MA/CSSE 473 Day 25

HeapSort

Problem Reduction

Space-time Tradeoffs

String Search Intro

MA/CSSE 473 Day 25

Assignment	Old due date	New due date
10	Tuesday, Oct 19	Wednesday, Oct 20
Convex Hull	Thursday, Oct 21	Friday, Oct 22
11	Saturday, Oct 23	Tuesday, Oct 26
12	Tuesday, Oct 26	Thursday, Oct 28
13	Thursday, Oct 28	Wednesday, Nov 3

- **Why the delays?** See the email from Sunday night
- **Take-home exam** available by Oct 29 (Friday) at 9:55 AM, due Nov 1 (Monday) at 8 AM.
- **Student Questions**
- Heapsort conclusion
- Problem reduction
- String Search

Recap: Binary Heap Quick Review

See also Weiss, Chapter 21

- An almost-complete Binary Tree
  - All levels, except possibly the last, are full
  - On the last level all nodes are as far left as possible
  - No parent is smaller than either of its children
  - A great way to represent a Priority Queue
- Representing a binary heap as an array:

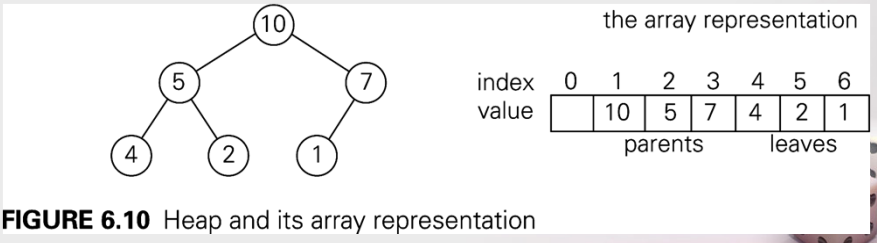


FIGURE 6.10 Heap and its array representation

Q1-3

Insertion and RemoveMax

- Insertion:
  - Insert at the next position to maintain an almost-complete tree, then "percolate up" to restore heap property.
- RemoveMax:
  - Move last element of the heap to the root, then "percolate down" to restore heap property.
- Both operations are Θ(log n).
- Demo: http://www.cs.auckland.ac.nz/software/AlgAnim/heaps.html

Heapsort: Quick look at code

```
def percolateUp(a,n):
    'Assume that elements 1 through n-1 are a heap; add element n and "re-heapify"'
    # compare to parent and swap until not larger than parent.
    current = n
    while current > 1:  # or until this value is in the root.
        if a[current//2] >= a[current]:
            break
        swap(a, current, current//2)
        current //= 2
```

```
def percolateDown(a,i, n):
  """Within the n elements to be "heapified", the two subtrees of A[i] are already maxheaps.
     Repeatedly exchange the element currently in a[i] with the largest of its children
     until the tree whose root is a[i] is a max heap. """
  current = i # root position for subtree we are heapifying
  lastNodeWithChild = n//2  # if a node number is higher than this, it is a leaf.
  while current <= lastNodeWithChild:
      max = current
      if a[max] < a[2*current]:  # if it is larger than its left child.
          max = 2*current
      if 2*current < n and a[max] < a[2*current+1]:  # But right child may be larger than eithe
          max = 2*current + 1                        # if there is a right child.
      if max == current:
          break  # larger than its children, so we are done.
      swap(a, current, max) # otherwise, exchange and move down the tree to check again.
      current = max
```

Code is on-line, linked from schedule page

HeapSort

- Arrange array into a heap
- Starting from the root, remove each element from the heap and move to the end of the array.
- Animation: http://www.cs.auckland.ac.nz/software/AlgAnim/heapsort.html
- Faster heap building algorithm: **buildheap** http://students.ceid.upatras.gr/~perisian/data_structure/HeapSort/heap_applet.html

Q10-12

More HeapSort Code

```
# The next two functions do the same thing; take an unordered array and turn it into a max-hea
# In HW 10, problem 5, you show that one of these is a lot more efficient than the other.
# So this first function is not actually called in this code.
def heapifyByInsert(a, n):
    """ Repeatedly insert elements into the heap.
        Worst case number of element exchanges: sum of depths of nodes."""
    for i in range(2, n+1):
        percolateUp(a, i)

def buildHeap(a, n):
    """ Each time through the loop, each of node i's two subtreees is already a heap.
        Find the right position to move the root down to to "reheapify."
        Worst case number of element exchanges: sum of heights of nodes."""
    for i in range (n//2, 0, -1):
        percolateDown(a, i, n)

def heapSort(a, n):
    buildHeap(a, n)
    for i in range(n, 1, -1):
        swap(a, 1, i)
        percolateDown(a, 1, i-1)
```

Q4

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 Evaluation

Problem Reductiion

TRANSFORM AND CONQUER

Recap: Horner's Rule

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

Recap: Horner's Rule Code

- This is clearly Θ(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

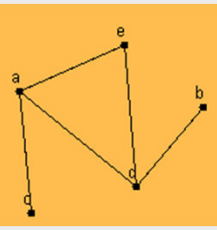
- Express an instance of a problem in terms of an instance of another problem that we already know how to solve.
- There needs to be a one-to-one mapping between problems in the original domain and problems in the new domain.
- **Example:** In quickhull, we reduced the problem of determining whether a point is to the left of a line to the problem of computing a simple 3x3 determinant.
- **Example:** Moldy chocolate problem in HW 10. The big question: What problem to reduce it to? (You'll answer that one in the homework)

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 lcm(m,n)·gcd(m,n) = m·n
- This makes it easy to find 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 the applet from http://oneweb.utc.edu/~Christopher-Mawata/petersen2/lesson7.htm, which is no longer accessible

Q5

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 $x_1,\ldots,x_n$.
- 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 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, …, n

Often using a little more space saves a lot of time

SPACE-TIME TRADEOFFS

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:

Q6

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: **Θ(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
```

Q7

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:** abracadabtabradabracadabcadaxbrabbracadabraxxxxxxabracadabracadabra
    Pattern: abracadabra
    abracadabra
    abracadabra
    abracadabra
- Can we shift farther?
  Like Boyer-Moore, Horspool does the comparisons in a counter-intuitive order (moves right-to-left through the pattern)

Q8-11

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: CSSE473

How Far to Shift?

- Look at first (rightmost) character in the part of the text that is compared to the pattern:
- The character is not in the pattern

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

  BAOBAB
- 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