Algorithm Analysis [2]: if-else statements, recursive algorithms

COSC 2011, Winter 2004, Section N
Instructor: N. Vlajic
Algorithm Analysis

“for-loop” Running Time

The running time of a simple loop

\[
\text{for (int } i=0; i<n; i++) \{
    \text{body; }
\}\]

is: \( RT(\text{loop}) \times RT(\text{body}) = n \times RT(\text{body}) \)

“if-else” Running Time

The running time of an if-else

\[
\text{if (cond) S1}
\text{else S2}
\]

is never more than: \( RT_{\text{worst-case}} = RT(\text{cond}) + \max\{RT(\text{S1}),RT(\text{S2})\} \)

is never less than: \( RT_{\text{best-case}} = RT(\text{cond}) + \min\{RT(\text{S1}),RT(\text{S2})\} \)
Example 1  [ running time of a simple loop with a nested if-else ]

```c
int k=0;
for (int i=0; i<n; i++) { /* outer_loop */
    if even(i) { for (int j=0; j<n; j++) /* inner_loop */
        k++; }
    else k *= 2;
}
```

\[ RT(n) = c + RT(\text{outer\_loop}) \times RT(\text{body}) = c + n \times RT(\text{body}) \]

\[ RT_{\text{worst-case}}(\text{body}) = c + \max\{ RT(\text{inner\_loop}), c \} = c + n \times c \]

\[ RT(n) \leq n \times RT_{\text{worst-case}}(n) = c + n \times (c + n \times c) \implies RT(n) = O(n^2) \]
Algorithm Analysis (cont.)

What is the best-case RT of the given algorithm, i.e. lower bound on RT??

```java
int k=0;
for (int i=0; i<n; i++) { /* outer_loop */
    if even(i) { for (int j=0; j<n; j++) /* inner_loop */
        k++; }
    else k *= 2;
}
```

$$RT_{\text{best-case}}(n) = c + n \times [\text{cond} + \min\{ S_1, S_2 \}] = c + n \times [c + c]$$

$$RT(n) \geq RT_{\text{best-case}}(n) \Rightarrow RT(n) = \Omega(n)$$
Algorithm Analysis (cont.)

What is the actual RT of the given algorithm, in big-Θ notation ??

Assume n odd.

```c
int k=0;
for (int i=0; i<n; i++) {
    if (i==0) { for (int j=0; j<n; j++) k++; };
    if (i==2) { for (int j=0; j<n; j++) k++; };
    ...
    if (i==n-1) { for (int j=0; j<n; j++) k++; };

    if (i==1) { k *= 2; };
    if (i==3) { k *= 2; };
    ...
    if (i==n) { k *= 2; };
}
```

\[
RT(n) = c + \frac{n+1}{2} \cdot (c+c \cdot n) + \frac{n-1}{2} \cdot (c+c) = c + c \cdot \left[ \frac{n^2}{2} + 2n - \frac{1}{2} \right] = \Theta(n^2)
\]
Algorithm Analysis (cont.)

The actual number of executed operations is less than what was roughly estimated, in the “worst case” analysis.

However, it still increases quadratically with n!
Algorithm Analysis (cont.)

**Recursive Algorithm** - algorithm that calls itself
- consists of a sub-problem, which is of exactly the same type as the original problem
- we attempt to solve the sub-problem and build up a solution to the entire problem

**RT of Algorithms using Recursion**
(sub-problem size: n-1)

\[
RT(n) = \begin{cases} 
O(1), & \text{if } n \text{ small enough & algorithm terminates} \\
RT(n-1) + C & \text{condition evaluation and other operations}
\end{cases}
\]

**RT of Algorithms using Recursion**
(sub-problem size: n/a)

\[
RT(n) = \begin{cases} 
O(1), & \text{if } n \text{ small enough & algorithm terminates} \\
RT(n/a) + C & \text{condition evaluation and other operations}
\end{cases}
\]
Algorithm Analysis (cont.)

Example 2 [ running time of a recursive algorithm ]

fact(n) = n*fact(n-1) = n*(n-1)*fact(n-2) = … = n*(n-1)*(n-2)*…*fact(1) = n!

```java
public int fact(int n) {
    if n<=1 return 1;
    else return n*fact(n-1);
}
```

- RT(1) = C (condition evaluation + assignment)
- RT(2) = C + RT(1) = 2C (condition evaluation + execution of fact(1))
- RT(3) = C + RT(2) = 3C (condition evaluation + execution of fact(2))
- ...
- RT(n) = RT(n-1) + C = C + (n-1)C = nC

RT(n) = O(n)
Time Complexity of Linear and Binary Search
Linear Search

Linear Search on **Unsorted Array** - compare each array component in turn with the target value, until either of the two occurs:

1) the target value is found
2) the end of array is reached

```java
static int linearSearch(int target, int[] a) {
    for (int i=0; i<a.length; i++) {
        if (target == a[i]) return i;
    }
    return -1;
}
```

**Performance of Linear Search**

1) Best Case RT(n): ~ 1
2) Worst Case RT(n): ~ n

\[ RT(n) = O(n) \]

If an array is not sorted, there is no better algorithm than “linear search” for finding an element in it.
Linear Search (cont.)

Linear Search on Sorted Array - by sorting an array, we can improve linear search operation slightly

- if current a[i] is greater than target, the search should be terminated immediately

```java
static int linearSearch(int target, int[] a) {
    for (int i=0; i<a.length; i++) {
        if (target == a[i]) return i;
        if (target < a[i]) return -1;
    }
    return -1;
}
```

The time complexity of linear search on a sorted array is still O(n), although it requires fewer comparisons than if the array was unsorted.
Binary Search

Binary Search algorithm can be used only if the array is sorted (in ascending order).

Binary Search - analogous to the way we search through a dictionary

1) compare target with the middle element; if they are not equal, divide the array into 2 sub-arrays

2) if target is less (greater) than the middle element, repeat the procedure in the left (right) sub-array

3) repeat the whole procedure until target is found or a sub-array of zero dimension is reached

target: 22

1 3 10 19 23 27 45 53 61 72 99

middle
Binary Search (cont.)

Implementation: Iterative Programming

```java
static int iterativeBinSearch(int target, int[] a, int left, int right) {
    while (left <= right) {
        int middle = (left + right) / 2;
        if (target == a[middle]) return middle;
        else if (target < a[middle]) right = middle - 1;
        else left = middle + 1;
    }
    return -1;
}
```

Target: 22

![Array and search process diagram]
Implementation: Recursive Programming

static int recursiveBinSearch(int target, int[] a, int left, int right) {
    if (left > right) return -1;
    int middle = (left+right) / 2;
    if (target == a[middle]) return middle;
    else if (target < a[middle])
        return recursiveBinSearch(target, a, left, middle -1);
    else
        return recursiveBinSearch(target, a, middle+1, right);
}

In order to find RT, we can employ the recursive formula from pp. 5 (a=2).
Complexity of Binary Search on Array of Size n

Complexity Analysis

\[ RT(n) = RT\left(\text{floor}\left(\frac{n}{2}\right)\right) + D(n) \]

\( RT(n) \) = time complexity of binary search on array of size \( n \).

\( D(n) \) = time complexity of 1) find mid-point, and 2) compare mid-point with target, ...

\[ RT(n) = RT\left(\frac{n}{2}\right) + C = RT\left(\frac{n}{4}\right) + 2C = RT\left(\frac{n}{8}\right) + 3C = ... \]

Let us assume an array of size \( n = 2^k \).
Then, in \( k = \log(n) \) steps \( n = 1 \) will be reached.

\[ RT(n) = RT\left(2^{k-1}\right) + C = RT\left(2^{k-2}\right) + 2C = RT\left(2^{k-3}\right) + 3C = ... = RT(1) + kC \]

\[ RT(n) = O(k) = O(\log_2 n) \]
### Time Complexity

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble Sort</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \cdot \log(n))$</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>$O(n \cdot \log(n))$</td>
</tr>
<tr>
<td>...</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

We will take a closer look at some of these algorithms later on ...