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

- Five common algorithms used to solve problems
- It is quite likely that at least one of these methods will work for a given problem
    - Greedy Algorithms
    - Divide and Conquer
    - Dynamic Programming
    - Randomized Algorithms
    - Backtracking Algorithms

Greedy Algorithms

- Take what is best for you now
- Greedy algorithms work in phases. In each phase, a decision is made that appears to be good, without regard for future consequences
- Generally, some local optimum is chosen. in hope that the local optimum finally leads to the global optimum
- Sometimes, a greedy algorithm only gives suboptimal solutions

Huffman Codes

Suppose we have a file that contains only the characters a, e, i, s, t, plus blank spaces and newlines

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

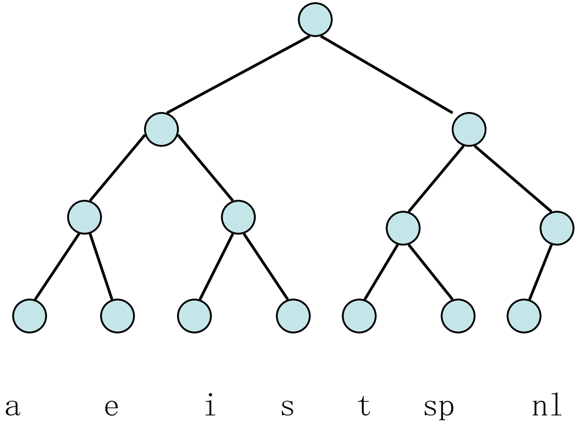
- Objective: reducing the file size in the case where we are transmitting it over a slow phone line
- In the previous example, total bits 174 (58 symbols)
- The general strategy : allow the code length to vary from character to character and to ensure that the frequently occurring characters have short codes
- If all the characters occur with the same frequency, then there are not likely to be any savings

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
- Given a set of symbols and weights, find a prefix-free binary code (a set of codewords) with minimum expected codeword length
- Shannon's source coding theorem: the optimal code length for a symbol is $-\log_b P$
- Information entropy $H = -\sum_i P_i \log P_i$ gives the lower bound of the expected codeword length

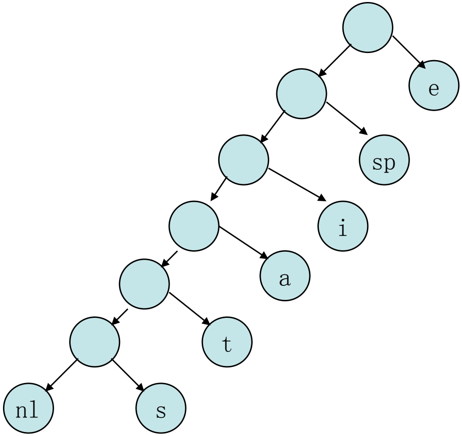
编码树

Represent coding as a tree



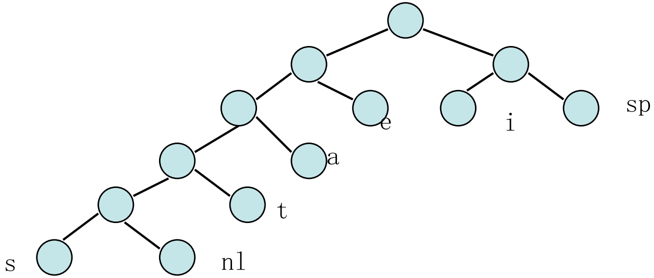
Greedy Try

Most frequent comes first



Huffman Tree

- Constraints: symbols should be decoded unambiguously
- Basic problem: find the full binary tree of minimum total cost (as defined above), where all characters are contained in the leaves



哈夫曼编码

Total cost: 146 bits

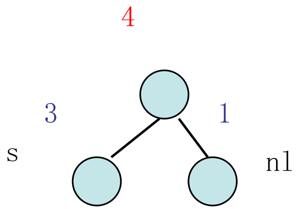
Character	Code	Frequency	Total Bits
a	001	10	30
e	01	15	30
i	10	12	24
s	00000	3	15
t	0001	4	16
space	11	13	26
newline	00001	1	5

Huffman Algorithm

- We maintain a forest of trees. The weight of a tree is equal to the sum of the frequencies of its leaves. C − 1 times, select the two trees, T_1 and T_2, of smallest weight, breaking ties arbitrarily, and form a new tree with subtrees T_l and T_2
- At the beginning of the algorithm, there are C single-node trees–one for each character. At the end of the algorithm there is one tree, and this is the optimal Huffman coding tree

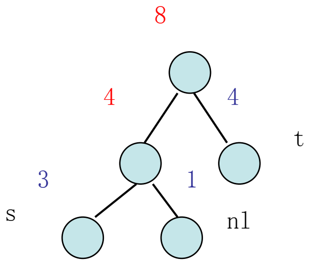
Constructing Huffman Tree

- Start from singleton trees
- Sort them by weights on the roots, {1, 3, 4, 10, 12, 13, 15}
- Select the two trees, with the smallest weights, to form a new tree



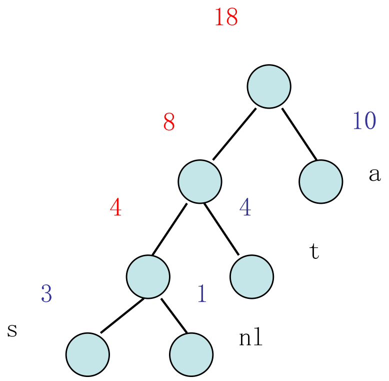
Constructing Huffman Tree

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



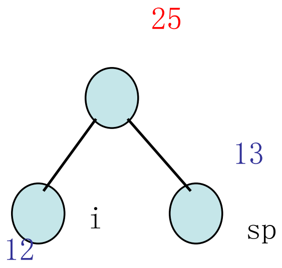
Constructing Huffman Tree

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



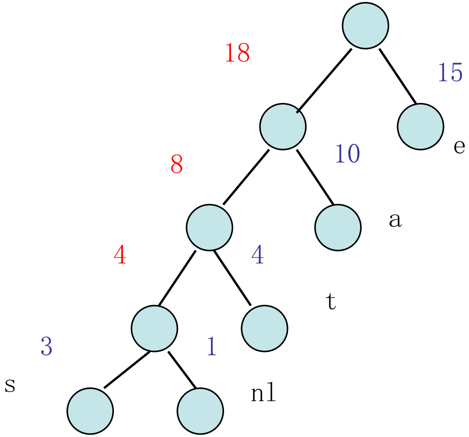
Constructing Huffman Tree

For {12, 13, 15, 18}



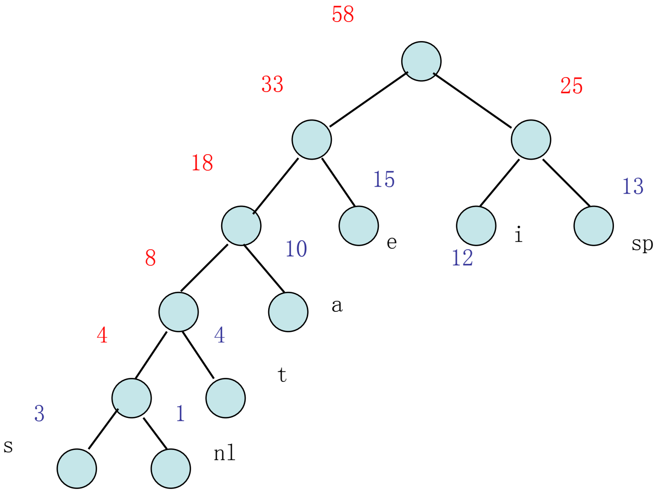
Constructing Huffman Tree

{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

The proof can be completed by using an induction argument. As trees are merged, we consider the new character set to be the characters in the roots. Thus, in our example, after four merges, we can view the character set as consisting of e and the meta-characters T_3 and T_4. This is probably the trickiest part of the proof

Roadmap

Divide and Conquer

- Divide the original problem into smaller sub-problems
- Solve the sub-problems
- Use the solutions to sub-problems to construct a solution to the original one
- Usually we may use recursion to implement a divide and conquer algorithm, and use T(n) notations to analyze its complexity

The Master Theorem

- Assumptions:
  - n is the size of the problem
  - a is the number of subproblems in the recursion
  - n/b is the size of each subproblem
  - f (n) is the non-recursive cost

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

- 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: $T(n) = 2T(n/2) + n^2$,
  $a = 2, b = 2, f(n) = n^2, \ \rightarrow T(n) = O(n^2)$
- Inadmissible: in many circumstances, none of the above cases apply, for instance, $T(n) = 2^n T(n/2) + n^n$

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
}
```

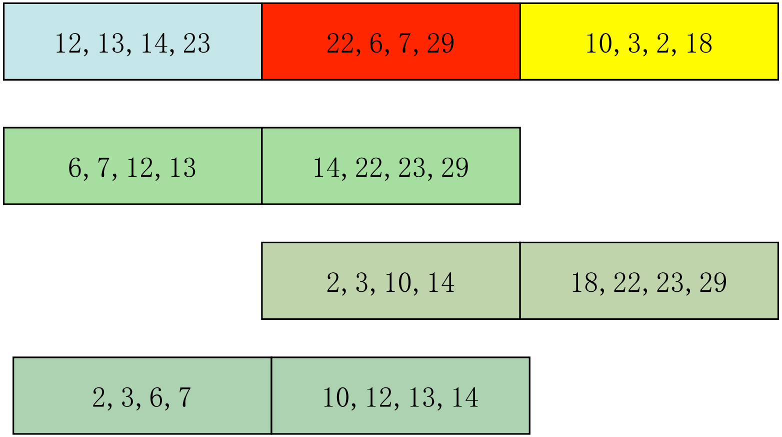
Stooge-sort

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

12, 13, 14, 23	22, 6, 7, 29	10, 3, 2, 18

Stooge-sort

Three recursive calls



Does it work?

- Trivial case, obviously OK
- Assume array A consists of: first 1/3, mid 1/3, last 1/3, and the algorithm works for smaller size arrays
- After the first recursive call, bigger elements have gone to the second 1/3
- After the second call, the biggest ones have gone to the last 1/3 and are sorted
- After the third call, all smaller ones are sorted

Does it work well?

- Trivial case: T(1) = 1
- Recursively, T(n) = 3T(2n/3) + 1
- We have $\log_{3/2} n$ recursions before reaching the base
- Think of a 3-tree with height $\log_{3/2} n$, each node has 1 non-recursive operation
- Total cost = # 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

- The selection problem: given an integer k and an array $x_1, \ldots, x_n$ of n elements, find the k-th smallest element in the array
- Similar to quick sort, pick up a pivot first
- The pivot partitions the array into 3 parts, $\{S_{left}, pivot, S_{right}\}$
  - If k <= $|S_{left}|$, do quick_select (k, S_{left})
  - If k = $|S_{left}|$+1, return pivot,
    Else do quick_select (k−$|S_{left}|$ − 1, S_{right})
- Note: unlike quick sort, only one recursion is needed

Quick Selection Example

- Find the 5th smallest one in array $\{11, 9, 8, 20, 15, 3, 7, 32, 12\}$
- Pivot 12, $\{11, 9, 8, 3, 7\}, 12, \{20, 15, 32\}$, k = 5
- Pivot 8, $\{3, 7\}, 8, \{9, 11\}$, k = 2
- Pivot = 9, $\{\}, 9, \{11\}$, k = 1
- Return 11

Complexity of quick_select

- Worst case, pivot is always the minimum or maximum of the array (rare cases)
- Complexity $O(n^2)$, why?
- Best case, the pivot is selected (rare case too), O(n)
- Then, what happens to the average case?
- If T(n) = T(0.9n) + n, T(1) = 1, T(n) = T(0.81n) + 0.9n + n = 1 + ...+0.81n +0.9n+n = O(n), is this above or below average?

Complexity of Average Case

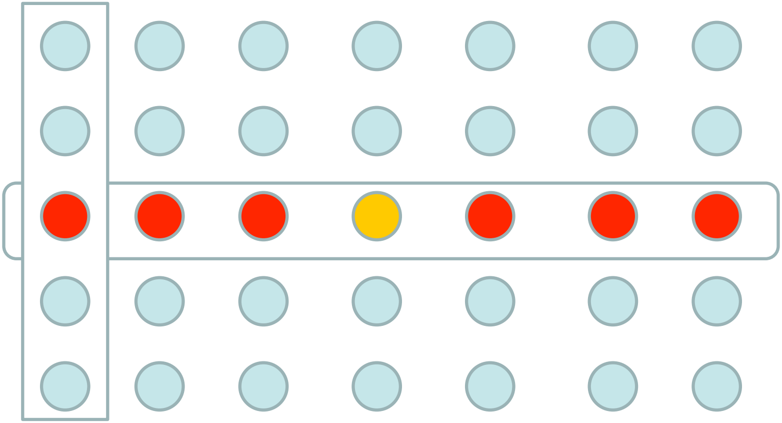
- T(n) = T(n-1) + n, if the pivot is the 1st smallest/biggest element
- T(n) = T(n-2) + n, if the pivot is the 2nd smallest/biggest element
- T(n) = T(n/2) + n, if the pivot is the median element
- Suppose each element has the equal possibility to be selected as the pivot, ie, probability = 1/n for each case
  $T(n) = (2/n)\sum_{i=1}^{n/2} T(n-i) + n$

Complexity of Average Case

- Claim: T(n) is O(n)
- Proof by induction.
  Fact $\sum_{i=1}^{n/2} n - i = \frac{n(n+2)}{2} - \frac{(n/2)(n/2+1)}{2} <= \frac{3}{8}n^2$
  If $T(k) < ck$ holds for all k < n
  $T(n) < \frac{2}{n}\sum_{i=1}^{n/2} T(n-i) + n \leq \frac{3c}{4}n + n = cn + (n - \frac{c}{4}n)$
- We may choose c > 4, so as to make $T(n) < cn$

Median of medians

Select the median quickly $\rightarrow$ worst case O(n)

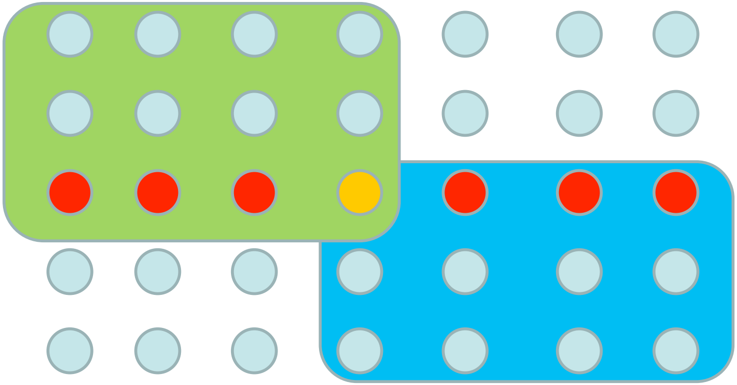


O(n) for Median Selection

- Idea: recursively select median for a good pivot:
    - Divide an array into n/5 sub-arrays of 5 elements each
    - Find the median for each sub-array
    - Recursively find the median of the medians
- Good choice as the pivot

Better Partitioning

At least 30% elements >= median of medians, 30% <= median of medians. Remember 0.3n > n/4



Select(i, n)

1. Divide the n elements into groups of 5. Find the median of each 5-element group
2. Recursively select the median x of the n/5 group medians to be the pivot.
3. Partition around the pivot x. Let k = rank(x)

4.
   ```
   if i = k then return x
   else if i < k
       then recursively select the ith smallest
           element in the lower part
   else recursively select the (i - k)th
       smallest element in the upper part
   ```

Complexity

- Total complexity = T(n)
  - Step 1, $\Theta(n)$
  - Step 2, $T(n/5)$
  - Step 3, $\Theta(n)$
  - Step 4, $T(3n/4)$
- $T(n) = T(n/5) + T(3n/4) + n$, $T(1) = 1$
- Proof by induction, $T(n) < cn$ for some constant c

Roadmap

Dynamic Programming

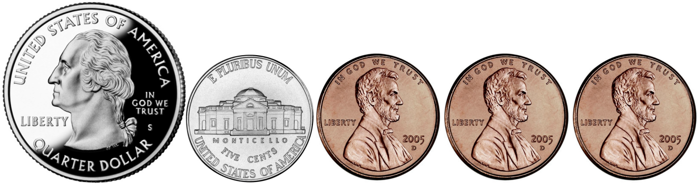
- 回顾分治算法
  - Break problem into independent subproblems
  - Recursively solve subproblems (subproblems are smaller instances of main problem)
  - Combine solutions
- 动态规划: appropriate when we have recursive subproblems that are not independent

Dynamic Programming

- Richard Bellman coined the term dynamic programming in 1957
- Solves problems by combining the solutions to subproblems that contain common subproblems
- Difference between DP and Divide-and-Conquer:
    - Using Divide and Conquer to solve these problems is inefficient as the same common subproblems have to be solved many times
    - DP will solve each of them once and their answers are stored in a table for future reference

硬币找零问题

- Objective: given currency denominations, say, 1, 5, 10, 25, 100, devise a method to pay amount to customer using fewest number of coins
- Cashier's algorithm: at each iteration, add coin of the largest value that does not take us past the amount to be paid
- Example: 33 cents



Cashier's Algorithm

```
SORT n coin denominations so that
    $c_1 < c_2 < \ldots < c_n$
$S \leftarrow \emptyset$
WHILE $x > 0$
  $k \leftarrow$ largest coin denomination $c_k$ such that
        $c_k \le x$
  IF no such k, RETURN "no solution"
  ELSE
    $x \leftarrow x - c_k$
    $S \leftarrow S \cup \{k\}$
RETURN S
```

Coin Changing

Is the greedy Cashier's algorithm optimal?

Theorem

Cashier's algorithm is optimal for U.S. coins: 1, 5, 10, 25, 100.

Proof by induction on x

In Other Case

- Cashier's algorithm may not work for other situations
- Consider U.S. postage: 1, 10, 21, 34, 70, 100, 350, 1225, 1500
  - Cashier's algorithm: 140c = 100 + 34 + 1 + 1 + 1 + 1 + 1 + 1
  - Optimal: 140c = 70 + 70
- It may not even lead to a feasible solution: 15c = 9 + ?

Dynamic Programming

- To make change for n cents, we are going to figure out how to make change for every value x < n first
- We then build up the solution out of the solution for smaller values
  - Let C[n] be the minimum number of coins needed to make change for n cents.
  - Let x be the value of the first coin used in the optimal solution
  - Then C[n] = 1 + C[n − x]
- The problem is: we don't know x

Dynamic Programming Algorithm

We will try all possible x and take the minimum

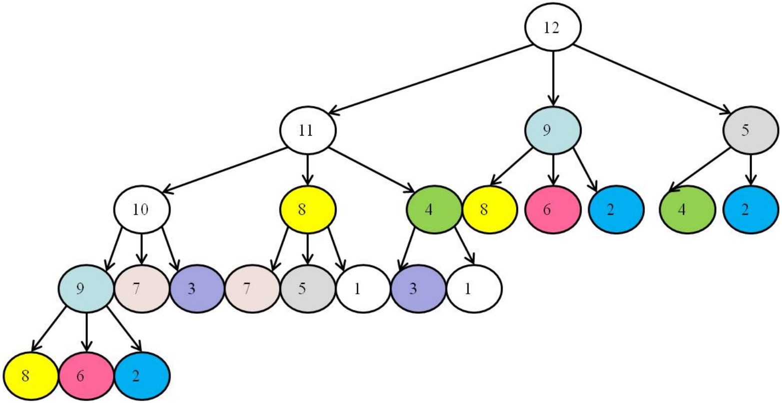
$$C[n] = \begin{cases} \min_{i:d_i \le n}\{C[n - d_i] + 1\} & \text{if } n > 0 \\ 0 & \text{if } n = 0 \end{cases}$$

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)));
}
```

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



Elements of Dynamic Programming

DP is used to solve problems with the following characteristics:

- Optimal sub-structure (Principle of Optimality): an optimal solution to the problem contains within it optimal solutions to sub-problems
- Overlapping subproblems: there exist some places where we solve the same subproblem more than once

Dynamic Programming Steps

1. Characterize optimal sub-structure
2. Recursively define the value of an optimal solution
3. Compute the value bottom up
4. (if needed) Construct an optimal solution

Memoization

- Memoization is one way to deal with overlapping subproblems
  - After computing the solution to a subproblem, store it in a table
  - Subsequent calls just do a table lookup
- Can modify recursive algorithm to use memoziation
- Change() has a lot of repeated work, use a table to make the algorithm 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]);
}
```

Build Up from Bottom

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

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						

Build Up from Bottom

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

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

Dynamic Programming vs Greedy Algorithms

- DP is suitable for problems with:
    - Optimal substructure: optimal solution to problem consists of optimal solutions to subproblems
    - Overlapping subproblems: few subproblems in total, many recurring instances of each
- Solve bottom-up, building a table of solved subproblems that are used to solve larger ones
- Greedy is top-down, dynamic programming can be overkill; greedy algorithms tend to be easier to code

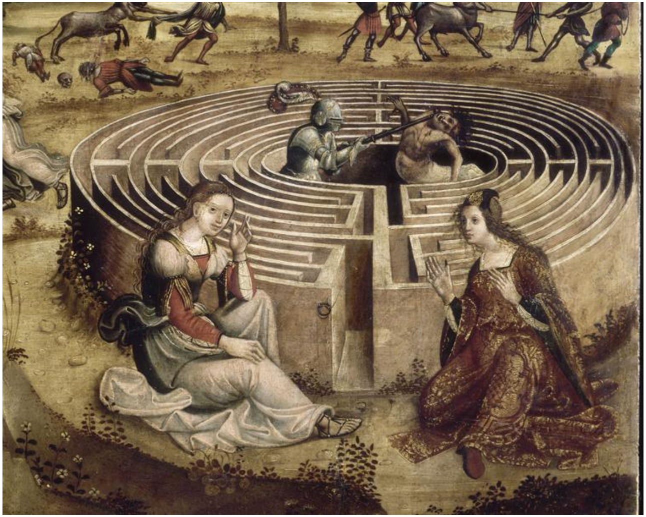
Roadmap

Backtracking

- Problem space consists of states (nodes) and actions (paths that lead to new states)
- When in a node can can only see paths to connected nodes
- If a node only leads to failure go back to its "parent" node. Try other alternatives
- If these all lead to failure then more backtracking may be necessary

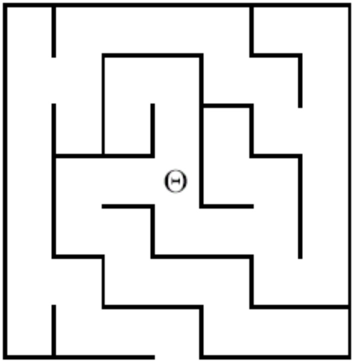
Labyrinth

Theseus and Minotaur



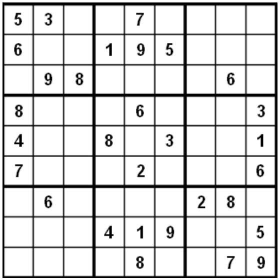
Right Hand Rule

```
To escape from a maze:
Put your right hand against a wall
while (you have not yet escaped from the maze)
  Walk forward keeping your right hand on a wall
```



Sudoku

- Sudoku: 9 by 9 matrix with some numbers filled in
- all numbers must be between 1 and 9
- Goal: Each row, each column, and each mini matrix must contain the numbers between 1 and 9 once each with no duplicates



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

Brute Force Sudoku Solution

- If not open cells, solved
- Scan cells from left to right, top to bottom for first open cell
- When an open cell is found trying through digits 1 to 9
- When a digit is placed check if the set up is legal
- When the search reaches a dead end, backs up to the previous cell it was trying to fill and goes onto to the next digit

Recursive Backtracking

- Brute force algorithms are slow because they don't employ a lot of logic
- But brute force algorithms are fairly easy to implement as a first pass solution
- Problems such as Sudoku can be solved using recursive backtracking
- Later versions of the problem are just slightly simpler than the original
- Backtrack if we have to try different alternatives

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

I dwell in Possibility —
A fairer House than Prose —
More numerous of Windows —
Superior —— for Doors —

Of Chambers as the Cedars —
Impregnable of Eye —
And for an Everlasting Roof
The Gambrels of the Sky —

Of Visitors —— the fairest —
For Occupation —— This —
The spreading wide of narrow Hands
To gather Paradise —

考试