CS 410/584, Algorithm Design & Analysis: Lecture 1

Algorithm Design & Analysis

David Maier
Portland State University

Let's Solve a Problem

Longest Common Subsequence (LCS)

$S = a_1 a_2 \ . \ . \ . \ a_n$

A <u>subsequence</u> is

$a_{i_1} a_{i_2} \ldots a_{i_k}$ $\quad i_1 < i_2 < \cdots < i_k$

A B S Q U A T U L A T E

A S T L T

B S Q U

Common Subsequence

$S_1 = a_1\ a_2\ .\ .\ .\ a_n$

$S_2 = b_1\ b_2\ .\ .\ .\ b_m$

$S = c_1\ c_2\ \ldots\ c_k$

where S is a subsequence of S_1 and of S_2

A B S Q U A T U L A T E

C A L E F A C T I O N

AAT

Longest Common Subsequence

GA

AGIO

G A N G L I O N

H A G I O L A T R Y

Given sequences S1 and S2, want to find the length of the longest common subsequence

Enumeration Algorithm

```
LCS(S1, S2)
max ← 0
for every subseq. R1 of S1
  for every subseq. R2 of S2
    if R1 = R2 and length(R1) > max
      then max ← length(R1)
return(max)
```

Go from large to small

where $|R1| > max$

test if this R1 is a subsequence of S2

time complexity = $O(2^n \cdot 2^m)$ with improvement $O(2^n \cdot m)$

$|S_1| = n$ $|S_2| = m$

Divide and Conquer

Capitulate

S_1 . . . . . S_2 . . . . . .

S_1' . . . . S_1'' . . . . S_2' . . . . S_2'' . . . .

$LCS(S_1, S_2) = LCS(S_1', S_2') + LCS(S_1'', S_2'')$

Recursion

```
R   a  b  b  c  d  a
S   a  e  b  b  a  e  e
```

R[i] = ith symbol of R

LCS(i,j) length of the LCS of first i symbols of R + first j symbols of S.

```
R[i] ≠ S[j]
   LCS(i,j) = max(LCS(i-1,j),LCS(i,j-1))
   LCS(6,7) = max(LCS(5,7), LCS(6,6)) = 4
```

3 abb 4 abba

```
R[i] = S[j]
   LCS(i,j) = LCS(i-1,j-1) + 1
   LCS(6,5) = LCS(5,4) + 1
```

3 abb + 1 = 4

Recursion 2

```
LCS(i,j)
   if i=0 or j=0 then LCS ← 0
   else if R[i] ≠ S[j] then
      LCS ← max(LCS(i-1,j),LCS(i,j-1))
   else
      LCS ← LCS(i-1,j-1) + 1
```

Count comparisons

$$c(LCS(i,j)) \geq 1 + c(LCS(i-1,j)) + c(LCS(i,j-1))$$
$$\geq 2 \cdot c(LCS(i-1,j-1))$$
$$\geq 2^n$$

Dynamic Programming

Have matrix `LCS(0-m,0-n)` and fill it in

```
R = a a b c a b
S = c a b b
```

a b b

	Ri		a	a	b	c	a	b
Sj		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
c	1	0	0	0	0	1	1	1
a	2	0	1	1	1	1	2	2
b	3	0	1	1	2	2	2	3
b	4	0	1	1	2	2	2	$\boxed{3}$

Algortithm Longest

```
1. Set LCS[i,j] ← 0 if i=0 or j=0
2. for j = 1 to n do
     for i = 1 to m do
       if R[i] ≠ S[j] then
       LCS[i,j]
         ← max(LCS[i-1,j],LCS[i,j-1])
       else
         LCS[i,j] ← LCS[i-1,j-1]+1
3. return LCS[ m, n ]
```

bottom corner

Time complexity $O(m \cdot n)$

Logistics

Course web page is
http://www.cs.pdx.edu/~maier/cs584

Expect class email list to be
cs584@cs.pdx.edu
(Watch the web page for sure.)

Will try to post weekly lecture slides by the evening before Monday class

Class Activity and Quiz

Work in groups on a sample execution of the Freq algorithm

Will have a short quiz on it –
The quiz doesn't count, but should help you assess if you are prepared for this class

I assume you know the material from CS350

Freq(S)
0. If $|S| = 0$, return (null, 0)

1. Pick some element $e \in S$.

2. Split S into two subsets:
   $S\_low \leftarrow \{d \in S \mid d < e\}$
   $S\_high \leftarrow \{d \in S \mid d > e\}$

3. Set $f \leftarrow |S| - |S\_low| - |S\_high|$
   Set (e_low, f_low) ← Freq(S_low)
   Set (e_high, f_high) ← Freq(S_high)

4. If $f > f\_low$ **and** $f > f\_high$, **return** (e, f)
   else if $f\_low > f\_high$, **return** (e_low, f_low)
   else return (e_high, f_high)

Study of Algorithms

- Design techniques
  Divide + Conquer, Dynamic Program. Greedy
- Specific algorithms
  Sorting, Geometric Algs, FFT
- Proofs of correctness
- Complexity analysis
  - measure code size, level of nesting?
  - time it takes, space it consumes
    - as a function of input size
    - need computation model
    - asymptotic behavior
  - worst case
  - average case

Study of Problems

- What's the best any algorithm can do?
  - lower bound - difficulty
  - upper bound - exhibit algorithm
- Relatedness: one problem can be used to solve another
  transitive closure & boolean matrix multiply
- Intractability of problems
  no good algorithms
  no known good algorithms

Complexity of an Algorithm

Amount of time and space required as a function of size of input

`T(n)` = maximum time required
over all inputs of length n

`S(n)` = maximum space required
over all inputs of length n

Input size
number of values
number of bits to represent those values
size of data structure

Time measures
uniform complexity
- basic ops all count 1
logarithmic cost
- op depends on # bits
specific op: comparisons

Time Complexity of a Problem

Minimum time over all algorithms
for the problem

to get
upper bound: exhibit an algorithm

lower bound: harder

Asymptotic Complexity

order notation as a set of fcns $f(n) \in O(g(n))$

f(n) has O(g(n)) if there is $c > 0$ and integer n_0

$f(n) \le c \cdot g(n)$

for all $n > n_0$

$f(n) = n^2 + 3n + 4$ ($\le n^2$, $\le 3n^2$, $\le 4n^2$)

f(n) has $O(n^2)$

f(n) $\le 8n^2$

$n \ge 1$

f(n) has $O(n^3)$ - yes

f(n) has $O(n)$? - no

Asymptotic Notation

"order"

$O(g(n)) = \{ f(n)$ there are positive c and n_0 such that $f(n) \le c \cdot g(n)$ when $n > n_0$ $\}$

$n^2 + 3n + 4 = O(n^2)$

means

$n^2 + 3n + 4 \in O(n^2)$

$\Theta(g(n)) = \{ f(n) \mid$ there are positive c_1, c_2 and n_0 such that

$c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ $\}$

$n > n_0$

$n^2 + 3n + 4 \in \Theta(n^2)$

$c_1 = 1$ $c_2 = 8$ $n_0 = 1$

$1 \cdot n^2 \le n^2 + 3n + 4$ $n > 1$

Asymptotic Notation 2

Note

$n^2 + 3n + 4 \in O(n^3)$

$n^2 + 3n + 4 \notin \Theta(n^3)$

$\Theta(n^3) \subseteq O(n^3)$

$\Omega(g(n)) = \{ f(n) \mid$ there are positive c and n_0 such that $c \cdot g(n) \le f(n)$ when $n > n_0$ $\}$

asymptotic

- upper bound O
- tight bound Θ
- lower bound Ω

Asymptotic Notation 3

Theorem

$f(n) \in O(g(n))$ and
$f(n) \in \Omega(g(n))$
imply
$f(n) \in \Theta(g(n))$

Other facts

$f(n) \in \Theta(h(n))$ and $h(n) \in \Theta(g(n))$
imply $f(n) \in \Theta(g(n))$

$f(n) \in \Theta(g(n))$
implies $g(n) \in \Theta(f(n))$

Heapsort

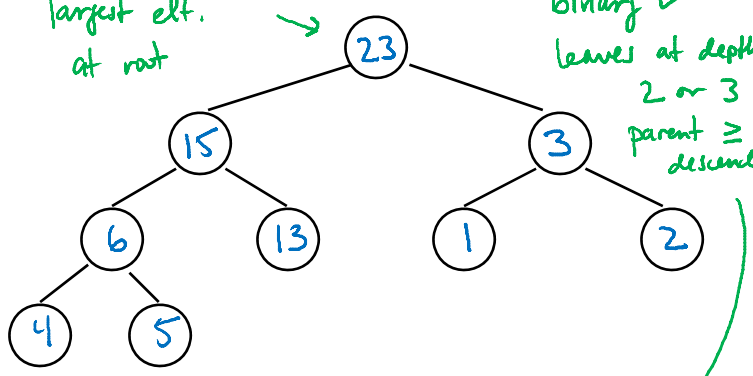
Heap: binary tree where all leaves at depth d or $d-1$

node labels are values to sort

value at node $\geq$ all values in subtrees

Example Heap

largest elt. at root



binary ✓
leaves at depth 2 or 3 ✓
parent ≥ descendents

parent ≥ children

Heapsort Algorithm

0. make a heap

Heapsort

1. `remove` largest value from root
2. `move` label of some leaf to root
3. `delete` that leaf
4. `remake` heap

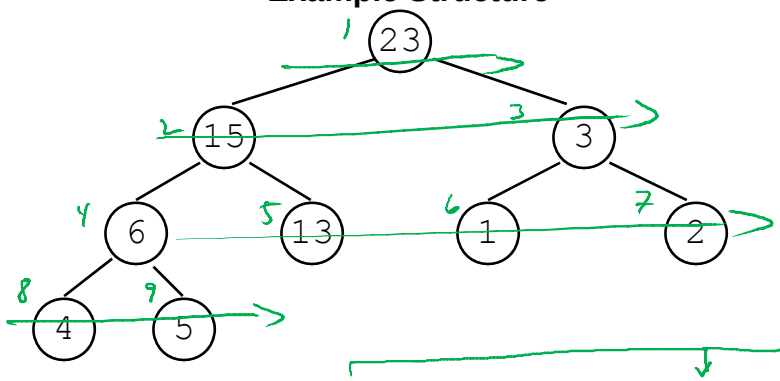
Implicit data structure

`A[1..n]` array of values

`A[1]` is root

left and right children of `A[i]` are `A[2i]` and `A[2i+1]`

Example Structure



i	1	2	3	4	5	6	7	8	9
A[i]	23	15	3	6	13	1	2	4	5

Restriction: leaves of depth d need to be to the left

Remove Root Value

Swap A[1] ↔ A[n]

Pretend array runs A[1] .. A[n-1]

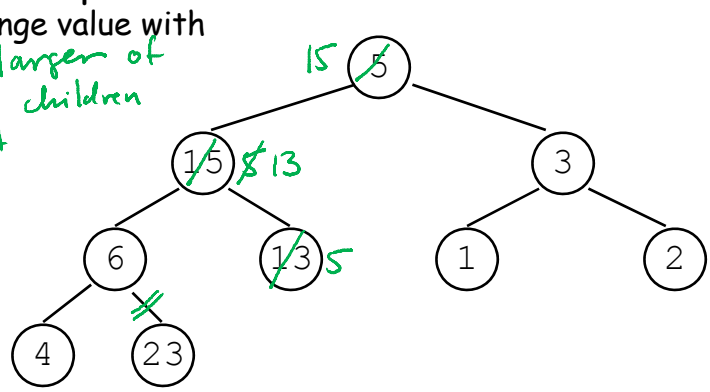
~~23~~ 5	15	3	6	13	1	2	4	~~5~~ 23

new end

Remake heap:

exchange value with the larger of two children

- repeat



Formally

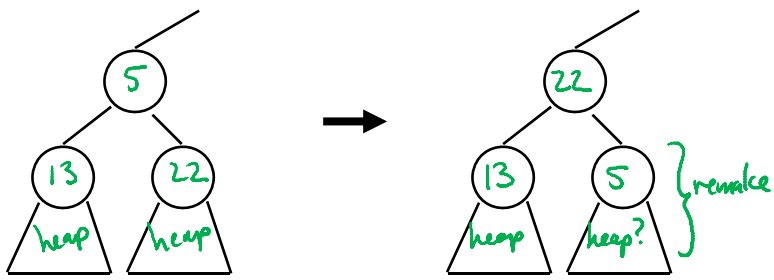
Input: `a1 a2 ... an` to sort

Initialize `A[i]` ← a_i

build-max-heap

want `A[i]` ≥ $A[2i]$, $A[2i+1]$

for `i` between `1` and `n/2`



Heap Construction

Assume: a1 a2 ... an to sort $A[i] = a_i$

Want: $A[i] \geq A[2i], A[2i+1]$ $1 \leq i \leq n/2$

max-heapify (i, j) make A[i] into root of heap if A[2i], A[2i+1] are ~~already~~ already heaps

current end of array

if `i` not a leaf **and** 2i ≤ j

a child of i has a greater value

then

let k be child with larger value

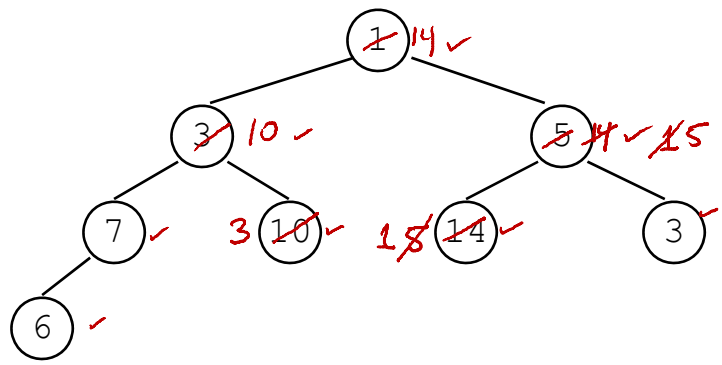
- $A[i] \leftrightarrow A[k]$
- max-heapify(k, j)

build-max-heap

for `i = n` **downto** `1` **do**

max-heapify(i, n)

Example of Build-max-heap



Complexity of Build-max-heap

Theorem: build-max-heap has `O(n)` time complexity

Define `T(h)` as the time to max-heapify a node of height `h` levels above a leaf

`T(h)` $\leq$ $T(h-1) + c$

`T(h)` $\in O(h)$ $T(h) \leq c \cdot h$

Build-max-heap calls max-heapify once per node.

How many nodes of height `i`? $\left\lceil \frac{n}{2^{i+1}} \right\rceil$

Heapsort Algorithm

current end of heap

```
heapsort
  build-max-heap
  for i = n to 2 do
```

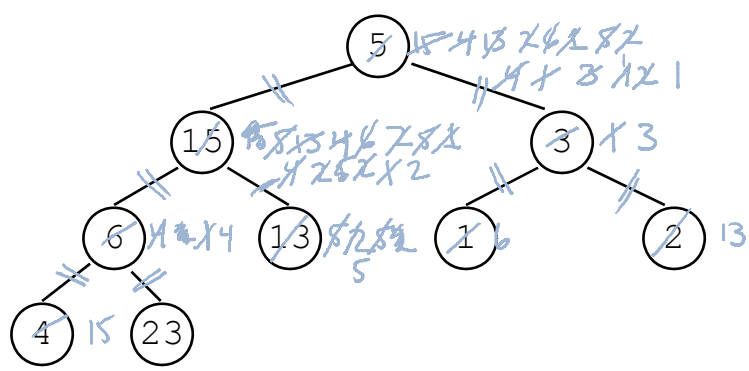
- A[1] ⟷ A[i]
- max-heapify (1, i-1)

Sorted list ends up in `A[1] ... A[n]`

Time complexity: $O(n + n \cdot \lg n) = O(n \cdot \lg n)$

$\lg n = \log_2 n$

Heapsort Example

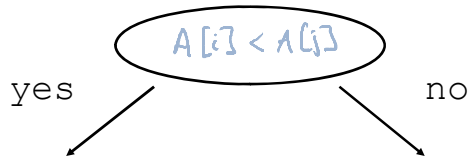


Lower Bounds on Sorting

Counting comparisons

Assume `A[1], A[2], ..., A[n]` are distinct

Represent algorithm by a decision tree



Fact: Binary tree of height `h` has at most 2^h leaves

Consequence

Thm: The decision tree for any algorithm that sorts n items has height at least lg(n!) — has at least n! leaves

Corollary: Any algorithm for sorting by comparisons needs $cn \lg n$ compares for some `c > 0` and large `n`.

$$n! \geq \underbrace{n(n-1) \ \ldots \ (n/2)}_{n/2} \geq (n/2)^{n/2}$$

$$\lg(n!) \geq \lg (n/2)^{n/2} = \frac{n}{2} \lg \frac{n}{2}$$

$$= \frac{n}{2}(\lg n - \underbrace{\lg 2}_{\text{constant}}) \geq \frac{1}{4}(n \lg n) \quad n > 2$$

c

$\lg(n!) \in \Omega(n \lg n)$

`n log n ≥ log n!` since

$= \lg n^n$

$n^n \geq n!$