**Computer Science Department**
**College of Computer and Information Sciences**
**King Saud University**

# CSC 311: Design and Analysis of Algorithms[1]

**Dr. Waleed Alsalih**

Divide and conquer is a recursion-based algorithm design technique in which a problem is broken down into smaller sub-problems, and then solutions to sub-problems are combined to come up with a solution to the main problem. Problems are broken down to smaller sub-problems until these sub-problems are so simple that they can be solved directly (usually in a constant time).
General plan for divide and conquer algorithms:

1. An instance of the problem is divided into a number of smaller instances of the problem.
2. The smaller instances of the problem are solved recursively (make sure you have an appropriate stop condition).
3. Solutions of smaller instances are combined to get a solution to the original instance.

## 5.1 Recursion

A recursive algorithm is simply one that calls itself. A recursive algorithm must have an appropriate stop condition that prevents it from calling itself forever!
Example:
Consider an algorithm that computes the factorial of a positive integer.

Example:
Consider an algorithm that finds the sum of an array of $n$ integer nuumbers.

[1]This is a summary of the material we cover from the textbook: *Introduction to the Design & Analysis of Algorithms*, A. Levitin, Second Edition, Pearson Addison-Wesley, 2006.

**Algorithm 1**: A recursive algorithm that computes the factorial of a positive integer.

```
Algorithm Factorial($n$)
if $n = 0$ then
    return 1;
else
    return $Factorial(n-1) * n$;
end
```

**Algorithm 2**: Finding the sum of $n$ integers.

```
Algorithm Sum($A[\,]$,$first$,$last$)
if $first = last$ then
    return $A[first]$;
else
    return Sum($A[\,]$, $first, first + \left\lceil \frac{last-first}{2} \right\rceil - 1$)+
    Sum($A[\,]$, $first + \left\lceil \frac{last-first}{2} \right\rceil, last$);
end
% First call to this algorithm should be Sum ($A[\,]$, 0, $n-1$);
```

Example:
Tower of Hanoi: "It consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape. The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- No disk may be placed on top of a smaller disk." [2]

[2] From Wikipedia.

**Algorithm 3**: Tower of Hanoi (moving $n$ disks from rod $x$ to rod $z$ through rod $y$).

```
Algorithm TOH(int $n$, char $x$, char $y$, char $z$)
if $n = 1$ then
    Println (“move the top disk from rod ” $x$ “to the top of rod ” $z$);
else
    TOH($n-1$,$x$,$z$,$y$);
    TOH(1,$x$,$y$,$z$);
    TOH($n-1$,$y$,$x$,$z$);
end
```

**Computer Science Department**
**College of Computer and Information Sciences**
**King Saud University**

**CSC 311: Design and Analysis of Algorithms**[1]
**Dr. Waleed Alsalih**

## 5.2 Binary search and binary trees [Sections 4.3 and 4.4]

Binary search is a well known algorithm to search for a particular value $K$ in a sorted list of elements $A$. $K$ is compared with the element in the middle of the list and based on that comparison, the search is limited in the next step to only one half of the list. This process continues until $K$ is found or there is no more items to be checked.

**Algorithm 1**: Binary search for an item $K$ in an array $A[\,]$.

```
Algorithm BinarySearch($A[0..n-1]$,$K$)
$l$:=0;
$r$:=$n$-1;
while $l \leq r$ do
    $m$:=$\left\lfloor \frac{l+r}{2} \right\rfloor$;
    if $K$=$A[m]$ then
        return $m$;
    else
        if $K < A[m]$ then
            $r$:=$m-1$;
        else
            $l$:=$m+1$;
        end
    end
end
return $-1$;
```

[1]This is a summary of the material we cover from the textbook: *Introduction to the Design & Analysis of Algorithms*, A. Levitin, Second Edition, Pearson Addison-Wesley, 2006.

**Algorithm 2**: Recursive binary search for an item $K$ in an array $A[l..r]$.

```
Algorithm BinarySearchRec($A[\,], l, r, K$)
if $l > r$ then
    return $-1$
else
    $m := \left\lfloor \frac{l+r}{2} \right\rfloor$;
    if $K = A[m]$ then
        return $m$;
    else
        if $K < A[m]$ then
            return BinarySearchRec($A[\,], l, m-1, K$);
        else
            return BinarySearchRec($A[\,], m+1, r, K$);
        end
    end
end
```

## Binary Trees

A tree is a connected graph with no cycles.
A binary tree $T$ is a set of nodes that is either empty or has a root and two disjoint binary trees: a left subtree $T.L$ and a right subtree $T.R$. Note that this is a recursive definition.
A leaf node in a binary tree is one with no children.
Height of a binary tree is the length (in terms of the number of nodes) of the longest path from the root to a leaf node. Now, consider an algorithm that computes the height of a binary tree.

**Algorithm 3**: Finding the height of a binary tree.

```
Algorithm Height($T$)
if $T = \phi$ then
    return 0;
else
    return MAX(Height($T.L$),Height($T.R$))+1;
end
```

**Computer Science Department**
**College of Computer and Information Sciences**
**King Saud University**

**CSC 311: Design and Analysis of Algorithms**[1]
**Dr. Waleed Alsalih**

## 5.3 Performance analysis of recursive algorithms [Section 2.4]

To analyze a recursive algorithm, we need to:

1. Identify the basic operation in the recursive algorithm.
2. Set up a recurrence relation (function), with an initial condition, for the number of times the basic operation is executed. The result is a recursive function $M$ of the input size.
   Example:
   $$M(n) = \begin{cases} 0 & \text{for } n = 1. \\ M(\lfloor \frac{n}{2} \rfloor) + 1 & \text{for } n > 1. \end{cases}$$
3. Solve the recurrence relation, i.e., remove the recursion from the definition of $M$. In other words, express $M$ as a non-recursive function of the input size.
   Example:
   $M(n) = \log n$.
   There are two ways to solve a recurrence relation:
   (a) Use backward substitutions.
   (b) Make a guess and use mathematical induction to prove it.
4. Find the order of growth of $M$ using $O$ and/or $\Theta$.

[1]This is a summary of the material we cover from the textbook: *Introduction to the Design & Analysis of Algorithms*, A. Levitin, Second Edition, Pearson Addison-Wesley, 2006.

Example:
The factorial algorithm.
Basic operation: multiplication.

**Algorithm 1**: A recursive algorithm that computes the factorial of a positive integer.

```
Algorithm Factorial($n$)
if $n = 0$ then
    return 1;
else
    return $Factorial(n-1) * n$;
end
```

A recurrence relation for the basic operation count:

$$M(n) = \begin{cases} 0 & \text{for } n = 0. \\ M(n-1)+1 & \text{for } n > 0. \end{cases}$$

Backward substitutions:

$$\begin{array}{llll} & M(n) & = & M(n-1)+1 \\ = & [M(n-2)+1]+1 & = & M(n-2)+2 \\ = & [M(n-3)+1]+2 & = & M(n-3)+3 \\ - & -------------------------- & - & --------- \\ = & M(n-n)+n & = & M(0)+n \\ = & n & & \end{array}$$

We can use mathematical induction to prove our solution (easy).
$M(n) \in \Theta(n)$.

Example:
Binary search.

**Algorithm 2**: Recursive binary search for an item $K$ in an array $A[l..r]$.

```
Algorithm BinarySearchRec($A[\,], l, r, K$)
if $l > r$ then
    return $-1$
else
    $m := \left\lfloor \frac{l+r}{2} \right\rfloor$;
    if $K = A[m]$ then
        return $m$;
    else
        if $K < A[m]$ then
            return BinarySearchRec($A[\,], l, m-1, K$);
        else
            return BinarySearchRec($A[\,], m+1, r, K$);
        end
    end
end
```

Basic operation: comparison.
A recurrence relation for the basic operation count:

$$M(n) = \begin{cases} 1 & \text{for } n = 1. \\ M(\lfloor \frac{n}{2} \rfloor) + 1 & \text{for } n > 1. \end{cases}$$

Since we divide by 2, we assume that $n = 2^k$.

$$M(2^k) = \begin{cases} 1 & \text{for } k = 0. \\ M(2^{k-1}) + 1 & \text{for } k > 0. \end{cases}$$

Backward substitutions:

$$\begin{array}{lll}
 & M(2^k) & = M(2^{k-1}) + 1 \\
= & [M(2^{k-2}) + 1] + 1 & = M(2^{k-2}) + 2 \\
= & [M(2^{k-3}) + 1] + 2 & = M(2^{k-3}) + 3 \\
- & ----------------------------- & - \; -------- \\
= & M(2^{k-k}) + k & = M(2^0) + k \\
= & 1 + k & = 1 + \log n
\end{array}$$

We can use mathematical induction to prove our solution (easy).
$M(n) \in O(\log n)$.

**Computer Science Department**
**College of Computer and Information Sciences**
**King Saud University**

**CSC 311: Design and Analysis of Algorithms**[1]
**Dr. Waleed Alsalih**

## 5.4 Divide-and-conquer sorting algorithms

### Merge sort

Merge sort is a divide-and-conquer sorting algorithm which is based on the idea of splitting the elements to be sorted into two parts of almost the same size, sorting each part, and then merging the two parts into one sorted list.

**Algorithm 1**: Merge Sort.

```
Algorithm MergeSort($A[\,], first, last$)
if $first == last$ then
    return $A[first..last]$;
else
    return Merge(MergeSort($A[\,], first, \lceil \frac{first+last}{2} \rceil - 1$),
    MergeSort($A[\,], \lceil \frac{first+last}{2} \rceil, last$));
end
```

Basic operation count:

$$M(n) = \begin{cases} 0 & \text{for } n = 1. \\ 2M(\frac{n}{2}) + n & \text{for } n > 1. \end{cases}$$

[1]This is a summary of the material we cover from the textbook: *Introduction to the Design & Analysis of Algorithms*, A. Levitin, Second Edition, Pearson Addison-Wesley, 2006.

**Algorithm 2**: Merge.

```
Algorithm Merge($B[0..p-1], C[0..q-1]$)
Array $A[0..p+q-1]$;
$i$:=0;
$j$:=0;
$k$:=0;
while $i < p$ and $j < q$ do
    if $B[i] \leq C[j]$ then
        $A[k]$:=$B[i]$;
        $i$:=$i+1$;
    else
        $A[k]$:=$C[j]$;
        $j$:=$j+1$;
    end
    $k$:=$k+1$;
end
if $i$=$p$ then
    copy $C[j..q-1]$ to $A[k..p+q-1]$;
else
    copy $B[i..p-1]$ to $A[k..p+q-1]$
end
return $A[0..p+q-1]$;
```

Exercise:
Solve this recurrence relation and show that $M(n) = n \log n$.
You can do this by backward substitutions or by mathematical induction, and you can assume that $n = 2^k$, for some positive integer $k$.

## Quick sort

Quick sort is another divide-and-conquer algorithm for sorting a list of elements. Merge sort has the following steps:

1. Pick any element and call it the pivot.
2. Rearrange the list so that all elements with values less than that of the pivot come before the pivot and all those with values larger than the pivot come after the pivot.
3. Recursively sort the sub-list before the pivot and the sub-list after the pivot.

**Algorithm 3**: Quick Sort.

```
Algorithm QuickSort($A[\,], l, r$)
if $l < r$ then
    $s$:=Partition($A[\,], l, r$);
    QuickSort($A[\,], l, s-1$);
    QuickSort($A[\,], s+1, r$);
end
```

Exercise:
What is the best case time complexity of the quick sort algorithm?
What is the worst case time complexity of the quick sort algorithm?

**Algorithm 4**: Partition.

```
Algorithm Partition($A[\,], l, r$)
$p$:=$A[l]$;
$i$:=$l + 1$;
$j$:=$r$;
while $i < j$ do
    while $A[i] \leq p$ and $i < j$ do
        $i$:=$i + 1$;
    end
    while $A[j] > p$ and $i < j$ do
        $j$:=$j - 1$;
    end
    swap ($A$,$i$,$j$);
end
if $A[i] \leq p$ then
    swap ($A$,$i$,$l$);
    return $i$;
else
    swap ($A$,$i - 1$,$l$);
    return $i - 1$;
end
```

**Computer Science Department**
**College of Computer and Information Sciences**
**King Saud University**

**CSC 311: Design and Analysis of Algorithms**
**Dr. Waleed Alsalih**

## 5.5 The master method

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be an asymptotically positive function, and let $M(n)$ be defined on the nonnegative integers by the recurrence:
$M(n) = aM(\frac{n}{b}) + f(n)$,
where we interpret $\frac{n}{b}$ to mean either $\lfloor \frac{n}{b} \rfloor$ or $\lceil \frac{n}{b} \rceil$. Then $M(n)$ can be bounded asymptotically as follows[1].

1. If $f(n) \in O(n^{log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $M(n) \in \Theta(n^{log_b a})$.
2. If $f(n) \in \Theta(n^{log_b a} \log^k n)$, with $k \geq 0$, then $M(n) \in \Theta(n^{log_b a} \log^{k+1} n)$.
3. If $f(n) \in \Omega(n^{log_b a + \epsilon})$ for some constant $\epsilon > 0$ AND $af(\frac{n}{b}) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large $n$, then $M(n) \in \Theta(f(n))$.

NOTE THAT $\epsilon > 0$ and not $\epsilon \geq 0$.

Example: $M(n) = 9M(\frac{n}{3}) + n$.
$a = 9$, $b = 3$, $f(n) = n$, and thus $f(n) \in O(n^{log_3 9 - \epsilon})$ where $\epsilon = 1$. So $M(n) \in \Theta(n^2)$.

Example: $M(n) = 3M(\frac{n}{4}) + n \log n$.
$a = 3$, $b = 4$, $f(n) = n \log n$, and thus $f(n) \in \Omega(n^{log_4 3 + \epsilon})$ where $\epsilon = 0.1$. It is true that

[1]This definition is from: *Introduction to Algorithms*; T. Cormen, C. Leiserson, R. Rivest, and C. Stein; Second Edition; McGraw-Hill Higher Education; 2001.

$3f(\frac{n}{4}) \leq cf(n)$, where $c = \frac{3}{4}$. So $M(n) \in \Theta(n \log n)$.

Example: $M(n) = 2M(\frac{n}{2}) + n \log n$.
$a = 2$, $b = 2$, $f(n) = n \log n$, and thus $f(n) \in \Theta(n^{log_2 2} \log n)$. So $M(n) \in \Theta(n \log^2 n)$.

Example: $M(n) = 2M(\frac{n}{2}) - n$.
$f(n)$ is not asymptotically positive. So the master theorem does not apply.