Space vs. time

- space/time tradeoffs
- examples: heap sort, data structure redundancy, hashing
- string matching
  - brute force, Horspool algorithm, Boyer-Moore algorithm

for many applications, it is possible to trade memory for speed
- i.e., additional storage can allow for a more efficient algorithm

- ArrayList wastes space when it expands (doubles in size)
  - each expansion yields a list with (slightly less than) half empty
  - improves overall performance since only log N expansions when adding N items

- linked structures sacrifice space (reference fields) for improved performance
  - e.g., a linked list takes twice the space, but provides O(1) add/remove from either end

- heap sort can perform an efficient sort of a list by building a heap then extracting

- hash tables can obtain O(1) add/remove/search if willing to waste space
  - to minimize the chance of collisions, must keep the load factor low
  - HashSet & HashMap resize (& rehash) if load factor every reaches 75%
Space vs. time

in fact, you have made these space/time tradeoffs often in your code

- **AnagramFinder** (find all anagrams of a given word)
  can preprocess the dictionary & build a map of words, keyed by sorted letters
  then, each anagram lookup is $O(1)$

- **BinaryTree** (implement a binary tree & eventually extend to BST)
  kept track of the number of nodes in a field (updated on each add/remove)
  turned the $O(N)$ size operation into $O(1)$

- **OKReads** (store book reviews & look up by reviewer or title)
  to allow for efficient searches by either, needed to build redundant data structures

- other examples?

String matching

many useful applications involve searching text for patterns

- word processing (e.g., global find and replace)
- data mining (e.g., looking for common parameters)
- genomics (e.g., searching for gene sequences)

TTAAGGACCGCATGCCCTCGAGGCTAATTAA

in general

- given a (potentially long) string $S$ and need to find (relatively small) pattern $P$
- an obvious, brute force solution exists: $O(|S| \cdot |P|)$
- however, utilizing preprocessing and additional memory can reduce this to $O(|S|)$
  in practice, it is often $< |S|$ HOW CAN THIS BE???
Brute force

String: FOOBARBIZBAZ
Pattern: BIZ

the brute force approach would shift the pattern along, looking for a match:

FOOBARBIZBAZ
FOOBARBIZBAZ
FOOBARBIZBAZ
FOOBARBIZBAZ
FOOBARBIZBAZ
FOOBARBIZBAZ
FOOBARBIZBAZ

at worst:
|S|-|P|+1 passes through the pattern
each pass requires at most |P| comparisons

\[ O(|S|+|P|) \]

Smart shifting

FOOBARBIZBAZ
BIZ

suppose we look at the rightmost letter in the attempted match
- here, compare Z in the pattern with O in the string
since there is no O in the pattern, can skip the next two comparisons

FOOBARBIZBBAZ
BIZ

again, no R in BIZ, so can skip another two comparisons

FOOBARBIZBBAZ
BIZ
Horspool algorithm

given a string S and pattern P,
1. Construct a shift table containing each letter of the alphabet.
   if the letter appears in P (other than in the last index) → store distance from end
   otherwise, store |P|

   e.g., P = BIZ

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>...</th>
<th>I</th>
<th>...</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>...</td>
<td>1</td>
<td>...</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

2. Align the pattern with the start of the string S, i.e., with $S_0S_1...S_{|P|-1}$

3. Repeatedly,
   compare from right-to-left with the aligned sequence $S_iS_{i+1}...S_{i+|P|-1}$
   if all |P| letters match, then FOUND.
   if not, then shift P to the right by shiftTable[$S_{i+|P|-1}$]
   if the shift falls off the end, then NOT FOUND

Horspool example 1

S = FOOBARBIZBAZ  P = BIZ

shiftTable for "BIZ"

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>...</th>
<th>I</th>
<th>...</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>...</td>
<td>1</td>
<td>...</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

FOOBARBIZBAZ
BIZ
Z and O do not match, so shift shiftTable[O] = 3 spots

FOOBARBIZBAZ
BIZ
Z and R do not match, so shift shiftTable[R] = 3 spots

FOOBARBIZBAZ
BIZ
pattern is FOUND

• total number of comparisons = 5
**Horspool example 2**

S = "FOZIZBARBIZBAZ"  P="BIZ"

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>...</th>
<th>I</th>
<th>...</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>shiftTable for &quot;BIZ&quot;</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>...</td>
<td>1</td>
<td>...</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

**FOZIZBARBIZBAZ**

- **BIZ**
  - Z and Z match, but not I and O, so shiftTable[Z] = 3 spots

**FOZIZBARBIZBAZ**

- **BIZ**
  - Z and B do not match, so shiftTable[B] = 2 spots

**FOZIZBARBIZBAZ**

- **BIZ**
  - Z and R do not match, so shiftTable[R] = 3 spots

**FOZIZBARBIZBAZ**

- **BIZ**
  - pattern is FOUND

- total number of comparisons = 7

---

**Horspool analysis**

**space & time**

- requires storing shift table whose size is the alphabet
- since the alphabet is usually fixed, table requires O(1) space

- worst case is O(|S|*|P|)
  - this occurs when skips are infrequent & close matches to the pattern appear often
- for random data, however, only O(|S|)

**Horspool algorithm is a simplification of a more complex (and well-known) algorithm: Boyer-Moore algorithm**

- in practice, Horspool is often faster
- however, Boyer-Moore has O(|S|) worst case, instead of O(|S|*|P|)
Boyer-Moore algorithm

based on two kinds of shifts (both compare right-to-left, find first mismatch)

- the first is bad-symbol shift (based on the symbol that caused the mismatch)

```
BIZFIZIBIZFIZBIZ
FIZBIZ  F and B don't match, shift to align F

BIZFIZIBIZFIZBIZ
FIZBIZ  I and Z don't match, shift to align I

BIZFIZIBIZFIZBIZ
FIZBIZ  I and Z don't match, shift to align I

BIZFIZIBIZFIZBIZ
FIZBIZ  F and Z don't match, shift to align F

BIZFIZIBIZFIZBIZ
FIZBIZ  FOUND
```

Bad symbol shift

bad symbol table is |alphabet|•|P|

- kth row contains shift amount if mismatch occurred at index k

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th></th>
<th>F</th>
<th></th>
<th>I</th>
<th></th>
<th>Y</th>
<th></th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>2</td>
<td>6</td>
<td>...</td>
<td>5</td>
<td>...</td>
<td>1</td>
<td>...</td>
<td>6</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>...</td>
<td>4</td>
<td>...</td>
<td>-</td>
<td>...</td>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>-</td>
<td>4</td>
<td>...</td>
<td>3</td>
<td>...</td>
<td>2</td>
<td>...</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>...</td>
<td>2</td>
<td>...</td>
<td>1</td>
<td>...</td>
<td>3</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>...</td>
<td>1</td>
<td>...</td>
<td>-</td>
<td>...</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
```

bad symbol table for FIZBIZ:

```
BIZFIZIBIZFIZBIZ
FIZBIZ  F and B don't match → badSymbolTable(F, 2) = 3

BIZFIZIBIZFIZBIZ
FIZBIZ  I and Z don't match → badSymbolTable(I, 0) = 1

BIZFIZIBIZFIZBIZ
FIZBIZ  I and Z don't match → badSymbolTable(I, 3) = 1

BIZFIZIBIZFIZBIZ
FIZBIZ  F and Z don't match → badSymbolTable(F, 0) = 5

BIZFIZIBIZFIZBIZ
FIZBIZ  FOUND
```
Good suffix shift

find the longest suffix that matches
- if that suffix appears to the left in $P$ preceded by a different char, shift to align
- if not, then shift the entire length of the word

<table>
<thead>
<tr>
<th>$P$</th>
<th>$Q$</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>BIZFIZIBIZFIZBIZ</code></td>
<td><code>FIZBIZ</code></td>
<td>IZ suffix matches, IZ appears to left so shift to align</td>
</tr>
<tr>
<td><code>BIZFIZIBIZFIZBIZ</code></td>
<td><code>FIZBIZ</code></td>
<td>no suffix match, so shift 1 spot</td>
</tr>
<tr>
<td><code>BIZFIZIBIZFIZBIZ</code></td>
<td><code>FIZBIZ</code></td>
<td>BIZ suffix matches, doesn't appear again so full shift</td>
</tr>
<tr>
<td><code>BIZFIZIBIZFIZBIZ</code></td>
<td><code>FIZBIZ</code></td>
<td>FOUND</td>
</tr>
</tbody>
</table>

Good suffix shift

good suffix shift table is $|P|$
- assume that suffix matches but char to the left does not
- if that suffix appears to the left preceded by a different char, shift to align
- if not, can shift the entire length of the word

good suffix table for `FIZBIZ`:

<table>
<thead>
<tr>
<th>$Q$</th>
<th>IZ</th>
<th>ZIB</th>
<th>BIZ</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FIZBIZ</code></td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td><code>BIZFIZIBIZFIZBIZ</code></td>
<td>IZ suffix matches $\Rightarrow$ goodSuffixTable(IZ) = 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>BIZFIZIBIZFIZBIZ</code></td>
<td>no suffix match $\Rightarrow$ goodSuffixTable() = 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>BIZFIZIBIZFIZBIZ</code></td>
<td>BIZ suffix matches $\Rightarrow$ goodSuffixTable(BIZ) = 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>BIZFIZIBIZFIZBIZ</code></td>
<td>FOUND</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Boyer-Moore string search algorithm

1. calculate the bad symbol and good suffix shift tables
2. while match not found and not off the edge
   a) compare pattern with string section
   b) shift1 = bad symbol shift of rightmost non-matching char
   c) shift2 = good suffix shift of longest matching suffix
   d) shift string section for comparison by max(shift1, shift2)

the algorithm has been proven to require at most $3|S|$ comparisons
- so, $O(|S|)$
- in practice, can require fewer than $|S|$ comparisons
- requires storing $O(|P|)$ bad symbol shift table and $O(|P|)$ good suffix shift table