CSIT113
Problem Solving

Week 7

Problem Decomposition

- 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.
- This problem ***decomposition*** approach takes 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 ***recombination***.

Reduce and Conquer

- In this approach we solve a problem by progressively reducing its size by one until we reach a base case which can be readily solved.
- We then work back to our required solution.
- Reduce and conquer strategies often arise from induction and recursion which we looked at in week 5.
- Also, greedy strategies often use a reduce and conquer approach.
- Let us look at one of the problems from last week.

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.

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*.
  - $\mathbf{part}(x/y) = \frac{1}{\lceil y/x \rceil}$
  - The ceiling operator $\lceil y/x \rceil$ is evaluated as the smallest integer that is greater than or equal to y/x.
  - E.g. $\lceil 4/3 \rceil = 2$
- We can now solve the Egyptian fraction puzzle recursively:

Recursive Egyptians

- Our strategy is now expressed as follows:
  - Solve for f :
    - If $f = 0$ stop
      find **part**(*f*)
      write down (as part of our solution) **part**(*f*) +
      Solve for *f* minus **part**(*f*)
  - Note that the last step of this strategy, solve for *f* minus **part**(*f*), is to repeat the whole strategy again on a smaller number.
  - i.e. recursion.

Recursive Egyptians

- Let us see this in action.

Solve for 4/5

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

write down "1/2 "

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

Solve for 3/10

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

write down "1/4 "

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

Solve for 1/20

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

write down "1/20 "

solve for 1/20 - 1/20 = 0

Solve for 0

stop

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

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.

Permutations

- A permutation of a set is any sequence of the elements of the set.
- For example, consider the set of names {alan, bob, chris}:
  - The following are some of the permutations of this set:
    - alan, bob, chris
    - bob, alan, chris
    - chris, bob, alan
- 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 $\{s_1, s_2, \ldots, s_n\}$
- Let $S_i{*}$ be the set we get if we remove element s_i from set S
- E.g.
  - If S = {tom, dick, harry}
  - s_1 = tom, s_2 = dick, s_3 = harry
  - $S_2{*}$ = {tom, harry}

An observation on permutations

- 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} = permutations starting with tom + permutations starting with dick + 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 reduce and conquer strategy to list all permutations of a set?
- We 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.
  - The part we have yet to produce. Let’s call it the suffix.
- At the start of the process the prefix is empty
- At each step the prefix gets one element longer
- 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, reduce and conquer algorithm

Recursive permutation

- Let us define a recursive procedure **permute** which has two arguments, prefix and suffix.
- Prefix is a list of elements. [p_1, p_2 …]
- Suffix is a set of elements. {s_1, s_2, …}
- We construct the procedure 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})

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

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

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

Divide and Conquer

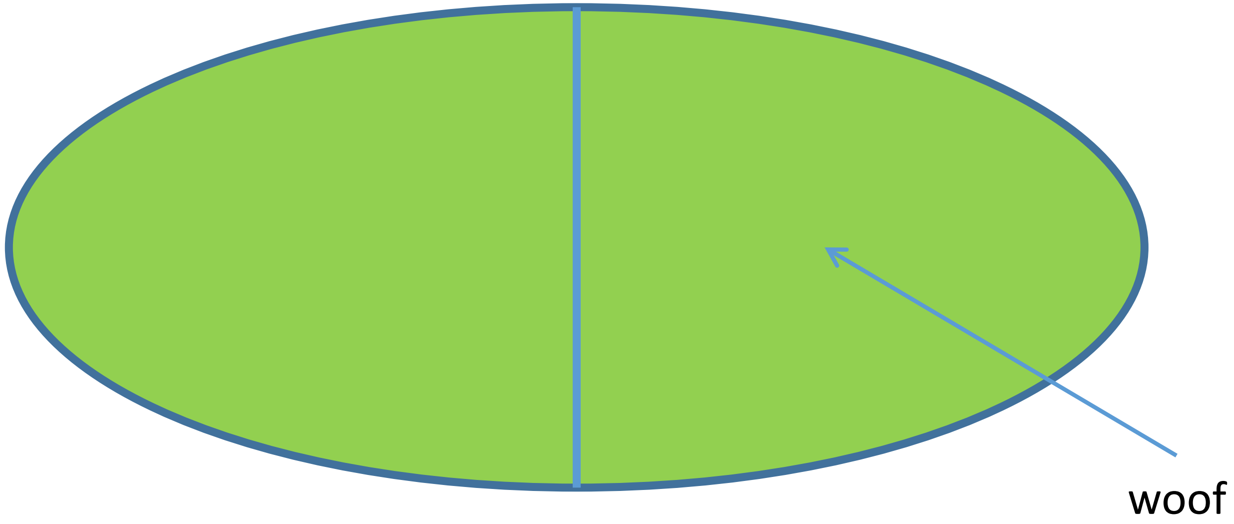
- The reduce and conquer strategy replaces a problem with a single, smaller problem.
- Another approach involves breaking a problem into multiple smaller parts.
- This is called the divide and conquer strategy.
- Let us start with a simple example:

Hunt 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 divide and conquer strategy to do this.

Fence the Grue

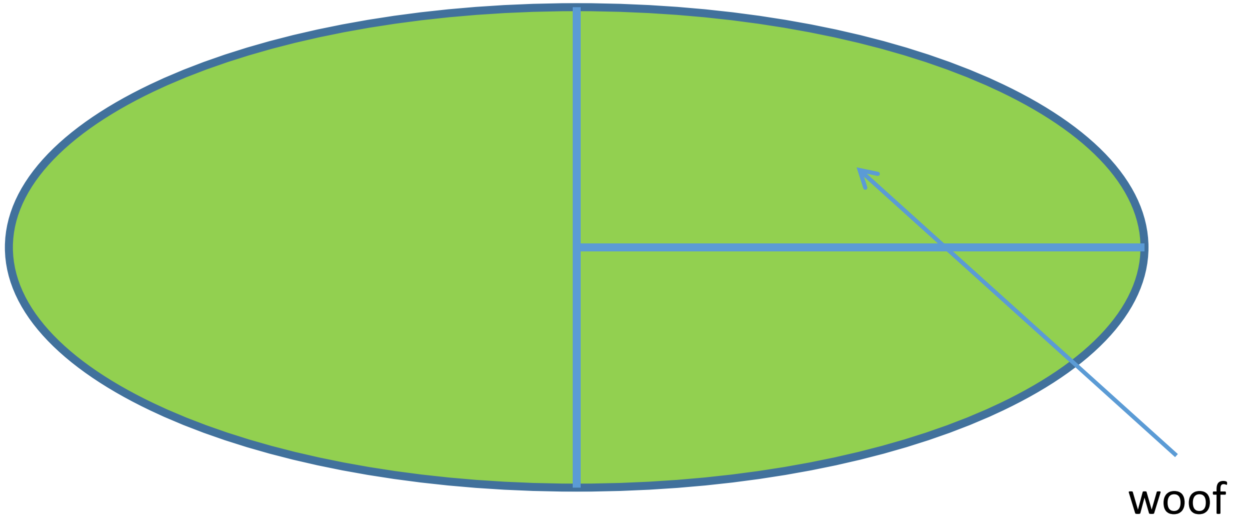
- Start by building a fence across the centre of the forest.



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

Fence the Grue

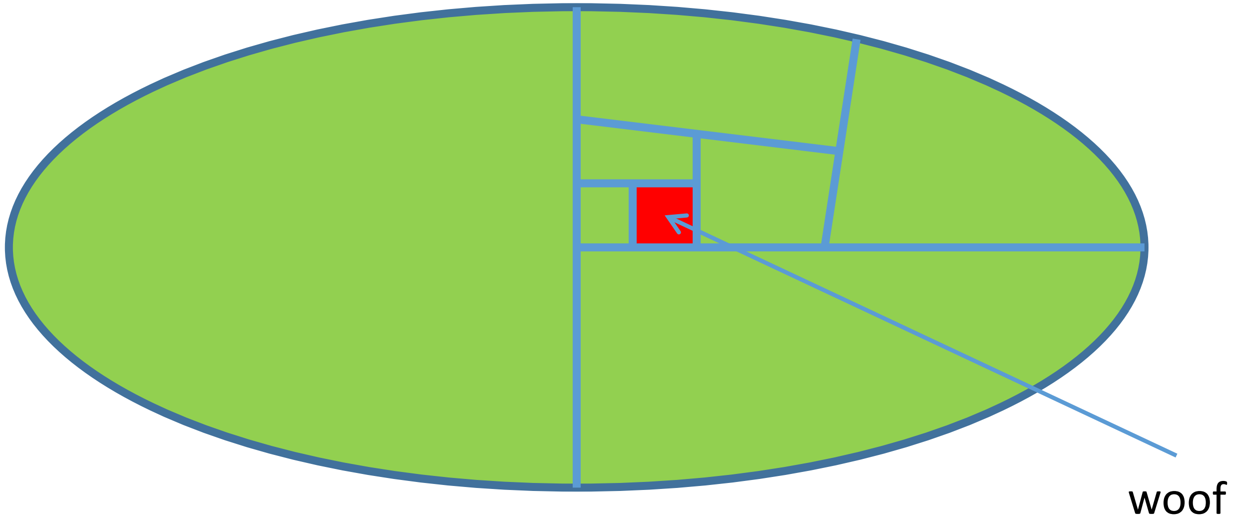
- Repeat the fencing operation of the half with the Grue.



- 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.

Divide and Conquer? Really?

- It could be argued that this example does not really divide the problem to solve.
- At each step the Grue is in **one** section of forest.
- This is the only section we further examine.
- Perhaps this is really just reduce and conquer.
- We can see a real divide and conquer approach emerge from our next example.

Computing Powers

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

Computing Powers

- We can use this to construct a reduce and conquer algorithm to solve the problem

- Power(m,n)

```
If n = 0 then
        return 1
Else
        return m*Power(m,n-1)
```

Computing Big Powers

- The previous algorithm involves n 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 (1000's of bits in length)
  - 2^{1000} is $1.071508607186267320948425049 06 \times 10^{301}$
- Maybe there is a faster way?

Computing Big Powers

- 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

- 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)

Multiplication made fast.

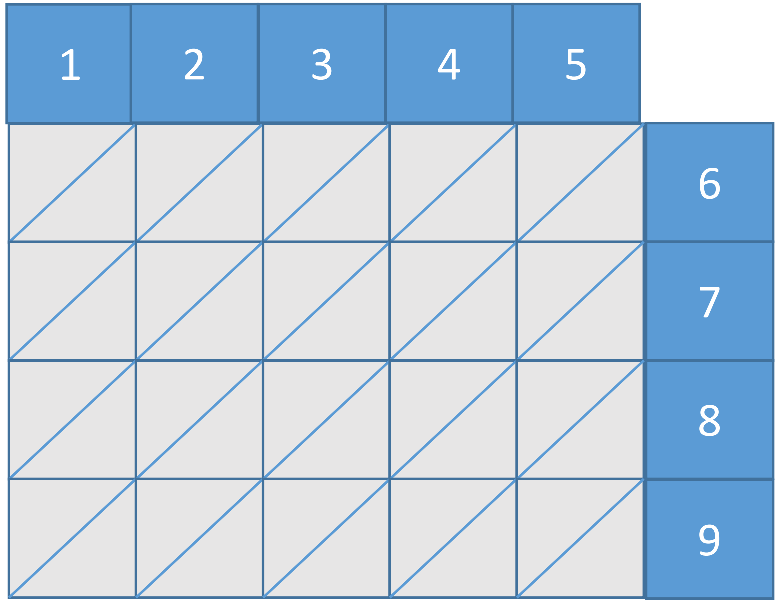
- We all know how to do multiplication of large integers:
- 12345*6789

```
   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.

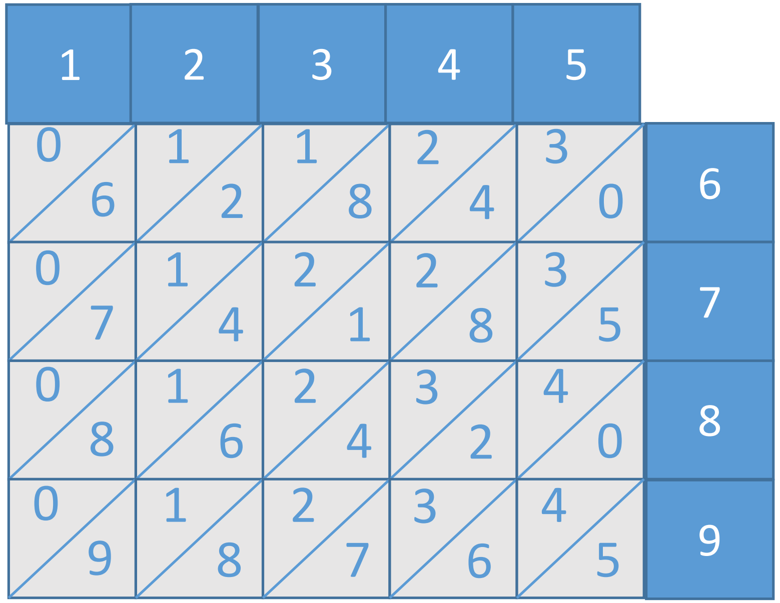
Multiplication made easy

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



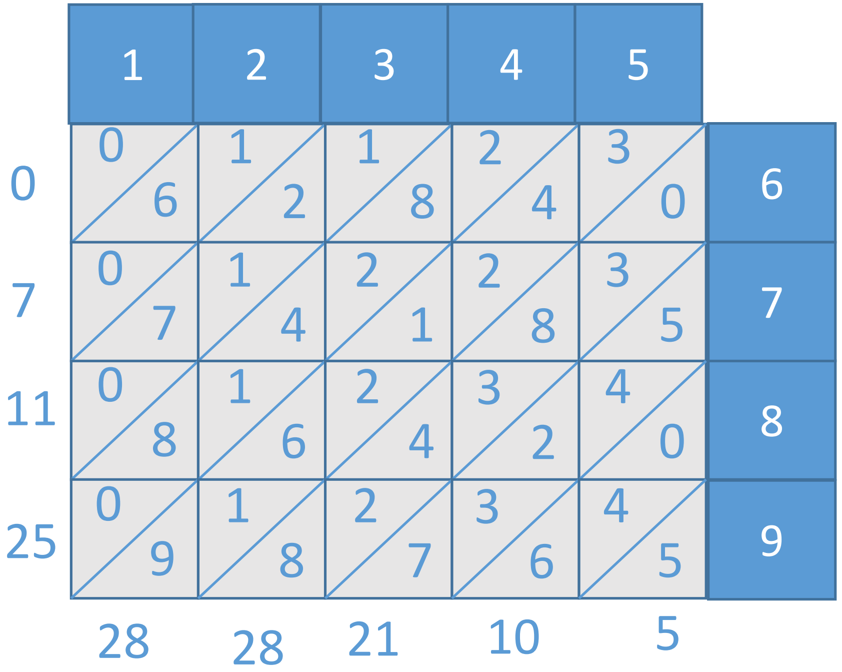
Multiplication made easy

- Multiply each pair of single digits...



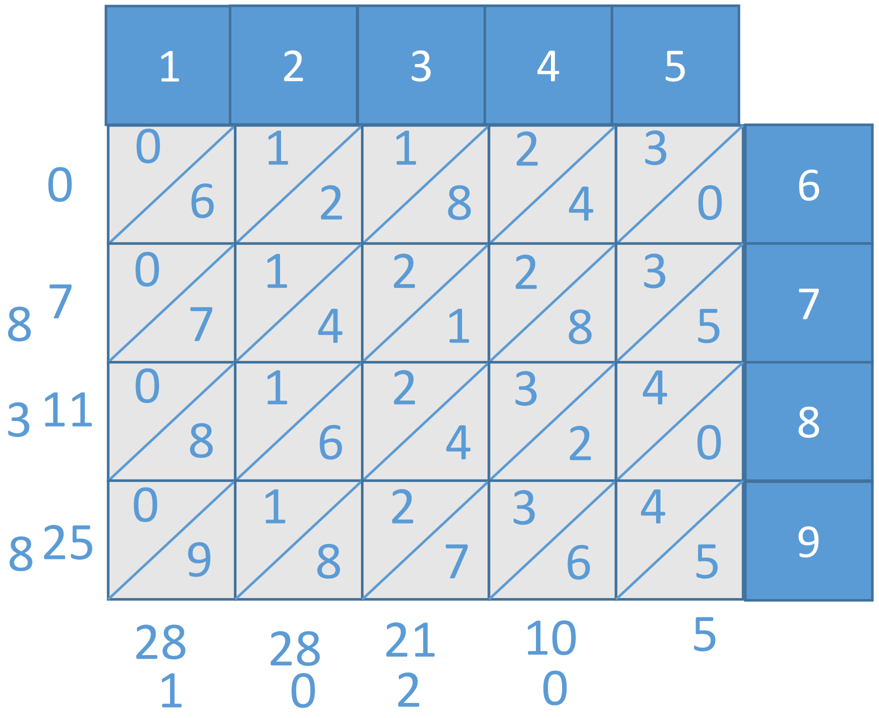
Multiplication made easy

- Add down the diagonals



Multiplication made easy

- Do the carries.



Multiplication made easy

- This gives us the result, (reading down and right):
- 083810205
- If we ignore the additions and carries we did $20 = 4 \times 5$ 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 $m \times n$?

Generalising multiplication

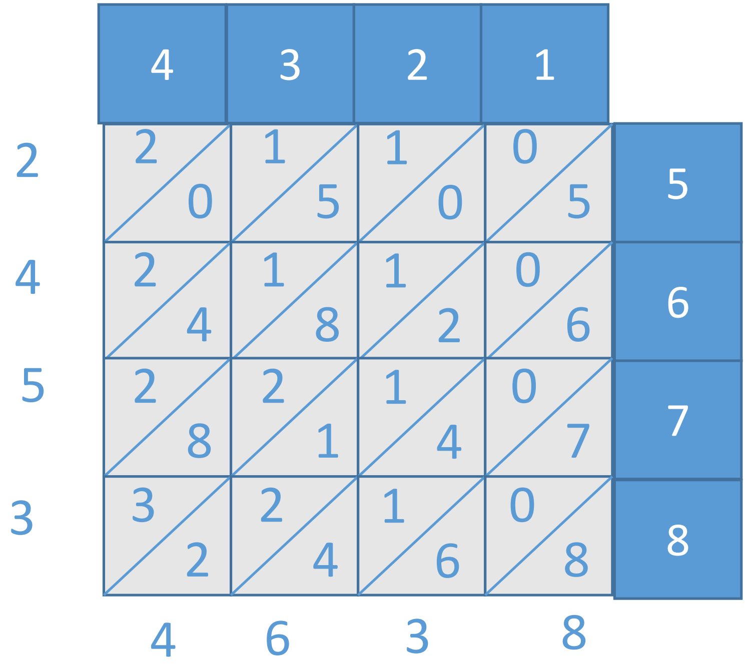
- Any multi-digit number can be split into two roughly equal parts:
- E.g. 123456 can be represented as $123 \times 1000 + 456 \times 1$
- We can use this to express multiplication as follows:
  - Multiply 2 numbers ab and cd where each of a, b, c, and d are k-digit sequences.
  - ab is really $a \times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

Generalising multiplication

- Let us calculate another product:
  - $(a + b) \times (c + d)$
  - This involves a single multiplication
  - This is the same as $(a \times c) + (a \times d) + (b \times c) + (b \times d)$
  - If we subtract $(a \times c)$ and $(b \times d)$ from this result we get $(a \times d) + (b \times c)$
  - So we only needed 3 multiplications instead of 4!
- We can use this approach repeatedly (recursively) until we have single-digit multiplications.
- This involves less multiplications at the expense of more additions (and subtractions).

A comparison

- Multiply 4321 by 5678



- To give 24534638 with 16 multiplications.

- Multiply 4321 by 5678
- Calculate 43×56, 21×78 and 64×134
- 43×56
  - Calculate 4×5, 3×6 and $7 \times 11 = 20, 18, 77$
  - $43 \times 56 = 20 \times 100$
    $+ (77 - 20 - 18) \times 10$
    $+ 18 =$ **240**8
- 21×78
  - Calculate 2×7, 1×8 and $3 \times 15 = 14$, 8 and 45
  - $21 \times 78 = 14 \times 100 + (45 - 14 - 8) \times 10 + 8 =$ **1638**
- $64 \times 134 = (43 + 21) \times (56 + 78)$
  - Calculate 6×13, 4×4 and $10 \times 17 = 78$, 16 and 170
  - $64 \times 134 = 78 \times 100 + (170 - 78 - 16) \times 10 + 16 =$ **8576**

- Multiply 4321 by 5678
- Calculate 43×56, 21×78 and $64 \times 134 = 2408$, 1638 and 8576
- 4321×5678

$$
\begin{aligned}
&= 2408 \times 10000 \\
&\qquad + (8576 - 2408 - 1638) \times 100 \\
&\qquad\qquad + 1638 \\
&= 24534638
\end{aligned}
$$

- This involved a total of 9 single-digit multiplications
  - Ok, I cheated a bit, it's really 13.
  - But still less than 16!