CS 483 - Data Structures and Algorithm Analysis
Lecture V: Chapter 5, part 1

R. Paul Wiegand

George Mason University, Department of Computer Science

February 22, 2006
Outline

1. Introduction to Decrease-And-Conquer
2. The insertionSort Algorithm
3. Depth-First and Breadth-First Searching
4. Homework
Decrease-And-Conquer

- Decrease-and-conquer exploits the relationship between a solution to a given problem instance and a solution to a smaller instance of the same problem.

- Divide-and-conquer attempts to solve separate pieces of the problem, then combine the pieces into an answer, while Decrease-and-conquer attempts to say something about the total solution in terms of the solution to the smaller piece.

- Can be approached top-down (recursively) or bottom-up.

- Three variations:
  - decrease by a constant — Each iteration, the size of a problem instance is reduced by a constant (e.g., $n-1$).
  - decrease by a constant factor — Each iteration, the size of a problem instance is reduced by a constant factor (e.g., $\frac{n}{2}$).
  - variable size decrease — The reduction pattern varies with each iteration (e.g., Euclid).
Consider the problem of computing \( f(n) = a^n \):

- **Decrease by a constant:**
  \[
  f(n) = \begin{cases} 
  f(n-1) \cdot a & \text{if } n > 1 \\ 
  a & \text{if } n = 1 
  \end{cases}
  \]

- **Decrease by a constant factor:**
  \[
  a^n = \begin{cases} 
  (a^{n/2})^2 & \text{if } n > 0 \text{ is even} \\
  (a^{(n-1)/2})^2 \cdot a & \text{if } n > 1 \text{ is odd} \\
  a & \text{if } n = 1 
  \end{cases}
  \]
Simple Examples of Decrease-and-Conquer

Consider the problem of computing $f(n) = a^n$:

- Decrease by a constant:
  $$f(n) = \begin{cases} 
  f(n-1) \cdot a & \text{if } n > 1 \\
  a & \text{if } n = 1 
  \end{cases}$$

- Decrease by a constant factor:
  $$a^n = \begin{cases} 
  \left(\frac{a^n}{2}\right)^2 & \text{if } n > 0 \text{ is even} \\
  \left(\frac{a^{n-1}}{2}\right)^2 \cdot a & \text{if } n > 1 \text{ is odd} \\
  a & \text{if } n = 1 
  \end{cases}$$

- We are not solving each piece
- We are using knowledge about how the solution to the piece relates to the whole problem
- $O(\lg n)$
Specifying **InsertionSort**

**InsertionSort**($A[0 \ldots n-1]$)

for $i \leftarrow 1$ to $n-1$ do

$v \leftarrow A[i]$

$j \leftarrow i - 1$

while $j \geq 0$ and $A[j] > v$ do

$A[j+1] \leftarrow A[j]$

$j \leftarrow j - 1$

$A[j+1] \leftarrow v$

- $v = 21$
- $i$
Specifying **InsertionSort**

**InsertionSort**(A[0…n − 1])

for \(i \leftarrow 1\) to \(n − 1\) do
  \(v \leftarrow A[i]\)
  \(j \leftarrow i − 1\)
  while \(j \geq 0\) and \(A[j] > v\) do
    \(A[j + 1] \leftarrow A[j]\)
    \(j \leftarrow j − 1\)
  \(A[j + 1] \leftarrow v\)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(j \rightarrow\)

\(A\)

\(v = 21\)

\(i\)

\(47\)

\(87\)

\(59\)

\(50\)

\(69\)
Specifying \textsc{InsertionSort}

\textsc{InsertionSort}(A[0 \ldots n - 1])

\begin{align*}
\text{for } & i \leftarrow 1 \text{ to } n - 1 \text{ do} \\
& v \leftarrow A[i] \\
& j \leftarrow i - 1 \\
& \quad \text{while } j \geq 0 \text{ and } A[j] > v \text{ do} \\
& \quad A[j + 1] \leftarrow A[j] \\
& \quad j \leftarrow j - 1 \\
& A[j + 1] \leftarrow v
\end{align*}
Specifying **InsertionSort**

**InsertionSort**($A[0 \ldots n - 1]$)

for $i \leftarrow 1$ to $n - 1$ do
  $v \leftarrow A[i]$
  $j \leftarrow i - 1$
  while $j \geq 0$ and $A[j] > v$ do
    $A[j + 1] \leftarrow A[j]$
    $j \leftarrow j - 1$
  $A[j + 1] \leftarrow v$

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>69</td>
<td>59</td>
</tr>
<tr>
<td>47</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

$v = 50$

$i \leftarrow j$
Specifying **InsertionSort**

**InsertionSort**\((A[0 \ldots n-1])\)

\[
\text{for } i \leftarrow 1 \text{ to } n-1 \text{ do }
\]

\[
v \leftarrow A[i]
\]

\[
j \leftarrow i - 1
\]

\[
\text{while } j \geq 0 \text{ and } A[j] > v \text{ do }
\]

\[
A[j + 1] \leftarrow A[j]
\]

\[
j \leftarrow j - 1
\]

\[
A[j + 1] \leftarrow v
\]
Specifying **INSERTIONSORT**

**INSERTIONSORT**\((A[0...n-1])\)

```plaintext
for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        j ← j − 1
    A[j + 1] ← v
```

```plaintext
<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
</tr>
<tr>
<td>50</td>
</tr>
<tr>
<td>69</td>
</tr>
<tr>
<td>87</td>
</tr>
<tr>
<td>47</td>
</tr>
</tbody>
</table>
```

\(v = 59\)
Specifying **InsertionSort**

**InsertionSort**\((A[0 \ldots n-1])\)

\[
\text{for } i \leftarrow 1 \text{ to } n-1 \text{ do}
\]

\[
v \leftarrow A[i]
\]

\[
j \leftarrow i - 1
\]

\[
\text{while } j \geq 0 \text{ and } A[j] > v \text{ do}
\]

\[
A[j + 1] \leftarrow A[j]
\]

\[
j \leftarrow j - 1
\]

\[
A[j + 1] \leftarrow v
\]
Specifying InsertionSort

InsertionSort(A[0...n − 1])

for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        j ← j − 1
    A[j + 1] ← v
### Specifying **InsertionSort**

**InsertionSort**($A[0\ldots n-1]$)

```
for i ← 1 to n - 1 do
    v ← $A[i]$
    j ← i - 1
    while $j \geq 0$ and $A[j] > v$ do
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j - 1$
    $A[j+1] \leftarrow v$
```

#### Example

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>v</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>47</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>59</td>
<td>59</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>69</td>
<td>69</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>87</td>
<td>87</td>
</tr>
</tbody>
</table>

---

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

CS483 Lecture II
Specifying **InsertionSort**

**InsertionSort**(\(A[0\ldots n-1]\))

for \(i \leftarrow 1\) to \(n-1\) do

\(v \leftarrow A[i]\)

\(j \leftarrow i - 1\)

while \(j \geq 0\) and \(A[j] > v\) do

\(A[j + 1] \leftarrow A[j]\)

\(j \leftarrow j - 1\)

\(A[j + 1] \leftarrow v\)

\[
\begin{array}{c|c|c|c|c|c}
\hline
\text{A} & v = 47 \\
\hline
21 & 47 & 50 & 59 & 69 & 87 \\
\hline
\end{array}
\]

It's like arranging cards in your hand!
When dealing with $A[n - 1]$, we assume that the $A[0 \ldots n - 2]$ problem has already been solved.

We find an appropriate position for $A[n - 1]$ and insert it.

The idea of this algorithm is recursive, but a bottom-up, iterative implementation is typically best.

One way to speed up insertion is to use `BinarySearch` to find the position (aka *binary insertion sort*).
Analyzing (Straight) InsertionSort

We count $A[j] > v$ comparisons, analysis depends on data ...

- Worst case:

- Best case:

- Average case:
Analyzing (Straight) **InsertionSort**

We count $A[j] > v$ comparisons, analysis depends on data ...  

- **Worst case:**
  - All elements in the sublist are shifted every insertion
  - This occurs when $A$ is initially strictly decreasing
  - $C_{\text{worst}}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} i = \frac{(n-1)n}{2} \in \Theta(n^2)$

- **Best case:**

- **Average case:**
Analyzing (Straight) **INSERTIONSORT**

We count $A[j] > v$ comparisons, analysis depends on data ...

- **Worst case:**
  - All elements in the sublist are shifted every insertion
  - This occurs when $A$ is initially strictly decreasing
  - $C_{\text{worst}}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} i = \frac{(n-1)n}{2} \in \Theta(n^2)$

- **Best case:**
  - We check each insertion, but no shift is necessary
  - $C_{\text{best}}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$

- **Average case:**

R. Paul Wiegand  
George Mason University, Department of Computer Science
Analyzing (Straight) **INSERTION SORT**

We count $A[j] > v$ comparisons, analysis depends on data ...

- **Worst case:**
  - All elements in the sublist are shifted every insertion
  - This occurs when $A$ is initially strictly decreasing
  - $C_{\text{worst}}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} i = \frac{(n-1)n}{2} \in \Theta(n^2)$

- **Best case:**
  - We check each insertion, but no shift is necessary
  - $C_{\text{best}}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$

- **Average case:**
  - Investigate number of pairs of elements that are out of order
  - On randomly ordered arrays, **INSERTION SORT** makes on average half as many comparisons as on decreasing arrays
  - $C_{\text{avg}}(n) \approx \frac{n^2}{4} \in \Theta(n^2)$
Searching Graphs

- Solutions to many problems involve searching through a graph
- There are a variety of ways of to search a graph ...
- But there's a simple generalization for many methods:

\[
\text{GraphSearch}(G, a)
\]

\[
\begin{align*}
\text{WaitingList} & \leftarrow \langle a \rangle \\
\text{VisitedList} & \leftarrow \langle \rangle \\
\text{while not } & \text{ EMPTY(WaitingList) } \\
\text{ } & \quad v \leftarrow \text{ GETANDREMOVEITEM(WaitingList)} \\
\text{ChildrenList} & \leftarrow \text{ GETCHILDVERTICES}(G, v) \\
\text{ADDLISTTOLIST(WaitingList, ChildrenList)} \\
\text{ADDMETHODLIST(WistedList, v)}
\end{align*}
\]
Solutions to many problems involve searching through a graph
There are a variety of ways of to search a graph ...
But there’s a simple generalization for many methods:

\[
\text{GraphSearch}(G, a)\\
\]

\[
\begin{align*}
\text{WaitingList} & \leftarrow \langle a \rangle \\
\text{VisitedList} & \leftarrow \langle \rangle \\
\text{while not not Empty(WaitingList)} \quad & \\
\text{v} & \leftarrow \text{GetAndRemoveItem(WaitingList)} \\
\text{ChildrenList} & \leftarrow \text{GetChildVertices}(G, v) \\
\text{AddListToList(WaitingList, ChildrenList)} \\
\text{AddItemToList(VistedList, v)}
\end{align*}
\]

DFS, BFS, Best-First, and A* are all instances of this method, depending on the list structure. DFS uses a Stack; BFS uses a queue.
Solutions to many problems involve searching through a graph
There are a variety of ways of to search a graph ...
But there’s a simple generalization for many methods:

```
GRAPHSEARCH(G, a)
WaitingList ← ⟨a⟩
VisitedList ← ⟨⟩
while not EMPTY(WaitingList)
    v ← GETANDREMOVEITEM(WaitingList)
    ChildrenList ← GETCHILDVERTICES(G, v)
    ADDLISTTOLIST(WaitingList, ChildrenList)
    ADDITEMTOLIST(VistedList, v)
```

DFS, BFS, Best-First, and A* are all instances of this method, depending on the list structure. DFS uses a Stack; BFS uses a queue.

WARNING: The algorithms in the book are presented differently, but they are the same in spirit.
Depth-First Searching: An Example

- For adjacency matrix representation, traversal time is $\Theta(|V|^2)$
- For adjacency list representation, traversal time is $\Theta(|V| + |E|)$
- We can use the algorithm to check for connectivity & cycles, and to find articulation points
Depth-First Searching: An Example

For adjacency matrix representation, traversal time is $\Theta(|V|^2)$.

For adjacency list representation, traversal time is $\Theta(|V| + |E|)$.

We can use the algorithm to check for connectivity & cycles, and to find articulation points.

Stack = \langle cde \rangle

Visited = \langle a \rangle

Dead = \langle \rangle
For adjacency matrix representation, traversal time is $\Theta(|V|^2)$.
For adjacency list representation, traversal time is $\Theta(|V| + |E|)$.
We can use the algorithm to check for connectivity & cycles, and to find articulation points.

Stack = $\langle dfe \rangle$
Visited = $\langle ac \rangle$
Dead = $\langle \rangle$
Depth-First Searching: An Example

For adjacency matrix representation, traversal time is $\Theta(|V|^2)$

For adjacency list representation, traversal time is $\Theta(|V| + |E|)$

We can use the algorithm to check for connectivity & cycles, and to find articulation points

Stack = $\langle fe \rangle$
Visited = $\langle acd \rangle$
Dead = $\langle d \rangle$
Depth-First Searching: An Example

For adjacency matrix representation, traversal time is $\Theta(|V|^2)$

For adjacency list representation, traversal time is $\Theta(|V| + |E|)$

We can use the algorithm to check for connectivity & cycles, and to find articulation points.
Depth-First Searching: An Example

For adjacency matrix representation, traversal time is $\Theta(|V|^2)$
For adjacency list representation, traversal time is $\Theta(|V| + |E|)$
We can use the algorithm to check for connectivity & cycles, and to find articulation points
Depth-First Searching: An Example

- For adjacency matrix representation, traversal time is $\Theta(|V|^2)$
- For adjacency list representation, traversal time is $\Theta(|V| + |E|)$
- We can use the algorithm to check for connectivity & cycles, and to find articulation points

Stack = $\langle \rangle$
Visited = $\langle acdfbe \rangle$
Dead = $\langle de \rangle$
Depth-First Searching: An Example

For adjacency matrix representation, traversal time is $\Theta(|V|^2)$

For adjacency list representation, traversal time is $\Theta(|V| + |E|)$

We can use the algorithm to check for connectivity & cycles, and to find articulation points.

Stack = $\langle \rangle$

Visited = $\langle acdfbe \rangle$

Dead = $\langle debfca \rangle$
Breadth-First Searching: An Example

For adjacency matrix representation, traversal time is $\Theta(|V|^2)$

For adjacency list representation, traversal time is $\Theta(|V| + |E|)$

We can use the algorithm to check for connectivity & cycles, and to find minimum paths
Breadth-First Searching: An Example

- For adjacency matrix representation, traversal time is $\Theta(|V|^2)$
- For adjacency list representation, traversal time is $\Theta(|V| + |E|)$
- We can use the algorithm to check for connectivity & cycles, and to find minimum paths

Queue = ⟨cde⟩
Visited = ⟨a⟩
Breadth-First Searching: An Example

- For adjacency matrix representation, traversal time is $\Theta(|V|^2)$
- For adjacency list representation, traversal time is $\Theta(|V| + |E|)$
- We can use the algorithm to check for connectivity & cycles, and to find minimum paths
Breadth-First Searching: An Example

- For adjacency matrix representation, traversal time is $\Theta(|V|^2)$
- For adjacency list representation, traversal time is $\Theta(|V| + |E|)$
- We can use the algorithm to check for connectivity & cycles, and to find minimum paths
Breadth-First Searching: An Example

- For adjacency matrix representation, traversal time is $\Theta(|V|^2)$
- For adjacency list representation, traversal time is $\Theta(|V| + |E|)$
- We can use the algorithm to check for connectivity & cycles, and to find minimum paths
Breadth-First Searching: An Example

- For adjacency matrix representation, traversal time is $\Theta(|V|^2)$
- For adjacency list representation, traversal time is $\Theta(|V| + |E|)$
- We can use the algorithm to check for connectivity & cycles, and to find minimum paths
Breadth-First Searching: An Example

- For adjacency matrix representation, traversal time is $\Theta(|V|^2)$
- For adjacency list representation, traversal time is $\Theta(|V| + |E|)$
- We can use the algorithm to check for connectivity & cycles, and to find minimum paths
This week’s assignments:

- Section 5.1: Problems 4, 6, and 9
- Section 5.2: Problems 1, 4, and 7