CS 483 - Data Structures and Algorithm Analysis
Lecture VII: Chapter 6, part 2

R. Paul Wiegand

George Mason University, Department of Computer Science

March 22, 2006
Outline

1. Balanced Trees
2. Heaps & HeapSort
3. Horner’s Rule & Binary Exponentiation
4. Problem Reduction
5. Homework
Binary Search Trees

- **binary search tree** — A binary tree in which, given some node, all nodes in the left subtree of that node have a smaller key value and all the nodes in the right subtree of a greater key value

- Operations: **Search, Insert, & Delete**

- Average case for these: $\Theta(\lg n)$
Binary Search Trees

- **binary search tree** — A binary tree in which, given some node, all nodes in the left subtree of that node have a smaller key value and all the nodes in the right subtree of a greater key value.

- Operations: **Search, Insert, & Delete**

- Average case for these: $\Theta(\lg n)$

- Worst case for these: $\Theta(n)$

- This occurs when the tree is **unbalanced** (wide diversity of path lengths from leaf nodes to root).

![Binary Search Tree Diagram]

R. Paul Wiegand  
George Mason University, Department of Computer Science
Binary Search Trees

- **binary search tree** — A binary tree in which, given some node, all nodes in the left subtree of that node have a smaller key value and all the nodes in the right subtree of a greater key value.

- **Operations**: Search, Insert, & Delete

- Average case for these: $\Theta(\lg n)$

- Worst case for these: $\Theta(n)$

- This occurs when the tree is *unbalanced* (wide diversity of path lengths from leaf nodes to root)

- In the most severe case, the tree becomes a list whose height is $O(n)$
Binary Search Trees

- **Binary search tree** — A binary tree in which, given some node, all nodes in the left subtree of that node have a smaller key value and all the nodes in the right subtree of a greater key value.

- **Operations**: **Search**, **Insert**, & **Delete**

- Average case for these: $\Theta(\lg n)$

- Worst case for these: $\Theta(n)$

- This occurs when the tree is **unbalanced** (wide diversity of path lengths from leaf nodes to root)

- In the most severe case, the tree becomes a list whose height is $O(n)$

- Two high-level for avoiding unbalanced trees:
  - Balance an unbalanced tree (instance simplification)
  - Allow more elements in a node (representation change)
AVL Trees

- Methods for transforming unbalanced trees to balanced trees include AVL trees, red-black trees, and splay trees.

- **Balance factor** — the difference between the heights of the left and right subtrees.

- **AVL tree** — a binary search tree in which the balance factor of every node is \{+1, 0, −1\}.

- The trick is to maintain the AVL property when nodes are inserted or deleted.

- To do so, there are four special transformations:
  - Single-right, single-left rotation
  - Double left-right, double right-left rotation
Right & Left Rotations

**Single Right Rotation**

Before Rotation:

```
    3
   / 
  2   0
 /   /
1    2
```

After Rotation:

```
    0
   / 
  1   3
```

**Single Left Rotation**

Before Rotation:

```
    1
   / 
  2   0
 /   /
3    0
```

After Rotation:

```
    0
   / 
  1   3
```

R. Paul Wiegand
George Mason University, Department of Computer Science

CS483 Lecture II
Left-Right & Right-Left Rotations

Double Left-Right Rotation

Double Right-Left Rotation
General Single-Right Rotation
General Double Left-Right Rotation

The diagram illustrates a general double left-right rotation in balanced trees. The rotation involves four subtrees, denoted as $T_1$, $T_2$, $T_3$, and $T_4$. The rotation moves nodes $c$ and $r$ in the tree structure, with possible alternative configurations indicated by 'or'. The rotation maintains the balance of the tree by ensuring that the subtrees are properly aligned after the operation.
Analyzing AVL Trees

- Rotations are complicated operations, but still constant time
- Tree traversal efficiency depends on height of the tree
- The Height $h$ of any AVL tree with $n$ nodes can be bound by $\lg n$
- So `Search`, `Insert`, and even `Delete` are in $\Theta(\lg n)$.
- Cost: Frequent rotations (high constant values in running-time)
Analyzing AVL Trees

- Rotations are complicated operations, but still constant time
- Tree traversal efficiency depends on height of the tree
- The Height $h$ of any AVL tree with $n$ nodes can be bound by $\log n$
- So Search, Insert, and even Delete are in $\Theta(\log n)$.
- Cost: Frequent rotations (high constant values in running-time)

Something to Ponder:
Is it better to accept a linear worst case situation when the average is $\Theta(\log n)$ (binary search tree), or to slow all operations down by a constant factor to ensure a $\log n$ bound in all cases (AVL tree)?
2-3 Trees

One may also change the representation by allowing more nodes (e.g., 2-3 trees, 2-3-4 trees, and B-trees)

2-node — Contains a single key $K$ and (up to) two subtrees. The left subtree contains nodes with key values less than $K$, the right contain values greater than $K$

3-node — Contains two keys $K_1$ and $K_2$, and (up to) three subtrees. The left subtree contains nodes with key values less than $K_1$, the right contain values greater than $K_2$, the middle contain values in $(K_1, K_2)$
Searching in 2-3 Trees

- For a 2-node: Compare the search key to the key at the node
  - If they are the same, return the node
  - If the search key is less, traverse left
  - If the search key is greater, traverse right

- For a 3-node: Compare the search key to two keys at the node
  - If the search key is equal to either node keys, return the node
  - If the search key is less than the first node key, traverse left
  - If it is between the two keys, traverse middle
  - If it is greater than the second node key, traverse right
Inserting in 2-3 Trees

- If tree is empty, make a 2-node at the root for the inserted key
- Otherwise,
  - Insert at a leaf (i.e., SEARCH)
  - If the leaf is a 2-node, insert the key in that node in the correct order
  - If the leaf is a 3-node, split the node up
    - The smallest key becomes a left 2-node
    - The largest key becomes a right 2-node
    - The middle key is promoted to the parent
    - Note: This promotion can force a split in the node above
Example: Inserting in a 2-3 Tree

Inserting: \langle9, 5, 8, 3, 2, 4, 7\rangle:
Example: Inserting in a 2-3 Tree

Inserting: \(\langle 9, 5, 8, 3, 2, 4, 7 \rangle:\)
Example: Inserting in a 2-3 Tree

Inserting: \(\langle 9, 5, 8, 3, 2, 4, 7 \rangle:\)

```
9, 5, 8
```

```
8
\downarrow
5 \rightarrow 9
```
Example: Inserting in a 2-3 Tree

Inserting: \( \langle 9, 5, 8, 3, 2, 4, 7 \rangle \):
Example: Inserting in a 2-3 Tree

Inserting: \langle 9, 5, 8, 3, 2, 4, 7 \rangle:
Example: Inserting in a 2-3 Tree

Inserting: \langle 9, 5, 8, 3, 2, 4, 7 \rangle:
Example: Inserting in a 2-3 Tree

Inserting: ⟨9, 5, 8, 3, 2, 4, 7⟩:
Analyzing 2-3 Trees

Consider a 2-3 tree of height $h$ with $n$ nodes in it.

- **Upper bound**: All nodes are 2-nodes,
  \[ n \geq 1 + 2 + \ldots + 2^h = 2^{h+1} - 1 \]
  \[ \therefore h \leq \log_2(n + 1) - 1 \]

- **Lower bound**: All nodes are 3-nodes,
  \[ n \leq 2 \cdot 3^0 + 2 \cdot 3^1 + \cdots + 2 \cdot 3^h = 3^{h+1} - 1 \]
  \[ \therefore h \geq \log_3(n + 1) - 1 \]

- So the height is bounded by $\Theta(\log n)$
- Basic operations are, as well
Introduction to Heaps

- Heaps are *incompletely* ordered data structures suitable for *priority queues*
  - FIND item with highest priority
  - DELETE item with highest priority
  - ADD NEW ITEM TO THE SET

**Definition**

A *heap* can be defined as a binary tree that meets the following conditions:

1. It is *essentially complete* (all \( h - 1 \) levels are full, level \( h \) has only left-most leaves)
2. *Parental dominance* — Key at each node is \( \geq \) its children
Introduction to Heaps

- Heaps are *incompletely* ordered data structures suitable for *priority queues*
  - **Find** item with highest priority
  - **Delete** item with highest priority
  - **Add new item to the set**

**Definition**

A *heap* can be defined as a binary tree that meets the following conditions:

1. It is *essentially complete* (all $h - 1$ levels are full, level $h$ has only left-most leaves)
2. *Parental dominance* — Key at each node is $\geq$ its children
Fun Facts about Heaps

- The height of an essentially complete binary tree with \( n \) nodes is always \( \lfloor \lg n \rfloor \)
- The root node of a heap always has the largest key value
- Any subtree of a heap is also a heap
- A heap can be implemented as an array
  - Store values top-down, left-to-right
  - Parent nodes in first \( \lfloor n/2 \rfloor \) positions, leaf keys in last \( \lceil n/2 \rceil \)
  - Children of a key in position \( i \in [1, \lfloor n/2 \rfloor] \) will be at \( 2i \) and \( 2i + 1 \)
  - A parent of a key in position \( j \in [\lceil n/2 \rceil, n] \) will be at \( \lfloor n/2 \rfloor \)
  - Alternate heap definition:
    \[
    H[i] \geq \max\{H[2i], H[2i + 1]\} \quad \forall i \in [1, \lfloor n/2 \rfloor]
    \]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i )</td>
<td>10</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>( H[i] )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bottom-Up Heap Construction

Bottom-up heap construction takes a non-heap and turns it into a heap.

- Starting with the last parental node, work toward the root ($i$)
- Check the parental dominance of the node under consideration ($j$)
- If condition not met:
  - Exchange keys with the larger child
  - Check again for node in new position
  - Repeat until satisfied (wc: to the leaf)
- Move to the immediate (array) predecessor and repeat

$$C_{worst}(n) = 2(n - \log(n + 1))$$
$$\therefore C(n) \in O(n)$$
Bottom-Up Heap Construction

Bottom-up heap construction takes a non-heap and turns it into a heap.

- Starting with the last parental node, work toward the root ($i$)
  - Check the parental dominance of the node under consideration ($j$)
  - If condition not met:
    - Exchange keys with the larger child
    - Check again for node in new position
    - Repeat until satisfied (wc: to the leaf)
- Move to the immediate (array) predecessor and repeat

$$C_{\text{worst}}(n) = 2(n - \log(n + 1))$$

$$\therefore C(n) \in O(n)$$
Bottom-Up Heap Construction

Bottom-up heap construction takes a non-heap and turns it into a heap.

- Starting with the last parental node, work toward the root ($i$)
  - Check the parental dominance of the node under consideration ($j$)
  - If condition not met:
    - Exchange keys with the larger child
    - Check again for node in new position
    - Repeat until satisfied (wc: to the leaf)
  - Move to the immediate (array) predecessor and repeat

$$C_{worst}(n) = 2(n - \log(n + 1))$$

$\therefore C(n) \in O(n)$
Bottom-Up Heap Construction

Bottom-up heap construction takes a non-heap and turns it into a heap.

- Starting with the last parental node, work toward the root \((i)\)
  - Check the parental dominance of the node under consideration \((j)\)
  - If condition not met:
    - Exchange keys with the larger child
    - Check again for node in new position
    - Repeat until satisfied \((\text{wc: to the leaf})\)
  - Move to the immediate (array) predecessor and repeat

\[
C_{\text{worst}}(n) = 2(n - \log(n + 1))\]

\[
\therefore C(n) \in O(n)
\]
Bottom-up heap construction takes a non-heap and turns it into a heap.

- Starting with the last parental node, work toward the root (i)
  - Check the parental dominance of the node under consideration (j)
  - If condition not met:
    - Exchange keys with the larger child
    - Check again for node in new position
    - Repeat until satisfied (wc: to the leaf)
- Move to the immediate (array) predecessor and repeat

\[
C_{worst}(n) = 2(n - \log(n + 1))
\]
\[
\therefore C(n) \in O(n)
\]
Bottom-Up Heap Construction

Bottom-up heap construction takes a non-heap and turns it into a heap.

- Starting with the last parental node, work toward the root ($i$)
  - Check the parental dominance of the node under consideration ($j$)
  - If condition not met:
    - Exchange keys with the larger child
    - Check again for node in new position
    - Repeat until satisfied (wc: to the leaf)
- Move to the immediate (array) predecessor and repeat

$$C_{worst}(n) = 2(n - \log(n + 1))$$

$\therefore C(n) \in O(n)$
Top-Down Heap Construction

Top-down heap construction maintains heap properties as nodes are inserted.

- Repeatedly insert new nodes at the bottom of the heap
- Each insert:
  - Compare inserted node to parent
  - If parental dominance condition is not met, swap nodes
  - Repeat until condition met or root is reached

\[
C_{\text{insert}}(n) = O(\log n) \\
\therefore C(n) \in O(n \log n)
\]
Top-Down Heap Construction

Top-down heap construction maintains heap properties as nodes are inserted.

- Repeatedly insert new nodes at the bottom of the heap
- Each insert:
  - Compare inserted node to parent
  - If parental dominance condition is not met, swap nodes
  - Repeat until condition met or root is reached

Comparisons needed for inserts are bounded by the heap height:

\[ C_{\text{insert}}(n) = O(\lg n) \]

\[ \therefore C(n) \in O(n \lg n) \]
Top-Down Heap Construction

Top-down heap construction maintains heap properties as nodes are inserted.

- Repeatedly insert new nodes at the bottom of the heap
- Each insert:
  - Compare inserted node to parent
  - If parental dominance condition is not met, swap nodes
  - Repeat until condition met or root is reached

![Heap Diagram]

Comparisons needed for inserts are bounded by the heap height:

\[ C_{\text{insert}}(n) = O(\lg n) \]

\[ \therefore C(n) \in O(n \lg n) \]
Deleting from a Heap

- Removing the largest heap element:
  1. Exchange the root with the last node in the heap
  2. Decrease the heap size by 1 (i.e., remove the last node)
  3. Sift the new root down the tree using the *heapify* procedure from bottom-up heap construction

Comparisons needed for delete are bounded by twice the height:

\[ C_{\text{delete}}(n) = O(\lg n) \]
Deleting from a Heap

- Removing the largest heap element:
  1. Exchange the root with the last node in the heap
  2. Decrease the heap size by 1 (i.e., remove the last node)
  3. Sift the new root down the tree using the heapify procedure from bottom-up heap construction

Step 1

Comparisons needed for delete are bounded by twice the height:

$$C_{\text{delete}}(n) = O(\lg n)$$
Deleting from a Heap

- Removing the largest heap element:
  1. Exchange the root with the last node in the heap
  2. Decrease the heap size by 1 (i.e., remove the last node)
  3. Sift the new root down the tree using the heapify procedure from bottom-up heap construction

Comparisons needed for delete are bounded by twice the height:

$$C_{\text{delete}}(n) = O(\log n)$$
Deleting from a Heap

- Removing the largest heap element:
  1. Exchange the root with the last node in the heap
  2. Decrease the heap size by 1 (i.e., remove the last node)
  3. Sift the new root down the tree using the heapify procedure from bottom-up heap construction

- Comparisons needed for delete are bounded by twice the height:
  \[ C_{\text{delete}}(n) = O(\lg n) \]
Deleting from a Heap

Removing the largest heap element:

1. Exchange the root with the last node in the heap
2. Decrease the heap size by 1 (i.e., remove the last node)
3. Sift the new root down the tree using the heapify procedure from bottom-up heap construction

Comparisons needed for delete are bounded by twice the height:

\[ C_{\text{delete}}(n) = O(\lg n) \]
Deleting from a Heap

- Removing the largest heap element:
  1. Exchange the root with the last node in the heap
  2. Decrease the heap size by 1 (i.e., remove the last node)
  3. Sift the new root down the tree using the *heapify* procedure from bottom-up heap construction

![Heap Diagram]

Comparisons needed for delete are bounded by twice the height:

\[ C_{\text{delete}}(n) = O(\lg n) \]
HEAPSORT

- Two stage process:
  1. Construct a heap
  2. Apply root-deletion \( n - 1 \) times
- Bottom-up heap construction is \( O(n) \)
- The deletes are \textit{slightly} more complicated to analyze because the size changes with each deletion:

\[
C(n) \leq 2 \left\lfloor \log(n - 1) \right\rfloor + 2 \left\lfloor \log(n - 2) \right\rfloor + \cdots + 2 \left\lfloor \log 1 \right\rfloor
\leq 2 \sum_{i=1}^{n-1} \log i
\leq 2 \sum_{i=1}^{n-1} \log(n - 1) = 2(n - 1) \log(n - 1)
\leq 2n \log n
\]

\[C(n) \in O(n \log n)\]
Evaluating Polynomials

\[ p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \]

- Given some polynomial evaluate it at a specified \( x \)
- Example: \( p(x) = 2x^2 - 3x + 1 \)
- Brute force: \( p(2) = 2*(2*2) - 3*(2) + 2 = 4 \)
Evaluating Polynomials

\[ p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \]

- Given some polynomial evaluate it at a specified \( x \)
- Example: \( p(x) = 2x^2 - 3x + 1 \)
- Brute force: \( p(2) = 2 \times (2 \times 2) - 3 \times (2) + 2 = 4 \)
  
 3 multiplications
Evaluating Polynomials

\[ p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \]

- Given some polynomial evaluate it at a specified \( x \)
- Example: \( p(x) = 2x^2 - 3x + 1 \)
- Brute force: \( p(2) = 2 \times (2 \times 2) - 3 \times (2) + 2 = 4 \)
- In general for brute force:
  - \( a_n x^n = a_n \times x \times x \times x \cdots \) requires \( n \) multiplications
  - \( a_{n-1} x^{n-1} \) requires \( n - 1 \) multiplications
  - ...
Evaluating Polynomials

\[ p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \]

- Given some polynomial evaluate it at a specified \( x \)
- Example: \( p(x) = 2x^2 - 3x + 1 \)
- Brute force: \( p(2) = 2 \times (2 \times 2) - 3 \times (2) + 2 = 4 \) \( \text{3 multiplications} \)
- In general for brute force:
  - \( a_n x^n = a_n \times x \times x \times x \times \cdots \) requires \( n \) multiplications
  - \( a_{n-1} x^{n-1} \) requires \( n - 1 \) multiplications
  - \( \ldots \)
  - \( \sum_{i=0}^{n} i \in O(n^2) \)
Evaluating Polynomials

\[ p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \]

- Given some polynomial evaluate it at a specified \( x \)
- Example: \( p(x) = 2x^2 - 3x + 1 \)
- Brute force: \( p(2) = 2 \times (2 \times 2) - 3 \times (2) + 2 = 4 \) [3 multiplications]

- In general for brute force:
  - \( a_n x^n = a_n \times x \times x \times x \cdots \) requires \( n \) multiplications
  - \( a_{n-1} x^{n-1} \) requires \( n - 1 \) multiplications
  - \( \cdots \)
  - \( \sum_{i=0}^{n} i \in O(n^2) \)

- Is there a better way?
Horner’s Rule

We can successively take a common factor in the remaining polynomials of smaller degree:

\[ p(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0 \]
Horner’s Rule

- We can successively take a common factor in the remaining polynomials of smaller degree:

\[ p(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0 \]

\[ = (a_n x + a_{n-1}) x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0 \]
Horner’s Rule

- We can successively take a common factor in the remaining polynomials of smaller degree:

\[
p(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0
\]

\[
= (a_n x + a_{n-1}) x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0
\]

\[
= ((a_n x + a_{n-1}) x + a_{n-2}) x^{n-2} + \cdots + a_1 x + a_0
\]
Horner’s Rule

- We can successively take a common factor in the remaining polynomials of smaller degree:

\[ p(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0 \]
\[ = (a_n x + a_{n-1}) x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0 \]
\[ = ((a_n x + a_{n-1}) x + a_{n-2}) x^{n-2} + \cdots + a_1 x + a_0 \]
\[ = (\ldots (a_n x + a_{n-1}) x + \ldots) x + a_0 \]
Horner’s Rule

We can successively take a common factor in the remaining polynomials of smaller degree:

\[
p(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0
\]

\[
= (a_n x + a_{n-1}) x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_1 x + a_0
\]

\[
= (((a_n x + a_{n-1}) x + a_{n-2}) x^{n-2} + \cdots + a_1 x + a_0
\]

\[
= \cdots (a_n x + a_{n-1}) x + \ldots ) x + a_0
\]

One multiplication (& one addition) per coefficient
∴ \(O(n)\)

For example: \(p(x) = 2x^4 - x^3 + 3x^2 + x - 5\). What is \(p(3)\)?

\[
\begin{array}{cccccc}
\tilde{a} & 2 & -1 & 3 & 1 & -5 \\
\hline
x & P = a_4 & P = Px + a_3 & P = Px + a_2 & P = Px + a_1 & P = Px + a_0 \\
3 & 2 & 2 \cdot 3 - 1 = 5 & 5 \cdot 3 + 3 = 18 & 18 \cdot 3 + 1 = 55 & 55 \cdot 3 - 5 = 160 \\
\end{array}
\]
A degenerate polynomial evaluation problem of interest is $a^n$
A degenerate polynomial evaluation problem of interest is \( a^n \).

Suppose we have a representation of \( n \) as a binary string of length \( \ell \):

\[
n = b_\ell b_{\ell-1} \cdots b_i \cdots b_0
\]

\( \ell \) = length of \( n \)

\( b_i \) are bits of \( n \)

e.g., \( n = 13 = 1101_2 \)
Binary Exponentiation Basics

- A degenerate polynomial evaluation problem of interest is $a^n$

- Suppose we have a representation of $n$ as a binary string of length $\ell$:
  
  $n = b_\ell b_{\ell-1} \cdots b_i \cdots b_0$

  e.g., $n = 13 = 1101_2$

- Can interpret bits as coefficients, write a polynomial where $x = 2$:
  
  $p(x) = b_\ell x^\ell + \cdots b_i x^i + \cdots b_0$

  e.g., $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
Binary Exponentiation Basics

- A degenerate polynomial evaluation problem of interest is $a^n$.
- Suppose we have a representation of $n$ as a binary string of length $\ell$: $n = b_\ell b_{\ell-1} \cdots b_i \cdots b_0$; e.g., $n = 13 = 1101_2$.
- Can interpret bits as coefficients, write a polynomial where $x = 2$: $p(x) = b_\ell x^\ell + \cdots b_i x^i + \cdots b_0$; e.g., $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$.
- We can now rewrite $a^n$: $a^{p(x)} = a^{b_\ell x^\ell + \cdots b_i x^i + \cdots b_0}$; e.g., $a^{1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0}$.
A degenerate polynomial evaluation problem of interest is $a^n$

Suppose we have a representation of $n$ as a binary string of length $\ell$:
$n = b_\ell b_{\ell - 1} \cdots b_i \cdots b_0$

e.g., $n = 13 = 1101_2$

Can interpret bits as coefficients, write a polynomial where $x = 2$:
$p(x) = b_\ell x^\ell + \cdots b_i x^i + \cdots b_0$

e.g., $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$

We can now rewrite $a^n$:
$a^{p(x)} = a^{b_\ell x^\ell + \cdots b_i x^i + \cdots b_0}$

e.g., $a^{1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0}$

So we can accumulate the product in the exponent by Horner’s rule
A degenerate polynomial evaluation problem of interest is $a^n$

Suppose we have a representation of $n$ as a binary string of length $\ell$:

$$n = b_\ell b_{\ell-1} \cdots b_i \cdots b_0$$

e.g., $n = 13 = 1101_2$

Can interpret bits as coefficients, write a polynomial where $x = 2$:

$$p(x) = b_\ell x^\ell + \cdots b_i x^i + \cdots b_0$$

e.g., $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$

We can now rewrite $a^n$:

$$a^{p(x)} = a^{b_\ell x^\ell + \cdots b_i x^i + \cdots b_0}$$

e.g., $a^{1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0}$

So we can accumulate the product in the exponent by Horner's rule

Writing $p$ as the current product, we recognize that:

$$a^{2p + b_i} = a^{2p} \cdot a^{b_i} = (a^p)^2 \cdot a^{b_i} = \begin{cases} (a^p)^2 & \text{if } b_i = 0 \\ (a^p)^2 \cdot a & \text{if } b_i = 1 \end{cases}$$
**Left-to-Right Binary Exponentiation**

### Algorithm

**LEFTTORIGHTEXP**(\(a, b(n)\))

\[ p \leftarrow a \]

for \(i \leftarrow \ell\) downto 0 do

\[ p \leftarrow p \cdot p \]

if \(b_i = 1\) then \(p \leftarrow p \cdot a\)

return \(p\)

- Number of multiplications bounded by the number of 1-bits
- This is bounded by \(\ell\), the length of \(b\)
- \(\ell - 1 = \lfloor \lg n \rfloor\)
- \(\therefore M(n) = O(\lg n)\)
- But we must have binary string to begin with!

For example: \(a^{13}\) where \(n = 13 = 1101_2\):

<table>
<thead>
<tr>
<th>binary digits of (n)</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>product accumulator</td>
<td>(a)</td>
<td>(a^2 \cdot a = a^3)</td>
<td>((a^3)^2 = a^6)</td>
<td>((a^6)^2 \cdot a = a^{13})</td>
</tr>
<tr>
<td>example</td>
<td>3</td>
<td>((9) \cdot 3 = 27)</td>
<td>((27)^2 = 729)</td>
<td>((729)^2 \cdot 3 = 1,594,323)</td>
</tr>
</tbody>
</table>
Right-to-Left Binary Exponentiation

- Can re-express $a^n$:
  \[ a^{b_\ell x^\ell + \cdots + b_i x^i + \cdots + b_0} = a^{b_\ell 2^\ell} \cdots a^{b_i 2^i} \cdots a^{b_0} \]

- We recognize that:
  \[ a^{b_i 2^i} = \begin{cases} 
  a^{2^i} & \text{if } b_i = 1 \\
  1 & \text{if } b_i = 0 
\end{cases} \]

- This is also $O(\log n)$

- Also relies on having an available binary string
Right-to-Left Binary Exponentiation

- Can re-express $a^n$:
  $$a^{b_ℓ x^ℓ + \cdots + b_i x^i + \cdots + b_0} = a^{b_ℓ 2^ℓ \cdots a^{b_i 2^i} \cdots a^{b_0}}$$

- We recognize that:
  $$a^{b_i 2^i} = \begin{cases} a^{2^i} & \text{if } b_i = 1 \\ 1 & \text{if } b_i = 0 \end{cases}$$

- This is also $O(\lg n)$

- Also relies on having an available binary string

**RightToLeftExp**(a, b(n))

1. $t \leftarrow a$
2. If $b_0 = 1$ then $p \leftarrow a$
3. Else $p \leftarrow 1$
4. For $i \leftarrow 1$ to $\ell$ do
   1. $t \leftarrow t \cdot t$
   2. If $b_i = 1$ then $p \leftarrow p \cdot t$
5. Return $p$
Right-to-Left Binary Exponentiation

- Can re-express $a^n$:
  
  $$a^{b_\ell \cdot 2^\ell + \cdots + b_i \cdot 2^i + \cdots + b_0} = a^{b_\ell 2^\ell} \cdot \cdots \cdot a^{b_i 2^i} \cdot \cdots \cdot a^{b_0}$$

- We recognize that:
  
  $$a^{b_i 2^i} = \begin{cases} a^{2^i} & \text{if } b_i = 1 \\ 1 & \text{if } b_i = 0 \end{cases}$$

- This is also $O(\lg n)$

- Also relies on having an available binary string

For example: $a^{13}$ where $n = 13 = 1101_2$:

<table>
<thead>
<tr>
<th>$1$</th>
<th>$1$</th>
<th>$0$</th>
<th>$1$</th>
<th>binary digits of $n$</th>
<th>terms of $a^{2^i}$</th>
<th>product accumulator</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a^8$</td>
<td>$a^4$</td>
<td>$a^2$</td>
<td>$a$</td>
<td>$a^5 \cdot a^8 = a^{13}$</td>
<td>$a \cdot a^4 = a^5$</td>
<td>$3^5 \cdot 3^8 = 1,594,323$</td>
<td>$3 \cdot 3^4 = 243$</td>
</tr>
</tbody>
</table>

```python
def RightToLeftExp(a, b(n)):
    t ← a
    if $b_0 = 1$ then $p ← a$
    else $p ← 1$
    for $i ← 1$ to $\ell$ do
        $t ← t \cdot t$
        if $b_i = 1$ then $p ← p \cdot t$
    return $p$
```

For example:

- $a^{13}$ where $n = 13 = 1101_2$:
  - $a^8 \cdot a^4 = a^{13}$
  - $a \cdot a^4 = a^5$
  - $3^5 \cdot 3^8 = 1,594,323$
  - $3 \cdot 3^4 = 243$
  - $3$

R. Paul Wiegand
George Mason University, Department of Computer Science
“Reducing” Problems

- Not called “reducing” because the problem gets smaller or even (necessarily) easier.
- Comp Sci’s transform one problem into another as a means of classifying problems.
- Properly classified, the space of unique problems is reduced.
“Reducing” Problems

- Not called “reducing” because the problem gets smaller or even (necessarily) easier
- Comp Sci’s transform one problem into another as a means of classifying problems
- Properly classified, the space of unique problems is reduced
- Also reduce problems as a means of solving problems using known & proven methods
- Or when another view gives us some additional insight about the original problem
Least Common Multiple

The least common multiple of two positive integers $m$ and $n$, $\text{lcm}(m, n)$, is the smallest integer that is divisible by both $m$ and $n$.

- Middle school method:
  - Compute the prime factors of $m$ and $n$
  - Multiply common factors by the uncommon factors

\[
\begin{align*}
24 &= 2 \cdot 2 \cdot 3 \cdot 2 \\
60 &= 2 \cdot 2 \cdot 3 \cdot 5 \\
\text{lcm}(24, 60) &= (2 \cdot 2 \cdot 3) \cdot (2 \cdot 5)
\end{align*}
\]
Least Common Multiple

The *least common multiple* of two positive integers $m$ and $n$, lcm$(m, n)$, is the smallest integer that is divisible by both $m$ and $n$.

- **Middle school method:**
  - Compute the prime factors of $m$ and $n$
  - Multiply common factors by the uncommon factors

- **Alternatively:**
  - Note: The product of lcm$(m, n)$ and gcd$(m, n)$ includes every factor exactly once
  - In other words: lcm$(m, n) \cdot$ gcd$(m, n) = m \cdot n$
  - So, if we can solve gcd, we can solve lcm
  - gcd can be computed efficiently via Euclid’s algorithm

\[
24 = 2 \cdot 2 \cdot 3 = 2 \\
60 = 2 \cdot 2 \cdot 3 \cdot 5 \\
lcm(24, 60) = (2 \cdot 2 \cdot 3) \cdot (2 \cdot 5)
\]
How many paths of length $k$ are there between any pair of nodes in a graph?

We could perform a graph search and count the paths ...

But there’s a cool little trick:

- Consider the adjacency matrix $A$

\[
A = \begin{bmatrix}
0 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
\end{bmatrix}
\]
Counting Paths in a Graph

- How many paths of length \( k \) are there between any pair of nodes in a graph?
- We could perform a graph search and count the paths ...
- But there’s a cool little trick:
  - Consider the adjacency matrix \( A \)
  - Recall: \( A^2 = A \cdot A \) and \( A_{ij} = \{0, 1\} \) \( \forall i, j \)
  - So by matrix multiplication, \( A^2_{ij} \) is the sum of all situations in which the \( i \) is connected to some other node \textit{and} that node is connected to \( j \)

\[
A = \begin{bmatrix}
a & 0 & 1 & 1 & 1 \\
b & 1 & 0 & 0 & 0 \\
c & 1 & 0 & 0 & 1 \\
d & 1 & 0 & 1 & 0 \\
\end{bmatrix}
\]

\[
A^2 = \begin{bmatrix}
a & 3 & 0 & 1 & 1 \\
b & 0 & 1 & 1 & 1 \\
c & 1 & 1 & 2 & 1 \\
d & 1 & 1 & 1 & 2 \\
\end{bmatrix}
\]
Counting Paths in a Graph

- How many paths of length $k$ are there between any pair of nodes in a graph?
- We could perform a graph search and count the paths ...
- But there’s a cool little trick:
  - Consider the adjacency matrix $A$
  - Recall: $A^2 = A \cdot A$ and $A_{ij} = \{0, 1\} \ \forall i, j$
  - So by matrix multiplication, $A_{ij}^2$ is the sum of all situations in which the $i$ is connected to some other node and that node is connected to $j$
  - $A^k = A \cdot A \cdot A \cdots$
  - The value at $A_{ij}^k$ will be the number of paths of length $k$ that connect $i$ and $j$
Optimization

- One optimization problem is *maximization* — \( \text{argmax}\{f(x)\} \), find the argument value for \( x \) that gives us \( \max\{f(x)\} \)
- We may also be asked to *minimize* a function
- It turns out that this is the *same problem*:
  \[
  \max\{f(x)\} = -\max\{-f(x)\}
  \]
- This works for virtually any domain — so if you can solve maximization, you can solve minimization
- Moreover, the standard calculus method is a type of reduction:
  - Calculate the derivative, \( f'(x) = \frac{d}{dx} f(x) \)
  - Solve for \( f'(0) \)
  - Assuming the derivatives can be calculated, this reduces to the problem of finding critical points
Linear Programming

- Linear programming problems involve optimizing a linear function subject to linear constraints.
- There exists a general form for many LP problems:
  \[
  \text{maximize} \quad c_1 x_1 + \cdots + c_n x_n \\
  \text{subject to} \quad a_{i1} x_1 + \cdots + a_{in} x_n \{\leq, =, \geq\} b_i \quad \forall i \in [1, m] \\
  x_1 \geq 0, \ldots, x_n \geq 0
  \]
- Many (many) problems in computer science can be reduced to such problems (e.g., the fractional knapsack problem).
- There are a variety of well-known methods for solving them:
  - The simplex method, which has an exponential worst-case bound, but whose average case is typically quite good.
  - Karmarkar’s algorithm, which guarantees a polynomial worst-case bound and has done well empirically.
- A much harder, related class of problems are integer linear programming, which are known to be NP-hard in general (e.g., the 0-1 knapsack problem).
Book Topics Skipped in Lecture

- In section 6.6:
  - Reduction to Graph Problems (pp. 239–240)
This week’s assignments:

- Section 6.3: Problems 1, 4, & 7
- Section 6.4: Problems 1 & 6
- Section 6.5: Problems 4, 6, 7 & 8
- Section 6.6: Problems 1, 8, & 9