RECURSION

Recall

Problem solving steps:

1) Identify a real world problem e.g. finding the maximum of n numbers;
   Note that real world problems are not CS problems, our first task is to convert the problem into a CS problem i.e. computers work on computational tasks so unless the problem is computational we may not get the solution. In real world the person simply needs to look at the numbers [70, 21, 40, 52, 91, 120, 30] and say that the, maximum is 120.
2) Derive a computational problem: The computer can only "compute" i.e. can only solve problems that are computational by nature, In the core of every real world problem there is a simple, basic computation that goes on to solve it these computations could be in the form of;
   - Logical: comparisons e.g. above example, we know that at one point we will have to compare something for us to determine which is greater before ultimately, we have the overall max.
   - Arithmetic: E.g. in counting from 0 to n, in the real world a young boy knows that the first will be 0, 2^{nd} 1, 3^{rd} 2, 4^{th} 3, … , nth n-1; and the boy can figure out those symbols in his mind but in computational terms we only reach n from 0 by adding 1 to the previous till we reach n:
     Therefore the counting problem is an arithmetic task i.e. starts at 0 then repetitively adds 1 till you reach n.
3) Problem Decomposition: Representing the problem into a series of steps that can otherwise be translated into a program; at this step we seek to break down our problem onto simpler, more manageable subtasks.
4) Develop Initial Solution: Would consist of a range of possible solutions probably on average; 2 to 3, that are ***correct*** by definition i.e. they give correct answers given any inputs.

5) Solution Refinement: Consideration of the set of solutions in the initial solution, then choosing one or an improved version of the solution this improved solution is reached at by considering efficiency and complexity factors of : time and space i.e. A detailed analysis of time taken by each computation within the algorithm and the memory required by data used for these computations is done and a derived solution which is most efficient: takes least time to respond and consumes less space is chosen.

6) Translation of the step 4 results into a computer program

What is a solution?

A solution consists of two components:

1) An Algorithm: A step by step specification of a method to solve a problem within ***finite*** amount of time. An algorithm could operate on a collection of data {put new data into a collection, remove data from a collection, or ask questions about a collection of data}
2) Data Structuring (Ways to store the data): You need to do much more than simply store data. Part of your solution should give considerable amount of time to considering how to store the data in such a way that the algorithm will easily be applied to the data.

When you design a solution to a given problem, there are several techniques you can employ in making your task easier.

What is a Good Solution?

The aim of studying the problem solving techniques that follow is to make you a good problem solver i.e. to equip you with tools and knowledge on how to write "good solutions" to problems so what exactly is a good solution?

Since a computer program is the final form of your solution, the issue here is to consider what constitutes a good compute program. Programs are written to perform some task in the course of which some **cost** is incurred in the form of what includes:

- Computing time : the time the processor will take to finally come up with your solution
- Memory: Amount of space required for the running of your solution.
- Difficulties encountered by users of the program
- Consequences of a program that does not behave correctly.

This factors are costs incurred during operation but a good solution should encompass a consideration of costs for all phases of software development (Development, refinement, coding, debugging, and testing; and also include the cost of maintaining, modifying and expanding)

Deff: ***A good solution is that solution such that the total cost it incurs over all phases of its life is minimal.***

Efficiency: When there are several approaches to solving a problem then, we need to choose the best… to do this we consider the efficiency of each, if there is considerable difference in efficiency then we choose the most efficient approach. In situations when the efficiencies of the different approaches are close, then we consider other factors.

The stages of the problem solving process at which you should be concerned about efficiency are:

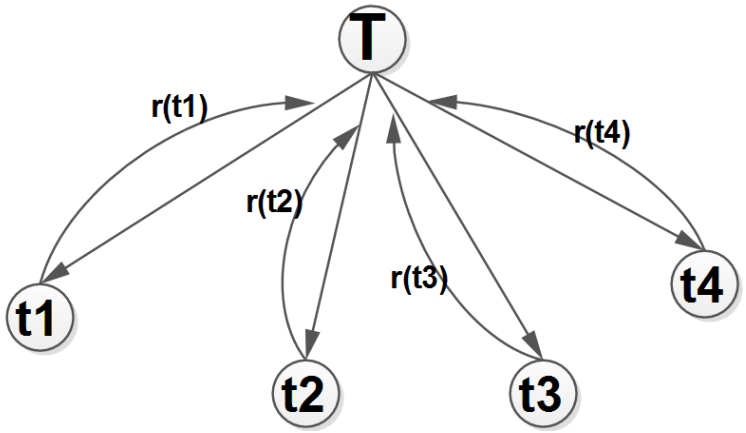
- The choice of a solution's component.

- The algorithms and ways to store data.

Rather than the code you write.

Problem Decomposition

Constitutes of a problem solving approach by which we find a solution by combining the results of solving smaller subtasks i.e given a task T, the solution, **sln(T)**, can be found by:



$$\mathbf{sln(T)} = \{r_{t1}, r_{t2}, r_{t3}, \ldots, r_{tn}\}$$

Therefore the final solution is attained by the recomposition of the smaller subtasks. Problem decomposition constitutes a major step in ***programming methodology***, also refered to as ***algorithm design*** used to generate an initial solution.

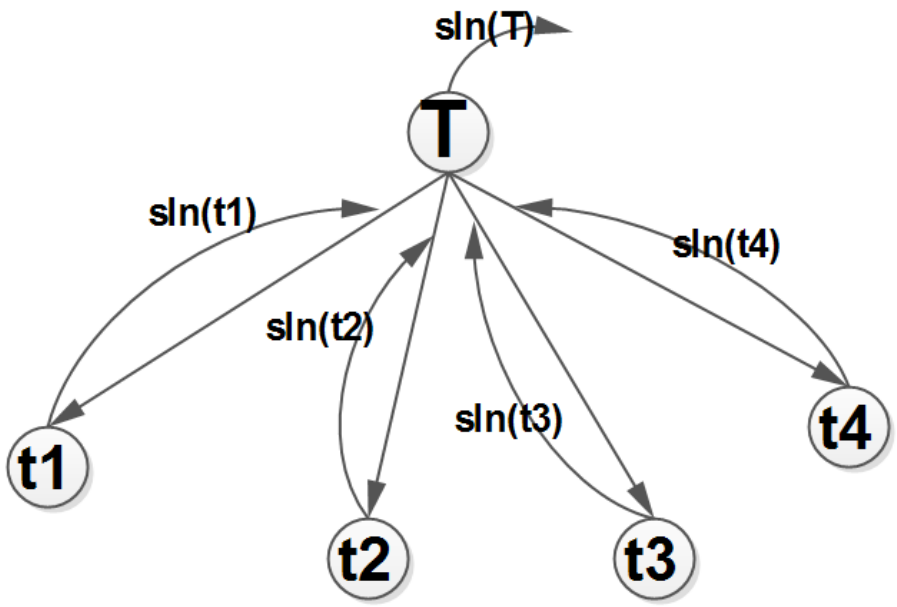
The next step after problem decomposition is refinement of the solution which includes choise of data structures.

RECURSIVE APPROACH TO PROBLEM DECOMPOSITION

A special form of problem decomposition for which the subtasks are identical to the initial task. Recursion can be defined as an approach to problem solving where the solution to a task is obtained by solving identical subtasks of the same problem.

Given the above Task T, the solution ; **sln(T)**, can be found by combining the solutions obtained by solving subtasks of T in a similar way we would have solved T and combining the results.

$$\mathbf{sln(T)} = \{\mathbf{sln}(t1), \mathbf{sln}(t2), \mathbf{sln}(t3), \ldots, \mathbf{sln}(t4)\}$$



Like top-down design; recursion breaks a problem into several smaller problems, the striking thing about recursion is that these smaller problems are exactly the same type (nature) as the original problem - Think about mirror images.

The last subtask of decomposition is the case for which the solution is small enough to be trivial and hence is automatically obtained, this is then used to solve the preceding subtask (it's parent) and so on until we get back to our original task.

Parts of a Recursive solution

1. **Base Case / Basis/ Degenerate Case :** The part of the solution that is obvious, The subtask of our task that is too trivial to an extent that it's solution is obtained by simply picking. The base case are mainly formed by facts about the problem, or sections of the problem definition that explain a solution to it's smallest instance. This follows therefore that the base case is a special case whose solution is known before hand.
2. **Iterative / Inductive Case (Part)**
   The part of the solution that repeatedly breaks down the task (decomposes) into smaller identicall subtask and recomposes the solutions at each step.

Questions to answer When Constructing a Recursive Function

1. How can you define the problem in terms of smaller tasks of same type?
2. How does each Recursive call diminish the size of the problem?
3. What instance of the problem can serve as the base case?
4. As the problem size diminishes; will you reach this base case?

Examples

1. Factorial

The factorial of a number n is given by:

$$fact(n) = \begin{cases} n = 0; & 1 \\ n > 0; & n * fact(n-1) \end{cases}$$

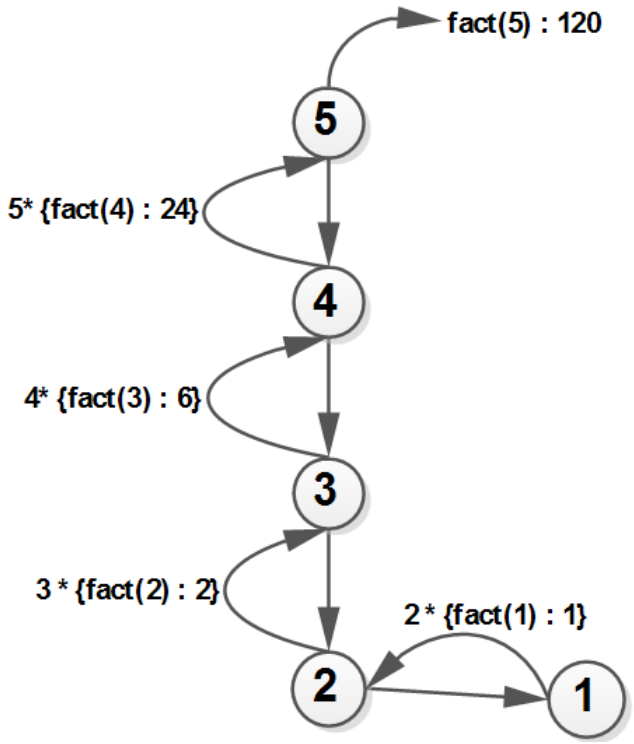
Base case:

- When n=0 then it's trivial i.e. there is no computation required, the result is definitely 1.
- Given this situation, we can include 1 in the base case because a recursive call to 0 from 1 will definitely lead to 1*1 computation which is a waste of time; hence we can eliminate this extra computation by including n=1 in the base case. But remember that n=0 is the proper basis.

Inductive Case:

The case for which n>0 must be further broken down to sub case since factorial of a number is found by multiplying that number by the factorial of its predecessor. This break down is iterative until the current predecessor is the trivial case; then recomposition begins. For factorial the combination for recomposition is achieved by a product of the factorial of predecessor and the current number.

$$fact(n) \leftarrow n * fact(n-1)$$



Algorithm fact(n) : get the factorial of number n

1. fact (n)
2. if n=0 or n=1
   return 1
3. else
   return n * fact(n-1)

Some Code

```
int fact(int n){
  if((n==0) || (n==1))
        return 1;
  else
        return n * fact(n-1);
}
```

2. Fibonacci Numbers

Fibonacci numbers are thus defined: {1,1,2,3,5,8,13,21,34,…..}

The series begin with two numbers at the first two positions {1 & 1} then the rest of the numbers is the series are computed by finding the sum of the two previous numbers in the series i.e. the 1st previous number and the 2nd previous number. We add a zero (0) at the beginning of the series to complete it, hence the zero, being not "really" part of the series is taken to be at position zero. {0,1,1,2,3,5,8,13,21,….}

The series can therefore be defined as;

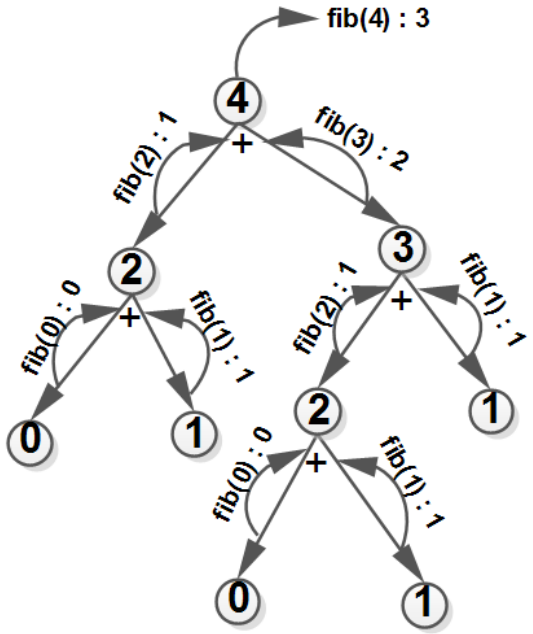
$$fib(n \mid n\text{: pisition of number in series}) = \begin{cases} n = 0; & 0 \\ n = 1; & 1 \\ n > 0; & fib(n-1) + fib(n-2) \end{cases}$$

Base case:

- When n=0 then it's trivial i.e. there is no computation required, the result is definitely 0
- Also when n=1 then we definitely give 1 as the result.

Inductive Case:

The case for which n>0 must be further broken down to sub case since fibonacci of a number is found by adding the Fibonacci number at position (n-1) to the Fibonacci number at position (n-2). This break down is iterative until (n-1 gives 0 or 1) or (n-2 gives 0 or 1) whereby these are the trivial case; then recomposition begins. For Fibonacci, the combination for recomposition is achieved by a sum of the two consecutive predecessors.



Algorithm fib(n) : gets fibonacci number at pos n

1. fib (n)
2. if n=0
   return 0
3. else if n=1
   return 1
4. else
   return fib(n-1) + fib(n-2)

Some Code

```
int fib(int n){
  if((n==0)  return 0;
  else if(n==1) return 1;
  else
        return fib(n-1) + fib(n-2);
}
```

EXERCISES FOR THE STUDENTS

The following tasks will be given in class as exercises:

i) The Task of obtaining the K-smallest element in an array.

ii) The tax of obtaining the Maximum or the sum of N integers.

iii) The task of searching for a key in a sorted list

iv) The task of solving the **Towers of Hanoi** (Both the Algorithm definition and the Code for it)

NOTES ON RECURSION

1. General approach:

1. Split a problem into smaller similar, sub problems.
2. Repeat until the sub problems are trivial.
    - "Trivial" means solved with no computation.
3. Recombine sub problem solutions into a solution for the originating (sub) problem.

This approach is called recursion.

1.1 Recursion Format

- The trivial sub computation in step 2 is called the base case.
- The recursive steps 1 and 3
    - Simplifies a problem into smaller similar sub problems.
    - Recursively applies the algorithm to the sub problems. **{Decomposition}**
    - Combines the sub problem solutions into a more complete solution. **{Recomposition}**

1.2 Recursion Advantages

The advantages of exploiting recursion include:

- **Simplicity**: the trivial problem, dividing problems and recombining solutions.
- **Wide applicability**: recursion can be discovered in surprising places.
- **Transparency**: correct recursive algorithms are correct (almost) by inspection.
- **Cliched**: the same damn thing over and over.

1.3 Recursion Disadvantages

Problems with recursion include

- Computational expense. (Easily and (usually) automatically transformed into more efficient forms).
- Opacity, particularly among the uninitiated. (Oh well)
- Subtle design errors. (Few hiding places).

2. Finding Recursion

Recursion relates directly to the problem's structure.
There are three general recursion sources:

1. The problem or data structure; {**structural recursion}**.
2. Linear structures; **{inductive recursion}**, a special case of structural recursion.
3. optimized (usually) semi-recursive algorithms; **{degenerate recursion}**.

2.1. Structural Recursion

1. Structural recursion is (almost) a gimme; it follows from the problem or data structure.
    - Although the problem or data structure may not be the best one to use.
2. Examples:
    - The max sum path problem: a triangle is an apex with left and right sub-triangles.
    - Trees: a tree is a node with a set of trees as children.

Recursive Data Structures

1. A recursive data structure *D* is defined by
    - A base case giving the simplest possible instance of *D*.
    - A constructive case that combines one or more *D* instances and other data into a new, single *D* instance.
2. A recursive algorithm undoes the structure created by the constructive case.
    - And stops recursing at the base case.

Example

- An *n*-array tree of type *T* is
    - Nothing (the base case).
    - A value of type *T* and up to *n* *n*-ary trees of type *T*.

A recursive algorithm:

```
bool find(v, root)
  if root == nil
    return false
  if root.value == v
    return true
  return find(v, root[0]) || ...
        find(v, root[n-1])
```

2.2. Inductive Recursion

Inductive Recursion is structural recursion over linear structures.

- Arrays, sequences, and the like.

The subdivision is usually (but not always) into a single element and the rest of the structure.
With care, inductive recursion can be automatically transformed into efficient looping code (tail-call optimization).

Example

- Find an element *x* in an array.
- Base case: finding an element in an empty array is trivial.
- Recursive case:
    - Splint an *n*-element array into a 1-element array and an array with *n* - 1 elements.
    - If *x* is in the 1-element array, return true.
    - Otherwise, return the result of recursing on the array with *n* - 1 elements.

Example Code

```
bool find(x, a[])
  if a.size == 0
    return false
  else
    if a[0] == x
      return true;
    else
      return find(x, a[1..a.size()])  {remove the first element from the array because it's already compared}
```

2.3. Degenerate Recursion

Sometimes extra information can be used to modify the recursive structure to produce a better design.

- But possibly not producing something following the typical recursive pattern.

- Example: searching for *x* in an unordered array vs. an ordered array.

Unordered Array Searching

No extra knowledge about an unordered array; use straight inductive recursion.

```
bool find(x, a[])
  if a.size == 0
    return false
  n = a.size/2
  if a[n] == x
    return true
  return find(x, a[0..n - 1]) ||
         find(x, a[n + 1..a.size])
```

Ordered Array Searching

The extra ordering information helps eliminate the sub problems.

```
bool find(x, a[])
  if a.size == 0
    return false
  n = a.size/2
  if a[n] == x
    return true
  if a[n] < x
    return find(x, a[n + 1..a.size])
  else
    return find(x, a[0..n - 1])
```

3. Recursion Requirements

To use recursion successfully, should have

- A base case (a trivial sub problem).
- A way to divide the problem into sub problems.
- A way to combine sub problem solutions.

4. Recursive Design

Make sure the base case is

- Really a base case (that is, a trivial instance of the problem). **{Mutative}**
- Is correctly solved.

Make sure the recursive step

- Creates sub problems related into the original problem. **{identical subtasks}**
- Combines sub problem solutions into a solution for the larger problem. **{Recomposition}**

5. Recursion Pitfalls

Although it's simple, it's possible to get recursion wrong.

- The base case is wrong, incomplete, or incorrectly solved.
- The sub problems are incorrectly derived from the original problem.
- Sub problem solutions are incorrectly combined into an invalid solution for the original problem.

5.1 Wrong Base Cases

The base case for factorial is 0, not 1.

```
int factorial(n)
  if n == 1
    return 1
  return n*factorial(n - 1)
```

Right base case

```
int factorial(n)
  if n == 0
    return 1
  return n*factorial(n - 1)
```

5.2 Incomplete Base Cases

How many 3- and 5-cent stamps equal any amount over 7 cents?

- For example, 11 cents equals one 5-cent and two 3-cent stamps.

The recursion seems easy:

- The base case is 8 = 3 + 5.
- For any value $n > 7$, $n = n' + 3$.
- One more 3-cent stamp than needed for n'.

Stamp Counting Code

That seems easy enough; here's the code.

```
int-pair stamps(n)
  if n == 8
    return (1, 1)
  return (1, 0) + stamps(n - 3)
```

Unfortunately, this code is wrong. Why?

Revised Code

There are three base cases:

- 8 = one 3¢ stamp + one 5¢ stamp.
- 9 = three 3¢ stamps + no 5¢ stamps.
- 10 = no 3¢ stamps + two 5¢ stamps.

```
int-pair stamps(n)
  if n == 8
    return (1, 1)
  if n == 9
    return (3, 0)
  if n == 10
    return (0, 2)
  return (1, 0) + stamps(n - 3)
```

5.3 Bad Subdivisions

The sub problems must be strictly smaller than the original problem.

- If not, the recursion won't terminate.

The sub problems must be really related to original problem.

- Otherwise the sub problem solutions won't be related to the original problem's solution.