Outline

Today:

- Design Patterns: Brute-Force and Greedy
- Counting Change Problem
- Pattern Matching Algorithms as Examples of 2 Design Patterns
- Brute Force
- Simplified Boyer-Moore

Sugih Jamin (jamin@eecs.umich.edu)

Design Patterns

What is a design pattern?

Design patterns we look at in this course:

- brute force
- divide and conquer
- recursive
- amortized
- greedy, usually involving “heuristics”
- branch and bound
- backtracking
- dynamic programming

Problem: Counting Change

Cashier has a collection of "coins" of various denominations
Want: return a specified sum *using the smallest number of coins*

Formally:

- A: sum to be returned
- n coins
- the coins, $P = \{p_1, p_2, p_3, \ldots, p_n\}$
- the denominations, $D = \{d_{p_1}, d_{p_2}, d_{p_3}, \ldots, d_{p_n}\}$
  (can have repetition (two dimes, three pennies))
- the change, $C \subset P$
- the selection, $S = \{s_i = 1$ if $p_i \in C$, 0 otherwise $\}$
- **Want:** minimize $\sum s_i$ (# of coins) such that $\sum d_{c_i} = A$

Counting Change: Example

- $A = 43$
- $n = 13$
- $P =$ coins of different sizes
- $D = 10, 1, 1, 25, 10, 1, 5, 1, 1, 1, 5, 1, 1$
- $C = 10, 1, 1, 25, 1, 5$
- $S = 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0$

Solution: Brute-force Approach

Try all subsets of P:

- $S_1 = 1, 0, 1, 0, 0, 1, \ldots$
- $S_2 = 0, 1, 1, 1, 0, 0, \ldots$
- $S_3 = 1, 0, 1, 1, 1, 0, \ldots$
- $\ldots$
- How many possible subsets are there?

Feasible solution set: all S_i's for which $\sum d_{c_i} = A$
Objective function: the S_i that minimizes $\sum s_i$
What is the time complexity to compute the sums?

Total time complexity of this approach:

- worst case:
- best case:

Bruce-force Algorithm

Solves a problem in the most simple, direct, or obvious way

- does not take advantage of structure or pattern in the problem
- usually involves exhaustive search of the solution space
- pro: often simple to implement
- con: usually not the most efficient way

Greedy Approach

Pick coin with largest denomination first:

- return largest coin p_i from P such that $d_{p_i} \leq A$
- $A -= d_{p_i}$
- find next largest coin

What is the time complexity of the algorithm?

Solution not necessarily optimal:

- consider $A = 20$ and $D = \{15, 10, 10, 1, 1, 1, 1, 1\}$
- greedy returns 6 coins, optimal requires only 2 coins!

Solution not guaranteed:

- consider $A = 20$ and $D = \{15, 10, 10\}$
- greedy picks 15 and finds no solution!

Greedy Approach

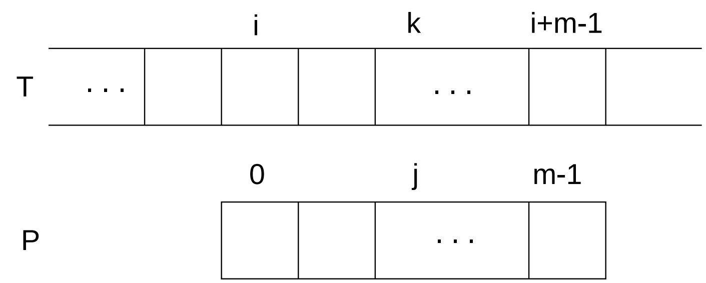
Algorithm decides what is the best thing to do at each step (local maxima), and never reconsiders its decisions

- pro: may run significantly faster than brute-force
- con: may not lead to the optimal (or even correct) solution (global maxima)

Usually requires some initial pre-computation to set up the problem, to take advantage of special structure/pattern in the problem or solution space

Pattern Matching

Given a text string (T) of length n,
and a pattern string (P) of length m,
determine if P is a substring of T



A match means:
$T[i] == P[0], T[i+1] == P[1], \ldots, T[i+m-1] == P[m-1]$
If match found, return i (first match)

Example strings:

- “The quick brown fox jumped over the lazy dog”
- “cagacagacagata”
- “1011110000101011100011100101”

Sugih Jamin (jamin@eecs.umich.edu)

Alphabet Space

The string doesn't have to consist only of alphabets in a human language

Alphabet space Σ:

- English language: "The quick brown fox jumped over the lazy dog"
- DNA sequence: "cagacagacagata"
- binary data: "1011110000101011100011100101"

Alphabet size, $|\Sigma|$:

- English language: 26 alphabets
- DNA sequence: 4 characters ('c', 'g', 'a', 't')
- binary data: 2 digits ('1', '0')

Pattern Matching Algorithms

- Brute-force
- Simplified Boyer-Moore: greedy, but falls back to brute-force
- Knuth-Morris-Pratt: memoized
- Original Boyer-Moore: memoized

Brute-force Pattern Matching

```
T: a a b c b d a a a a b c a c b a a c
P: a c b a a c   a c b a a c
   1 2          11 12
     a c b a a c   a c b a a c
     3 4          13 14
       a c b a a c   a c b a a c
       5            15 16
         a c b a a c   a c b a a c
         6            17
           a c b a a c   a c b a a c
           7            18
             a c b a a c   a c b a a c
             8            19 . . . . 24
               a c b a a c
               9 10
```

```
int // index of matching start in T
bfmatch(char *T, char *P) // T text, P pattern
```

What is the time complexity of the algorithm?

- best case: - worst case:

Sugih Jamin (jamin@eecs.umich.edu)

Simplified Boyer-Moore: Intuition

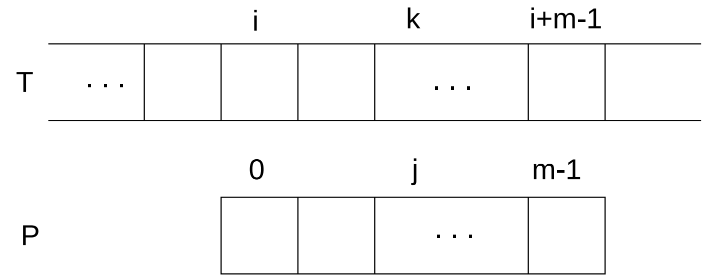
```
T: a a b c b d a a a a b c a c b a a c
P: a c b a a c                          ; right-to-left
             1
               a c b a a c              ; skip no match
                       3 2
                   a c b a a c          ; skip to rightmost match
                         6 5 4
                     a c b a a c        ; falls back to brute force
                               7
                             a c b a a c ; skip to rightmost match
                            13 . . . . 8
```

Compare against 24 CMPs with brute-force

What is a **heuristic**?
Origin: *heuriskein* (Gr), to find, to discover, Archimedes: Heureka!
A process that may solve a given problem, but offers no guarantee of doing so (in terms of time and/or quality of solution, the opposite of algorithm); a *technique* that improves the average case but not necessarily the worst-case performance

Simplified-BM Algorithm

Heuristics used by the SBM algorithm:



- **SBM-0:** match pattern backwards
- when $T[k] \neq P[j]$:

  SBM-1: if $T[k] \notin P$, shift P to the right, past $T[k]$, restart matching from $T[k+m]$

  SBM-2: if $T[k] == P[l]$ and $T[k] \notin P[l+1,\ldots,m-1]$ (rightmost l)

  SBM-2.1: if $l < j$, shift P right to align $P[l]$ with $T[k]$, restart matching from $T[k+(m-1-l)]$ (shift right by l)

  SBM-2.2: else $l > j$, shift P right by 1, restart matching from $T[k+(m-j)]$ (fall back to brute-force)

Sugih Jamin (jamin@eecs.umich.edu)

Simplified-BM Example

```
T: a a b c b d a a a a b c a c b a a c
P: a c b a a c                         ; d != c by SBM-0
               a c b a a c             ; by SBM-1
                   a c b a a c         ; by SBM-2.1
                     a c b a a c       ; by SBM-2.2
                           a c b a a c ; by SBM-2.1

--------------------------------------------------------

T: a a b c b d a a a a b c a c b a a c
P: a c b a a c                         ; last[d] = -1
               a c b a a c             ; last[b] = 2 < 4
                   a c b a a c         ; last[c] = 5 > 3
                     a c b a a c       ; last[b] = 2 < 5
                           a c b a a c ; match!
```

Simplified-BM `last[]` Computation

How do you determine l?

Just as in the greedy count-change algorithm,
pre-compute the information (heuristics) you need

In this case, pre-compute l for every letter of the alphabet,
store these in `last[]`:

- initialize each member of `last[|Σ|]` to -1
- go thru P in reverse to determine the last occurrence of each alphabet

For the example P in previous slide, `last[]`:

a	b	c	d
4	2	5	-1

Simplified BM Time Complexity

What is the worst-case time complexity of the algorithm?

What is the average-case time complexity?
Works well for large alphabet, longish pattern with few different characters; empirically, for English words, SBM requires about $0.3n$ CMPs

Sugih Jamin (jamin@eecs.umich.edu)