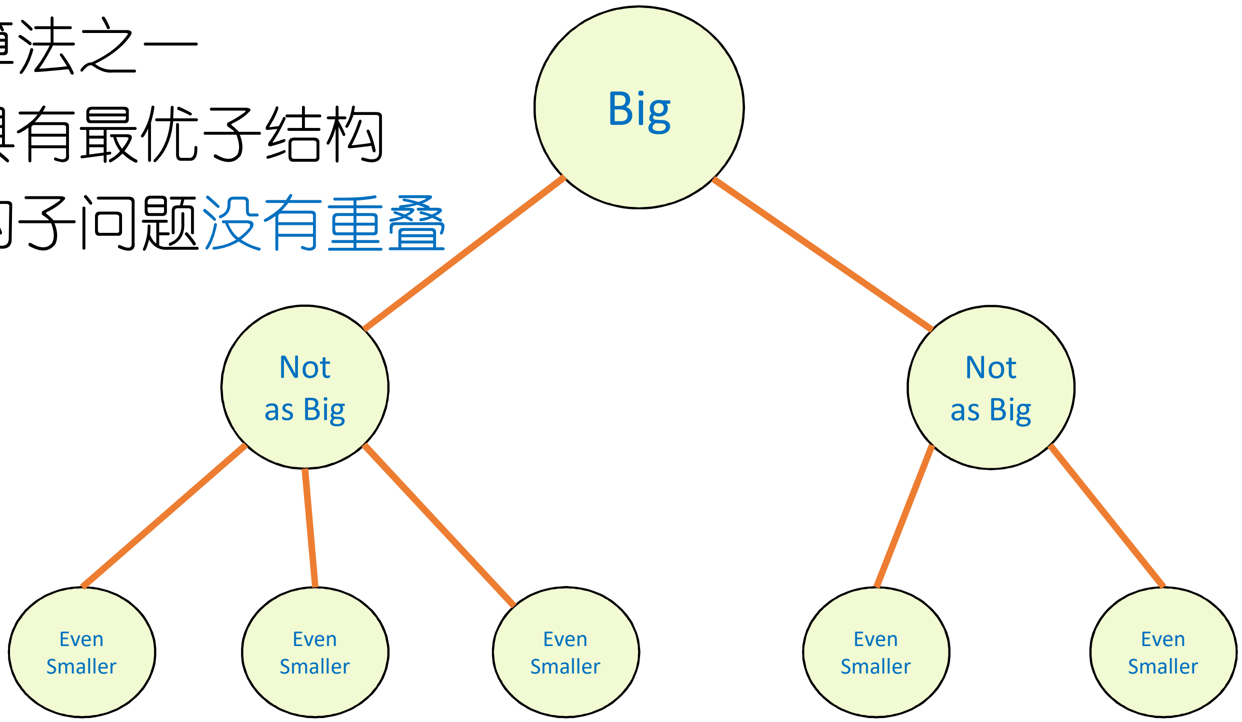
- 有些问题可以全局最优，比如
  - 任务安排
  - Huffman coding
  - 图里的最小生成树MST
  - 等等

解决问题思路

- 分治法
- 动态规划
- 贪婪算法

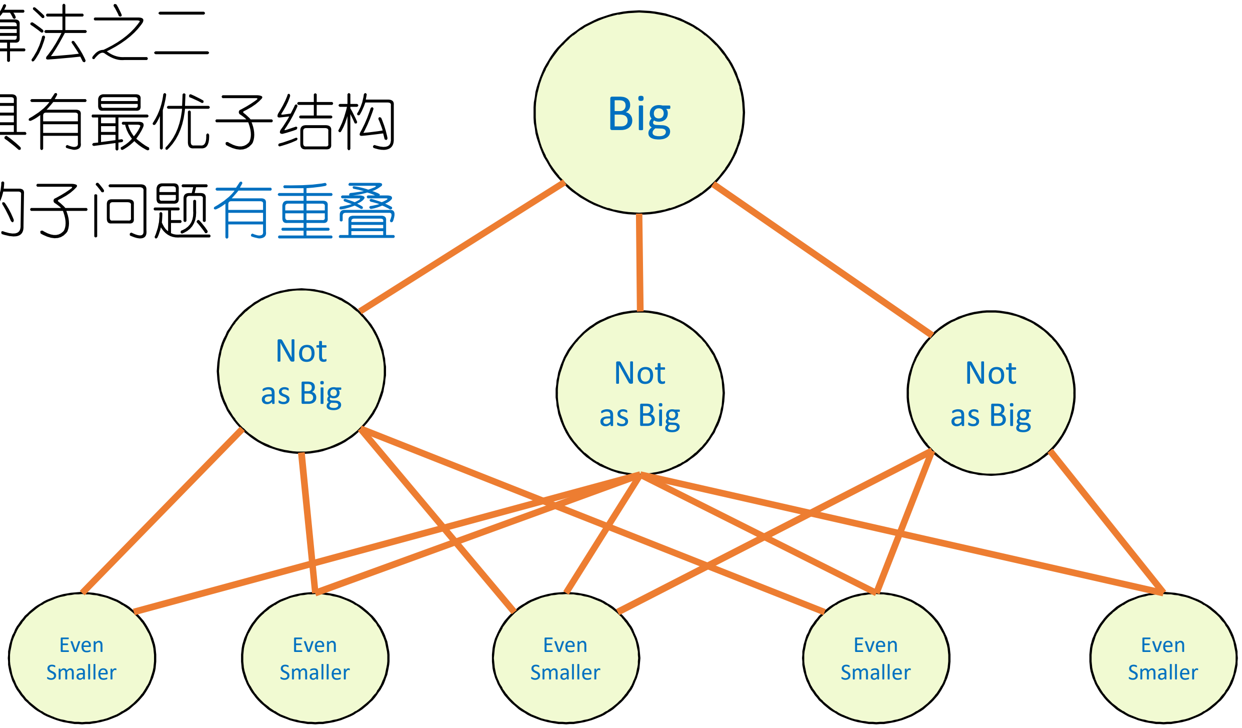
分治法 Divide and Conquer

- 基础算法之一
- 问题具有最优子结构
- 依赖的子问题没有重叠



动态规划 Dynamic Programming

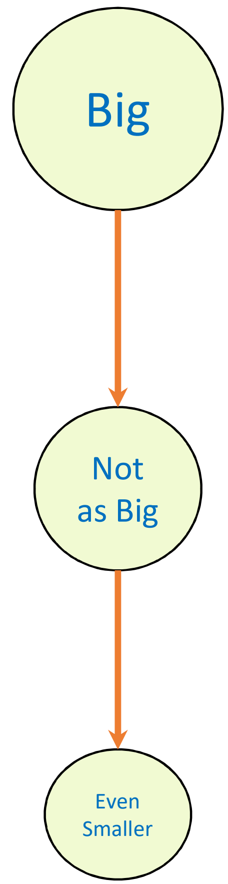
- 基础算法之二
- 问题具有最优子结构
- 依赖的子问题有重叠



贪婪算法

- 基础算法之三
- 问题具有最优子结构
- 原问题只依赖于一个子问题
- 即一个子问题能组成原问题的最优解

- 效率最高
- 如何每一步的选择
- 证明最优解相对最难



小结

- 解决问题的方法是同一个
- 都是分析原问题，并把大问题分解
- 同一个问题可以不同方式分解
- 子问题划分的好坏，决定了最后效率的高低

霍夫曼树回顾

Huffman Tree

问题

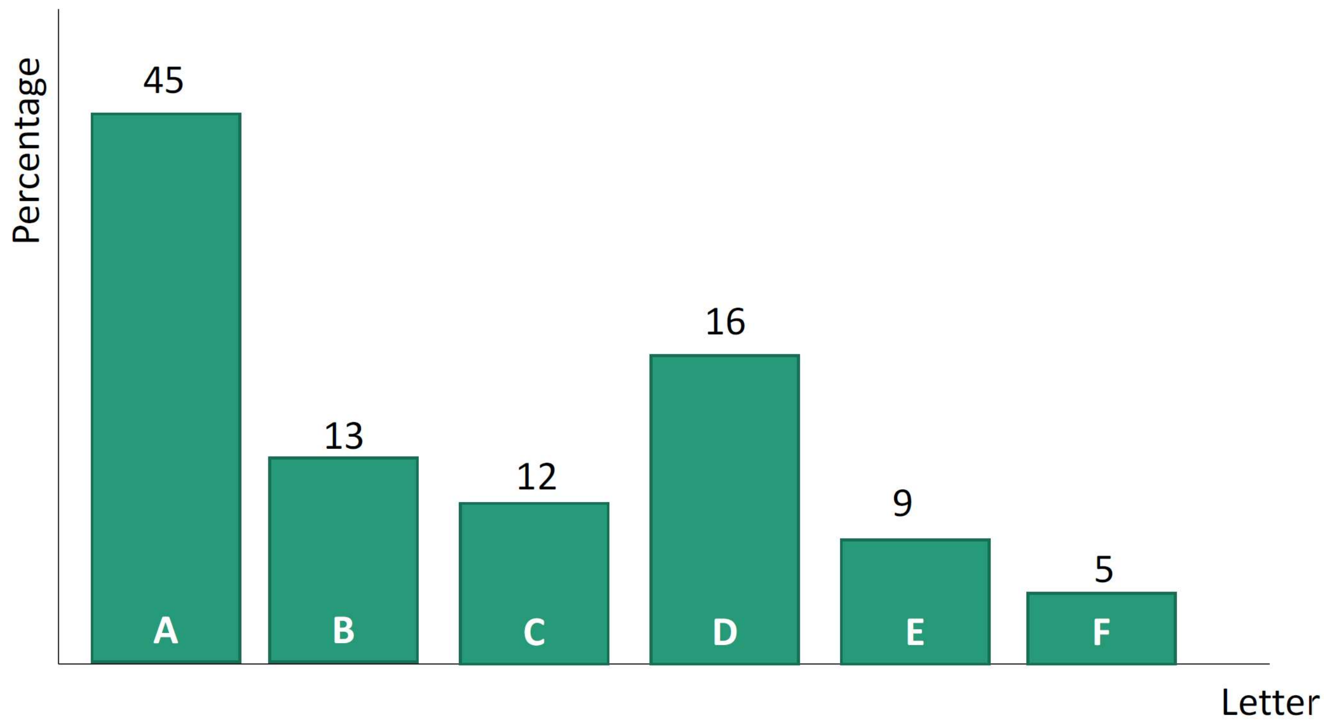
- 对文本文件怎么压缩保存，比如基因序列
- 特点：不同字符个数比较少（ACGT）

- ASCII字符一个字节，比较浪费空间

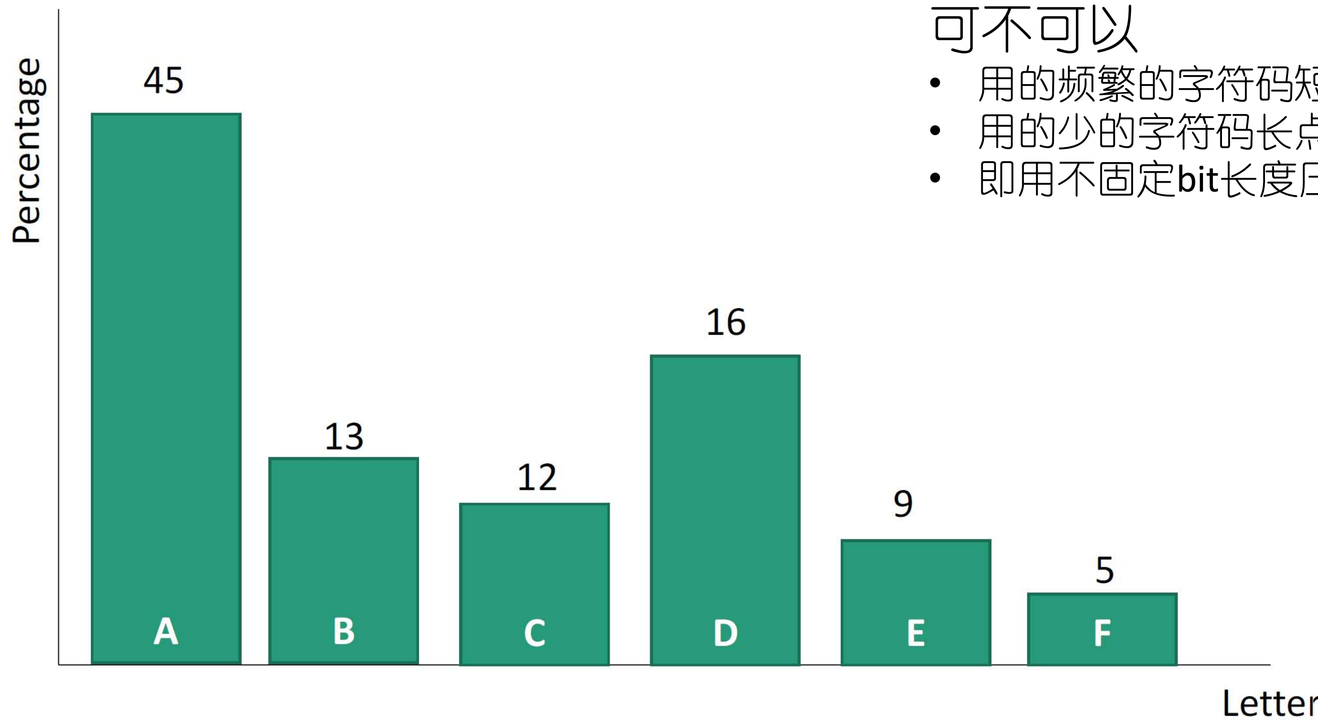
01100101 01110110 01100101 01110010 01111001 01100101 01100001 01111001 00100000 01100101 01101110 01100101 01101100 01100101 01110011 01101000 01100101 01110011 01100101 01101110 01110100 01100101 01101110 01100011 01100101

- 解码比较简单

假如我们知道用到的字符的分布



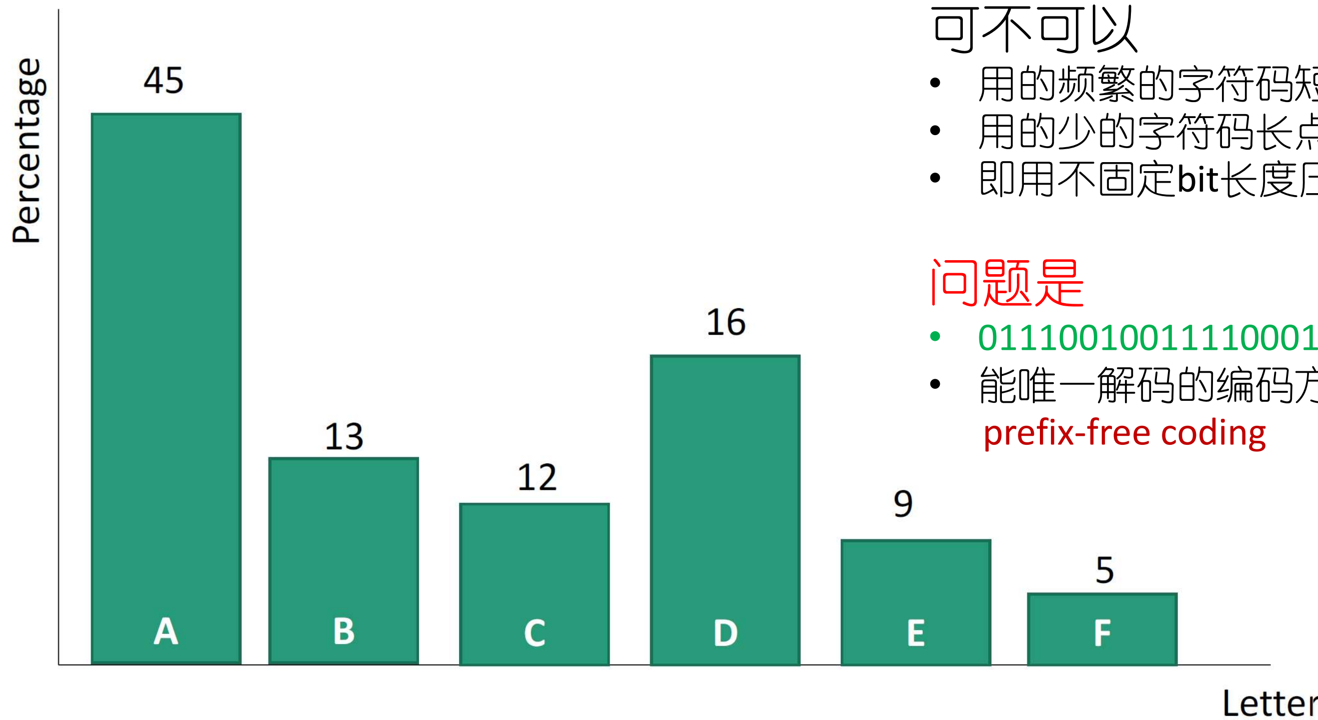
假如我们知道用到的字符的分布



可不可以

- 用的频繁的字符码短点
- 用的少的字符码长点
- 即用不固定bit长度压缩

假如我们知道用到的字符的分布



可不可以

- 用的频繁的字符码短点
- 用的少的字符码长点
- 即用不固定bit长度压缩

问题是

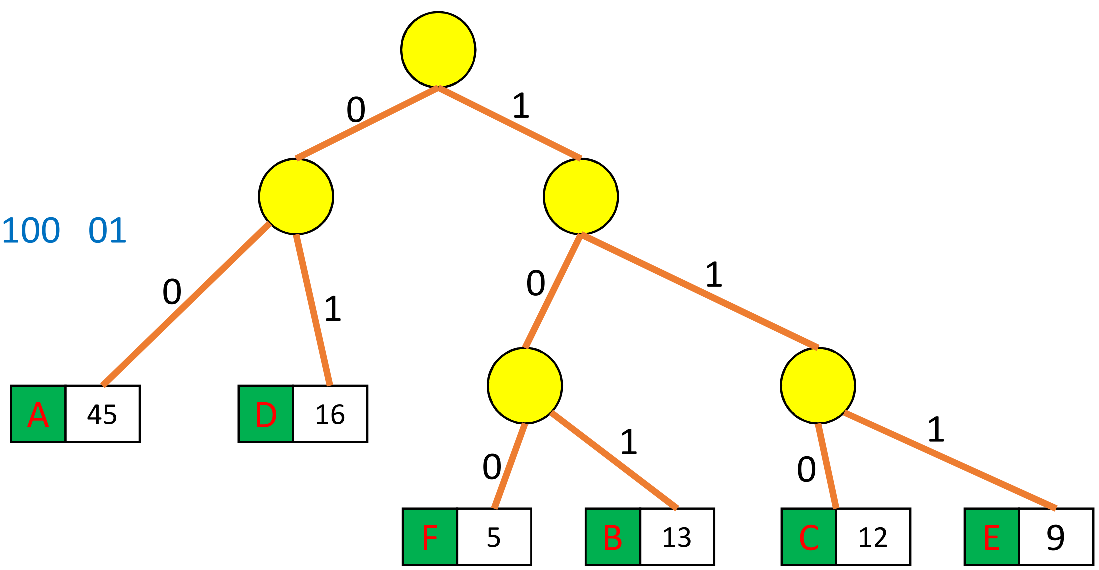
- 01110010011110001怎么解码？
- 能唯一解码的编码方式叫 prefix-free coding

Prefix-free coding

- 一个字符代表的码不是另一个字符代表的码的前缀
- 那么在所有字符码生成的Trie
- 只要字符是在Trie的叶节点，就是prefix-free coding
- 可以唯一解码

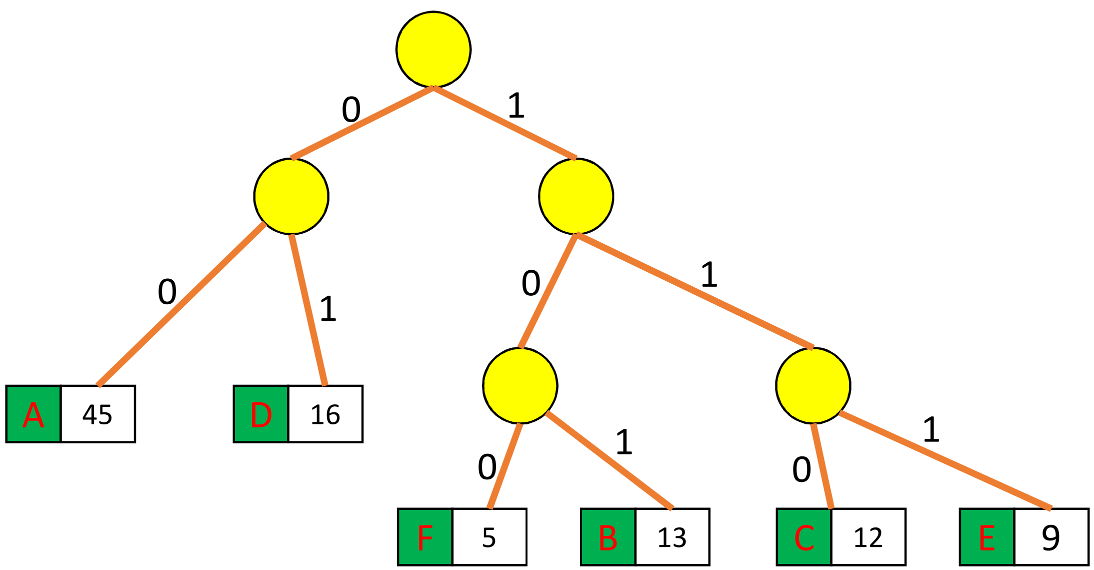
01110010011110001

01 110 01 00 111 100 01



问题转化成

- 找一种prefix-free coding，使得在给定字符使用率下，压缩率最高
  - 编码：根节点到字符叶节点路径对应的码
  - 解码：用同样的Trie对二进制码翻译
- 用数学公式表示，是最小化 Cost $= \sum_x \boldsymbol{P(x)} \times \boldsymbol{depth(x)}$



首先，直接结论

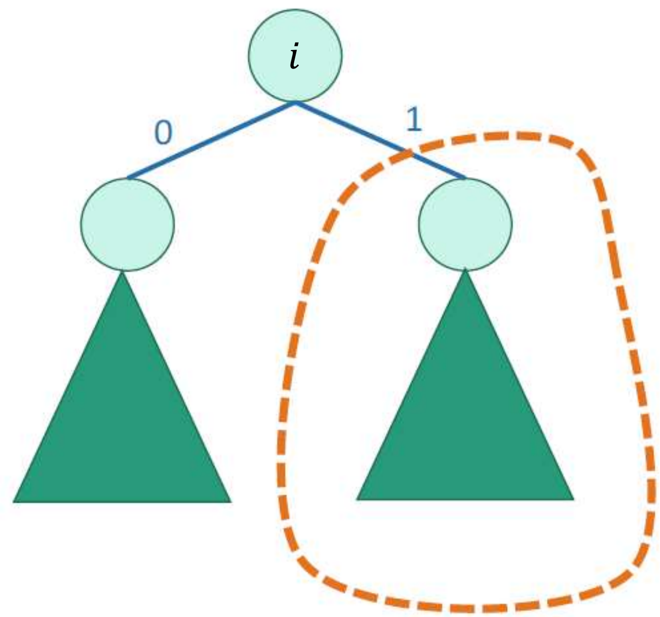
- 最优的prefix-free coding生成的树一定是满二叉树
  - 反证法：如果某个内部节点只有一个子节点，那么我们可以调整相应的叶节点，使整体Cost更小
- 假设满二叉树有n个叶节点，那么中间节点是$n-1$个
  - 数学归纳法
- n个叶节点的满二叉树有很多，我们要找Cost最小的编码树

DP方式求解

- 子树对应的字符集上找最优的编码树做为子问题

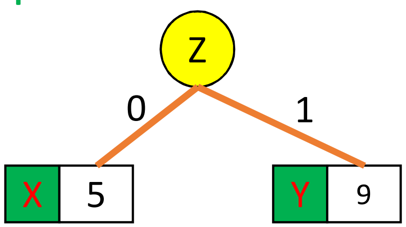
- 需要遍历根节点的可能性
- 需要遍历左右子树包含的字符集的可能性

- 复杂度高（即使改进）



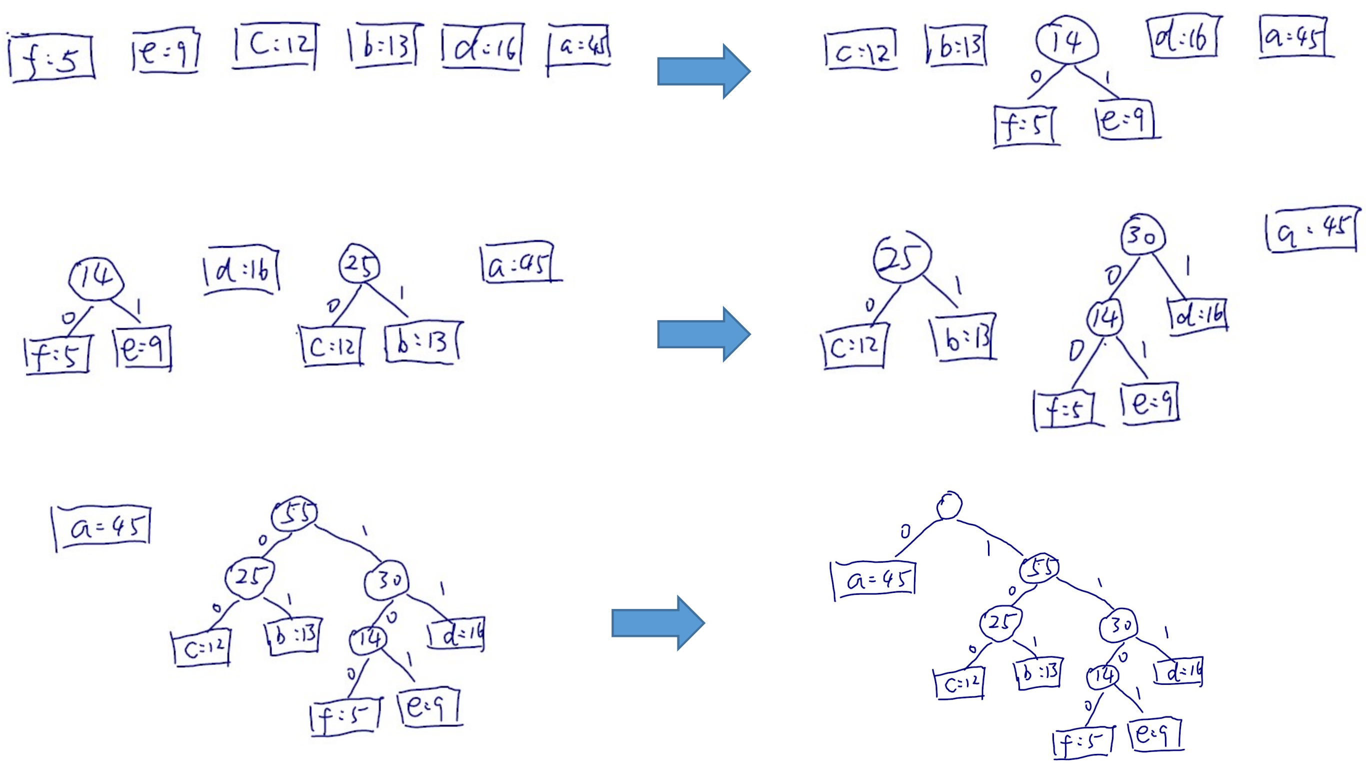
贪婪算法：Huffman coding算法

- 先对每个编码字符建立n个叶节点
- 对这n个节点按使用频率做为优先级建立最小堆
- **While** 最小堆里还有2个以上节点，
  1. 从堆里取出优先级最小的两个节点X， Y
  2. 建立一个新的节点Z， 使得
     - X， Y的父节点是Z，
     - Z的优先级是X和Y的优先级的和
  3. 然后把Z放入最小堆



- 每次循环， 都选出两个节点， 最小堆大小减一（子问题）
- 最后最小堆里的节点是huffman树的根节点

实例

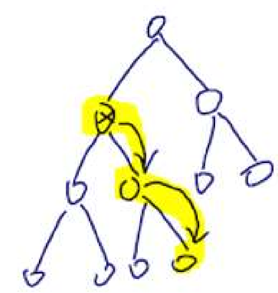


2.Huffman树

```cpp
class Heap
{
public:
    // add void element at index 0.
    Heap() { m_data.push_back(0); }

    // to extract the root which is the minimum element
    BinaryTree * ExtractMin();

    // Inserts a new key: pointer to BinaryTree
    void InsertKey(BinaryTree * nd);

    bool IsEmpty() { return m_data.size() <= 1; }

private:
    // A recursive method to heapify a subtree with root at given index
    // This method assumes that the subtrees are already heapified
    void MinHeapify(int i);

    int Parent(int i) { return i/2; }
    // to get index of left child of node at index i
    int Left(int i) { return 2*i; }
    // to get index of right child of node at index i
    int Right(int i) { return (2*i + 1); }

    std::vector<BinaryTree *> m_data;
};
```

```cpp
void Heap::MinHeapify(int i)
{
    int l = Left(i);
    int r = Right(i);
    int min = i;
    if (l < m_data.size() && m_data[l]->CompareRootKey(m_data[min]) < 0)
    {
        min = l;
    }
    if (r < m_data.size() && m_data[r]->CompareRootKey(m_data[min]) < 0)
    {
        min = r;
    }

    if (min != i)
    {
        std::swap(m_data[i], m_data[min]);
        MinHeapify(min);
    }
}
```

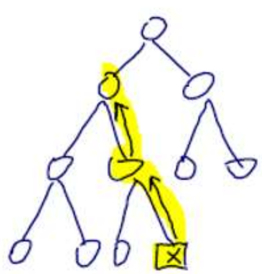
② min_heapify



2.Huffman树

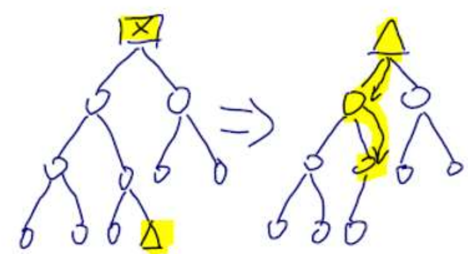
```
void Heap::InsertKey(BinaryTree * nd)
{
    // First insert the new key at the end
    m_data.push_back(nd);
    int i = m_data.size() - 1;

    // Fix the min heap property if it is violated
    while (i > 1 && m_data[Parent(i)]->CompareRootKey(m_data[i]) > 0)
    {
       std::swap(m_data[i], m_data[Parent(i)]);
       i = Parent(i);
    }
}
```

① Insert



```
// to extract the root which is the minimum element
BinaryTree * Heap::ExtractMin()
{
    // data[0] is a place holder.
    if (m_data.size() <= 1)
    {
        return nullptr;
    }

    // Store the minimum value, and remove it from heap
    BinaryTree * root = m_data[1];
    m_data[1] = m_data.back();
    m_data.pop_back();

    if (m_data.size() > 2)
    {
        MinHeapify(1);
    }

    return root;
}
```

③ Extract min



2.Huffman树

```cpp
BinaryTree(char ch, int key) { _pRoot = new Node(key, ch); }
void HuffmanMerge(const BinaryTree * pChild);
int CompareRootKey(const BinaryTree * other)
{
    return _pRoot->key - other->_pRoot->key;
}
```

```cpp
void BinaryTree::HuffmanMerge(const BinaryTree * pChild)
{
    Node * newRoot = new Node(_pRoot->key + pChild->_pRoot->key);
    newRoot->left = _pRoot;
    newRoot->right = CopyTree(pChild->_pRoot);
    _pRoot = newRoot;
}
```

```cpp
int main()
{
    vector<char> alphabets = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'};
    vector<int> distributes = {18, 12, 14, 14, 2, 3, 11, 26};

    Heap pq;
    for (int i = 0; i < alphabets.size(); i++)
    {
        pq.InsertKey(new BinaryTree(alphabets[i], distributes[i]));
    }

    BinaryTree * root = nullptr;
    while (!pq.IsEmpty())
    {
        BinaryTree * top = pq.ExtractMin();
        if (root == nullptr)
        {
            root = top;
            continue;
        }
        root->HuffmanMerge(top);
        delete top;
        pq.InsertKey(root);
        root = nullptr;
    }

    root->LevelOrder();

    return 0;
}
```

2. Huffman树

```
string BinaryTree::Translate(string binary)
{
    string result;
    Node * nd = _pRoot;

    for (int i = 0; i < binary.size(); )
    {
        if (nd->alpha == 0)
        {
            if (binary[i] == '0')
            {
                nd = nd->left;
            }
            if (binary[i] == '1')
            {
                nd = nd->right;
            }
            i++;
        }

        if (nd->alpha > 0)
        {
            result.append(1, nd->alpha);
            nd = _pRoot;
        }
    }

    assert(nd == _pRoot);
    return result;
}
```

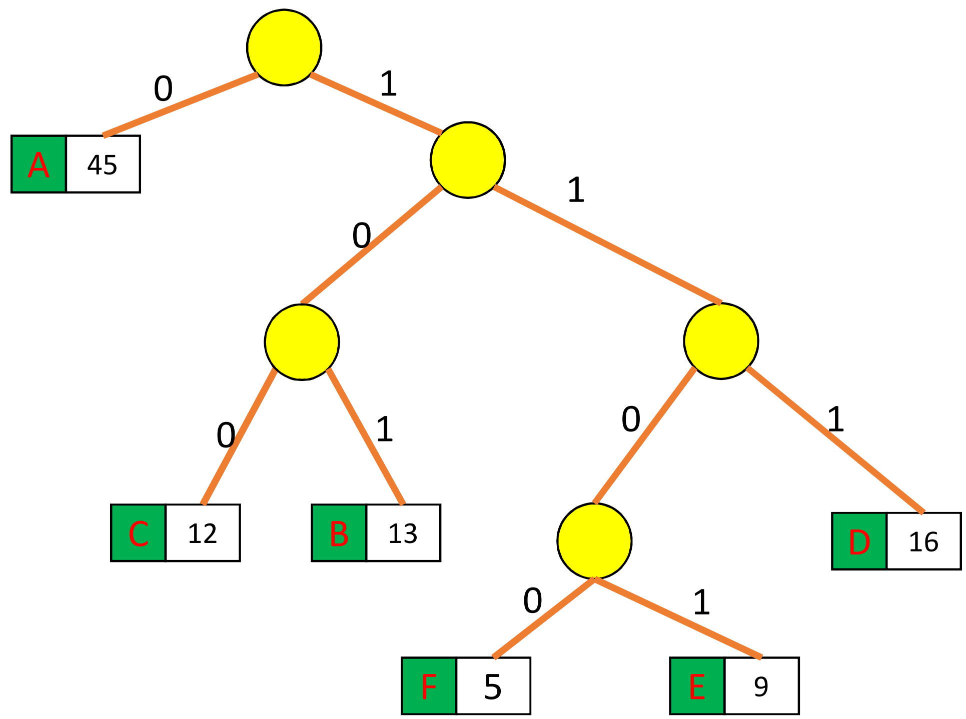
Huffman树

- 一种最优树，使得编码Cost最小

$$\text{Cost} = \sum_{x} \boldsymbol{P(x)} \times \boldsymbol{depth(x)}$$

$$\mathbf{1 * 0.45 + 3 * 0.41 + 4 * 0.14}$$

$$\mathbf{= 2.24}$$



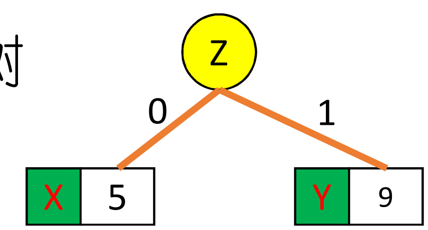
分解原问题

$$\text{Cost} = \sum_{x} \boldsymbol{P(x)} \times \boldsymbol{depth(x)}$$

按字符集数目分解
- 子问题：在k个字符集上找一个Cost最小的编码树
- 假如我们可以在$k-1$个字符集上找一个Cost最小的编码树

- Huffman算法的分解方式，
  - 可以在k字符集上选择频率最小的x和y，
  - 新建一个字符z替换x和y，其频率等于x+y，
  - 这样问题变成在$k-1$字符集，找最优编码树

- 根据归纳，我们可以找到$k-1$字符集的最优编码树
- 那么，把z替换为子树就是原问题的一个解



证明最优解

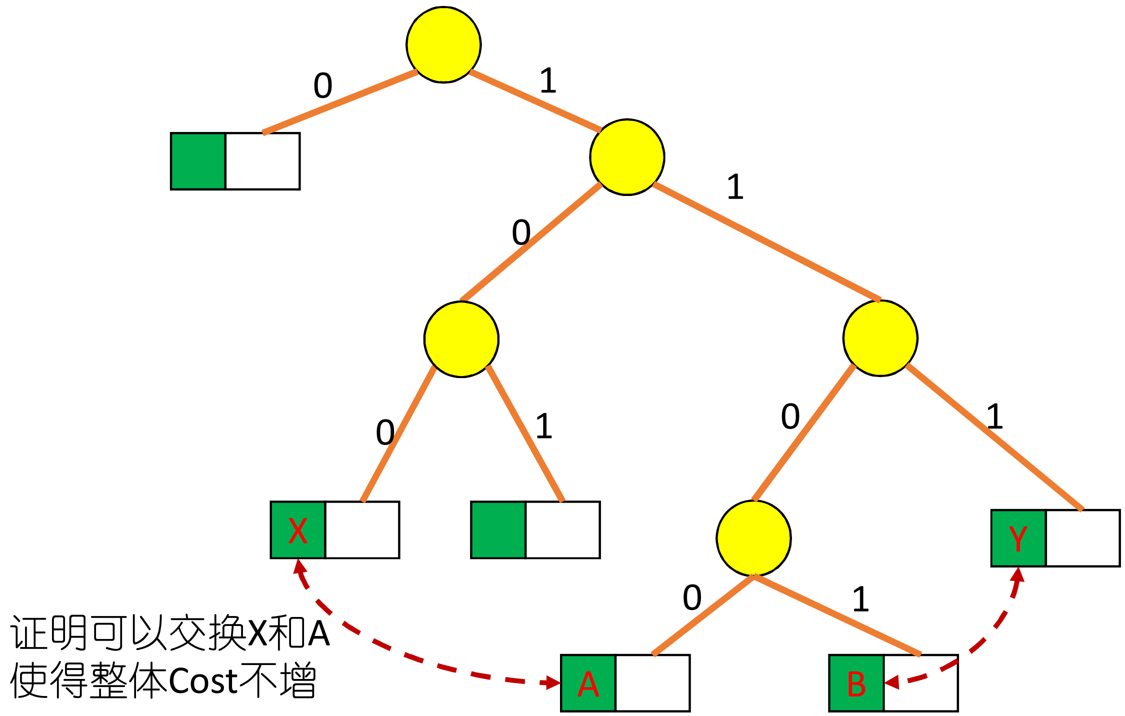
从子问题还原过来的解是最优解

- Lemma：如果在k个字符集上频率最小的x和y，那么存在一个最优解，使得x和y是兄弟节点
- Claim：在$k-1$个字符集上的最优解 + 把z替换成子树，同样是在k个字符集上的最优解

Lemma

- 如果在k个字符集上频率最小的x和y，那么存在一个最优解，使得x和y是兄弟节点
- 选最深的两个节点A和B

$$\text{Cost} = \sum_{x} \boldsymbol{P(x)} \times \boldsymbol{depth(x)}$$



Claim

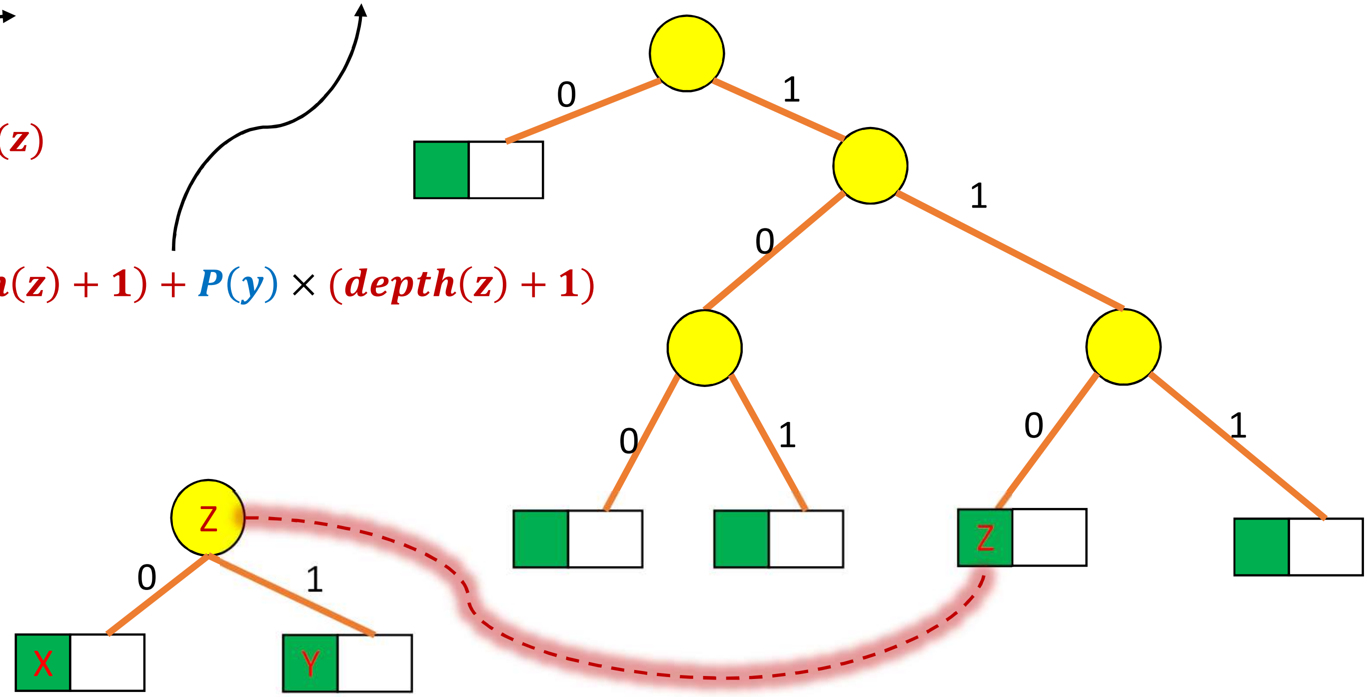
$$\text{Cost} = \sum_x \boldsymbol{P(x)} \times \boldsymbol{depth(x)}$$

- 在$k-1$个字符集上的最优解T_{k-1}，把z替换成子树，得到在k个字符集上的一颗编码树T_k，那么T_k也是最优的，且

$$\text{Cost}(T_{k-1}) = \text{Cost}(T_k) - P(x) - P(y)$$

$\boldsymbol{P(z)} \times \boldsymbol{depth(z)}$

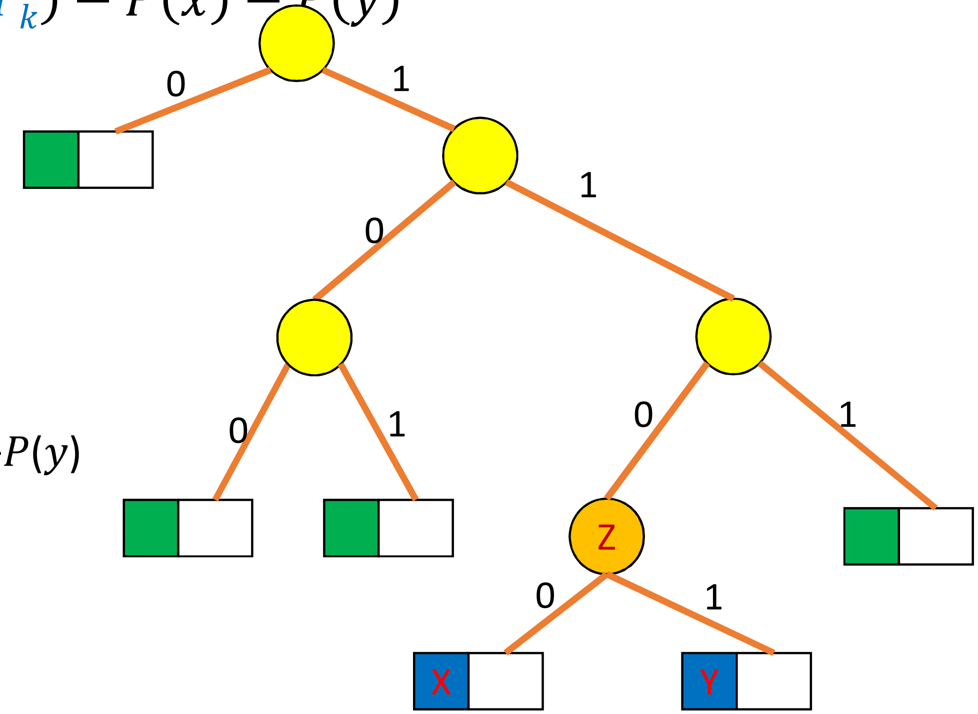
$\boldsymbol{P(x)} \times \boldsymbol{(depth(z)+1)} + \boldsymbol{P(y)} \times \boldsymbol{(depth(z)+1)}$



Claim

$$\text{Cost} = \sum_x P(x) \times depth(x)$$

- 在$k-1$个字符集上的最优解$T_{k\text{-}1}$，把z替换成子树，得到在k个字符集上的一颗编码树T_k，那么T_k也是最优的，且
$$\text{Cost}(T_{k\text{-}1}) = \text{Cost}(T_k) - P(x) - P(y)$$



- 假设T_k不是原问题的最优解
- 存在最优解T'_k，满足
  - x和y是兄弟节点
  - $\text{Cost}(T'_k) < \text{Cost}(T_k)$
- 然后我们把T'_k转换成子问题T'_{k-1}
  - 建一个父节点z替换，使其频率=$P(x)$+$P(y)$
- 那么$\text{Cost}(T'_{k-1}) = \text{Cost}(T'_k) - P(x) - P(y)$
  $< \text{Cost}(T_k) - P(x) - P(y) = \text{Cost}(T_{k\text{-}1})$

小结

- 原问题：在n个字符集上找最优编码树（prefix-free）
- 每一步都选择两个最“好”的节点，合并
- 依赖于一个子问题：在新$n-1$个字符集上找最优编码树
- 也就是说，如果子问题可解，原问题的解就是把合并节点拆开成子树