数据结构与算法分析

华中科技大学软件学院

2017年秋

Binary Tree Isomorphism

```
BOOL isIsomorphic (BinTreeNode *r1, BinTreeNode *r2)
{
    if (r1 == NULL && r2 == NULL)
        return (TRUE);

    if (r1 == NULL || r2 == NULL)
        return (FALSE);

    if (r1->value == r2->value)
    {
        if (((isIsomorphic (r1->left, r2->left) &&
                (isIsomorphic (r1->right, r2->right))
             || ((isIsomorphic (r1->left, r2->right) &&
                (isIsomorphic (r1->right, r2->left)))
        {
            return (TRUE);
        }
    }
    return (FALSE);
}
```

Binary Tree Isomorphism

```
BOOL isIsomorphic (BinTreeNode *r1, BinTreeNode *r2)
{
    if (r1 == NULL && r2 == NULL)
        return (TRUE);

    if (r1 == NULL || r2 == NULL)
        return (FALSE);

    if (r1->value != r2->value)
        return (FALSE);

    if (r1->left && r2->left &&  r1->left->value == r2->left->value)
    {
        return ((isIsomorphic (r1->left, r2->left) &&
                 (isIsomorphic (r1->right, r2->right)));
    }

    return ((isIsomorphic (r1->left, r2->right) &&
                 (isIsomorphic (r1->right, r2->left)));
}
```

大纲

课程计划

- 已经学习了
  - 算法时间复杂度及其分析
  - 线性表：堆栈、队列
  - 非线性数据结构：树、优先队列、图
  - 散列和排序算法

课程计划

- 已经学习了
  - 算法时间复杂度及其分析
  - 线性表：堆栈、队列
  - 非线性数据结构：树、优先队列、图
  - 散列和排序算法
- 即将学习算法设计思想
  - 贪婪算法
  - 分治算法
  - 动态规划
  - 回溯算法

Roadmap

Algorithm Design Techniques

- 用于解决问题的五种常用算法
- 这些方法中至少有一种很有可能对给定的问题起作用。
  - 贪婪算法
  - 分治算法
  - 动态规划
  - 随机化算法
  - 回溯算法

Greedy Algorithms

- 采用当下最好的步骤
- 贪婪算法分阶段工作。在每一个阶段， 做出一个看起来不错的决定，而不考虑将来的结果。
- 一般情况下，选择局部最优。 希望局部最优最终导致全局最优。
- 有时，贪婪算法只给出次优解。

Huffman Codes

假设我们有一个文件，其中只包含字符a, e, i, s, t, 加上空格和换行符。

Character	Code	Frequency	Total Bits
a	000	10	30
e	001	15	45
i	010	12	36
s	011	3	9
t	100	4	12
space	101	13	39
newline	110	1	3

Encoding

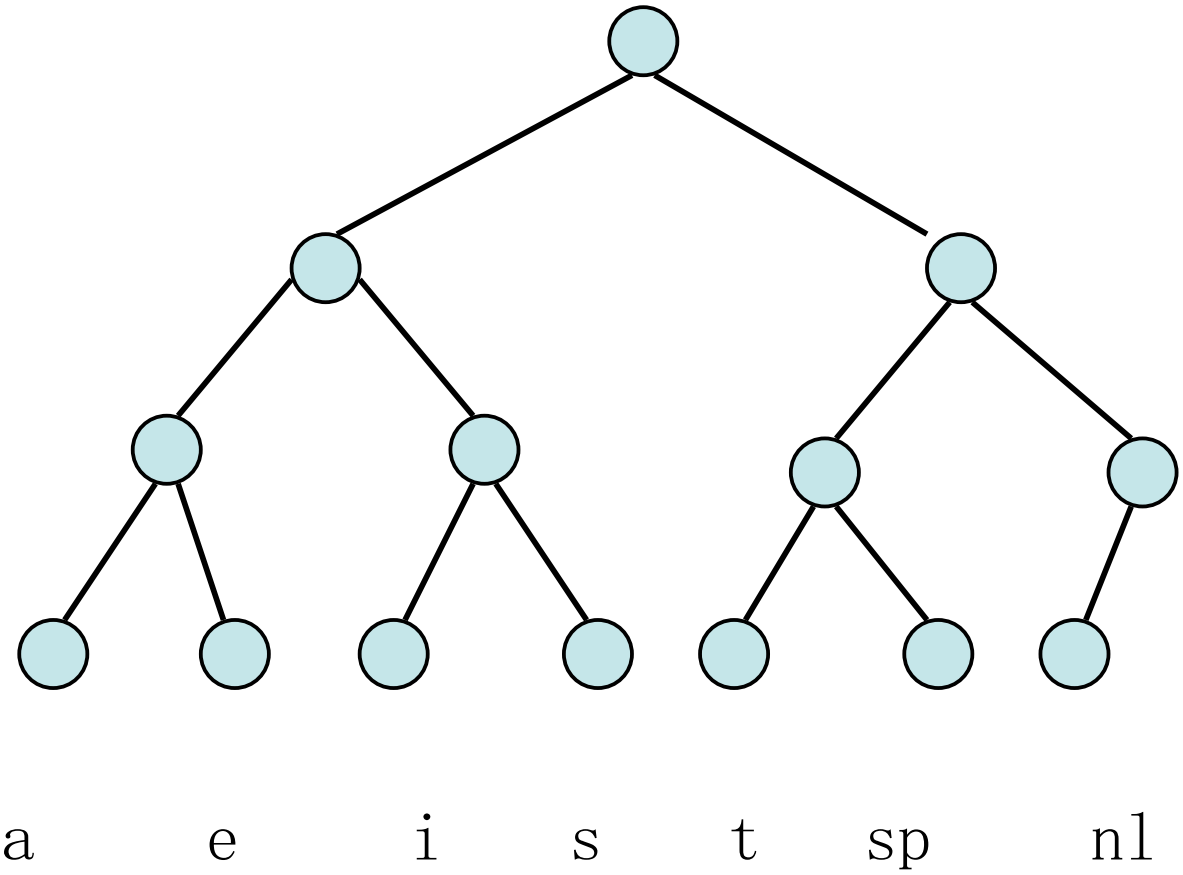
- 目标：在我们通过慢速电话线传输的情况下，减少文件大小。
- 在前面的示例中，总计有174 bits(58个符号)
- 一般策略：允许代码长度因字符而异， 并确保频繁出现的字符具有短代码。
- 如果所有字符都以相同的频率出现， 则不太可能节省任何费用。

Huffman Codes

- Developed by David Huffman while a Ph.D. student at MIT, published in the 1952 paper A Method for the Construction of Minimum-Redundancy Codes
- 给定一组符号和权重，找到一组无前缀二进制码(一组码字)，其最小期望码字长
- Shannon源编码定理：符号的最佳编码长度是 $-\log_b P$
- 信息熵 $H = -\sum_i P_i \log P_i$ 给出了期望码字长度的下限

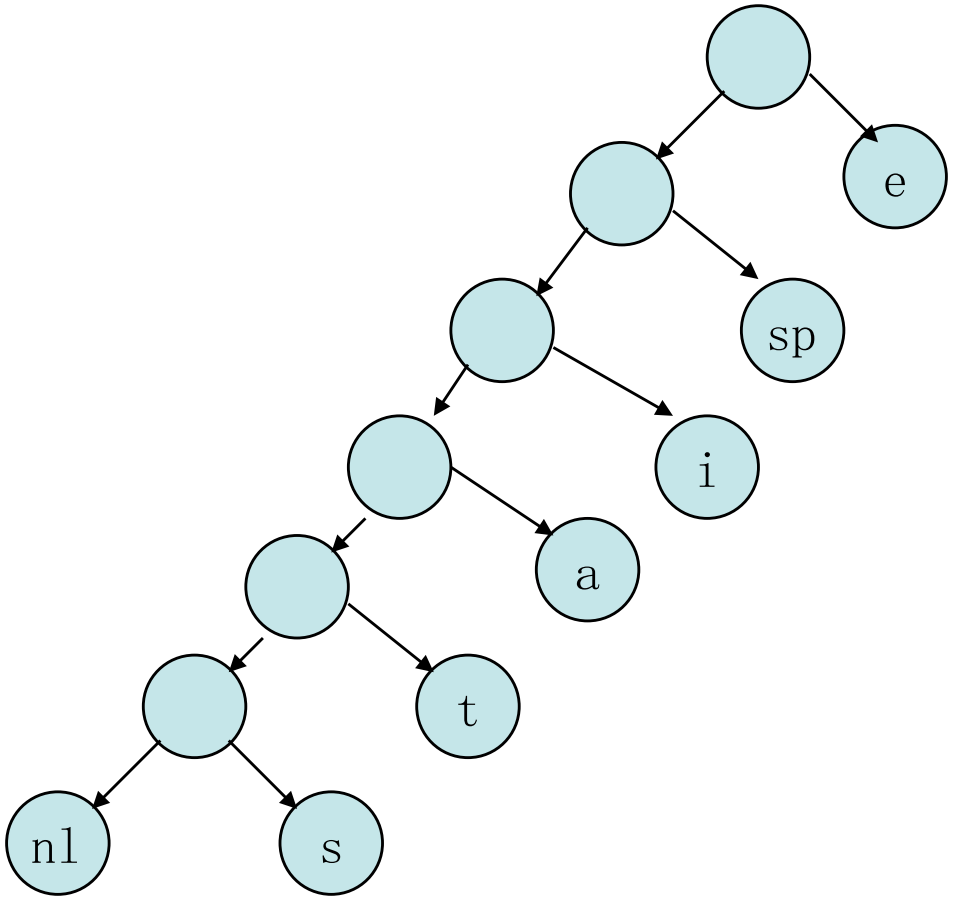
编码树

Represent coding as a tree



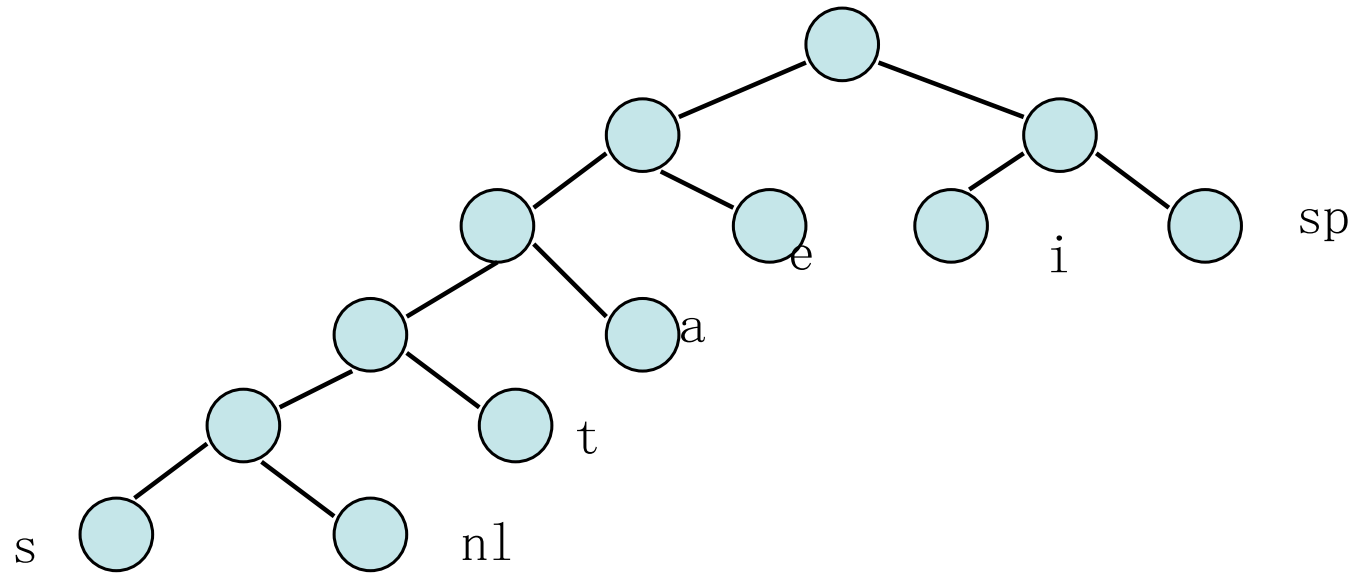
Greedy Try

Most frequent comes first



Huffman Tree

- 约束：符号应明确解码
- 基本问题：找到总成本最低的完整二叉树(如上文所定义)， 其中所有字符都包含在树叶中。



哈夫曼编码

Total cost: 146 bits

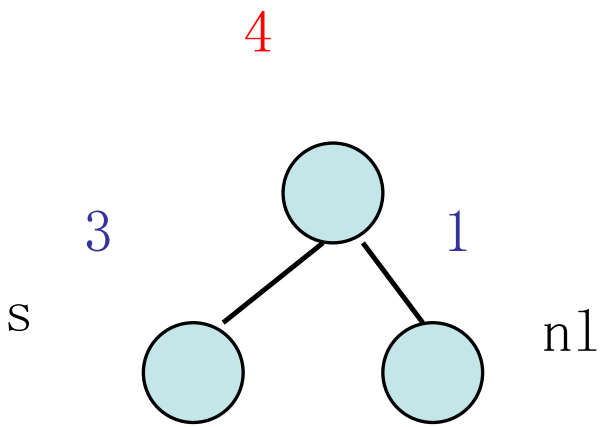
Character	Code	Frequency	Total Bits
a	001	10	30
e	01	15	30
i	10	12	24
s	00000	3	15
t	0001	4	16
space	11	13	26
newline	00001	1	5

Huffman Algorithm

- 我们维持一片森林. 一棵树的权重等于树叶的频率之和. C - 1 次, 选择两棵权重最小的树, T_1 and T_2, 任意地断开关系, 并且形成一棵 带有子树T_1 and T_2的 新树
- 在算法的开始, 有C棵单节点树 - 每个字符一棵 在算法的最后有一棵树, 这是最佳的Huffman coding tree

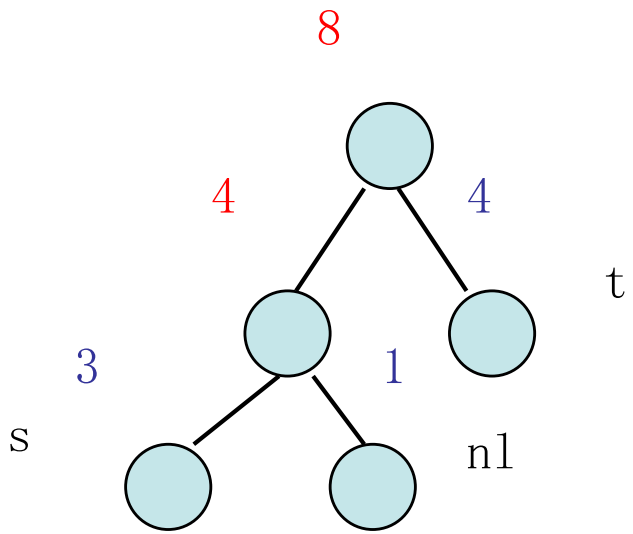
Constructing Huffman Tree

- 从单树开始
- 把它们按根上的重量分类，{1，3，4，10，12，13，15}
- 选择两棵权重最小的树，形成一棵新树



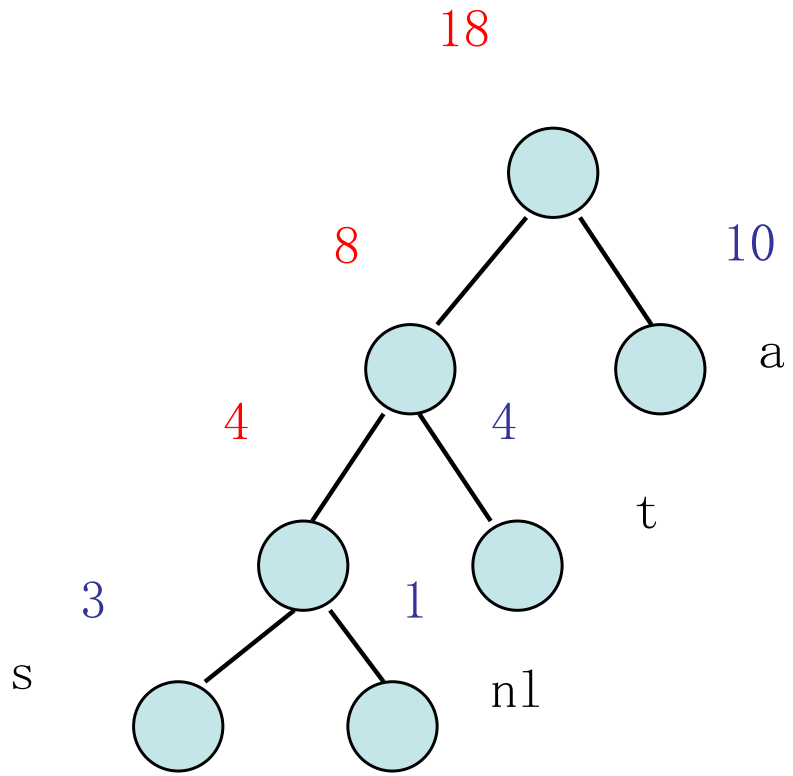
Constructing Huffman Tree

Carry on this process in {4, 4, 10, 12, 13, 15}



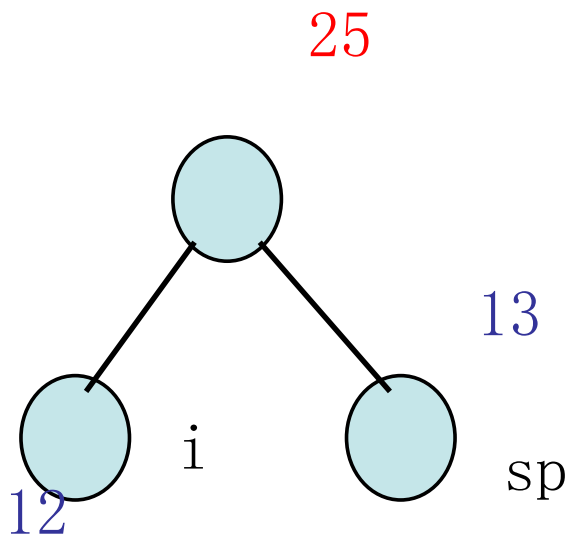
Constructing Huffman Tree

Now we have {8, 10, 12, 13, 15} left



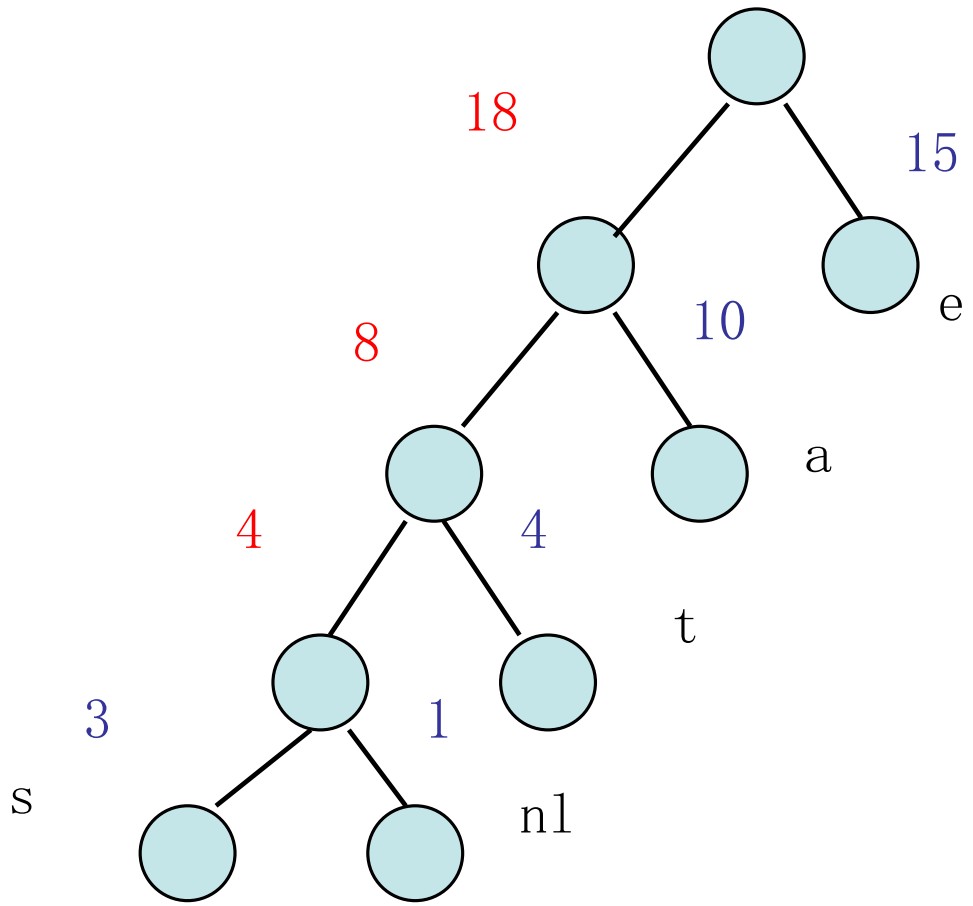
Constructing Huffman Tree

For {12, 13, 15, 18}



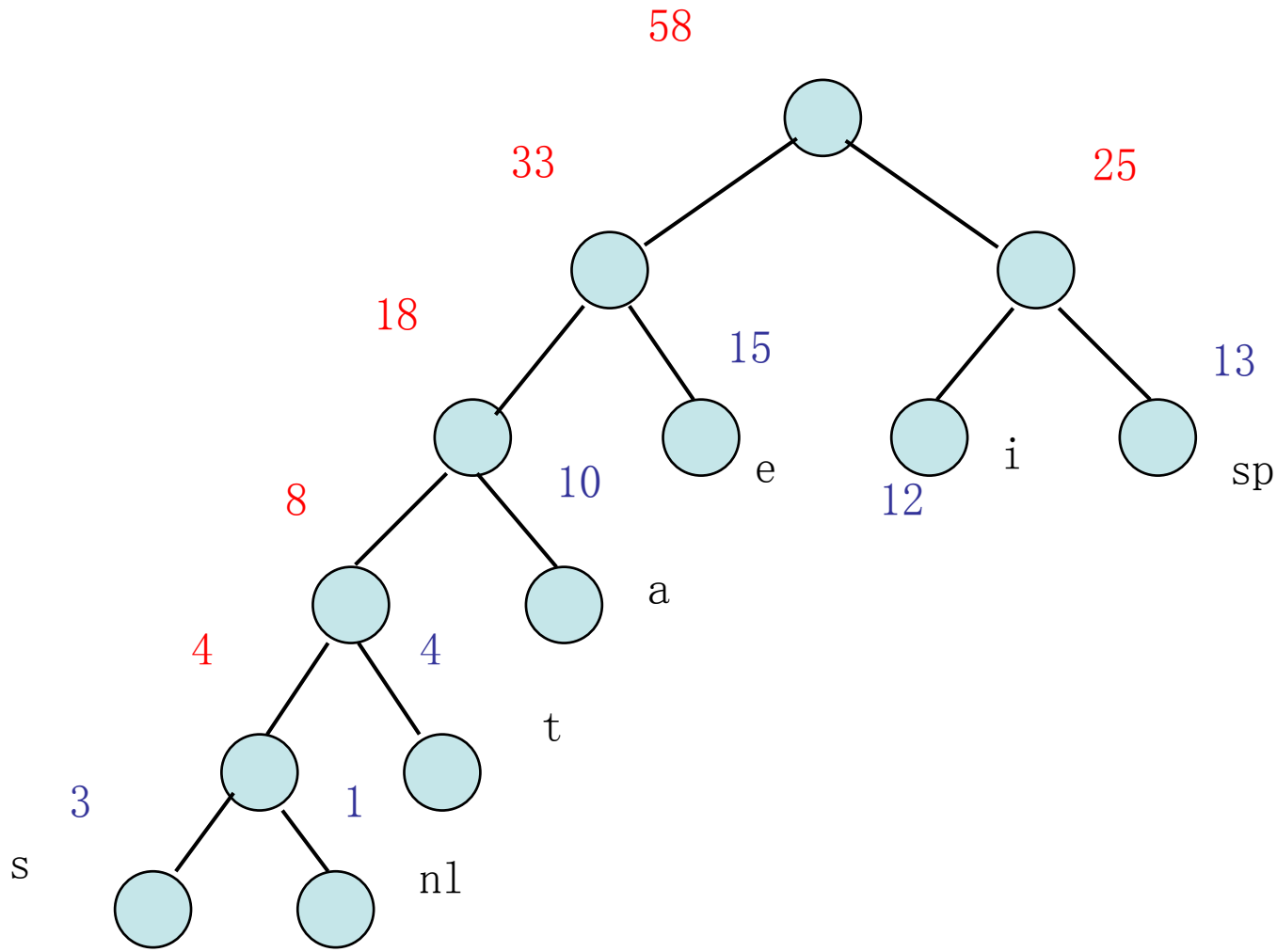
Constructing Huffman Tree

{15, 18, 25}



Constructing Huffman Tree

Final tree



Exercise

Draw a Huffman tree for symbols with frequency list {12, 13, 14, 23, 22, 6, 7, 29, 10, 3, 2, 18}

Huffman Algorithm

这个证明可通过归纳法来论证。 当树被合并时，我们认为新的字符集是根上的字符。 因此，在我们的示例中，经过四次合并后， 我们可以将字符集看作由 e 和元字符T_3 and T_4 组成。 这恐怕是证明中最微妙的部分

Roadmap

Divide and Conquer

- 将原始问题划分为更小的子问题
- 解决子问题
- 使用子问题的解决方案来构造原问题的解决方案
- 通常我们可以用递归来实现分治算法， 并用T(n)符号来分析其复杂度。

The Master Theorem

- 假定:
  - n 是原问题的规模
  - a 是递归中的子问题的数目
  - n/b 是每个子问题的规模
  - f (n) 是非递归成本(时间复杂度)

$$T(n) = aT(\frac{n}{b}) + f(n), \text{当 } a \geq 1, b > 1$$

- Case 1: $T(n) = \Theta(n^{\log_b a}), \text{if } f(n) = O(n^c)$ with $c < \log_b a$
- Case 2: $T(n) = \Theta(n^c \log^{k+1} n)$, if $f(n) = \Theta(n^c \log^k n)$ with $c = \log_b a$
- Case 3: $T(n) = \Theta(f(n)), \text{if } f(n) = \Omega(n^c)$ with $c > \log_b a$ and $af(n/b) \leq kf(n)$ for large n and $k < 1$

Examples

- Case 1: $T(n) = 3T(n/2) + n$,
  $a = 3, b = 2, f(n) = n, \rightarrow T(n) = O(n^{\log 3})$
- Case 2: $T(n) = 2T(n/2) + n$,
  $a = 2, b = 2, f(n) = n, \rightarrow T(n) = O(n \log n)$
- Case 3: $T(n) = 2T(n/2) + n^2$,
  $a = 2, b = 2, f(n) = n^2, \rightarrow T(n) = O(n^2)$
- 不可行的：在许多情况下，上述情况都不适用，
  如$T(n) = 2^n T(n/2) + n^n$

Stooge-sort

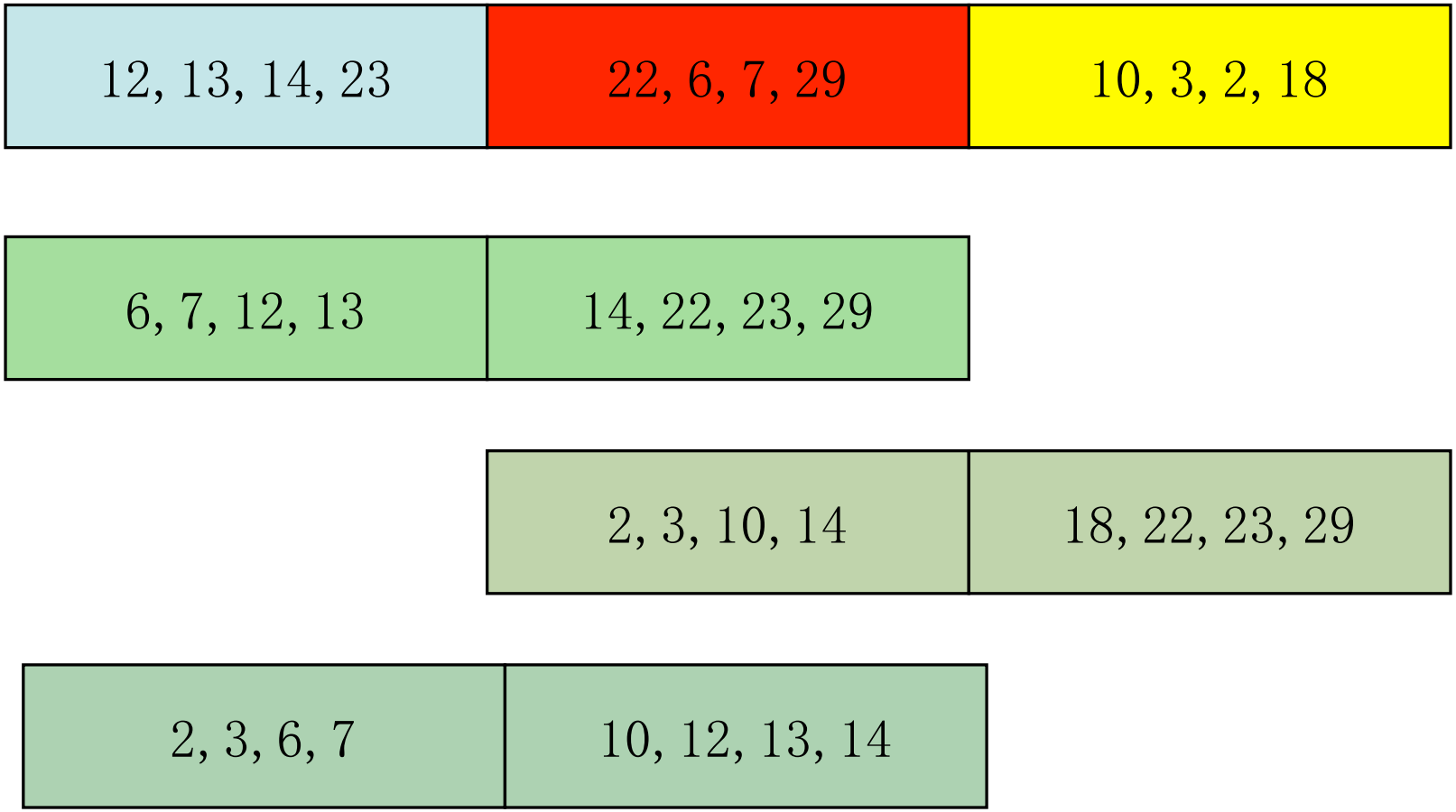
```
Stooge-sort (A, i, j)
{
     if A[i] > A[j]
          then exchange A[i], A[j];
     if i + 1  >= j
          then return;

     k = (j - i + 1) / 3;  //round down
     Stooge-sort (A, i, j - k);  //first 2/3
     Stooge-sort (A, i + k , j); //last 2/3
     Stooge-sort (A, i, j - k);  //first 2/3 again
}
```

Stooge-sort

Try this algorithm for {12, 13, 14, 23, 22, 6, 7, 29, 10, 3, 2, 18}

Stooge-sort

Three recursive calls



Does it work?

- 平凡情况，显然 OK
- 假设数组A包括：前1/3、中1/3、最后1/3，算法适用于较小的数组
- 在第一次递归调用之后，更大的元素进入了第二个1/3
- 在第二次调用之后，最大的元素已经到了最后的1/3，并被排序
- 在第三次调用之后，所有较小的元素被排序。

Does it work well?

- 平凡情况: $T(1) = 1$
- 递归地, $T(n) = 3T(2n/3) + 1$
- 我们有 $\log_{3/2} n$ 次递归, 在到达根本之前
- 假设有一高度为$\log_{3/2}n$的3-tree, 每个节点都有一次非递归操作
- 总耗费 = # of nodes:
  $3^{\log_{3/2} n} = 3^{\log_{3/2}(3^{\log_3 n})} = 3^{\log_3 n * \log_{3/2} 3}$
- $T(n) = O(n^{\log_{3/2} 3}) > O(n^{2.7})$

Quick Selection

- 选择问题： 给定一个整数k和一个n元数组$x_1, \ldots, x_n$ 找到数组中第k小的元素
- 类似于快速排序，先选择一个枢纽元
- 该中心将数组划分为3个部分。 $\{S_{left}, pivot, S_{right}\}$
  - 如果 k <= $|S_{left}|$, quick_select (k, S_{left})
  - 如果 k = $|S_{left}|+1$, 返回枢纽元, 否则 quick_select (k-$|S_{left}|$ - 1, S_{right})
- 注意:不像快速排序，只需要一个递归。

Quick Selection Example

- 在数组中找到第五小的元素
  $\{11, 9, 8, 20, 15, 3, 7, 32, 12\}$
- 枢纽元 12， $\{11, 9, 8, 3, 7\}, 12, \{20, 15, 32\}$， $k = 5$
- 枢纽元 8， $\{3, 7\}, 8, \{9, 11\}$， $k = 2$
- 枢纽元 = 9， $\{\}, 9, \{11\}$， $k = 1$
- 返回 11

Complexity of quick_select

- 最坏情况下，枢纽元始终是数组的最小值或最大值(少数情况)
- 复杂度 $O(n^2)$，为什么?
- 最好的情况，枢纽元是被选择的(罕见的例子)，$O(n)$
- 那么，平均情况会怎样呢?
- If $T(n) = T(0.9n) + n$, $T(1) = 1$, $T(n) = T(0.81n) + 0.9n + n = 1 + \ldots + 0.81n + 0.9n + n = O(n)$，这是平均水平还是低于平均水平?

Complexity of Average Case

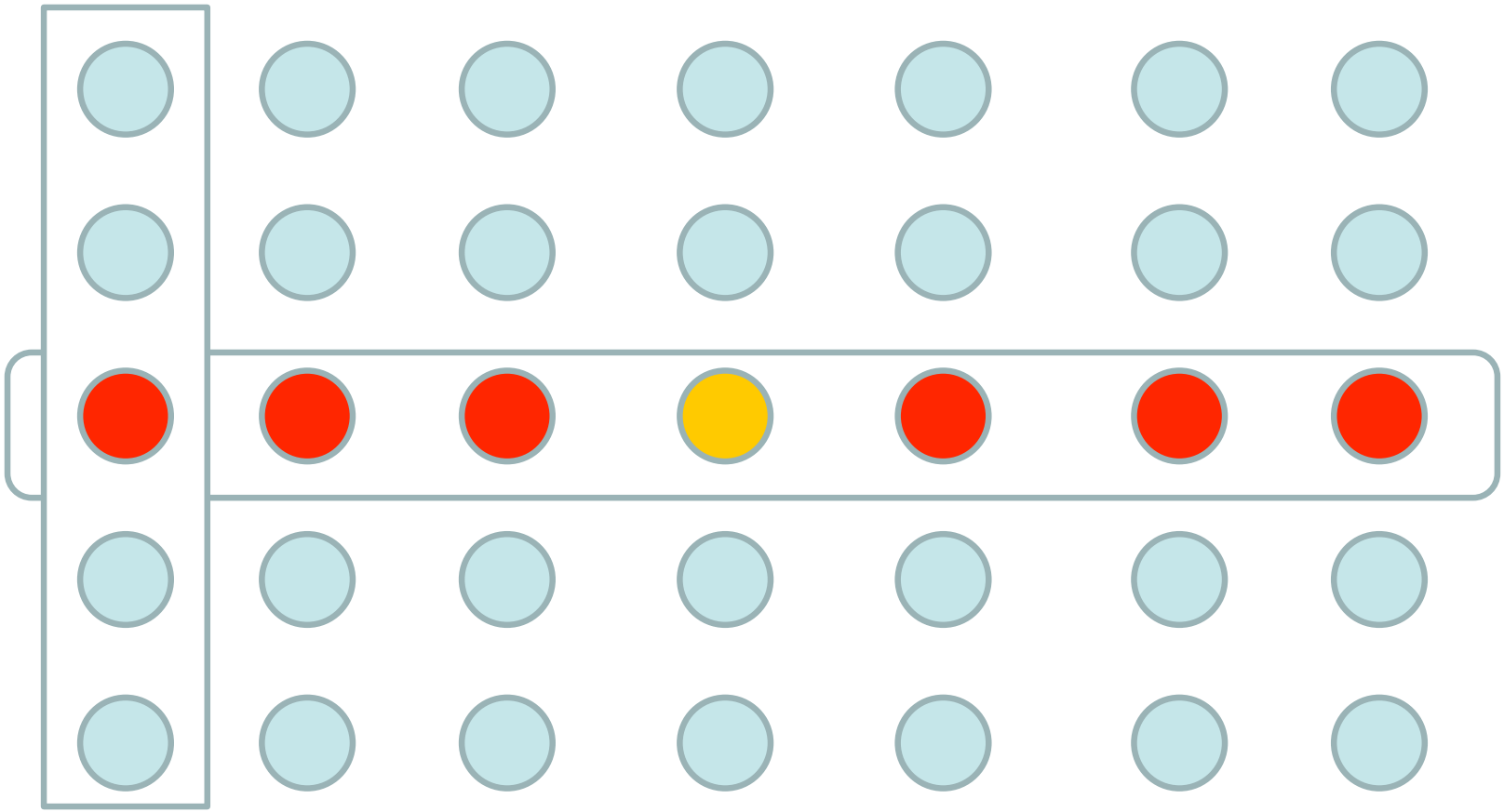
- T(n) = T(n-1) + n, 如果枢轴元是第一个最小/最大的元素。
- T(n) = T(n-2) + n, 如果枢纽元是第2小/最大的元素。
- T(n) = T(n/2) + n, 如果枢轴元是中值元素。
- 假设每个元素都有相等的可能性被选为主元，即每个情况的概率= 1/n。
  $T(n) = (2/n)\sum_{i=1}^{n/2} T(n-i) + n$

Complexity of Average Case

- 声明:T(n)是O(n)
- 用归纳法证明
  事实 $\sum_{i=1}^{n/2} n - i = \frac{n(n+2)}{2} - \frac{(n/2)(n/2+1)}{2} \leq = \frac{3}{8}n^2$
  假设 $T(k) < ck$ 适用于所有$k < n$
  （可以验证$T(1) < c$，如果$c > 1$）
  $T(n) = \frac{2}{n}\sum_{i=1}^{n/2} T(n-i) + n \leq \frac{2}{n}\sum_{i=1}^{n/2} c(n-i) + n$
  $\leq \frac{3c}{4}n + n = cn + (n - \frac{c}{4}n)$
- 我们可能选择 $c > 4$，使 $T(n) < cn$

Median of medians

快速选择中位数→ 最坏情况 O(n)

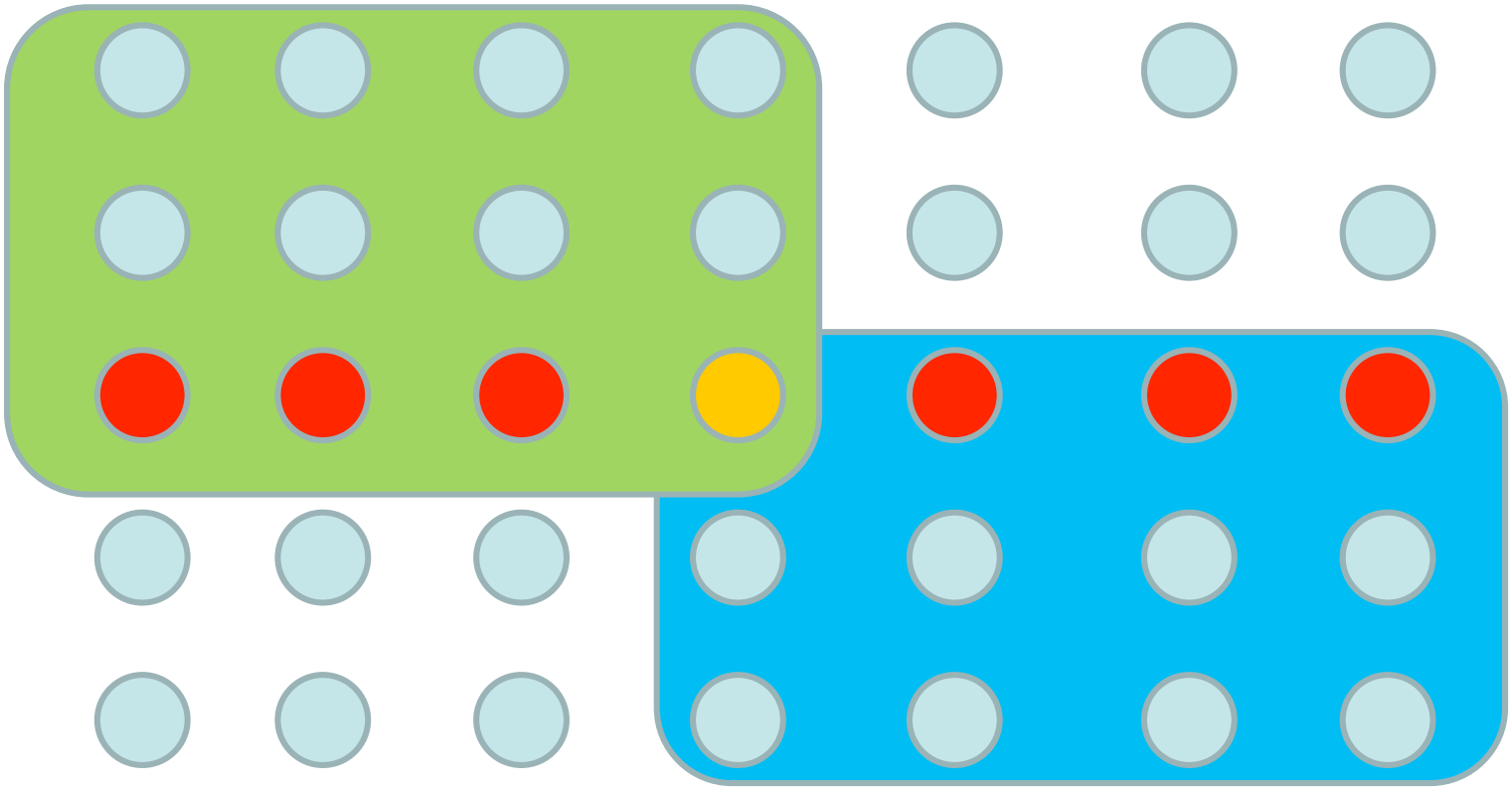


O(n) for Median Selection

- 想法:递归地选择一个好的枢纽元，中位数的中位数:
  - 将一个数组分成5个元素的n/5个子数组
  - 找到每个子数组的中值。不妨用insort ()先排序，得到的中位值组成一个长度为n/5的中位数数组
  - 递归地寻找中位数数组的中位数。调用quick_select ()，如果数组长度小于5，停止递归，返回insort ()找到的结果
- 找到的中位数的中位数是一个较好的选择，可以作为枢纽元（给quick_select ()函数找中位值时使用）

Better Partitioning

最少 30% 的元素 >= 中位数，30% <= 中位数. 且 0.3n > n/4



Select(i, n)

1. 将n个元素分成5个组。找到每个5个元素组的中位数。
2. 递归地选择n/5组中值的x值作为枢纽元。
3. 在枢纽元x得到的划分中，让k = rank(x)
4. 如果i = k，则返回x
   否则若 i < k
   递归地选择第i个最小值。
   下部元素
   否则递归地选择(i−k)th。
   上半部分最小的元素。

Complexity

- 假设总的复杂度 = T(n)
  - 第一步，Θ(n)
  - 第二步，T(n/5)
  - 第三步，Θ(n)
  - 第四步，T(3n/4)（最坏情况，选择了较大的划分里递归）
- T(n) = T(n/5) + T(3n/4) + n, T(1) = 1
- 可以用归纳法证明，存在某些常数c来说T(n) < cn

Roadmap

Dynamic Programming

- 回顾分治算法
  - 将问题分解为 独立的 子问题
  - 递归解决子问题(子问题是主要问题的较小实例)
  - 合并子问题的解决方案
- 动态规划： 适用于 存在不独立的递归子问题

Dynamic Programming

- Richard Bellman在1957年 创造了术语：动态规划
- 通过组合包含常见子问题的子问题的解决方案来解决问题
- 动态规划与分治算法的区别：
  - 用分治算法解决这些问题是低效的，因为相同的普通子问题需要多次解决。
  - dp将一次解决一个问题，并将它们的答案存储在一个表中，以供将来参考。

硬币找零问题

- 目的：给定货币面值，例如，1，5，10，25，100，设计一种使用最少数量的硬币支付给顾客的方法。
- 收银员算法：在每次迭代中，添加最大值的硬币，它不会带我们超过所要支付的金额。
- 例子：33美分

收银员算法

把n种硬币分类

$$c_1 < c_2 < \ldots < c_n$$

$S \leftarrow \emptyset$

WHILE $x > 0$

  $k \leftarrow$ 最大的硬币 c_k 使

  $c_k \leq x$

  IF 找不到 k, RETURN "无解决方案"

  ELSE

    $x \leftarrow x - c_k$

    $S \leftarrow S \cup \{k\}$

RETURN S

硬币的变化

贪婪的收银员算法是最优的吗？

Theorem

收银员算法对美国硬币而言是最佳的：1，5，10，25，100。

归纳法证明 x

另一个例子

- 收银员算法对于其他情况可能不适用
- 考虑美国的邮费：1，10，21，34，70，100，350，1225，1500
  - 收银员算法：140c = 100 + 34 + 1 + 1 + 1 + 1 + 1 + 1
  - 最佳：140c = 70 + 70
- 它甚至不能导致一个可行的解决方案：15c = 9 + ?

动态规划

- 为了解决找零n美分问题，我们需要先搞清楚所有x<n的找零问题
- 然后我们在解决方案中为较小的值建立解决方案。
  - 设 C[n] 为n美分所需的最小硬币数.
  - 设x为最优解中使用的第一个硬币的值
  - 然后 C[n] = 1 + C[n - x]
- 问题：我们不知道x的值

动态规划算法

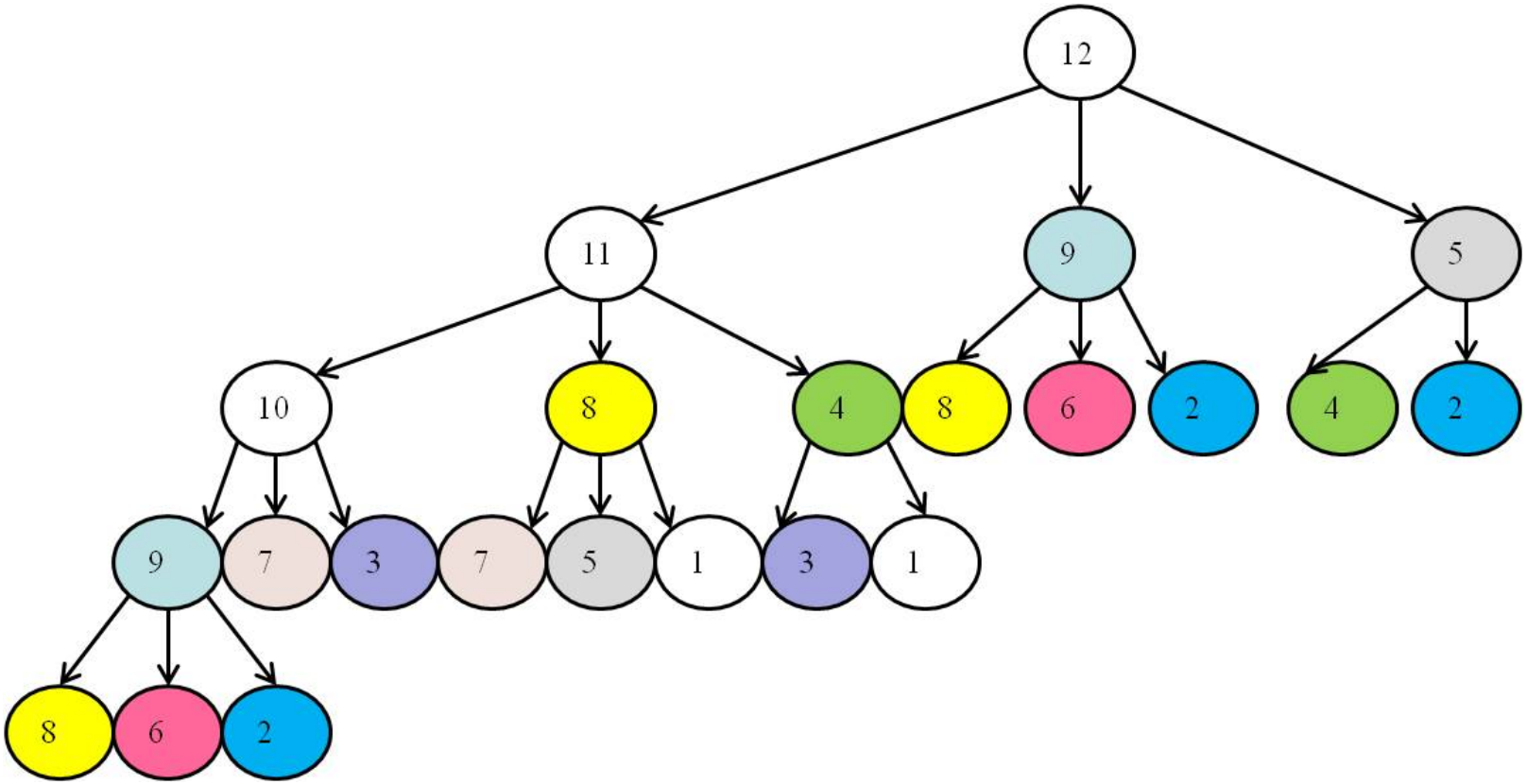
我们将尝试所有可能的x并取最小值。

$$C[n] = \begin{cases} \min_{i:d_i \le n}\{C[n-d_i]+1\} & \text{if } n > 0 \\ 0 & \text{if } n = 0 \end{cases}$$

Change

```
int Change (int n)
{
    if (n < 0)
        return (INFTY);
    else if (n == 0)
        return (0);

    return (1 + min (Change(n - d1),
        Change(n - d2), Change(n - d3)));
}
```

$n = 12, d_1 = 1, d_2 = 3, d_3 = 7$



动态规划的要素

动态规划用于解决具有以下特征的问题：

- 最优子结构（最优性原理）：问题的最优解包含子问题的最优解
- 重叠子问题：有些地方我们不止一次地解决同一个子问题

动态规划的步骤

1. 优化子结构特征
2. 递归定义最优解的值
3. 自下而上计算值
4. （如果需要）构造一个最优解

Memoization

- Memoization(储存化)是处理重叠的子问题的一种方法
  - 在计算子问题的解后，将结果存储在一个表中
  - 后续调用只执行表的查找
- 可以修改使用memoziation递归算法
- Change() 有很多重复的工作，用表格让算法达到O(nk)

DP_Change (n)

```
int DP_Change (int n)
{
    int tmp, i, j;

    for (i = 1, C[0] = 0; i <= n; i++)
    {
        tmp = INFTY;
        for (j = 0; j < k; j++)
            if (d[j] <= i &&
                C[i - d[j]] + 1 < tmp)
                tmp = C[i - d[j]] + 1;
        C[i] = tmp;
    }
    return (C[n]);
}
```

自下而上

$$C[n] = \min_i(1 + C[n - d_i])$$

n	0	1	2	3	4	5	6	7	8	9	10	11	12
C	0	1	2	1	2	3	2						
d_1	0	1	2	0	1	2	0						
d_2	0	0	0	1	1	1	2						
d_3	0	0	0	0	0	0	0						

自下而上

$$C[n] = \min_i(1 + C[n - d_i])$$

n	0	1	2	3	4	5	6	7	8	9	10	11	12
C	0	1	2	1	2	3	2	1	2	3	2	3	4
d_1	0	1	2	0	1	2	0	0	1	2	0	1	2
d_2	0	0	0	1	1	1	2	0	0	0	1	1	1
d_3	0	0	0	0	0	0	0	1	1	1	1	1	1

动态规划与贪婪算法

- 动态规划适用于：
  - 最优子结构：问题的最优解由子问题的最优解组成。
  - 重叠子问题：总共有几个子问题，每个问题都有重复的实例
- 自下而上解决问题，为了解决较大的问题而建立一个解决问题的子表
- 贪婪是自上而下的，动态规划可以矫枉过正；贪婪算法往往是容易编写的代码

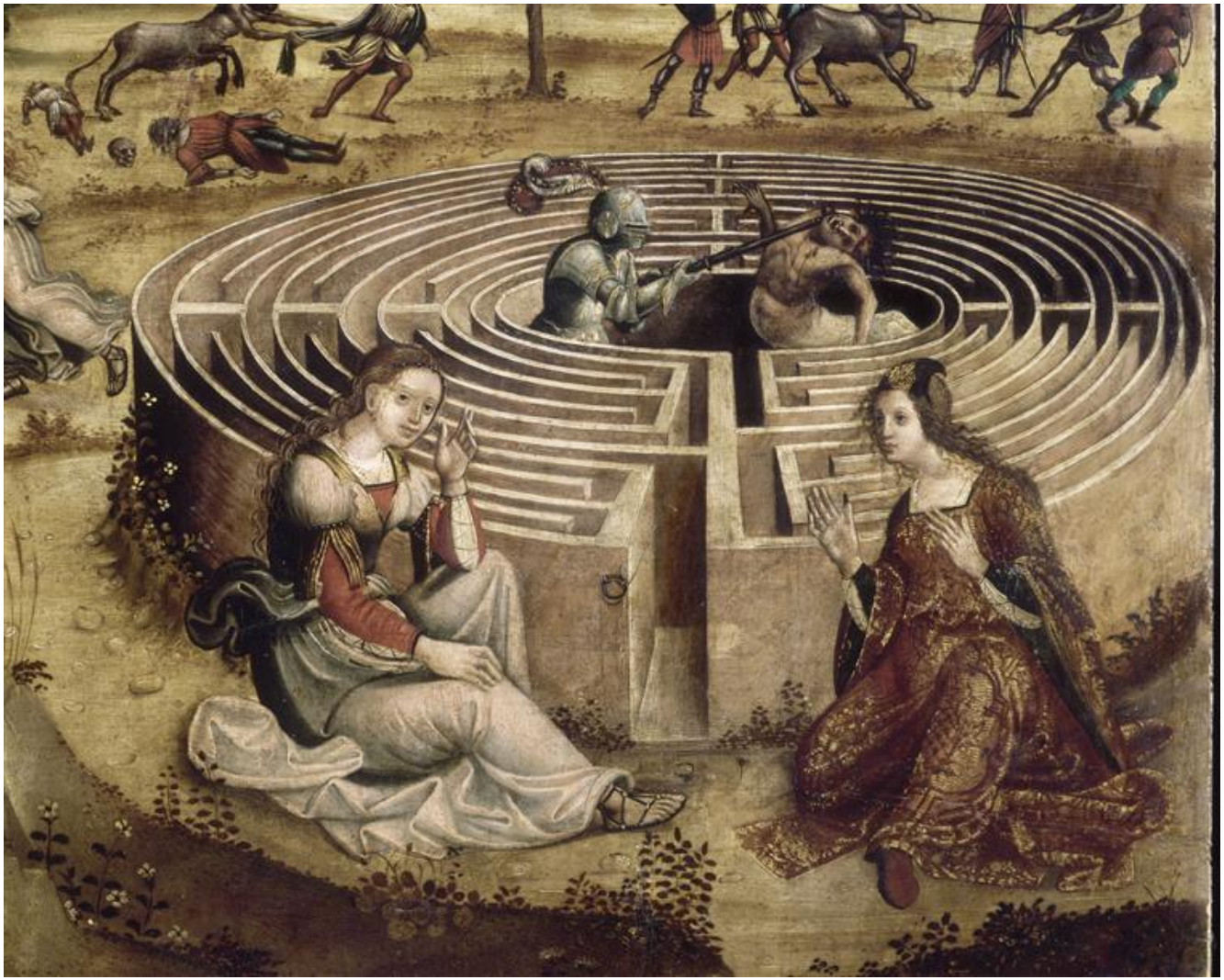
Roadmap

Backtracking

- 问题空间由状态（节点）和动作（导致新状态的路径）组成
- 当一个节点只能看到连接节点的路径时
- 如果一个节点只导致失败，返回到它的“父”节点。尝试其他的选择
- 如果这些都导致失败，那么可能需要更多的回溯。

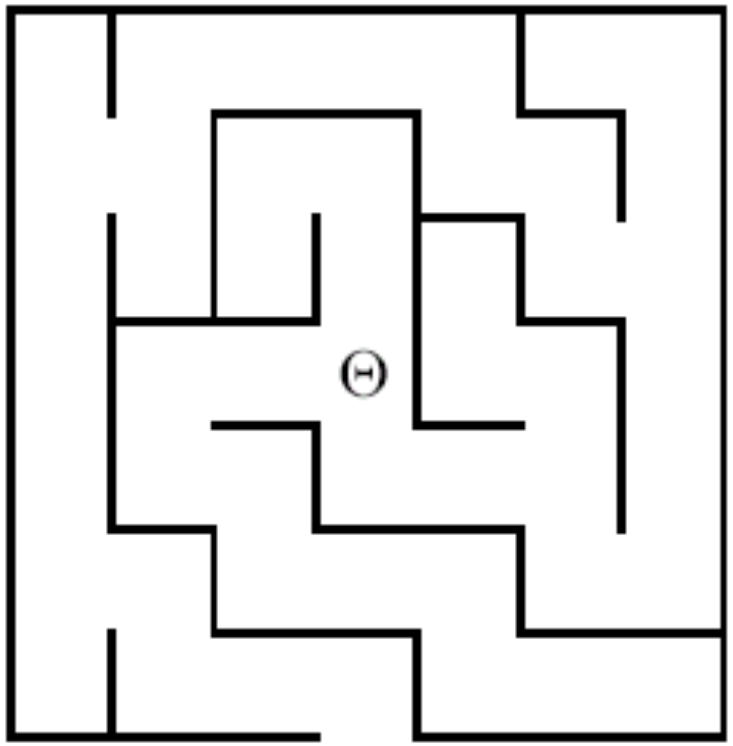
Labyrinth

特修斯和牛头人



右手定则

为了逃离迷宫：
将你的右手贴到墙上
while（你还没有从迷宫中逃出来）
　向前走，并把你的右手放在墙上



数独

- 数独：9乘9矩阵，填满数字。
- 所有数字必须在1到9之间
- 目标：每一行、每一列和每个小矩阵必须包含1到9之间的数字，而且每个数字不能重复。

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

暴力方法解决数独

- 如果没有空白的格子，问题解决
- 从左到右扫描单元格，从上到下找到第一个空白的单元格
- 如果找到空白的单元格，从1到9进行尝试
- 填入数字后检查时候符合规则
- 当搜索到达一个死胡同时，备份到前一个单元格，它试图填充并进入下一个数字。

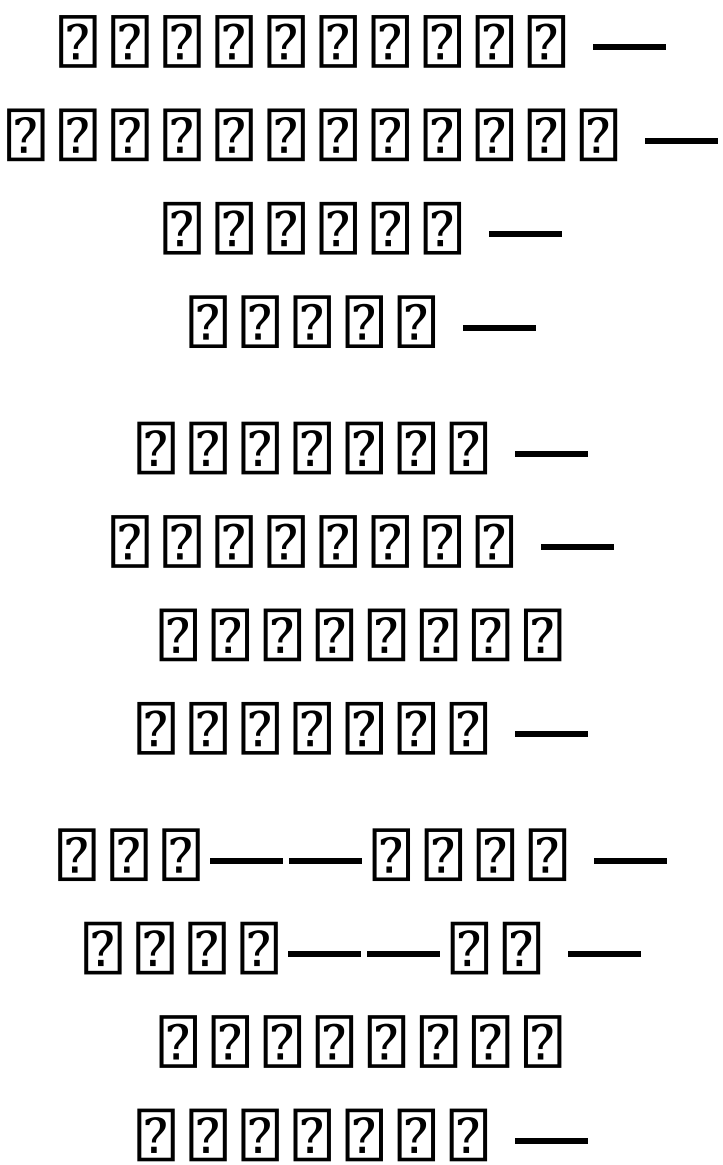
Recursive Backtracking

- 蛮力算法很慢，因为它不使用大量逻辑
- 但是蛮力算法是很容易实现的
- 用递归回溯法可以解决数独等问题
- 该问题的后一种版本比原始版本稍微简单一些
- 如果我们必须尝试不同的选择，用回溯法

小结

- 贪婪算法：局部最优可构成全局最优
- 分治算法：划分成独立的子问题单独求解
- 动态规划：存在重叠的需要优化的子问题
- 回溯算法：递归地尝试可能路径以寻找可行解

可能性的空间



考试