CS 173: Discrete Structures

Eric Shaffer

Office Hour: Wed. 12-1, 2215 SC

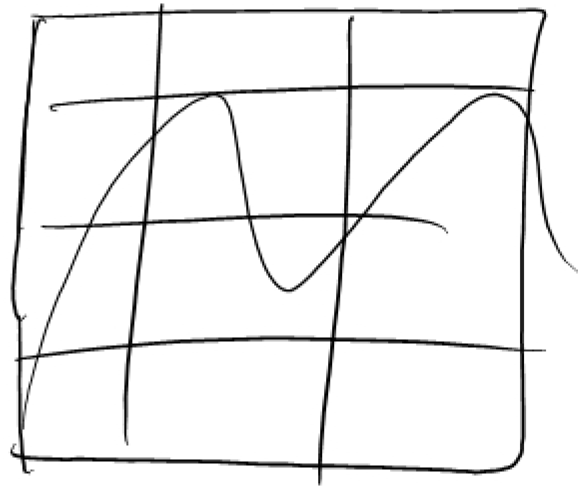
shaffer1@illinois.edu

Announcements

it's short

- HW 7 released today, due April 3
- Today:
  - Algorithms (Section 3.1, 3.3)
    - Sorting
  - Recursive algorithms (Section 4.4)

How parallelism impacts performance



- We parallelize an algorithm that takes time T(n)
  - Across m processors
  - T(n)/m time is the best we can hope for..usually
    - Superlinear speedup is sometimes (rarely) possible…why?
- Even linear speedup is rarely realized
  - Not every part of a program can be parallelized
    - Amdahl's Law: speedup=1/((1-P)+(P/m))
    - P=percentage of program parallelizable
    - **If P=90%, speedup maxes out at factor of 10**
  - Parallelizing also usually introduces communication overhead

Running times

But computers are getting faster! Maybe we can do better.

It's fun to make comparisons about the running times of algorithms of various complexities.

Inp size / compxity	10	20	30	40	50	60
$\Theta(n)$	.00001s	.00002s	.00003s	.00004s	.00005s	.00006s
$\Theta(n^2)$	.0001s	.0004s	.0009s	.0016s	.0025s	.0036s
$\Theta(n^5)$	.1s	3.2s	24.3s	1.7m	5.2m	13m
$\Theta(3^n)$	.059s	58m	6.5y	3855c	$2\text{x}10^8$c	$1.3\text{x}10^{13}$ c

Running times

Compare the sizes of problems solvable in 1 hour now, vs the size of problem we could solve if we had a 100 times faster machine.

Algorithmic complexity	Input size we can solve w today's machines (1hr)	Input size we can solve w. 100x faster machines (1hr)
n	N_1	$100xN_1$
n^2	N_2	$10xN_2$
n^5	N_3	$2.5xN_3$
3^n	N_4	$\boxed{N_4 + 4.19}$

A little more about algorithmic complexity

Tractable problems can be solved in polynomial time: class P

Intractable problems have worst case time complexity > polynomial

Understand the difference between general problem and a specific instance.

Problems in class NP can a have a solution checked in polynomial time

It is unknown if P = NP

one of the outstanding mathematical questions of our time

Some problems are unsolvable

the Halting Problem: will an arbitrary algorithm always halt

probably

$P \neq NP$

Binary Search

```
i := 1
j := n
while (i < j)
   m := ⌊(i + j)/2⌋ {midpt of range (i,j)}
   if x > a_m then i := m + 1
   else j := m
if x = a_i then position:=i
else position:=0 {not in list}
```

Binary search 4,7,8,10,12,14,20 for 14:

i	j	m
1	7	4
5	7	6
5	6	5
6	6	

Review: Searching

In analyzing algorithmic complexity we look at:

Number of operations as a function of input size

Name two types of analysis:

worst

average

Perform Binary Search on a 1,048,576 number list...

In the worst case, approximately how many comparisons ≈ 20

- for binary search
- for sequential search

1 048 576

Bubble Sort

Bubble Sort

Input: a unsorted array of real numbers $a_1, a_2, \ldots a_n$, $n>1$

Output: a sorted array of real numbers $a_1 \le a_2 \le \ldots \le a_n$

```
for i =1 to n-1
   for j = 1 to n-i
      if a_j > a_{j+1} then swap a_j and a_{j+1}
```

Bubble sort example 4,1,10,3

i=1 j = 1 to3

4, 1, 10, 3

1, 4, 10, 3

1 4 10 3

1 4 3 10

i=2 j = 1 to2

1 4 3 10

1 4 3 10

1 3 4 10

i=3 j=1 to 1

1 3 4 10

Bubble Sort

Bubble Sort

Input: a unsorted array of real numbers $a_1, a_2, \ldots a_n$, $n>1$

Output: a sorted array of real numbers $a_1 \leq a_2 \leq \ldots \leq a_n$

```
for i:=1 to n-1
   for j:= 1 to n-i
      if a_j > a_{j+1} then swap a_j and a_{j+1}
```

$\underbrace{\text{swap } a_j \text{ and } a_{j+1}}$ $\Big\} \Theta(1)$

comparisons & arithmetic ops $= \Theta(1)$

swap $= \Theta(1)$

Bubble sort worst case Complexity:

i	# compars
$i = 1$	$n-1$
2	$n-2$
$\vdots$	$\vdots$
$n-1$	1

$$\sum_{i=1}^{n-1} = \frac{(n-1)n}{2} \boxed{= \Theta(n^2)}$$

Bubble Sort

Bubble Sort

Input: a unsorted array of real numbers $a_1, a_2, \ldots a_n$, n>1

Output: a sorted array of real numbers $a_1 \leq a_2 \leq \ldots \leq a_n$

```
1. for i =1 to n-1
2.    for j = 1 to n-i
3.        if a_j > a_{j+1} then swap a_j and a_{j+1}
```

What is the bubble sort average case performance?

$\Theta(n^2)$

Review: Searching

We've seen two searching algorithms:

- binary search
- sequential (or linear) search.

What is an advantage of binary search over linear search?

$\Theta(\lg n)$ vs $\Theta(n)$

When is this not a such a great advantage?

if n is small

What is an advantage of sequential search over binary?

works on unsorted lists

Insertion Sort

Insertion Sort

Input: a unsorted array of real numbers $a_1, a_2, \ldots a_n$, $n>1$

Output: a sorted array of real numbers $a_1 \le a_2 \le \ldots \le a_n$

Example:

$\underset{\text{sorted}}{\boxed{7}}$, $\underbrace{2, 6, 4, 1}_{\text{unsorted}}$

$\underbrace{2, 7}$, 6, 4, 1

$\underbrace{2, 6, 7}$, 4, 1

$\underbrace{2, 4, 6, 7}$, 1

$\boxed{1, 2, 4, 6, 7}$

Insertion Sort

Insertion Sort

Input: a unsorted array of real numbers $a_1, a_2, \ldots a_n$, n>1

Output: a sorted array of real numbers $a_1 \le a_2 \le \ldots \le a_n$

```
for j:=2 to n
begin
    i:=1
    while aj>ai
        i:=i+1
    m := aj
    for k := 0 to j-i-1
        aj-k := aj-k-1
    ai:=m
end
```

finds the spot in sorted to insert a_j

copies a_j to temp

moves elements of list up

puts temp value in right place

Insertion Sort Analysis

```
for j =2 to n
   i=1
   while aj>ai
      i=i+1
   m = aj
   for k = 0 to j-i-1
      aj-k = aj-k-1
   ai=m
```

$a_1 \; a_2 \; a_3 \; a_4$

1, 2, 3, 4

2 compars

3

4

What input produces the best running time?

sorted order

$$\# \text{compares} = \sum_{i=2}^{n} i = \Theta(n^2)$$

What input produces the worst?

4, 3, 2, 1 still $\Theta(n^2)$

Recursive algorithms...

A recursive algorithm solves a problem by reducing it to another instance of the same problem on a smaller input.

<u>Example:</u>

Factorial (n)

Input: an integer n > 0.

Output: n!

```
If n = 1 then output:= 1
else
    output:= n(Factorial(n-1))
```

base case $\Theta(1)$

recursive call

$\Theta(1) + T(n-1)$

Analyzing recursive algorithms…

We can define a function for the complexity of a recursive algorithm as in the form of a **recurrence relation**.

<u>Example:</u>

Factorial (n)

function that tells us running time

Let $T(n)$ denote the running time of the algorithm on input of size n.

$C = \Theta(1)$

$$T(n) = C + T(n-1)$$
$$T(1) = C$$

$$T(n) = C + (C + T(n-2))$$
$$= C + (C + (C + T(n-3))) \;\dots = nC = \cancel{O(n)} = \Theta(n)$$

Create your own recursive algorithm!

How about computing a^n recursively?

Recursive Binary Search

BinarySearch($x,i,j, a_1,a_2,\ldots,a_n$)
Input: a sorted array A of **n** numbers and a number x
Output: location of x in A

```
if (n > 1)
  m := ⌊(i + j)/2⌋

    if x > a_m then ____________________________

    else ___________________________________

else
  if x = a_i then output:=I {x is at position i}
  else output:=0 {x is not in list}
```

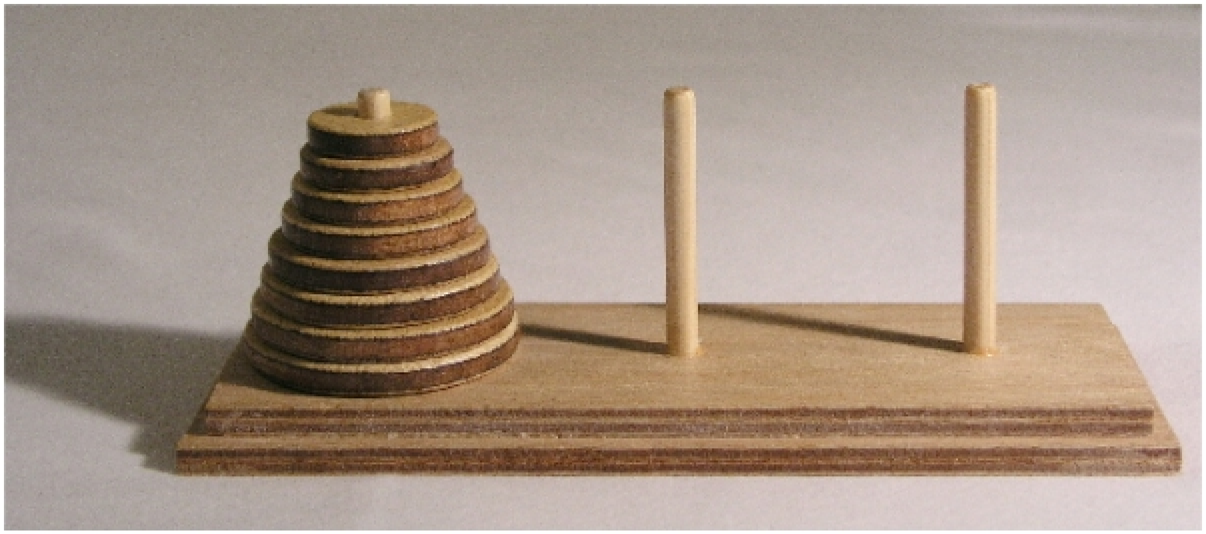
Recursive Binary Search

An appropriate recurrence relation for binary search is:

A. $T(n) = C + T(n-1)$
B. $T(n) = C + T(n/2)$
C. $T(n) = n + T(n/2)$
D. Depends on the data.

Tower of Hanoi…

The Tower of Hanoi invented by French mathematician, Edouard Lucas, in 1883.



TH (A, B, C,n)

```
If n = 1 then Move(A,B)
else
   TH(A,C,B,n-1)
   Move(A,B)
   TH(C,B,A,n-1)
```

CS173
On to recursive algorithms…

Now let's analyze the running time. $T(1) = C$

$T(n) = 2\ T(n-1) + C$

$= 2\ (2\ T(n-2) + C) + C = 4\ T(n-2) + 3C$

$= 4\ (2\ T(n-3) + C) + 3C = 8\ T(n-3) + 7C$

$\ldots = 2^k\ T(n-k) + (2^k-1)C$

$= 2^{n-1}\ T(1) + (2^{n-1}-1)C = (2^n-1)\ C$

$= O(2^n)$

Does it work…?

Is the algorithm correct?

Does it do the right thing for 1 disk?

Assume it does the right thing for n-1 disks. (IH)

And finally, it DOES do the right thing for n disks: move n-1 out of the way, move the biggest disk, move n-1 back.