LECTURE 07 – DECREASE & CONQUER

Paul Doliotis (PhD)
Adjunct Assistant Professor
Portland State University

What is Decrease & Conquer?

- Solves a problem instance of size n by:

What is Decrease & Conquer?

- Solves a problem instance of size n by:
  - decreasing n by a constant, e.g., 1, or

What is Decrease & Conquer?

- Solves a problem instance of size n by:
  - decreasing n by a constant, e.g., 1, or
  - decreasing n by a constant factor, e.g., 2, or

What is Decrease & Conquer?

- Solves a problem instance of size n by:
  - decreasing n by a constant, e.g., 1, or
  - decreasing n by a constant factor, e.g., 2, or
  - decreasing n by a variable amount

What is Decrease & Conquer?

- Solves a problem instance of size n by:
  - decreasing n by a constant, e.g., 1, or
  - decreasing n by a constant factor, e.g., 2, or
  - decreasing n by a variable amount (e.g., Euclid’s algorithm)

What is Decrease & Conquer?

- Solves a problem instance of size n by:
  - decreasing n by a constant, e.g., 1, or
  - decreasing n by a constant factor, e.g., 2, or
  - decreasing n by a variable amount (e.g., Euclid's algorithm)

… to get a problem instance of size $k < n$

What is Decrease & Conquer?

- Solves a problem instance of size n by:
  - decreasing n by a constant, e.g., 1, or
  - decreasing n by a constant factor, e.g., 2, or
  - decreasing n by a variable amount (e.g., Euclid’s algorithm)

… to get a problem instance of size k < n

- Solve the instance of size k, using the same algorithm recursively.

What is Decrease & Conquer?

- Solves a problem instance of size n by:
  - decreasing n by a constant, e.g., 1, or
  - decreasing n by a constant factor, e.g., 2, or
  - decreasing n by a variable amount (e.g., Euclid’s algorithm)

… to get a problem instance of size k < n

- Solve the instance of size k, using the same algorithm recursively.
- Use that solution to get the solution to the original problem.

What is Decrease & Conquer?

- Also known as the inductive or incremental approach

What is Decrease & Conquer?

- Also known as the inductive or incremental approach
- Implement it recursively (top-down)

What is Decrease & Conquer?

- Also known as the inductive or incremental approach
- Implement it recursively (top-down)
- Implement it iteratively (bottom-up)

Decrease & Conquer by one

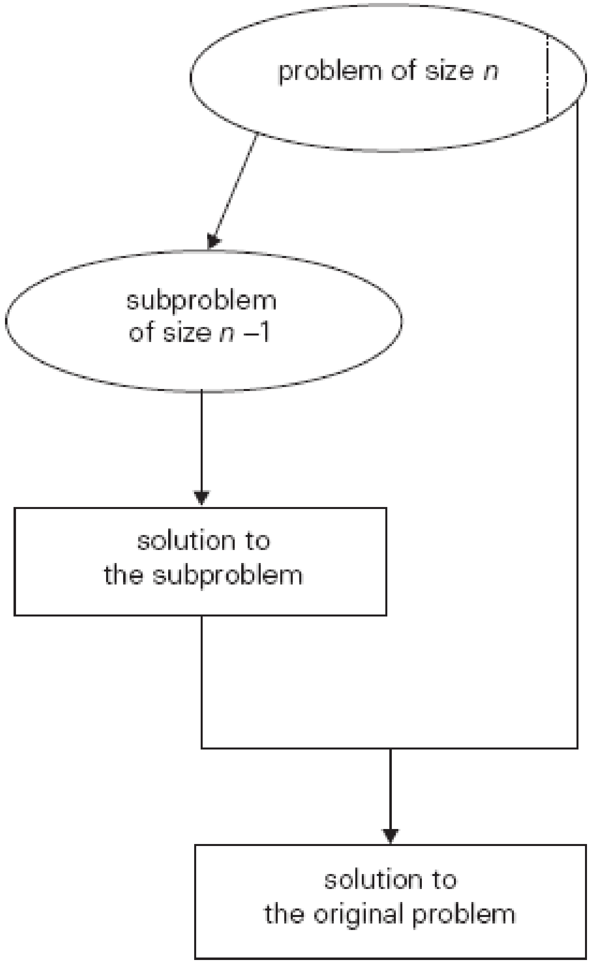

FIGURE 4.1 Decrease-(by one)-and-conquer technique.

Example - Decrease & Conquer by one

- Exponentiation using $a^n = a^{n-1} \times a$, $a \neq 0$, n non-negative integer

Example - Decrease & Conquer by one

- Exponentiation using $a^n = a^{n-1} \times a$, $a \neq 0$, n non-negative integer

FIGURE 4.1 Decrease-(by one)-and-conquer technique.

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0, \\ 1 & \text{if } n = 0, \end{cases} \qquad \textbf{(4.1)}$$

Example - Decrease & Conquer by one

- Exponentiation using $a^n = a^{n-1} \times a$, $a \neq 0$, n non-negative integer

FIGURE 4.1 Decrease-(by one)-and-conquer technique.

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0, \\ 1 & \text{if } n = 0, \end{cases} \qquad \textbf{(4.1)}$$

What about running time function complexity?

Example - Decrease & Conquer by one

- How does the decrease-and-conquer algorithm differ from the Brute-force algorithm?
  1. the brute-force algorithm is more efficient
  2. the decrease-and conquer algorithm is more efficient
  3. the two algorithms are identical
  4. the two algorithms have the same asymptotic efficiency, but decrease-and conquer has a better constant.

Example - Decrease & Conquer by constant

FIGURE 4.2 Decrease-(by half)-and-conquer technique.

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases} \quad \textbf{(4.2)}$$

Example - Decrease & Conquer by constant

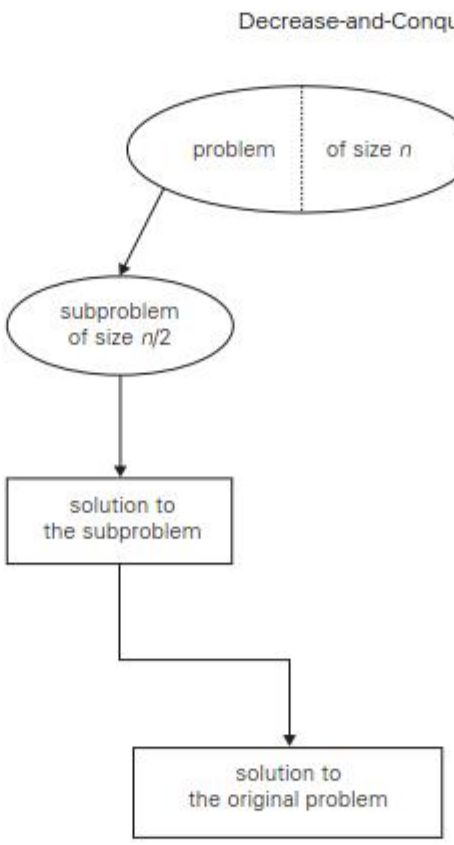

FIGURE 4.2 Decrease-(by half)-and-conquer technique.

Example - Decrease & Conquer by constant

FIGURE 4.2 Decrease-(by half)-and-conquer technique.

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases} \qquad \textbf{(4.2)}$$

What about running time function complexity?

Binary Search

- Search for an element K in an already sorted array A

$$\begin{array}{c} K \\ \updownarrow \\ \underbrace{A[0] \dots A[m-1]}_{\substack{\text{search here if} \\ K < A[m]}} \quad A[m] \quad \underbrace{A[m+1] \dots A[n-1]}_{\substack{\text{search here if} \\ K > A[m]}}. \end{array}$$

Binary Search

- Search for an element K in an already sorted array A
- Compare K with array’s middle element

$$\begin{array}{c} K \\ \updownarrow \\ \underbrace{A[0] \ldots A[m-1]}_{\substack{\text{search here if} \\ K < A[m]}} \quad A[m] \quad \underbrace{A[m+1] \ldots A[n-1]}_{\substack{\text{search here if} \\ K > A[m]}}. \end{array}$$

Binary Search

- Search for an element K in an already sorted array A
- Compare K with array’s middle element A[m]
- If they match stop. Otherwise do the same recursively for the first half if k < A[m], otherwise look at second half of the array

$$\underbrace{A[0]\ldots A[m-1]}_{\substack{\text{search here if}\\ K<A[m]}} \quad \overset{\overset{\displaystyle K}{\updownarrow}}{A[m]} \quad \underbrace{A[m+1]\ldots A[n-1]}_{\substack{\text{search here if}\\ K>A[m]}}.$$

Binary Search

ALGORITHM *BinarySearch*$(A[0..n-1], K)$

```
//Implements nonrecursive binary search
//Input: An array A[0..n − 1] sorted in ascending order and
//          a search key K
//Output: An index of the array's element that is equal to K
//          or −1 if there is no such element
l ← 0;   r ← n − 1
while l ≤ r do
    m ← ⌊(l + r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ← m − 1
    else l ← m + 1
return −1
```

Binary Search

ALGORITHM *BinarySearch*$(A[0..n-1], K)$

```
//Implements nonrecursive binary search
//Input: An array A[0..n − 1] sorted in ascending order and
//          a search key K
//Output: An index of the array's element that is equal to K
//          or −1 if there is no such element
l ← 0;   r ← n − 1
while l ≤ r do
    m ← ⌊(l + r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ← m − 1
    else l ← m + 1
return −1
```

- What is the basic operation?

Binary Search

ALGORITHM *BinarySearch*$(A[0..n-1], K)$

```
//Implements nonrecursive binary search
//Input: An array A[0..n − 1] sorted in ascending order and
//          a search key K
//Output: An index of the array's element that is equal to K
//          or −1 if there is no such element
l ← 0;   r ← n − 1
while l ≤ r do
    m ← ⌊(l + r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ← m − 1
    else l ← m + 1
return −1
```

- What is the basic operation?
- What about worst case complexity?

Binary Search

ALGORITHM *BinarySearch*$(A[0..n-1], K)$

//Implements nonrecursive binary search
//Input: An array $A[0..n-1]$ sorted in ascending order and
// a search key K
//Output: An index of the array's element that is equal to K
// or -1 if there is no such element
$l \leftarrow 0$; $r \leftarrow n-1$
while $l \leq r$ **do**
  $m \leftarrow \lfloor (l+r)/2 \rfloor$
  if $K = A[m]$ **return** m
  else if $K < A[m]$ $r \leftarrow m-1$
  else $l \leftarrow m+1$
return -1

- What is the basic operation?
- What about worst case complexity? $C(n) = C(n/2) + 1, n > 1$ $C(1)=1$

Insertion Sort

- Decrease-by-one, sort array A[0…n-1]

Insertion Sort

- Decrease-by-one, sort array A[0…n-1]
- If A[0….n-2] is sorted, how can we solve A[0….n-1]?

Insertion Sort

- Decrease-by-one, sort array A[0…n-1]
- If A[0….n-2] is sorted, how can we solve A[0….n-1]?
- All we have to do is put A[n-1] in the correct position.

Insertion Sort

- Decrease-by-one, sort array A[0…n-1]

Insertion Sort

- Decrease-by-one, sort array A[0…n-1]
- Decrease-by-one, A[0….n-2] is sorted:
  how can we solve A[0….n-1]?

Insertion Sort

- Decrease-by-one, sort array A[0…n-1]
- Decrease-by-one, A[0….n-2] is sorted:
  how can we solve A[0….n-1]?
- All we have to do is put A[n-1] in the correct position.

Insertion Sort

- Decrease-by-one, sort array A[0…n-1]
- Decrease-by-one, A[0….n-2] is sorted:
  how can we solve A[0….n-1]?
- All we have to do is put A[n-1] in the correct position.
  - Scan sorted subarray from right to left until you find a smaller element than A[n-1]
- Though insertion is based on a recursive idea its easier to implement bottom-up, iteratively.

Insertion Sort

89 | **45** 68 90 29 34 17

45 89 | **68** 90 29 34 17

45 68 89 | **90** 29 34 17

45 68 89 90 | **29** 34 17

29 45 68 89 90 | **34** 17

29 34 45 68 89 90 | **17**

17 29 34 45 68 89 90

FIGURE 4.4 Example of sorting with insertion sort. A vertical bar separates the sorted part of the array from the remaining elements; the element being inserted is in bold.

Insertion Sort

ALGORITHM *InsertionSort*($A[0..n-1]$)

    //Sorts a given array by insertion sort
    //Input: An array $A[0..n-1]$ of n orderable elements
    //Output: Array $A[0..n-1]$ sorted in nondecreasing order
    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**
            $A[j+1] \leftarrow A[j]$
            $j \leftarrow j-1$
        $A[j+1] \leftarrow v$

Insertion Sort

- Run animation: https://visualgo.net/bn/sorting

Runtime Analysis of Insertion Sort

- $C(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2)$

Runtime Analysis of Insertion Sort

- $C(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2)$
- This is <u>worst case</u> complexity (array of strictly decreasing values)

Runtime Analysis of Insertion Sort

- $C(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2)$
- This is worst case complexity (array of strictly decreasing values)
- In the best case, comparison A[j] > u is executed only once (array is already sorted in non-decreasing order)

Runtime Analysis of Insertion Sort

- $C(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2)$
- This is worst case complexity (array of strictly decreasing values)
- In the best case, comparison A[j] > u is executed only once (array is already sorted in non-decreasing order)

$$\sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

Runtime Analysis of Insertion Sort

- $C(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2)$
- This is worst case complexity (array of strictly decreasing values)
- In the best case, comparison A[j] > u is executed only once (array is already sorted in non-decreasing order)

$$\sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

  - almost-sorted files do arise in a variety of applications, and insertion sort preserves its excellent performance on such inputs.

Runtime Analysis of Insertion Sort

- $C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2)$, average case
- In the worst case Insertion Sort makes same number of comparisons as Selection Sort

Is Insertion Sort Stable?

- Is Insertion sort stable?

Is Insertion Sort Stable?

- Is Insertion sort stable?
  - Yes, it is stable

Euclid's Algorithm

```
int gcd(int m, int n)
{
  if (n == 0) return m;
  return gcd(n, m % n);
}
```

Euclid's Algorithm

```
int gcd(int m, int n)
{
  if (n == 0) return m;
  return gcd(n, m % n);
}
```

- One of the most ancient algorithms.

Euclid's Algorithm

```
int gcd(int m, int n)
{
  if (n == 0) return m;
  return gcd(n, m % n);
}
```

- One of the most ancient algorithms.
- Computes the greatest common divisor of two numbers.

Euclid's Algorithm

```
int gcd(int m, int n)
{
  if (n == 0) return m;
  return gcd(n, m % n);
}
```

- One of the most ancient algorithms.
- Computes the greatest common divisor of two numbers.
- It is based on the property that if T divides X and Y, then T also divides X mod Y.

Euclid's Algorithm

```
int gcd(int m, int n)
{
  if (n == 0) return m;
  return gcd(n, m % n);
}
```

- One of the most ancient algorithms.
- Computes the greatest common divisor of two numbers.
- It is based on the property that if T divides X and Y, then T also divides X mod Y.
- How is gcd(96, 36) evaluated?

Euclid's Algorithm

```
int gcd(int m, int n)
{
  if (n == 0) return m;
  return gcd(n, m % n);
}
```

- One of the most ancient algorithms.
- Computes the greatest common divisor of two numbers.
- It is based on the property that if T divides X and Y, then T also divides X mod Y.
- How is gcd(96, 36) evaluated?
- gcd(96, 36) = gcd(36, 24) = gcd(24, 12) = gcd(12, 0) = 12