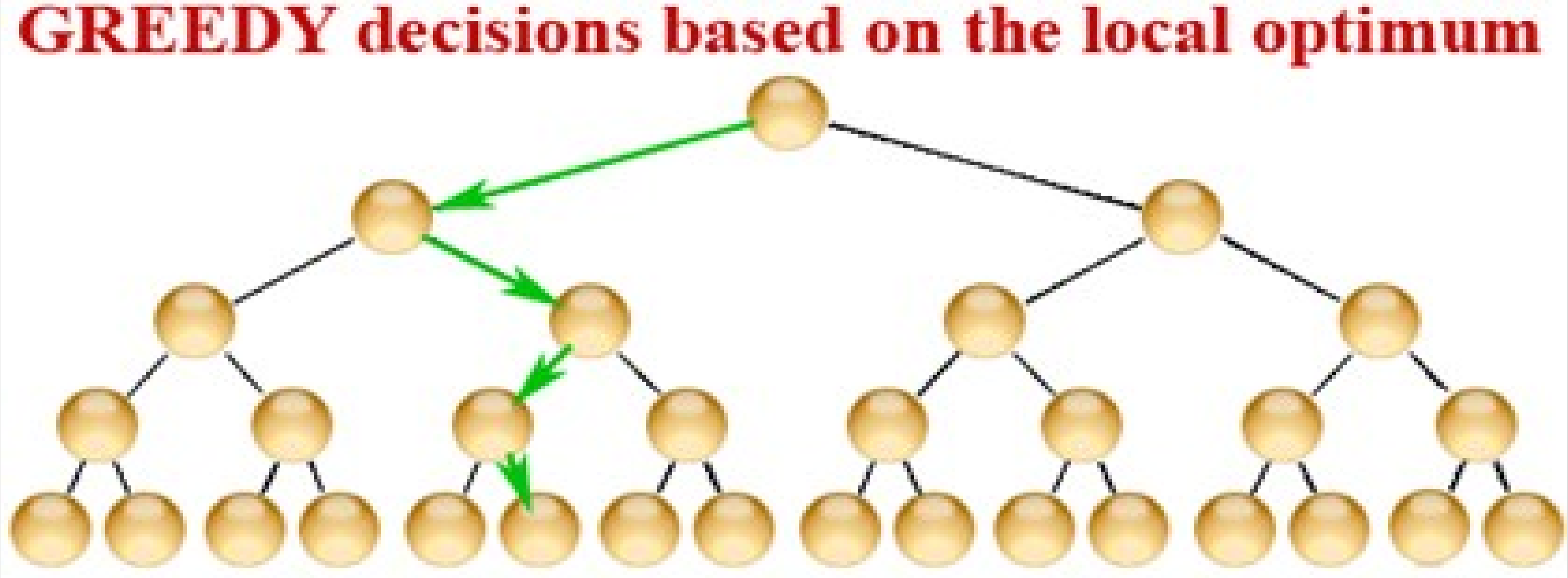
Greedy algorithms



Announcements

Programming assignment 1 posted
- need to submit a .sh file

The .sh file should just contain what you need to type to compile and run your program from the terminal

Greedy algorithms

Find the best solution to a local problem and (hope) it solves the global problem

Greedy algorithm

Greedy algorithms find the global maximum when:

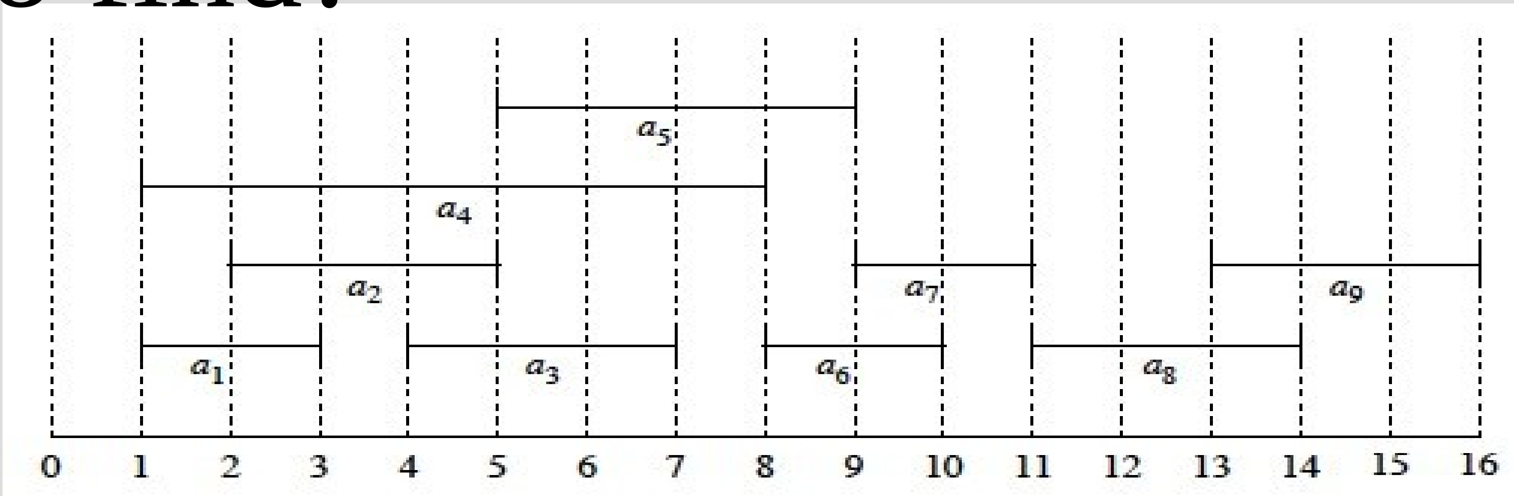
1. optimal substructure – optimal solution to a subproblem is a optimal solution to global problem
2. greedy choices are optimal solutions to subproblems

Activity selection

A list of tasks with start/finish times

Want to finish most number of tasks

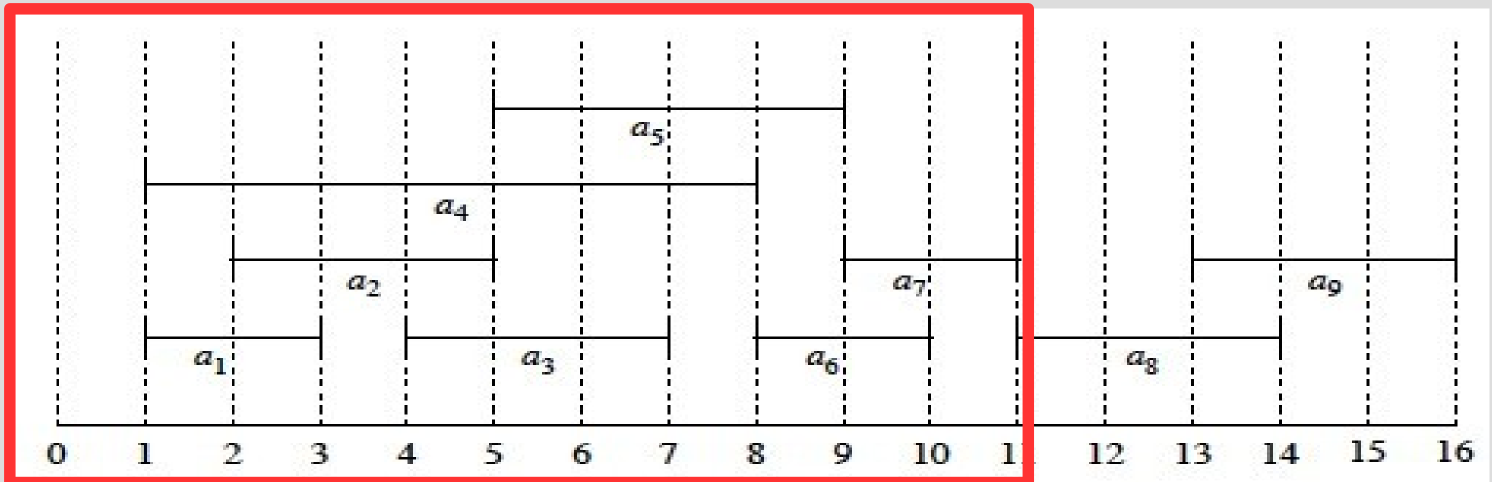
How to find?



Activity selection

Optimal substructure:

Finding the largest number of tasks that finish before time t can be combined with the largest number of tasks that start after time t



Activity selection

Greedy choice:

The task that finishes first is in a optimal solution

Proof:

Suppose we have optimal solution A. If quickest finishing task in A, done. Otherwise we can swap it in.

Activity selection

Greedy: select earliest finish time

Knapsack problem

A list of items with their values, but your knapsack has a weight limit

Goal: put as much value as you can in your knapsack

Knapsack problem

What is greedy choice?

Knapsack problem

What is greedy choice?

A: pick the item with highest value to weight ratio (value/weight) (only optimal if fractions allowed)

Knapsack problem

If you have to choose full items, the constraint of the fixed backpack size is infeasible for greedy solutions

Huffman code

Who has used a zip/7z/rar/tar.gz?

Compression looks at the specific files you want to compress and comes up with a more efficient binary representation

Huffman code

How many letters in alphabet?
How many binary digits do we need?

If we are given a specific set of letters, we can have variable length representations and save space:
aaabaaabaa : a=0,b=1->0001000100
or :aaab=1,a=0 -> 1100

Huffman code

Huffman code uses variable size letter representation compress binary representation on a specific file

letter:	a	b	c	d	e
count:	15	7	6	6	5

What is greedy choice?

Huffman code

We want longer representations for less frequently used letters

Greedy choice: Find least frequently used letters (or group of letters) and assign them an extra 1/0

Repeat until all letters unique encode

Huffman code

1. Merge least frequently used nodes into a single node (usage is sum)

2. Repeat until all nodes on a tree

Huffman code

1. Merge least frequently used nodes into a single node (usage is sum)

2. Repeat until all nodes on a tree

You try!

Huffman code

1. Merge least frequently used nodes into a single node (usage is sum)

2. Repeat until all nodes on a tree

Huffman code

Huffman coding length =
15 * 1 + 3 * 24 = 87

Original coding length =
15 * 3 + 3 * 24 = 117

25 percent compression

Dynamic programming

Greedy algorithms are closely related to dynamic programming

Greedy solutions depend on an optimal subproblem structure

Subproblem structure = recursion, which can be expensive

Dynamic programming

Dynamic programming is turning a recursion into a more efficient iteration

Consider Fibonacci numbers

Dynamic programming

Using recursion leads to repeated calculation: f(n) = f(n-1) + f(n-2)

Instead we can compute from the bottom up:

L=0, C = 1

for 1 to n

  N = C+L, L=C, C=N

Dynamic programming

You can often apply dynamic programming to greedy solutions

Consider the longest “common subsequence problem”:

A = {a, b, b, a, c, c, b, a}

B = {b, c, a, b, a, a, c, a}

Find most matches (in order)

Dynamic programming

Greedy recursive structure:
If end element the same, should always pick

Otherwise, find recursively comparing A with one less or B with one less

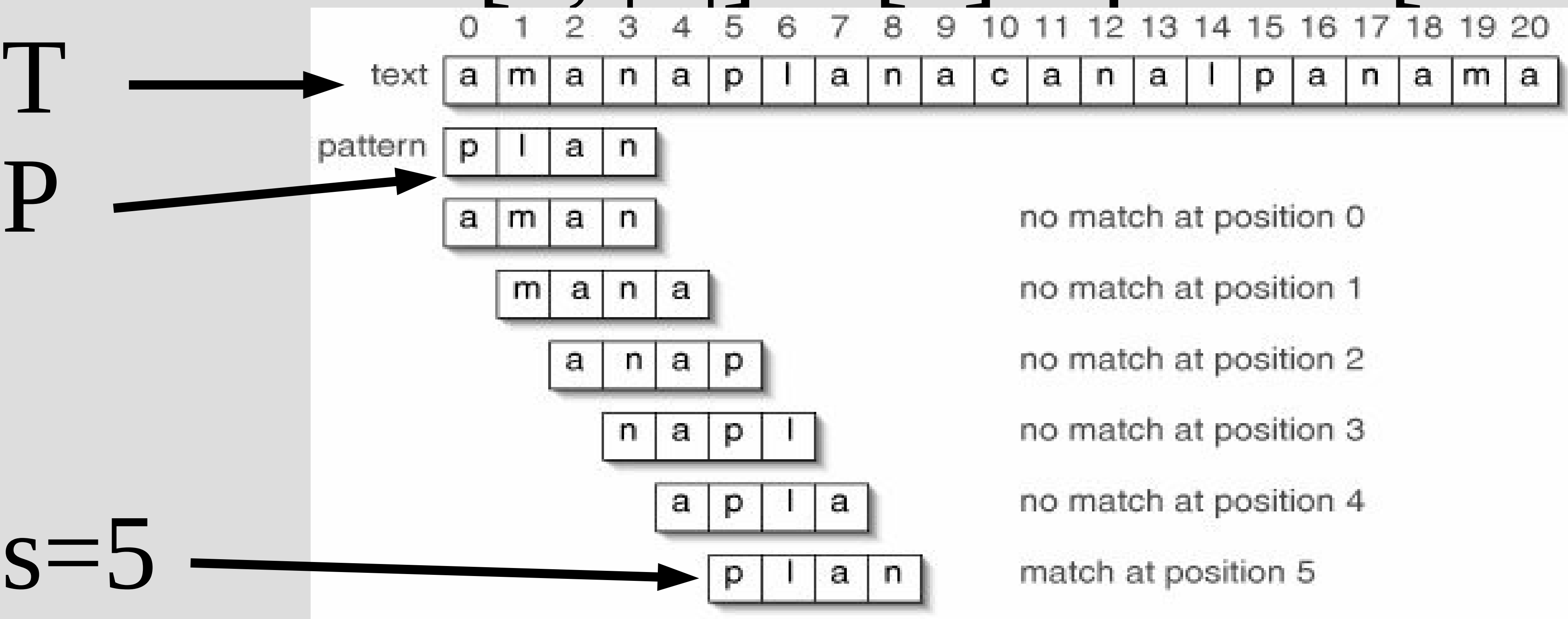
String matching

```
>>>
>>> import re
>>> r=re.compile(r"regexes\s(are|do)\s?(n[o']t)?\s(fun|boring)", re.I)
>>> def is_match(x): return x is not None
>>> is_match(re.match(r, "Regexes are fun!!!"))
True
>>> is_match(re.match(r, "Regexes are not fun!!!"))
True
>>> is_match(re.match(r, "Regexes aren't boring!!!"))
True
>>> is_match(re.match(r, "Obviously, regexes are boring."))
False
>>> is_match(re.search(r, "Obviously, regexes are boring."))
True
```

String matching

Some pattern/string P occurs with shift s in text/string T if:

for all k in [1, |P|]: P[k] equals T[s+k]



String matching

Both the pattern, P, and text, T, come from the same finite alphabet, ∑.

empty string (“”) = ε

$\sqsubset$

w is a prefix of x=w [x, means exists y s.t. wy = x (also implies |w| ≤ |x|)

(w] x = w is a suffix of x)

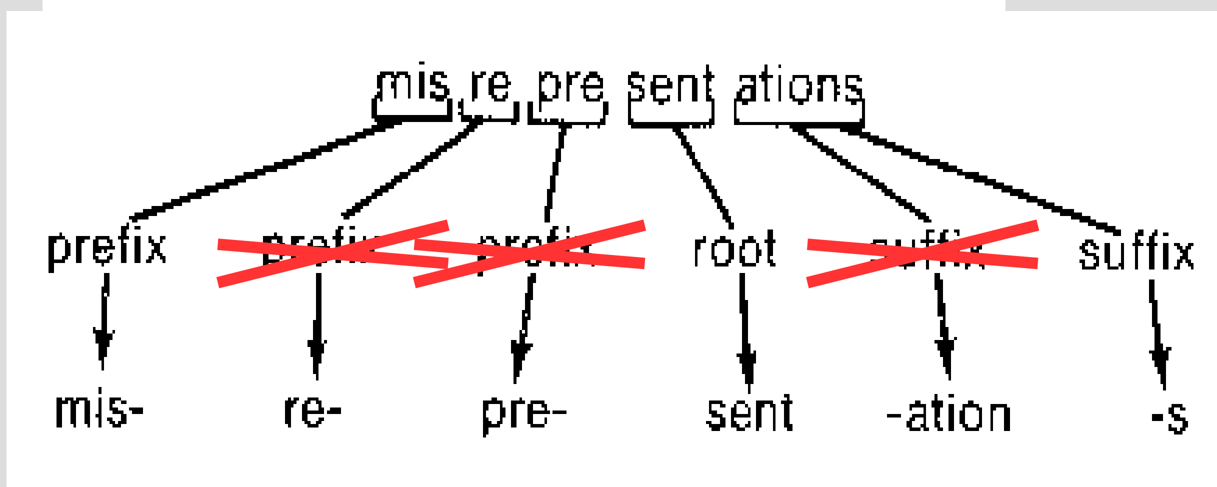
Prefix

w prefix of x means: all the first letters of x are w

x → “**bread**”

prefixes of x → **b , br , bre , brea**

suffixes of x → **read , ead , ad , d**



not english!

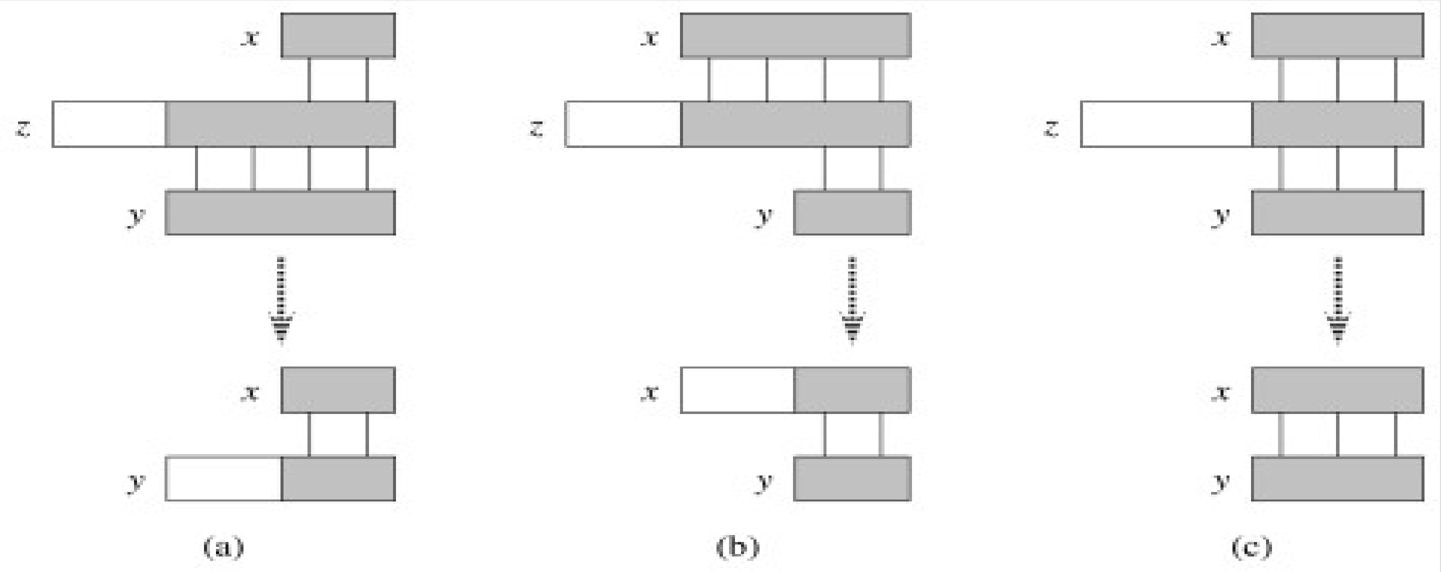
Suffix

If x] z and y] z, then:

(a) If $|x| \leq |y|$, x] y

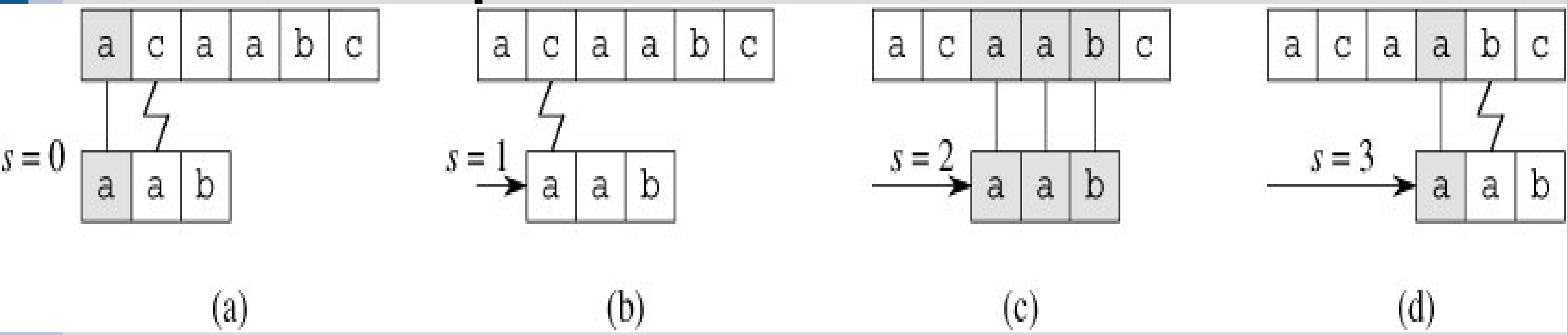
(b) If $|y| \leq |x|$, y] x

(c) If $|x| = |y|$, x = y



Dumb matching

Dumb way to find all shifts of P in T?
Check all possible shifts!

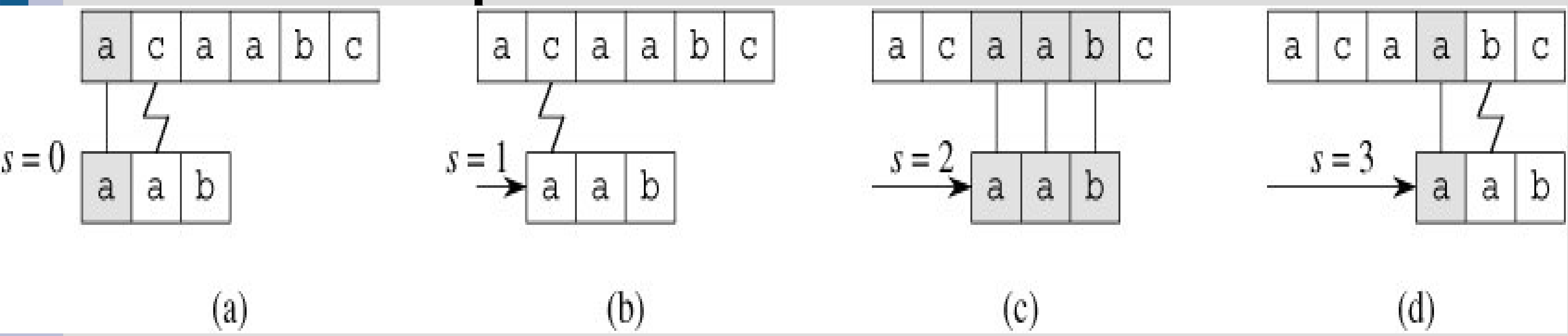


(see: naiveStringMatcher.py)
Run time?

Dumb matching

Dumb way to find all shifts of P in T?

Check all possible shifts!



(see: naiveStringMatcher.py)

Run time?

O(|P| |T|)

Rabin-Karp algorithm

A better way is to treat the pattern as a single numeric number, instead of a sequence of letters

So if P = {1, 2, 6} treat it as 126 and check for that value in T

Rabin-Karp algorithm

The benefit is that it takes a(n almost) constant time to get the each number in T by the following:

(Let t_s = T[s, s+1, ..., s+|P|])

$t_{s+1} = d(t_s – T[s+1]h) + T[s+|P|+1]$

where $d = | \sum |$, $h= d^{|P|-1}$

Rabin-Karp algorithm

Example: $\sum = \{0, 1, ..., 9\}$, $|\sum| = 10$

$T = \{1, 2, 6, 4, 7, 2\}$

$P = \{6, 4, 7\}$

$t_0 = 126$

$t_1 = 10(126\text{-}T[0+1]10^{3-1}) + T[0+|P|+1]$

$t_1 = 10(126\text{-}100) + T[0+3+1]$

$t_1 = 264$

Rabin-Karp algorithm

This is a constant amount of work if the numbers are small...

So we make them small!
(using modulus/remainder)

Any problems?

Rabin-Karp algorithm

This is a constant amount of work if the numbers are small...

So we make them small!
(using modulus/remainder)

Any problems?
x mod q=y mod q does not mean x=y

Hash functions

One way functions

Modulus is a one way function, thus computing the modulus is easy but recovering the original number is hard/impossible

127 % 5 = 2, or 127 mod 5 = 2 mod 5

However if we want to solve x%5=2, all we can say is x=2+5k or some k

One way functions

Other one way functions?

One way functions

Other one way functions?
- multiplication
- hashing

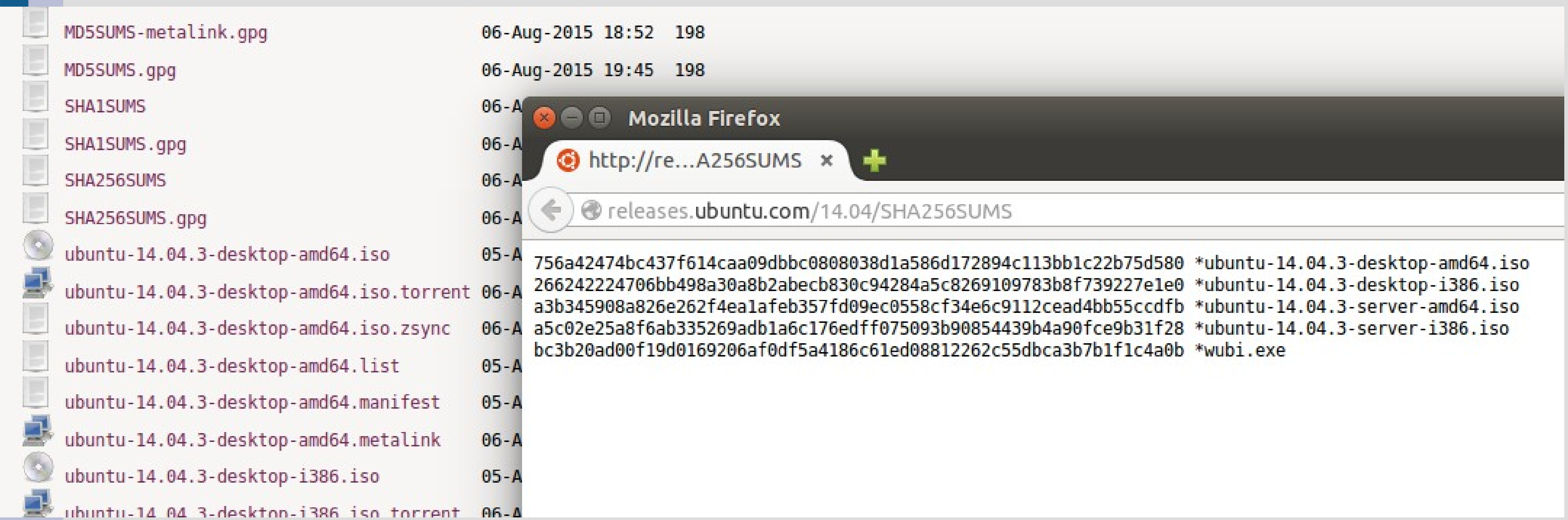
Multiplication is famous, as it is easy:
200*50 = 10,000
... yet factoring is hard:
132773= 31 * 4283 (what alg?)

One way functions

Hashing is another commonly used function for security/verification, as...

- -fast (low computation)
- -low collision chance
- -cannot easily produce a specific hash

One way functions



Hash functions

Rabin-Karp algorithm

Larger q (for mod):
- larger numbers = more computation
- less frequent errors

There are trade-offs, but we often pick $q > |P|$ but not $q >> |P|$

Pick a prime number as q

Rabin-Karp algorithm

Kabin-Karp-Matcher(T,P,|∑|,q,)

d=|∑|, h=d$^{|P|-1}$ mod q, p=0, t_0 = 0

for i=1 to |P| // “preprocessing”

  p = (dp + P[i]) mod q // for P

  t_0 = (dt_0 + T[i]) mod q // for T

for s = 0 to |T| - |P|

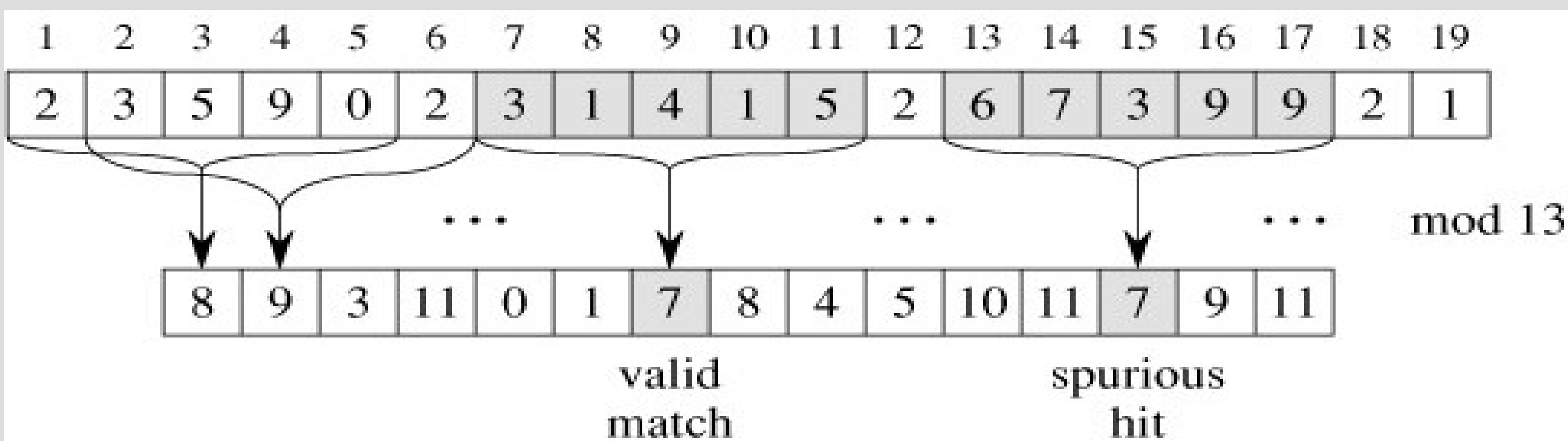
  if p == t_s, check brute-force match at s

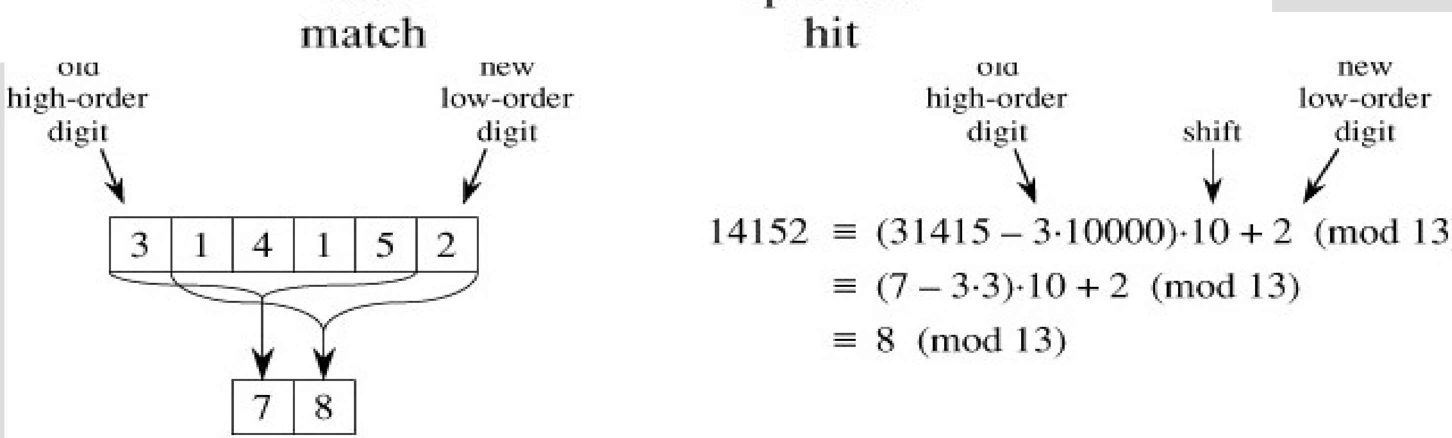
  if s < |T| - |P| then compute t_{s+1}

Rabin-Karp algorithm

To compute t_{s+1}:

t_{s+1}=(d(t_s-t[s+1]h)+T[s+|P|+1]) mod q





Rabin-Karp algorithm

Example: T = {1, 2, 5, 3, 5, 2, 6, 3}

P = {2, 5}, q = 5, assume base 10

Rabin-Karp algorithm

Example: $T = \{1, 2, 5, 3, 5, 2, 6, 3\}$

$P = \{2, 5\}$, $q = 5$, assume base 10

$\underline{P = 25 \bmod 5 = 0}$, $t_0 = 12 \bmod 5 = 2$

$t_{i+1}=10*(t_i-T[i+1]*10)+T[i+|P|+1]\%q$

$t_1 = 25 \bmod 5 = 0$, true match!

$t_2 = 53 \bmod 5 = 3$,

$t_3 = 35 \bmod 5 = 0$, false match

Rabin-Karp algorithm

$T = \{1, 2, 5, 3, 5, 2, 6, 3\}$, $P = \{2, 5\}$

$t_5 = 52 \bmod 5 = 2,$

$t_6 = 26 \bmod 5 = 1,$

$t_7 = 63 \bmod 5 = 3$

$t_{i+1}=10*(t_i-T[i+1]*10)+T[i+|P|+1]\%q$

So only s=1 is match

Rabin-Karp algorithm

Run time? (Average? Worst case?)

Rabin-Karp algorithm

Run time?
- "preprocessing" (first loop)= $O(|P|)$
- "matching" (second loop) = $O(|T|)$

So $O(|T|+|P|)$ and as $n>m$, $O(|T|)$ on average

Worst case: always a match $O(|T|\ |P|)$