

MA/CSSE 473

Day 25



Problem Reduction

**Space-time
Tradeoffs**

String Search

MA/CSSE 473 Day 25

- HW 11 due Friday
- Exam 2 is Friday, Oct 31
- **Student Questions**
- Problem reduction
- Space-time tradeoffs
- String search



Recap: HeapSort: Build Initial Heap

- Two approaches:
 - for $i = 2$ to n
 `percolateUp(i)`
 - for $j = n/2$ downto 1
 `percolateDown(j)`
- Which is faster, and why?
- What does this say about overall big-theta running time for HeapSort



Polynomial Multiplication: Horner's Rule

- We discussed it in class in response to problem 3.1.4 in Homework 3
- It involves a representation change.
- Instead of $a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$, which requires a lot of multiplications, we write
- $(\dots (a_n x + a_{n-1})x + \dots + a_1)x + a_0$
- code on next slide



Polynomial Multiplication: Horner's Rule

- This is clearly $\Theta(n)$.

```
def polyEvalHorner(p, x):  
    """ p is a list representing the coefficients.  
        p[i] is the coefficient of x^i.  
        x is where we are to evaluate p. """  
    sum = 0  
    for i in range(len(p)-1, -1, -1):  
        sum = sum * x + p[i]  
  
    return sum  
  
# evaluate 4x^3 + 3x^2 + 2x + 1 at x=2  
print polyEvalHorner([1, 2, 3, 4], 2)
```



Problem Reduction

- Express an instance of a problem in terms of an instance of another problem that we already know how to solve.
- Example: In quickhull, we reduced the problem of determining whether a point is to the left of a line to the problem of computing a simple 3×3 determinant.



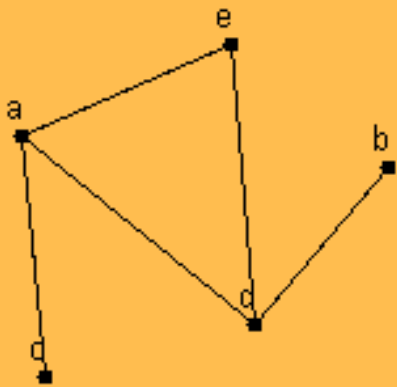
Least Common Multiple

- Let m and n be integers. Find their LCM.
- Factoring is hard.
- But we can reduce the LCM problem to the GCD problem, and then use Euclid's algorithm.
- Note that $\text{lcm}(m,n) \cdot \text{gcd}(m,n) = m \cdot n$
- This makes it easy to find $\text{lcm}(m,n)$



Paths and Adjacency Matrices

- We can count paths from A to B in a graph by looking at powers of the graph's adjacency matrix.



	a	b	c	d	e
a	0	0	1	1	1
b	0	0	0	1	0
c	1	0	0	0	0
d	1	1	0	0	1
e	1	0	0	1	0

	a	b	c	d	e
a	3	1	0	1	1
b	1	1	0	0	1
c	0	0	1	1	1
d	1	0	1	3	1
e	1	1	1	1	2

For this example, I used from an applet from <http://oneweb.utc.edu/~Christopher-Mawata/petersen2/lesson7.htm>



Linear programming

- We want to maximize/minimize a linear function $\sum_{i=1}^n c_i x_i$, subject to **constraints**, which are linear equations or inequalities involving the n variables x_1, \dots, x_n .
- The constraints define a region, so we seek to maximize the function within that region.
- If the function has a maximum or minimum in the region it happens at one of the vertices of the convex hull of the region.
- The simplex method is a well-known algorithm for solving linear programming problems. We probably will not deal with it in this course.
- The Operations Research courses cover linear programming in some detail.



Integer Programming

- A linear programming problem is called an **integer programming** problem if the values of the variables must all be integers.
- The knapsack problem can be reduced to an integer programming problem:
- maximize $\sum_{i=1}^n x_i v_i$ subject to the constraints
 $\sum_{i=1}^n x_i w_i < W$ and $x_i \in \{0, 1\}$ for $i=1, \dots, n$



Space vs time tradeoffs

- Often we can find a faster algorithm if we are willing to use additional space.
- Give some examples (quiz question)
- Examples:
 - Binary heap vs simple sorted array. Uses one extra array position
 - Merge sort
 - Radix sort and Bucket Sort
 - Anagram solver
 - Binary Search Tree (extra space for the pointers)
 - AVL Tree (extra space for the balance code)



Faster String Searching

- The problem: Search for the first occurrence of a **pattern** of length m in a **text** of length n .
- Usually, m is much smaller than n .
- Brute force: **worst case $m(n-m+1)$**
- Average: **$\Theta(mn)$**

```
def search(pattern, text):  
    n, m = len(text), len(pattern)  
    for i in range(n-m+1):  
        j = 0  
        while j < m and text[i+j] == pattern[j]:  
            j += 1  
        if j == m:  
            return i  
    return False
```



Horspool's Algorithm

- A simplified version of the Boyer-Moore algorithm
- A good bridge to understanding Boyer-Moore
- Published in 1980
- What makes brute force so slow?
 - When we find a mismatch, we can shift the pattern over by only one character position in the text.
 - **Text:** abracadabtabradabracadabcbadaxbrabbracadabraxxxxxxabracadabracadabra
Pattern: **abracadabra**
 abracadabra
 abracadabra
 abracadabra
- Like Boyer-Moore, Horspool does the comparisons in a counter-intuitive order (moves right-to-left through the pattern)



Horspool's Main Question

- If there is a character mismatch, how far can we shift the pattern, with no possibility of missing a match within the text?
- What if the last character in the pattern is compared with a character in the text that does not occur in the pattern at all?
- Text : . . . ABCDEFG . . .
Pattern: BOUTELL



How Far to Shift?

- Look at first (rightmost) character in text that was compared:

- The character is not in the pattern

. *C* (*C* not in pattern)

AOBAB

- The character is in the pattern (but not the rightmost)

. *O* (*O* occurs once in pattern)

BAOBAB

. *A* (*A* occurs twice in pattern)

BAOBAB

- The rightmost characters do match

. *B*

BAOBAB

