Let’s Solve a Problem

**Longest Common Subsequence (LCS)**

Let's denote the sequence as:

$$S = a_1 a_2 \ldots a_n$$

A **subsequence** is:

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

Example:

**ABSTRACT**

- **ASTLT**
- **BSQU**
Common Subsequence

$S_1 = a_1 a_2 \ldots a_n$

$S_2 = b_1 b_2 \ldots b_m$

$S = c_1 c_2 \ldots c_k$

where $S$ is a subsequence of $S_1$ and of $S_2$

Longest Common Subsequence

Given sequences $S_1$ and $S_2$, want to find the length of the longest common subsequence
Enumeration Algorithm

LCS(S1, S2)

max ← 0

for every subseq. R1 of S1 where |R1| > max

for every subseq. R2 of S2

if R1 = R2 and length(R1) > max

then max ← length(R1)

return(max)

time complexity = \(O(2^n \cdot 2^m)\) \(\approx O(2^{n+m})\) with improvement

\[ |S_1| = n \quad |S_2| = m \]

Divide and Conquer

LCS(S1, S2) = LCS(S1', S2') + LCS(S1'', S2'')
Recursion

R: a b b c d a
S: a e b b a e e

R[i] ≠ S[j]
LCS(i,j) = max(LCS(i-1,j), LCS(i,j-1))
LCS(6,7) = \text{max}(LCS(5,7), LCS(6,6)) = 4
R[i] = S[j]
LCS(i,j) = LCS(i-1,j-1) + 1
LCS(6,5) = LCS(5,4) + 1 = 4

Recursion 2

LCS(i,j)
if i=0 or j=0 then LCS \leftarrow 0
else if R[i] ≠ S[j] then
  LCS \leftarrow \text{max}(LCS(i-1,j), LCS(i,j-1))
else
  LCS \leftarrow LCS(i-1,j-1) + 1

Count comparisons

\[ c(LCS(i,j)) \geq 1 \cdot c(LCS(i-1,j)) + c(LCS(i,j-1)) \geq 2 \cdot c(LCS(i-1,j-1)) \geq 2^n \]
Dynamic Programming

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

\[
\begin{array}{cccccc}
R &=& a & a & b & c & a & b \\
S &=& c & a & b & b \\
\end{array}
\]

\[
\begin{array}{cccccccc}
Ri & a & a & b & c & a & b & \ \\
Sj & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
c & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
a & 2 & 0 & 1 & 1 & 1 & 1 & 1 \\
b & 3 & 0 & 2 & 2 & 2 & 2 & 2 \\
b & 4 & 0 & 2 & 2 & 2 & 2 & 2 & 2 \\
\end{array}
\]

Algorithm Longest

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

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
Algorithm Design & Analysis

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

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

2. Split $S$ into two subsets:
   - $S_{\text{low}} \leftarrow \{d \in S \mid d < e\}$
   - $S_{\text{high}} \leftarrow \{d \in S \mid d > e\}$

3. Set $f \leftarrow |S| - |S_{\text{low}}| - |S_{\text{high}}|$
   - Set $(e_{\text{low}}, f_{\text{low}}) \leftarrow \text{Freq}(S_{\text{low}})$
   - Set $(e_{\text{high}}, f_{\text{high}}) \leftarrow \text{Freq}(S_{\text{high}})$

4. If $f > f_{\text{low}}$ and $f > f_{\text{high}}$, return $(e, f)$
   - else if $f_{\text{low}} > f_{\text{high}}$, return $(e_{\text{low}}, f_{\text{low}})$
   - else return $(e_{\text{high}}, f_{\text{high}})$

Study of Algorithms

- Design techniques
  - Divide & Conquer, Dynamic Program, Greedy

- Specific algorithms
  - Sorting, Geometric, 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
  - 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

\[ T(n) = \text{maximum time required over all inputs of length } n \]
\[ S(n) = \text{maximum space required over all inputs of length } n \]

Input size
- number of values
- number of bits to represent these values
- size of data structure

Time measures
- uniform complexity
- basic unit all counts
- logarithmic cost
- op depends on # bits
- specific ops: comparisons
Time Complexity of a Problem

Minimum time over all algorithms to get

upper bound: exhibit an algorithm

lower bound: harder

Asymptotic Complexity

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

$f(n)$ has $O(g(n))$ if there is $C > 0$ and integer $N_0$

for all $n > N_0$

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

$f(n) = 8n^2$

$n \geq 1$

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

$f(n)$ has $O(n^3)$? - no
Asymptotic Notation

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

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

means

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

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

$\frac{c_1}{c_2} \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ when } n > n_0 \}

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

\begin{align*}
c_1 &= 1 & c_2 &= 8 & n_0 &= 1 \\
1 \cdot n^2 &\leq n^2 + 3n + 4 & n &> 1
\end{align*}

Asymptotic Notation 2

Note

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

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

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

$\Omega(g(n)) = \{ f(n) \mid \text{there are positive } c \text{ and } n_0 \text{ such that }$\n
\begin{align*}
c \cdot g(n) &\leq f(n) \text{ when } n > n_0 \}
\end{align*}

asymptotic

- upper bound $O$
- tight bound $\Theta$
- lower bound $\Omega$
Asymptotic Notation 3

Theorem
\[ f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n)) \]
imply
\[ f(n) \in \Theta(g(n)) \]

Other facts
\[ f(n) \in \Theta(h(n)) \text{ and } h(n) \in \Theta(g(n)) \]
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

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

<table>
<thead>
<tr>
<th>i</th>
<th>A[i]</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
<td>23</td>
<td>15</td>
<td>3</td>
<td>6</td>
<td>13</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>6</td>
<td>15</td>
<td>3</td>
<td>13</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>13</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>23</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>23</td>
<td>4</td>
<td>23</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Remake heap:
- exchange value with the larger of the two children.
- repeat

Swap: A[i] \rightarrow A[n]

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

Remove Root Value
Formally

Input: $a_1 \ a_2 \ \ldots \ \ a_n$ to sort
Initialize $A[i] \leftarrow a_i$
build-max-heap
  want $A[i] \geq A(2i), A(2i+1)$
  for $i$ between $1$ and $n/2$

Heap Construction
Assume: $a_1 \ a_2 \ \ldots \ \ a_n$ to sort $A[i] = a_i$
Want: $A[i] \geq A(2i), A(2i+1)$ $1 \leq i \leq \lfloor n/2 \rfloor$
max-heapify $(i, j)$
  if $i$ not a leaf and a child of $i$ has a greater value
    then
      let $k$ be child with larger value
      $A[i] \leftarrow A[k]$
      max-heapify $(k, j)$
build-max-heap
  for $i = n$ downto $1$ do
    max-heapify $(i, n)$
Algorithm Design & Analysis

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

$T(h) \leq T(h-i) + c$

$T(h) \in O(h)$

$T(n) \leq c \cdot n$

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

How many nodes of height $i$?

$\left\lfloor \frac{n}{2^{i+1}} \right\rfloor$
Heapsort Algorithm

heapsort
  build-max-heap
  for i = n to 2 do
    max-heapify (1, i-1)

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

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

\(\lg n = \log_2 n\)
Lower Bounds on Sorting

Counting comparisons
Assume $A[1], A[2], \ldots, A[n]$ are distinct
Represent algorithm by a decision tree


Fact: Binary tree of height $n$ has at most $2^n$ leaves

Consequence

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

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

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

$$n \log n \geq \log n! \quad \text{since}$$

$$n^n \geq n!$$