Four Common Algorithms

- Greatest Common Divisor (GCD)
- Subsets
- Permutations
- Combinations

Greatest Common Divisor

6 & 8 → 2

8 & 16 → 8

8 & 5 → 1

24 & 18 → 6

Examples: Reducing Fractions (12/18), Relatively Prime Numbers (8 & 15).

Naïve (obvious approach): given 650 and 550

Divide both by 550, then by 549, then by 548, …, until you succeed

Euclid's Algorithm

```
int gcd(int num1, int num2)
        int temp
        if (num1 < num2)
                swap(num1, num2)
        while (num2 > 0)
                temp = num1 % num2
                num1 = num2
                num2 = temp
        end while
        return(num1)
end gcd
```

num1	num2	temp (remainder)
650	550	
		100
550	100	
		50
100	50	
		0
50	0	

converges very quickly

Recursive (vs. Iterative)

```
int gcd(int num1, int num2)
        // we assume num1 >= num2
        if (num2 == 0)
                return(num1)
        return(gcd(num2, num1 % num2))
```

Least Common Multiple

lcm(a, b) = (a * b) / gcd(a, b) ←→ (a / gcd(a,b)) * b

Subsets

Given a set, generate all its subsets.

Examples:

Topping options for a pizza: {Pepperoni, Sausage, Mushroom}

{}, {P}, {S}, {M}, {P,S}, {P,M}, {S,M}, {P,S,M}

Topping options: {Pepperoni, Sausage, Mushroom, Onion}

{}, {P}, {S}, {M}, {O}, {P,S}, {P,M}, {P,O}, {S,M}, {S,O}, {M,O},
{P,S,M}, {P,S,O}, {P,M,O}, {S,M,O}, {P,S,M,O}

Pepperoni	Sausage	Mushroom	Subset
T	T	T	{P,S,M}
T	T	F	{P,S }
T	F	T	{P, M}
T	F	F	{P }
F	T	T	{ S,M}
F	T	F	{ S }
F	F	T	{ M}
F	F	F	{ }

Pepperoni	Sausage	Mushroom	Onion	Subset
T	T	T	T	{P,S,M,O}
T	T	T	F	...
T	T	F	...	
T	T	F		
T	F	...		
T	F			
T	F			
T	F			
F	...			
F				
F				
F				
F				
F				
F				
F				

Algorithm

At each recursive level, first generate all subsets that include the current element, then generate all subsets that exclude the current element.

Data Structures:

```
item[1..MAX_ITEM_COUNT] of datatype  /* note that we are
                                        using 1-based indexing */
in_subset[1..MAX_ITEM_COUNT] of Boolean
int count
```

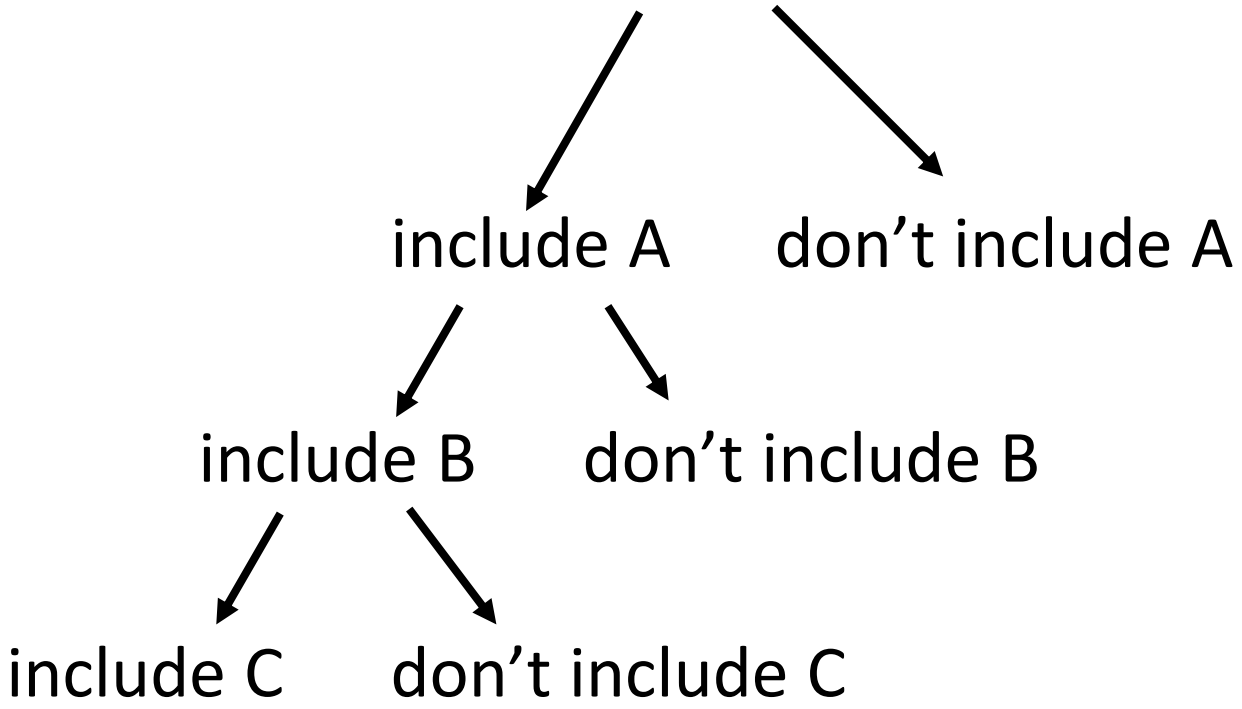
```
for k = 1 to count
        in_subset[k] = false
call subset(1)

subset(int index)
        if (index > count) then
                process subset, e.g., print it (the in_subset array tells
                you which items are in the subset)
```

```
        else
                /* put the element (item[index]) in the subset */
                in_subset[index] = true
                call subset(index + 1)
                /* don't put the element (item[index]) in the subset */
                in_subset[index] = false
                call subset(index + 1)
        end if
end subset
```

Recursion Tree

item: A,B,C



Permutations

Given a list of objects, generate all permutations (rearrangements) of the objects.

Examples:

Family of three (Father, Mother, Child) want to take a picture (all standing up); different ways of lining them up:

F M C

F C M

M F C

M C F

C F M

C M F

Family of four (Father, Mother, Son, Daughter):

```
F  M  S  D
F  M  D  S
F  S  M  D
F  S  D  M
F  D  M  S
F  D  S  M
M  ...
M
M
M
M
M
```

S ...
S
S
S
S
S
D ...
D
D
D
D
D

Algorithm

- Generate one permutation at a time.
- For a permutation, fill the positions one at a time; a position is filled by an element that is not used in any other position in the current permutation.

Data Structures:

```
item[1..MAX_ITEM_COUNT] of datatype    // input items
perm[1..MAX_ITEM_COUNT] of datatype  // permutation
used[1..MAX_ITEM_COUNT] of Boolean
int count
```

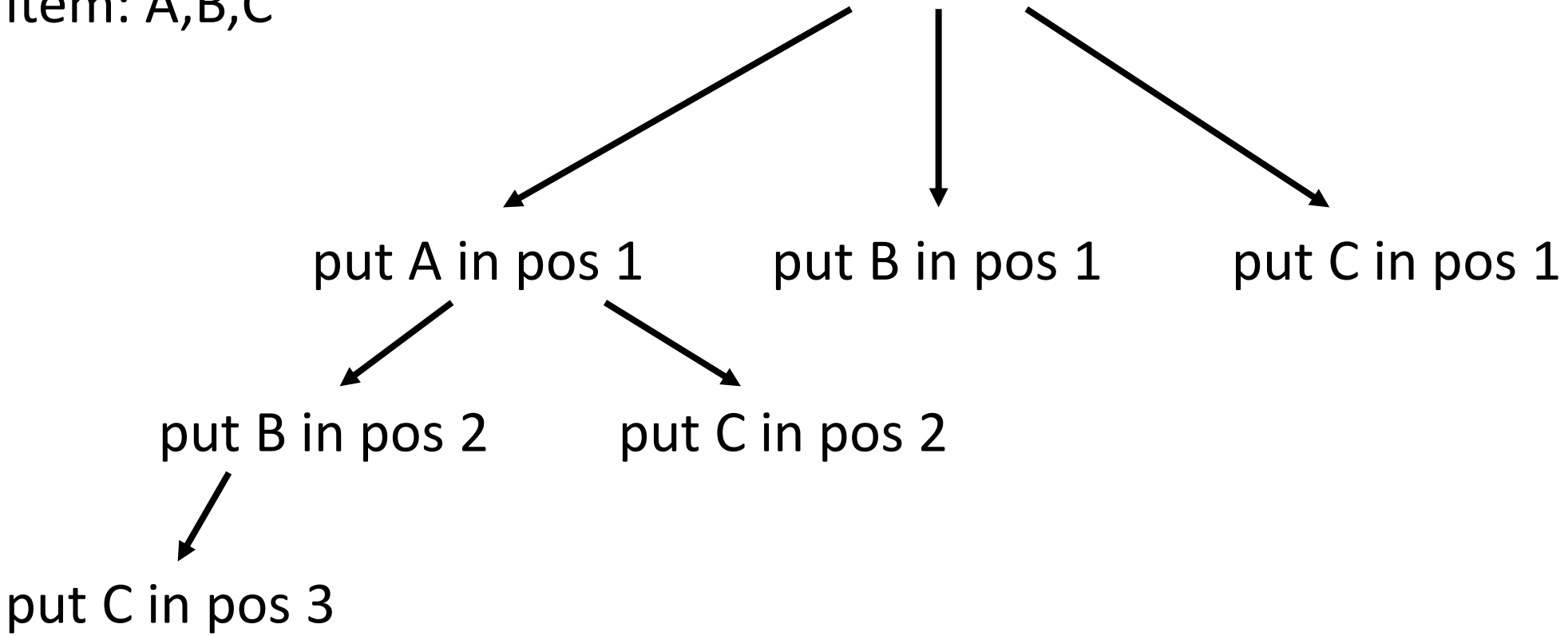
```
for k = 1 to count
        used[k] = false
call permute(1)

Permute(int position)
        int j
        if (position > count)
                process permutation, e.g., print it (the permutation is in
                the “perm” array)
```

```
        else
                /* Select an element to be used in “position” for the    */
                /* current permutation.                                  */
                /* Each element that has not been used in the current */
                /* permutation will be used for “position”.              */
                for j = 1 to count
                        if (used[j] == false) then
                                used[j] = true
                                perm[position] = item[j]
                                call permute(position + 1)
                                used[j] = false
                        end if
                end for
        end if
end permute
```

Recursion Tree

item: A,B,C



Combinations

Given n items, generate all combinations of size r; similar to subset (a given size subset).

Examples:

Let's assume there are four items (A,B,C,D) each costing $50. Let's say you have only $150, i.e., you can get only 3 of these 4:

A B C

A B D

A C D

B C D

Let’s say 5 items, take 3:

A	B	C	D	E
A	B	C		
A	B		D	
A	B			E
A		C	D	
A		C		E
A			D	E
	B	C	D	
	B	C		E
	B		D	E
		C	D	E

Algorithm

At each recursive level, first generate all combinations that include the current element, then generate all combinations that exclude the current element.

Data Structures:

```
item[1..MAX_ITEM_COUNT] of datatype  /* note that we are
                                        using 1-based indexing */
in_combin[1..MAX_ITEM_COUNT] of Boolean
int count, degree
```

```
for k = 1 to count
        in_combin[k] = false
call combination(1, 0)  /* first argument indicates ‘item_index’;
                           second argument indicates ‘#of_included’ */

combination(int index, int included)
        if (included == degree) then
                process combination, e.g., print it (the in_combin array
                tells you which items are in the combination)
```

```
        else if (index <= count) then
                /* put the element (item[index]) in the combination */
                in_combin[index] = true
                call combination(index + 1, included + 1)
                /* don't put the element (item[index]) in the combination */
                in_combin[index] = false
                call combination(index + 1, included)
        end if
end combination
```