CSC 421: Algorithm Design & Analysis

Spring 2014

Space vs. time

- space/time tradeoffs
- examples: heap sort, data structure redundancy, hashing
- string matching

  brute force, Horspool algorithm, Boyer-Moore algorithm

Space vs. time

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

- 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%

Heap sort

can utilize a heap to efficiently sort a list of values

- start with the ArrayList to be sorted
- construct a heap out of the elements
- repeatedly, remove min element and put back into the ArrayList

```
public static <E extends Comparable<? super E>>
void heapSort(ArrayList<E> items) {
    MinHeap<E> itemHeap = new MinHeap<E>();

    for (int i = 0; i < items.size(); i++) {
        itemHeap.add(items.get(i));
    }

    for (int i = 0; i < items.size(); i++) {
        items.set(i, itemHeap.minValue());
        itemHeap.remove();
    }
}
```

- N items in list, each insertion can require O(log N) swaps to reheapify
  →construct heap in O(N log N)
- N items in heap, each removal can require O(log N) swap to reheapify
  →copy back in O(N log N)

can get the same effect by copying the values into a TreeSet, then iterating

3

Other space vs. time examples

we have made similar space/time tradeoffs often in 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)
- KWIC Index (store and display titles based on keywords)
  to allow for efficient searches/displays, store a copy for each keyword
- other examples?

4

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)

TTAAGGACCGCATGCCCTCGAATAGGCTTGAGCTTGCAATTAACGCCCAC…

in general

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

5

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|)

6

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
```

7

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

A	B	C	…	I	…	X	Y	Z
3	2	3	…	1	…	3	3	3

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

8

Horspool example 1

S = FOOBARBIZBAZ P=BIZ

shiftTable for "BIZ"

A	B	C	…	I	…	X	Y	Z
3	2	3	…	1	…	3	3	3

```
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

9

Horspool example 2

S = "FOZIZBARBIZBAZ" P="BIZ"

shiftTable for "BIZ"

A	B	C	…	I	…	X	Y	Z
3	2	3	…	1	…	3	3	3

```
FOZIZBARBIZBAZ
BIZ
```
Z and Z match, but not I and O, so shift shiftTable[Z] = 3 spots

```
FOZIZBARBIZBAZ
   BIZ
```
Z and B do not match, so shift shiftTable[B] = 2 spots

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

```
FOZIZBARBIZBAZ
        BIZ
```
pattern is FOUND

- total number of comparisons = 7

suppose we are looking for BIZ in a string with no Z's how many comparisons?

10

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|)

11

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
```

12

Bad symbol shift

bad symbol table is |alphabet|•|P|

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

bad symbol table for FIZBIZ:

	A	B	C	…	F	…	I	…	Y	Z
0	6	2	6	…	5	…	1	…	6	3
1	5	1	5	…	4	…	-	…	5	2
2	4	-	4	…	3	…	2	…	4	1
3	3	3	3	…	2	…	1	…	3	-
4	2	2	2	…	1	…	-	…	2	2

```
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

```
BIZFIZIBIZFIZBIZ
FIZBIZ                IZ suffix matches, IZ appears to left so shift to align

BIZFIZIBIZFIZBIZ
   FIZBIZ                       no suffix match, so shift 1 spot

BIZFIZIBIZFIZBIZ
    FIZBIZ            BIZ suffix matches, doesn't appear again so full shift

BIZFIZIBIZFIZBIZ
          FIZBIZ      FOUND
```

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, then can shift the entire length of the word

good suffix table for FIZBIZ

IZBIZ	ZBIZ	BIZ	IZ	Z	
6	6	6	3	6	1

```
BIZFIZIBIZFIZBIZ
FIZBIZ              IZ suffix matches → goodSuffixTable(IZ) = 3

BIZFIZIBIZFIZBIZ
   FIZBIZ           no suffix match → goodSuffixTable() = 1

BIZFIZIBIZFIZBIZ
    FIZBIZ          BIZ suffix matches → goodSuffixTable(BIZ) = 6

BIZFIZIBIZFIZBIZ
          FIZBIZ    FOUND
```

15

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

16