# CSIT113
# Problem Solving

## UNIT 6
## REDUCE-AND-CONQUER AND DIVIDE-AND-CONQUER

UNIVERSITY OF WOLLONGONG AUSTRALIA

SIM GLOBAL EDUCATION



## Overview

- Problem Decomposition and Recombination
- Reduce-and-Conquer
- Divide-and-Conquer



## Problem Decomposition and Recombination

- Often, a problem seems to be too hard to solve simply because of its size.
- One useful way to attack such problems is to break them into smaller pieces.
- Induction is the general strategy to design algorithm to solve a problem from smaller sub-problems. Many these designs can also be carried out more specifically by taking the problem *decomposition* approach in two broad forms:
  - Reduce-and-conquer
  - Divide-and-conquer
- The remaining step is to build back up to the solution of the original problem. This is called *recombination*
- Both Reduce-and-conquer and Divide-and-conquer are powerful strategies for designing algorithms to solve many combinatorial problems.
- Though the solution leads naturally to a recursive implementation (from top-down), it can usually be implemented iteratively from bottom-up variation.



## Reduce-and-Conquer

- In this approach, we extend the solution of a sub-problem of a problem (**ONE smaller subproblem**) to form the solution for the problem.
- Once such a relationship is established, it can be exploited either top-down or bottom-up implementation. The former leads to recursive implementation and the latter leads to iterative implementation.
- In this way, we solve a problem by progressively reducing its size until we reach a base case which can be readily solved.
- Note that greedy strategies often use a reduce-and-conquer approach.
- Let us look at some problems and their solution designed from reduce-and-conquer technique.

## How to use Reduce-and-Conquer?

- It always solves a problem based on decomposition as follows:
  - ✓ **Cannot be decomposed further** : solve it directly.
  - ✓ **Otherwise:** Solve it by producing the solution of the whole problem from the solution of **ONE suitable smaller subproblem**.
- The smaller subproblem will be solved in the same way too.
- Identifying the suitable smaller subproblem is the key in using reduce-and-conquer.



## Real world reduction

- We often use reduce-and-conquer strategies to solve everyday problems.
- For example, finding the right key.
- We have a bunch of keys and need to unlock a toolbox. Unfortunately we don't know which key to use.
- The obvious strategy is to try each key in turn.
- But wait! This is a reduce-and-conquer strategy:
  - At each attempt we reduce the effective number of keys by one.



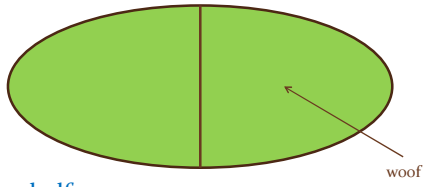
## Fence the Grue

- The grue is a fierce beast which lives in a large dark forest, surrounded by a grue-proof fence.
- It is also invisible.
- Luckily, we own a gruehound which barks when it shares an enclosure with the grue.
- Our aim is to trap the grue in a small section of the forest.
- We can use a reduce-and-conquer strategy to do this.



## Fence the Grue

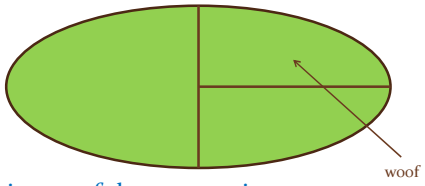
- Start by building a fence across the centre of the forest.




- The Grue must be in one half.
- The hound will tell us which.

## Fence the Grue

- Repeat the fencing operation of the half with the Grue.

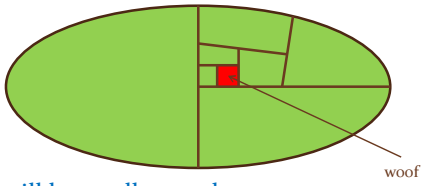



- The Grue must now be in one of the new sections.
- Again, the hound will tell us which.



## Fence the Grue

- Keep fencing in the section containing the grue.




- Eventually, the section will be small enough.



## The Use Reduce-and-Conquer in this Problem

- At each step the Grue is in **one** section of forest.
- This is the only section we further examine.
- At each step, we reduce the size of the area that we need to examine.
- Clearly, it is the use of reduce-and-conquer to find the solution.



## Egyptian Fractions revisited.

- In this problem, our task was to find a representation of a rational number by adding up a series of fractions all with a numerator of one.
  - 3/4 = 1/2 + 1/4
- The strategy we adopted was to repeatedly find the largest 1/x fraction that was less than or equal to the remaining part of our original number.
- If we look at this another way, we are recursively solving for progressively smaller numbers.
- Hence, we find the Egyptain Fractions of a rational number r by finding the Egyptain Fractions of a rational number that is smaller than r: That is, we solve the problem by solving a problem (**one only**) of reduced size (here, size is the value of the number). This is the use of reduce-and-conquer strategy.

## Egyptian Fractions revisited.

- Let us define a useful function, **part**($f$) as follows:
  - For any fraction $f = x/y$, **part**($f$) is the largest fraction with a numerator of one, less than or equal to $f$.
  - **part**$(x/y) = \frac{1}{\lceil y/x \rceil}$
  - The ceiling operator $\lceil y/x \rceil$ is evaluated as the smallest integer that is greater than or equal to $y/x$.
  - E.g. $\lceil 4/3 \rceil = 2$



## Recursive Egyptian Fractions

- Our strategy is now expressed recursively for solving the Egyptian fraction puzzle for fraction f as follows:

```
find(f) {
  if f = 0
    stop
  else {
    write down part(f)
    find(f - part(f))
  }
}
```

- Note that the last step of this strategy, solve for ($f$ -**part**($f$)), is to repeat the whole strategy again on a smaller number. That is, recursion.



## Recursive Egyptian Fractions

- Let us see this in action.

Solve for 4/5

**part**$(4/5) = \frac{1}{\lceil 5/4 \rceil} = 1/2$

write down "1/2 "

solve for 4/5 - 1/2 = 3/10

Solve for 3/10

**part**$(3/10) = \frac{1}{\lceil 10/3 \rceil} = 1/4$

write down "1/4 "

solve for 3/10-1/4 = 1/20

Solve for 1/20

**part**$(1/20) = \frac{1}{\lceil 20/1 \rceil} = 1/20$

write down "1/20 "

solve for 1/20 - 1/20 = 0

Solve for 0

stop

- So 4/5 = 1/2 + 1/4 + 1/20



## Computing Powers

- Given two integers m and n, calculate $m^n$
- We observe the following:
  - $m^n = 1$ if n = 0
  - $m^n = m^1 m^{n-1}$ if n > 0

## Computing Powers

- There are two ways to construct a reduce-and-conquer algorithm to solve the problem
- The first method is as follows:

```
Power(m, n) {
  if n = 0
    return 1
  else
    return m*Power(m, n-1)
}
```



## Computing Big Powers

- A better method is to use reduce-and-conquer in the following way instead.
- Consider:

$m^n = 1$ if n = 0

$m^n = (m^{n/2})^2$ if n is even

$m^n = m * m^{n-1}$ if n is odd



## Computing Big Powers – Algorithm for the Second Method

```
power_fast(m, n)  {
  if n = 0
    return 1
  else
    if n is even {
      temp = power_fast(m, n/2)
      return temp*temp }
    else
      return m*power_fast(m, n-1)
  }
```



## Divide-and-Conquer

- In this approach, we form the solution for a problem from the solutions for some of its sub-problems (**MULTIPLE subproblems**).
- Similar to reduce and-conquer, either top-down or bottom-up can be exploited to implement the solution. The former leads to recursive implementation and the latter leads to iterative implementation.
- Let us start with the computing big power problem that we use reduce-and-conquer to construct an algorithm earlier: by comparing the algorithm constructed from reduce-and- conquer and divide-and-conquer carefully, we should be able to see the differences between the two strategies.

## How to use Divide-and-Conquer?

- It always solves a problem based on decomposition as follows:
  - ✓ **Cannot be decomposed further** : solve it directly.
  - ✓ **Otherwise:** Solve it by producing the solution of the whole problem from the solutions of **MULTIPLE suitable smaller subproblems**.
- The smaller subproblems will be solved in the same way too.
- Identifying the multiple suitable smaller subproblems is the key in using divide-and-conquer.



## Computing Big Powers

- The previous algorithms on computing powers involves many function calls
- What's wrong with that?
  - Nothing if n is not too big
- Some applications (e.g. cryptography) involve computing powers in which both m and n are large numbers (up to 1000 bit values)
  - $2^{1000}$ is $1.071508607186267320948425049060 \times 10^{301}$
- Maybe there is a faster way?



## Computing Big Powers

- Another way is to use divide-and conquer instead by following the formulas below

- Consider:

$m^n = 1$ if $n = 0$

$m^n = m$ if $n = 1$

$m^n = m^{\lfloor n/2 \rfloor} \times m^{\lceil n/2 \rceil}$ if $n > 1$



## Computing Big Powers – Divide-and Conquer Algorithm

```
Algorithm DivConqPower(a, n) {
    //Computes a^n by a divide-and-conquer algorithm
    //Input: A positive number a and a positive integer n
    //Output: The value of a^n
    if n = 0
        return 1
    else
        if n = 1
            return a
        else
            return DivConqPower(a, ⌊n/2⌋) * DivConqPower(a, ⌈n/2⌉)
}
```

## Permutations

- All the permutations of the elements in the set.
- For example, consider the set of names {alan, bob, chris}:
  - The following are the permutations of this set:
    - alan, bob, chris
    - alan, chris, bob
    - bob, alan, chris
    - bob, chris, alan
    - chris, bob, alan
    - chris, alan, bob
- Our problem is to find a strategy for listing **all** permutations of a set.



## Notation

- Let us introduce a bit of notation:
- Let $S$ be a set of $n$ elements $\{s_1, s_2, ..., s_n\}$
- Let $S_i^*$ be the set we get if we remove element $s_i$ from set $S$. That is, $S_i^* = S - \{s_i\}$.
- E.g.
  - If $S$ = {tom, dick, harry}
  - $s_1$ = tom, $s_2$ = dick, $s_3$ = harry
  - $S_2^*$ = {tom, harry}



## An observation on permutation

- We notice (because we are observant) that we can divide all possible permutations into parts, where each part begins with a different element of the set.
  - E.g. permutations of {tom, dick, harry} = The combination of the permutations starting with tom, the permutations starting with dick and the permutations starting with harry.
- We also notice (because we are really observant) that each permutation which starts with tom ends with a permutation of {dick, harry}.
- Can we use this observation to find a divide-and-conquer strategy to list all permutations of a set?
- We will need to find a recursive strategy for this.



## Recursive permutation

- Each permutation of a set $S$ is of the form:
  - Some element of S followed by a permutation of the remaining elements of S.
- Using our earlier notation:
  - $s_i$ , permutation of $S_i^*$
- Wait a second!
- We can use the same idea to find a permutation of $S_i^*$
- At each step the size of the remaining set is reduced by one.
- Eventually we will end up with an empty set, { }.

## Some more notation

- We can split a permutation into two parts:
  - The part we have already produced. Let's call it the prefix that is a sequence.
  - The part we have yet to produce. Let's call it the suffix that is a set.
- At the start of the process the prefix is empty
- At each step the prefix gets one element longer
- We shall use Prefix + $s_i$ to denote the resulting sequence after appending $s_i$ *to* Prefix
- At each step the suffix gets one element shorter
- When the suffix is empty the prefix is a valid permutation
- We can use this to define a recursive, divide-and-conquer algorithm: find the permutations of the elements in S by finding the permutations of the elements in all the subsets of S with one element less than S, and construct the former permutations form the latter permutations. That is, if the size of S is n, we solve the problem by breaking the problem into n smaller problems and solve them. From the solutions of the subproblems, we construct the solution of the original problem.



## Recursive permutation

- Let us define a recursive procedure **permute** which has two arguments, prefix and suffix.
- Prefix is a sequence of elements. [$p_1$, $p_2$ ...] and Prefix + $s_i$ = *Append* $s_i$ *to Prefix*.
- Suffix is a set of elements, {$s_1$, $s_2$, ...} and $\text{Suffix}_i$* = Suffix – {$s_i$}.
- We construct the algorithm as follows:

```
permute(Prefix, Suffix) {
    if Suffix is empty
        write down Prefix
    else
        for each element, s_i, of Suffix
            permute(Prefix+ s_i, Suffix_i*)
}
```



## Recursive permutation

- Let us take a small example:
  - find all permutations of the set {a, b, c}
- At each step we will note the value the prefix and suffix.
- We start with **permute**([],{a, b, c})



## All Permutations of the Elements in {a,b,c}

```
permute([],{a,b,c})
    permute([a],{b,c})
        permute([ab],{c})
            permute([abc],{ }])      abc
        permute([ac],{b})
            permute([acb], { }])     acb
    permute([b],{a,c})
        permute([ba],{c})
            permute([bac], { }])     bac
        permute([bc],{a})
            permute([bca], { }])     bca
    permute([c],{a,b})
        permute([ca],{b})
            permute([cab],{ }])      cab
        permute([cb],{a})
            permute([cba],{ }])      cba
```

```
permute(Prefix, Suffix):
    if Suffix is empty
        write down Prefix
    else
        for each element, s_i, of Suffix
            permute(Prefix+ s_i, Suffix_i*)
```

- Which gives us the permutations abc, acb, bac, bca, cab and cba.
- This is a lot of recursion!

## Multiplication of Integers

- We all know how to do multiplication of large integers:
- 12345*6789

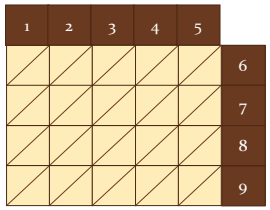
```
   12345
    6789
  111105
  987600
 8641500
74070000
83810205
```

- This involves getting four partial results and adding them together.
- Each partial result involves five single-digit multiplications.



## Gelosia Multiplication

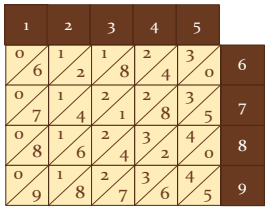
- We can see this more clearly if we use the Gelosia multiplication method.






## Gelosia Multiplication

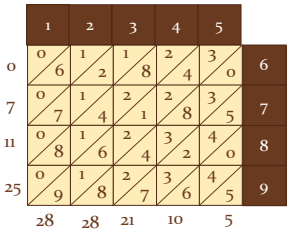
- Multiply each pair of single digits…





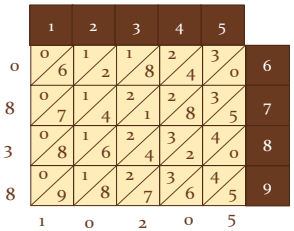

## Gelosia Multiplication

- Add down the diagonals



## Gelosia Multiplication

- Do the carries.






## Multiplication made Faster

- This gives us the result, (reading down and right):
- 083810205
- If we ignore the additions and carries we did 20 = 4 × 5 single-digit multiplications.
- In general if we multiply an $m$-digit number by an $n$-digit number we must carry out $m \times n$ single-digit multiplications.
- Can we do this with less than $m \times n$?



## Multiplication Made Faster

- Any multi-digit number can be split into two roughly equal parts:
- E.g. 123456 can be represented as 123 × 1000 + 456 × 1
- We can use this to express multiplication as follows:
  - Multiply 2 numbers $ab$ and $cd$ where each of $a$ *and* $c$ are $k_0$ digit sequences, b and $d$ are $k$ - digit sequences
  - $ab$ is really $a \times 10^k + b$
  - $ab \times cd = (a \times 10^k + b) \times (c \times 10^k + d)$
    $= (a \times c) \times 10^{2k} + ((a \times d) + (b \times c)) \times 10^k + (b \times d)$
  - Thus, splitting each number into two parts results in 4 multiplications.
- But we can be cleverer than that

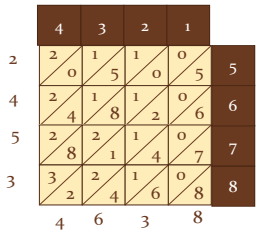


## Multiplication Made Faster

- Let us calculate another product:
  - $(a + b) \times (c + d)$ involves only a single multiplication
  - Since, $(a + b) \times (c + d) = (a \times c) + (a \times d) + (b \times c) + (b \times d)$
    We have, $(a \times d) + (b \times c) = (a + b) \times (c + d) - (a \times c) - (b \times d)$
  - *Hence,* $ab \times cd = (a \times c) \times 10^{2k} + ((a \times d) + (b \times c)) \times 10^k + (b \times d)$
    $= (a \times c) \times 10^{2k} + [(a + b) \times (c + d) - (a \times c) - (b \times d)] \times 10^k + (b \times d)$
  - So we only needed 3 multiplications, $(a \times c)$, $(b \times d)$ and $(a + b) \times (c + d)$ instead of 4!
- We can use this approach repeatedly (recursively) until we have single-digit multiplications.
- This method finds the product of two numbers by finding the products of three smaller numbers – using the divide-and-conquer strategy. We shall not cover the algorithm for this method.
- Though this method involves less multiplications at the expense of more additions (and subtractions), using algorithm analysis (not covered in this course), we can show that this method is faster for large numbers.

## Multiplication Made Faster: A comparison

- Multiply 4321 by 5678




- To give 24534638 with 16 multiplications.



## Multiplication Made Faster: A comparison

- Multiply 4321 by 5678
  - Calculate 43 × 56, 21 × 78 and 64 × 134
- 43 × 56
  - Calculate 4 × 5, 3 × 6 = 20, 18
  - Calculate 7 × 11
    - Calculate 0, 7 × 1 and 7 ×2 = 0, 7, 14
    - 7 × 11 = 0 × 100 + (14 – 0 – 7) × 10 + 7 = 77
  - 43 × 56 = 20 × 100 + (77 – 20 – 18) × 10 + 18 = **2408**
  - This involved a total of 4 single-digit multiplications
- 21 × 78 (Appendix shows the details of this multiplication and count the total no. of single-digit multiplications for this case = 4)
  - Calculate 2 × 7, 1 × 8 and 3 × 15 = 14, 8 and 45
  - 21 × 78 = 14 × 100 + (45 – 14 – 8) × 10 + 8 = **1638**



## Multiplication Made Faster: A comparison

- 64 × 134 (Appendix shows the details of this multiplication and count the total no. of single-digit multiplications for this case = 5)
  - Calculate 6 × 13, 4 × 4 and 10 × 17 = 78, 16 and 170
  - 64 × 134 = 78 × 100 + (170 – 78 – 16) × 10 + 16 = **8576**
- Multiply 4321 by 5678
  - Calculate 43 × 56, 21 × 78 and 64 × 134 = 2408, 1638 and 8576
  - 4321 × 5678 = 2408 × 10000 + (8576 – 2408 – 1638) × 100 + 1638
    = 24534638
- This involved a total of 13 single-digit multiplications
  - It is less than 16!



## Appendix

This appendix shows the steps on using "Multiplication Made Faster" and count the number of single-digit multiplications for each following multiplication:

a) 21×78
b) 64×134

a) 21 × 78
- Calculate 2 × 7, 1 × 8 = 14, 8
- Calculate 3 × 15
  - Calculate 0, 3 × 5 and 3 × 6 = 0, 15, 18
  - 3 × 15 = 0 × 100 + (18 – 0 – 15) × 10 + 15 = 45
- 21 × 78 = 14 × 100 + (45 – 14 – 8) × 10 + 8 = **1638**
- This involved a total of 4 single-digit multiplications

## Q Appendix

b) $64 \times 134$

- 64 divides into 6, 4
- 134 divides into 13, 4
- Calculate $6 \times 13$
  - Calculate 0, $6 \times 3$ and $6 \times 4 = 0, 18, 24$
  - $6 \times 13 = 0 \times 100 + (24 - 0 - 18) \times 10 + 18 = 78$
- Calculate $4 \times 4 = 16$
- Calculate $10 \times 17$
  - Calculate $1 \times 1$, 0 and $1 \times 8 = 1, 0, 8$
  - $10 \times 17 = 1 \times 100 + (8 - 1 - 0) \times 10 + 0 = 170$
- $64 \times 134 = 78 \times 100 + (170 - 78 - 16) \times 10 + 16 =$ **8576**
- This involved a total of 5 single-digit multiplications.