CSC 421: Algorithm Design Analysis

Spring 2017

Transform & conquer

- transform-and-conquer approach
  - presorting, balanced search trees, heaps, Horner's Rule
  - problem reduction
- space/time tradeoffs
  - heap sort, data structure redundancy, hashing
  - string matching: Horspool algorithm, Boyer-Moore algorithm

Transform & conquer

the idea behind transform-and-conquer is to transform the given problem into a slightly different problem that suffices

e.g., presorting data in a list can simplify many algorithms

- suppose we want to determine if a list contains any duplicates

  BRUTE FORCE: compare each item with every other item

  $(N-1) + (N-2) + \ldots + 1 = (N-1)N/2 \rightarrow O(N^2)$

  TRANSFORM & CONQUER: first sort the list, then make a single pass through the list and check for adjacent duplicates

  $O(N \log N) + O(N) \rightarrow O(N \log N)$

- finding the mode of a list, finding closest points, …

Balanced search trees

recall binary search trees – we need to keep the tree balanced to ensure O(N log N) search/add/remove

- OR DO WE?
- it suffices to ensure O(log N) height, not necessarily minimal height

transform the problem of "tree balance" to "relative tree balance"

several specialized structures/algorithms exist:

- AVL trees
- 2-3 trees
- red-black trees

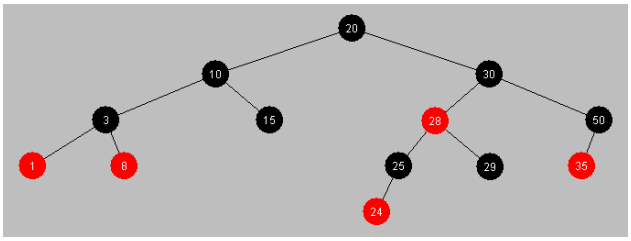
3

Red-black trees

a red-black tree is a binary search tree in which each node is assigned a color (either red or black) such that

1. *the root is black*
2. *a red node never has a red child*
3. *every path from root to leaf has the same number of black nodes*

- add & remove preserve these properties (complex, but still O(log N))
- red-black properties ensure that tree height < 2 log(N+1) → O(log N) search



4

TreeSets & TreeMaps

`java.util.TreeSet` uses *red-black trees* to store values

➔ O(log N) efficiency on add, remove, contains

`java.util.TreeMap` uses *red-black trees* to store the key-value pairs

➔ O(log N) efficiency on put, get, containsKey

thus, the original goal of an efficient tree structure is met

- even though the subgoal of balancing a tree was transformed into "relatively balancing" a tree

5

Scheduling & priority queues

many real-world applications involve optimal scheduling

- balancing transmission of multiple signals over limited bandwidth
- selecting a job from a printer queue
- selecting the next disk sector to access from among a backlog
- multiprogramming/multitasking

a *priority queue* encapsulates these three optimal scheduling operations:

✓ *add item (with a given priority)*
✓ *find highest priority item*
✓ *remove highest priority item*

- can be implemented as an unordered list
  → add is O(1), findHighest is O(N), removeHighest is O(N)
- can be implemented as an ordered list
  → add is O(N), findHighest is O(1), removeHighest is O(1)

6

Heaps

Java provides a `java.util.PriorityQueue` class

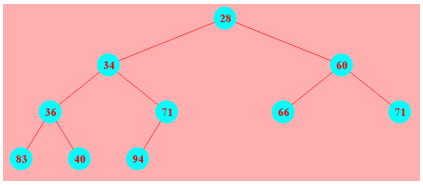
- the underlying data structure is not a list or queue at all
- it is a tree structure called a *heap*

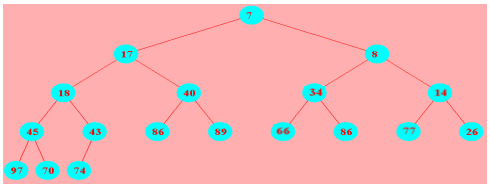
a *complete tree* is a tree in which

- all leaves are on the same level or else on 2 adjacent levels
- all leaves at the lowest level are as far left as possible
- note: a complete tree with N nodes will have minimal height = $\lfloor \log_2 N \rfloor + 1$

a *heap* is complete binary tree in which

- for every node, the value stored is ≤ the values stored in both subtrees
  (technically, this is a min-heap -- can also define a max-heap where the value is ≥)



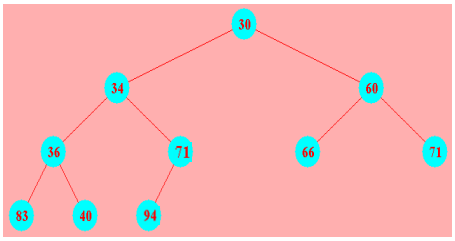


7

Implementing a heap

a heap provides for O(1) find min, O(log N) insertion and min removal

- also has a simple, List-based implementation
- since there are no holes in a heap, can store nodes in an ArrayList, level-by-level



30	34	60	36	71	66	71	83	40	94

- root is at index `0`
- last leaf is at index `size()-1`
- for a node at index `i`, children are at `2*i+1` and `2*i+2`
- to add at next available leaf, simply add at end

8

Horner's rule

polynomials are used extensively in mathematics and algorithm analysis

$p(x) = a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1 x + a_0$

how many multiplications would it take to evaluate this function for some value of x?

W.G. Horner devised a new formula that transforms the problem

$p(x) = (\ldots (((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0$

can evaluate in only n multiplications and n additions

9

Problem reduction

in CSC321, we looked at a number of examples of reducing a problem from one form to another

- e.g., generate the powerset (set of all subsets) of an N element set

$S = \{ x_1, x_2, x_3, x_4 \}$ $\quad powerset_S = \{ \{\}, \{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}, \{x_1, x_2\}, \{x_1, x_3\}, \{x_1, x_4\}, \{x_2, x_3\}, \{x_2, x_4\}, \{x_3, x_4\}, \{x_1, x_2, x_3\}, \{x_1, x_2, x_4\}, \{x_1, x_3, x_4\}, \{x_2, x_3, x_4\}, \{x_1, x_2, x_3, x_4\} \}$

- PROBLEM REDUCTION: simplify by reducing it to a problem about bit sequences
  can map each subset into a sequence of N bits: $b_i = 1 \rightarrow x_i$ in subset

$\{ x_1, x_4, x_5 \} \leftarrow\rightarrow 10011000\ldots0$

much simpler to generate all possible N-bit sequences

{ 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111,
1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111 }

10

lcm & gcd

consider calculating the least common multiple of two numbers m & n

- BRUTE FORCE: reduce each number to its prime factors
  then multiply (factors in both m & n) (factors only in m) (factors only in n)

  24 = 2 • 2 • 2 • 3
  60 = 2 • 2 • 3 • 5
  lcm(24, 60) = (2 • 2 • 3) • (2) • (5) = 12 • 2 • 5 = 120

- PROBLEM REDUCTION: can recognize a relationship between lcm & gcd

  lcm(m, n) = m x n / gcd(m, n)

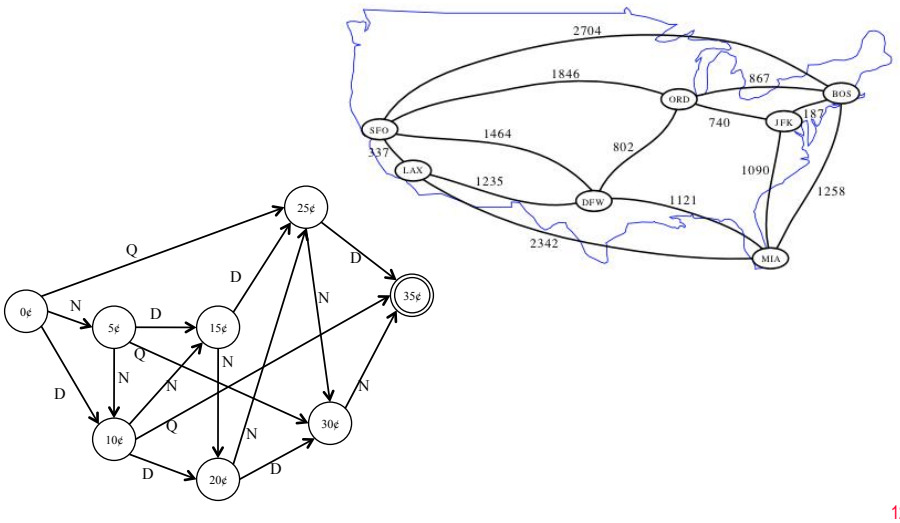
  lcm(24, 60) = 24 • 60 / 12 = 2 • 60 = 120

  gcd can be calculated efficiently using Euclid's algorithm:
  gcd(a, 0) = a gcd(a, b) = gcd(b, a % b)

11

Reduction to graph searches

many problems can be transformed into graph problems

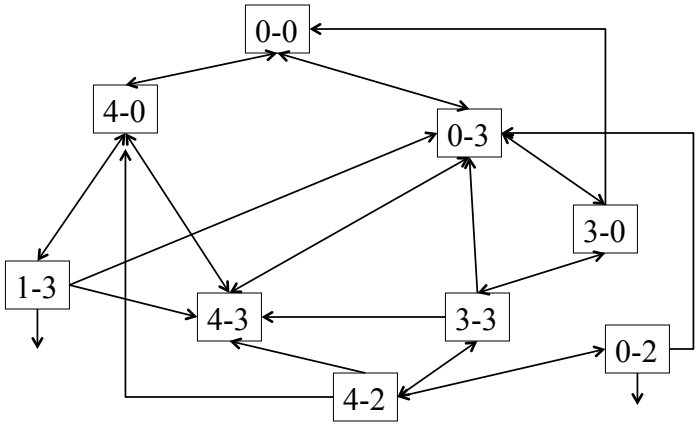


12

Water jug problem

recall from *Die Hard with a Vengeance*

- you have two empty water jugs (4 gallon & 3 gallon capacity) & water supply
- want to end up with exactly 2 gallons in a jug



13

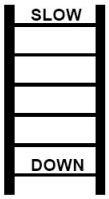
Other interesting examples

multiplication can be transformed into squaring

$$a \times b = ((a + b)^2 - (a - b)^2) / 4$$

find the shortest word ladder

- construct a graph:
  vertices are the words in the dictionary
  there is an edge between v_i and v_j if they differ by one letter
- perform a breadth first search, from start word to end word



Andrew's algorithm for complex hull

- transforms the graph into a sorted list of coordinates, then traverses & collects

14

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%

15

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

16

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)

- HoopsStats (HW1)
  could redundantly store team stats and individual stats

- other examples?

17

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?!?

18

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|)

19

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
```

20

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

21

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

22

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

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

- total number of comparisons = 7

23

Horspool exercise

String: BANDANABABANANAN

Pattern: BANANA

shift table for BANANA?

A	B	C	D	…	N	…	Y	Z
				…		…		

steps?

```
BANDANABABANANAN
BANANA
```

24

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|)

25

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 in S 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
```

26

Bad symbol shift

bad symbol table is |alphabet|•|P|

- kth row contains shift amount if mismatch occurred at index k (from right)

bad symbol table for FIZBIZ:

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

27

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
```

28

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
```

29

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

30

Boyer-Moore exercise

String: BANDANABABANANAN

Pattern: BANANA

bad symbol table for BANANA?

	A	B	C	D	…	N	…	Y	Z
0					…		…		
1					…		…		
2					…		…		
3					…		…		
4					…		…		

good suffix table for BANANA?

ANANA	NANA	ANA	NA	A	

steps?

BANDANABABANANAN

BANANA

31