RECURSION

By: Tamal Chakraborty

Recursive Functions

- A recursive function is one that manes a call to itself.
- For example:
  - We can define factorial of a positive integer n, as
  - Factorial(n) = 1, when n = 0, 1
  - Factorial(n) = n * Factorial(n – 1), when n > 1
- Notice that we are calculating Factorial(n) in terms of Factorial(n – 1).
- In other words, to find Factorial(5) we need to find Factorial(4) and multiply the result by 5.
- To find Factorial(4), we need to find Factorial(3) and multiply the result by 4, and so on.

Recursion Implementation

- A recursive factorial function loons line this:

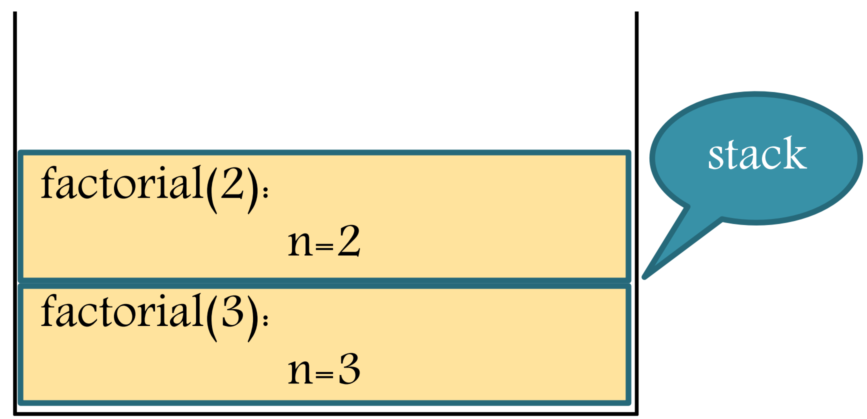
```
unsigned int factorial(unsigned int n) {
  if (n <= 1) return 1;
  else return n * factorial(n – 1);
}
```

- Recursive functions calls are not executed immediately.
- They are placed on a stack until the condition that terminates recursion is encountered.
- The function calls are executed in reverse order, as they are popped off the stack.

Example: factorial(3)

- Let us consider the invocation of factorial(3).
- To compute 3 * factorial(2) in the else statement, the computation of factorial(3) is suspended and factorial is invoked with n = 2.
- When computation of factorial(3) is suspended then the program state (i.e. local variables, program counter etc.) are pushed on recursion stack.
- Similarly invocation of factorial(2) is suspended, factorial is invoked with n = 1.
- factorial(1) returns 1, program state of factorial(2) is then popped off the stack to compute factorial(2).

```
unsigned int factorial(unsigned int n) {
        if (n <= 1) return 1;
        else return n * factorial(n – 1);
}
```



factorial(1) = 1

factorial(2) = 2 * factorial(1) = 2 * 1 = 2

factorial(3) = 3 * factorial(2) = 3 * 2 = 6

Recursion vs. Iteration

Recursion

```
unsigned int factorial(unsigned int n) {
        if (n <= 1) return 1;
        else return n * factorial(n – 1);
}
```

- Usually slower, due to overhead of stack manipulation and function calls.
- Has the risk of stack overflow, for too many function calls.
- Some problems can be more easily solved by recursion.

Iteration

```
unsigned int factorial(unsigned int n) {
        unsigned int result = 1;
        while (n > 1) {
                result *= n--;
        }
        return result;
}
```

- Runs faster as assignments are usually less costly than function calls.

Tail Recursion

- **tail recursion** (or **tail-end recursion**) is a special case of recursion in which the last operation of the function, i.e. the tail call, is the recursive call.
- In other words there are no more operations after the function calls itself.
- Such recursions can be easily transformed to iterations.
- Replacing recursion with iteration, can drastically decrease the amount of stack space used and improve program efficiency.

Tail Recursion: Example

```
unsigned int factorial(unsigned int n) {
        return tailFactorial(1, n);
}

unsigned int tailFactorial(unsigned int result, unsigned int n) {
        if (n <= 1) return result;
        else return tailFactorial(n * result, n - 1);
}
```

Tail Recursion & Iteration

Tail Recursion

```
unsigned int factorial(unsigned int n) {
		return tailFactorial(1, n);
}

unsigned int tailFactorial(unsigned int result, unsigned int n) {
		if (n <= 1) return result;
		else return tailFactorial(n * result, n – 1);
}
```

Let us trace the execution of this program to calculate factorial(3):

factorial(3)

*tailFactorial(**result = 1, n = 3**)*
*tailFactorial(**result = 3, n = 2**)*
*tailFactorial(**result = 6, n = 1**)*
return 6

Iteration

```
unsigned int factorial(unsigned int n) {
		unsigned int result = 1;
		while (n > 1) {
				result *= n--;
		}
		return result;
}
```

Look at how the variables change with each iteration while calculating factorial(3):

factorial(3)
result = 1, n = 3
result = 3, n = 2
result = 6, n = 1
return 6

Tail Recursion & Iteration

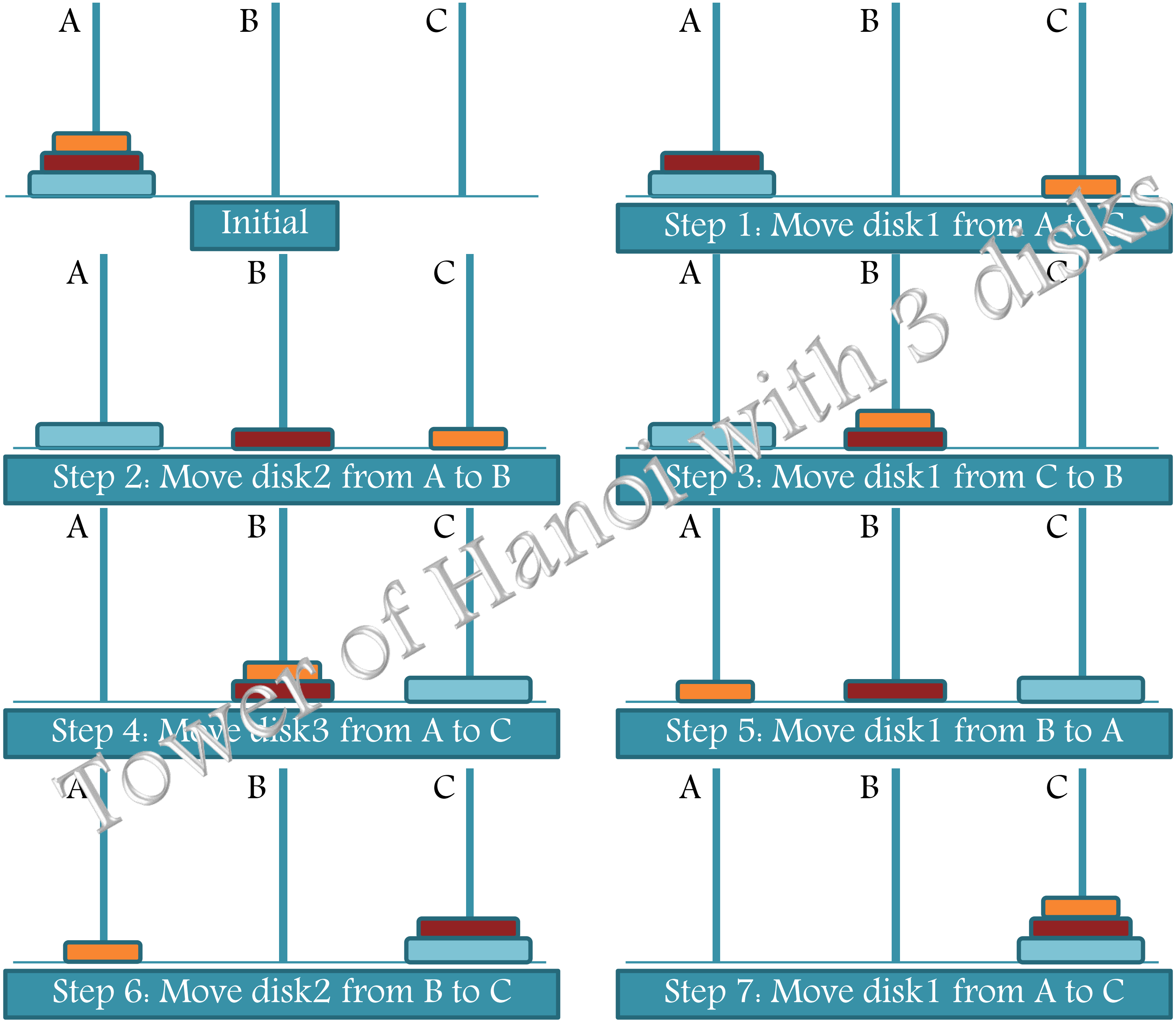
- As we can see the variables result and n are identical for iterative and tail-recursive methods.
- Hence tail recursion and iteration are equivalent.
- Thus a recursive algorithm can be converted to a tail recursive algorithm.
- A tail recursive algorithm can in-turn be converted to a iterative algorithm.

Tower of Hanoi

- It is a mathematical game, which consists of three towers, and a number of disks of different sizes which can slide onto any tower. The puzzle starts with the disks in a neat stack in ascending order of size on one tower, the smallest at the top, thus making a conical shape.
- The objective of the puzzle is to move the entire stack to another tower, 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 towers and sliding it onto another tower, on top of the other disks that may already be present on that tower.
  - No disk may be placed on top of a smaller disk.

Recursive Solution

- A key to solving this puzzle is to recognize that it can be solved by breaking the problem down into a collection of smaller problems and further breaking those problems down into even smaller problems until a solution is reached. The following procedure demonstrates this approach.
  1. label the towers A, B, C—these labels may move at different steps
  2. let *n* be the total number of disks
  3. number the disks from 1 (smallest, topmost) to *n* (largest, bottommost)
  4. To move n disks from tower A to tower C:
  5. move n−1 disks from A to B. This leaves disc #n alone on tower A
  6. move disk #n from A to C
  7. move n−1 disks from B to C so they sit on disk #n
- The above is a recursive algorithm.


Tower of Hanoi with 3 disks
A
B
C
Initial
Step 1: Move disk1 from A to C
Step 2: Move disk2 from A to B
Step 3: Move disk1 from C to B
Step 4: Move disk3 from A to C
Step 5: Move disk1 from B to A
Step 6: Move disk2 from B to C
Step 7: Move disk1 from A to C

Recursive Algorithm

```
void dohanoi(int N, int from, int to, int via)
{
    if (N > 0)
    {
        dohanoi(N-1, from, via, to);
        printf ("move %d --> %d\n", from, to);
        dohanoi(N-1, via, to, from);
    }
}
```

Recurrence Relations

- Let $T(n)$ be the number of moves needed to solve the puzzle with n disks.
- The recursive solution involves moving $n - 1$ disks from one tower to another twice, making one additional move in between.
- Thus it follows that:
- $T(n) = T(n - 1) + 1 + T(n - 1) = 2T(n - 1) + 1$
- Intuitively, $T(1) = 1$
- The equation above is called a **recurrence relation**.

n^{th} order linear recurrence relations

- Consider the following equation:

  $S(k) = c_1 S(k-1) + c_2 S(k-2) + \ldots + c_n S(k-n) + f(n)$

  Where $c_1, c_2, \ldots, c_n$ are numbers and f is a numeric function.

- Such an equation is called an n^{th} order linear recurrence relation, if $c_n \neq 0$
- For example:
  - $F_i - F_{i-1} - F_{i-2} = 0$ … order 2
  - $P(j) + 2P(j-3) = j^2$ … order 3
  - $a(n) = 2(a(n-1) + n))$ … order 1

Homogeneous recurrence relations

- Consider the following equation:

  $S(n) = c_1S(n-1) + c_2S(n-2) + \ldots + c_nS(n-n) + f(n)$
- If $f(n) = 0$, for all n, then this equation is called a homogeneous recurrence relation.
- For example, say:

  $S(k) - 7S(k-1) + 12S(k-2) = 0, S(0) = S(1) = 4$ *... (i)*

  Let $S(k) = b.a^k$ *be the solution, where a, b are non-zero constants*

  Then $S(k-1) = b.a^{k-1}, S(k-2) = b.a^{k-2}$

  Substituting in $(i), b.a^k$ *-7* $b.a^{k-1} + 12b.a^{k-2} = 0$

  Dividing by $b.a^{k-2}, a^2 - 7a + 12 = 0$ *... (ii)*

  This equation is called the characteristic equation of the recurrence.

  Solving (ii) yields, $(a - 3)(a - 4) = 0$

  Thus the general solution is : $S(k) = b_1.3^k + b_2.4^k$

  Using the initial conditions, $S(0) = S(1) = 4$*, we get* $b_1 = 12, b_2 = -8$

  Thus, $S(k) = 12.3^k - 8.4^k$

Algorithm for solving homogeneous recurrences

1. Write the characteristic equation of the recurrence Consider the following equation:

   $S(k) + c_1 S(k-1) + c_2 S(k-2) + \ldots + c_n S(k-n) = 0$

   Which is: $a^n + c_1 a^{n-1} + \ldots + c_n = 0$

2. Find all the roots of the characteristic equation.

3. If there are n characteristic roots $a_1, \ldots, a_n$ then general solution is

   $S(n) = b_1.a_1^k + b_2.a_2^k + \ldots + b_n.a_n^k$

Algorithm for solving homogeneous recurrences

4. If there are fewer than n characteristic roots then at least one root is a multiple root. If a_j is a double root then $b_j a_j^k$ is replaced by $(b_{j0} + b_{j1}k).a_j^k$
5. In general, if a_j is a root of multiplicity p, then $b_j a_j^n$ is replaced by $(b_{j0} + b_{j1}n + \ldots + b_{j(p-1)}.k^{p-1}).a_j^k$
6. If n initial conditions are given then form n linear equations and solve.

Solving homogeneous recurrences

$\mathrm{f}(n) = f(n-1) + 8f(n-2) - 12f(n-3) \ \dots (i)$
$initial\ conditions\colon\ f(0) = 0, f(1) = 1, f(2) = 3$

We have $f(n) - f(n-1) - 8f(n-2) + 12f(n-3) = 0$
The characteristic equation of this recurrence relation is:
$a^3 - a^2 - 8a + 12 = 0 \dots (ii)$

Solving (ii) we get, $(a-2)^2.(a+3) = 0$
Thus the characteristic roots are: $a = 2$, $a = 2$, $a = -3 \dots (iii)$

Since $a = 2$ *is a characteristic root with multiplicity 2, we write*
$f(n) = (b_o + b_1.n).2^n + b_2(-3)^n$
Applying the initial conditions, we can get the values of b_0, b_1 & b_2

Non-homogeneous recurrence relations

- Consider the following equation:

$$S(k) + c_1S(k-1) + c_2S(k-2) + \ldots + c_nS(k-n) = f(n)$$

- If $f(n) \neq 0$, for all n, then this equation is called a non-homogeneous recurrence relation.
- For example, $T(n) = 2T(n-1) + 1$

Algorithm for solving non-homogeneous recurrences

1. Write the associated homogeneous equation by putting the RHS, i.e. $f(n) = 0$, solve this using previous algorithm. Call this the homogeneous solution.
2. Obtain the particular solution by taking a guess by the form of RHS, i.e. $f(n)$.

f(n)	Particular Solution
Constant q	Constant d
Linear equation $q_0 + q_1k$	$d_0 + d_1k$
$q.a^k$	$d.a^k$

Algorithm for solving non-homogeneous recurrences

4. If RHS involves an exponential with base a, where a is a characteristic root with multiplicity p, then multiply particular solution by k^p, where k is the index of recurrence.
5. Substitute your guess into the recurrence relation.
6. Sum up the homogeneous solution and the particular solution to get the general solution.
7. Use initial conditions to evaluate constants.

Solving non-homogeneous recurrences

$f(k) + 5f(k-1) = 9$ … (i), *initial condition f(0) = 6*

Now, the homogeneous characteristic equation is: $a + 5 = 0$*, So,* $a = -5$
Thus the homogeneous solution is: $f(k) = b(-5)^k$ … *(ii)*

Since $f(n) = 9$*, we guess that the particular solution is d.*
Substituting in (i), $d + 5d = 9$*, i.e.* $d = 3/2$ … *(iii)*

Using (ii) + (iii) as the general solution we get
$f(k) = b(-5)^k + 3/2$
By initial conditions: $6 = b + 3/2$*, i.e.* $b = 9/2$

Hence, $f(k) = 9/2(-5)^k + 3/2$

Solving non-homogeneous recurrences

$f(k) - 9f(k-1) + 20f(k-2) = 2.5^k$ ... (i), *initial condition f(0) = 1, f(1) = 60*

Now, the homogeneous characteristic equation is:

$a^2 - 9a + 20 = 0$, i.e. characteristic roots are $a = 4, 5$

Thus the homogeneous solution is: $f(k) = b_0(4)^k + b_1(5)^k$ ... (ii)

Since $f(n) = 2.5^k$, we guess that the particular solution is $d.5^k$

But since 5 is a characteristic root with multiplicity 1, we multiply the particular solution by n, thus getting $dk.5^k$

Substituting in (i), $dk.5^k - 9d(k-1).5^{k-1} + 20d(k-2).5^{k-2} = 2.5^k$

So, $d = 10$, and particular solution is $10k.5^k$ ... (iii)

Using (ii) + (iii) as the general solution, we get

$f(k) = b_0(4)^k + b_1(5)^k + 10k.5^k$

Apply initial conditions to solve for b_0 & b_1

Solving the Tower-of-Hanoi recurrence relation

$\mathrm{T}(n) = 2T(n-1) + 1 \; \ldots (i)$, *initial condition T(1) = 1*

Now, the homogeneous characteristic equation is:
a – 2 = 0, i.e. a = 2
Thus the homogeneous solution is: $T(n) = b(2)^n \; \ldots (ii)$

Since RHS = 1, we guess that the particular solution is d.
Substituting in (i), d - 2d = 1, i.e. d = -1 … (iii)

Using (ii) + (iii) as the general solution, we get
$T(n) = b(2)^n - 1$
By initial conditions: b=1

Hence, $T(n) = 2^n - 1$
Remember that we needed 7 steps for 3 disks, which matches with our solution since $2^3 - 1 = 7$.

Recursion: Binary Search

```
bool bsearch(int a[ ], int first, int last, int key)
{
        if (key < a[first] || key > a[last]) return false;  // not found
        int mid = (first + last) / 2;
        if (a[mid] > key) return bsearch(a, first, mid - 1, key);
        else if (a[mid] < key) return bsearch(a, mid + 1, last, key);
        else return true;  // a[mid] == key
}
```

How to find the running time of such an algorithm?

Recursion: Binary Search

```
bool bsearch(int a[ ], int first, int last, int key)
{
        if (key < a[first] || key > a[last]) return false;  // not found
        int mid = (first + last) / 2;
        if (a[mid] > key) return bsearch(a, first, mid - 1, key);
        else if (a[mid] < key) return bsearch(a, mid + 1, last, key);
        else return true;  // a[mid] == key
}
```

- Let $T(n)$ be the time taken for input size n
- At each stage the algorithm divides the list of numbers in 2 halves
- Then it tries to find the key in the half it is likely to be present, by using binary search on that half
- $T(n/2)$ would be the time taken for any of the halves
- Finding the value of the mid (first + last) / 2 would take $O(1)$ time
- Hence, $T(n) = T(n/2) + 1$

Recursion

- We observed that the running time of the binary search algorithm can be expressed by the recurrence relation:

$$T(n) = T(n/2) + 1$$

- A more generic form of this equation is:

$$T(n) = a.T(n/b) + f(n), \qquad a>=1, b>1$$

- This form of recurrence relation is observed in many recursive algorithms

Recursion

- $T(n) = a.T(n/b) + f(n)$, $a>=1, b>1$
- In the analysis of a recursive algorithm, the constants and function take the following significance:
  - n = size of the problem
  - a = number of sub-problems in the recursion
  - n/b = size of each sub-problem
  - f(n) = cost of work done outside recursive calls

Observe the recurrence relation of binary search:

$T(n) = T(n/2) + 1$

Recursion: Master Theorem

- $T(n) = a.T(n/b) + f(n)$, $a>=1, b>1$
- T(n) can be bounded as follows:

1

- If $f(n) = O(n^{\log_b(a) - \varepsilon}), \varepsilon > 0$
- Then, $T(n) = \Theta(n^{\log_b(a)})$

2

- If $f(n) = \Theta(n^{\log_b(a)})$
- Then, $T(n) = \Theta(n^{\log_b(a)}.\lg(n))$

3

- If $f(n) = \Omega(n^{\log_b(a) + \varepsilon}), \varepsilon>0$
- And $a.f(n/b) <= c.f(n), c<1$
- Then, $T(n) = \Theta(f(n))$

Recursion: $T(n) = a.T(n/b) + f(n)$

- In each of the cases we are comparing $f(n)$ with $n^{\log_b(a)}$

1

- $f(n) = O(n^{\log_b(a) - \varepsilon})$, $\varepsilon>0$

- $f(n)$ is polynomially smaller than $n^{\log_b(a)}$ by a factor $\varepsilon>0$
- Then, $T(n) = \Theta(n^{\log_b(a)})$

Recursion: $T(n) = a.T(n/b) + f(n)$

- In each of the cases we are comparing $f(n)$ with $n^{\log_b(a)}$

3

- If $f(n) = \Omega(n^{\log_b(a) + \varepsilon})$, $\varepsilon>0$ and $a.f(n/b) <= c.f(n)$, $c<1$

- $f(n)$ is polynomially larger than $n^{\log_b(a)}$ by a factor $\varepsilon>0$
- Then, $T(n) = \Theta(f(n))$

Recursion: $T(n) = a.T(n/b) + f(n)$

- In each of the cases we are comparing $f(n)$ with $n^{\log_b(a)}$

2

- If $f(n) = \Theta(n^{\log_b(a)})$

- $f(n)$ and $n^{\log_b(a)}$ are of the same size,
- Then, $T(n) = \Theta(n^{\log_b(a)}.\lg(n))$

[multiply by a logarithmic factor lg(n)]

Recursion: Applying Master Method

$T(n) = 8T(n/2) + 100n^2$

We have $a = 8, b = 2$

$\log_b(a) = 3$

$n^{\log_b(a)} = n^3$

$f(n) = 100n^2 = O(n^2) = O(n^{3-1})$

Hence, $f(n) = O(n^{\log_b(a) - \varepsilon})$, $\varepsilon=1$

By, case 1 of Master Method

$T(n) = \Theta(n^3)$

Recursion: Applying Master Method

$T(n) = 2T(n/2) + 10n$

$We\ have\ a = 2, b = 2$

$log_b(a) = 1$

$n^{log_b(a)} = n^1$

$f(n) = 10n = \Theta(n)$

$Hence,\ by\ case\ 2\ of\ Master\ Method$

$T(n) = \Theta(n.lg(n))$

Recursion: Applying Master Method

$T(n) = 2T(n/2) + n^2$

We have $a = 2$, $b = 2$

$\log_b(a) = 1$

$n^{\log_b(a)} = n^1$

$f(n) = n^2 = n^{1+1}$

So, $f(n) = \Omega(n^{\log_b(a) + \varepsilon})$, $\varepsilon=1$

$a.f(n/b) = 2f(n/2) \quad = 2.\,(n^2 / 4)$

$\qquad\qquad\qquad\qquad = n^2 / 2 <= c.f(n)$, $c=1/2$

Hence, by case 3 of Master Method

$T(n) = \Theta(n^2)$

Recursion: Applying Master Method

- $T(n) = 2T(\sqrt{n}) + 1$

Can we solve this recurrence by applying the master method?

Solution: Re-arrange the variables

Recursion: Applying Master Method

$T(n) = 2T(\sqrt{n}) + 1$

Let $m = lg(n)$, i.e. $n = 2^m$

$T(2^m) = 2T(2^{m/2}) + 1$

Let $T(2^m) = S(m)$

Then we can re-write the recurrence as:

$S(m) = 2S(m/2) + 1$

$m^{\log_b(a)} = m$

$f(m) = 1 = O(m^{1-1})$

Hence, $f(m) = O(m^{\log_b(a) - \varepsilon})$, $\varepsilon=1$

By, case 1 of Master Method

$S(m) = \Theta(m)$

Therefore, $T(n) = \Theta(lg(n))$

Recursion: Applying Master Method

- For the binary search example:
- We had, $T(n) = T(n/2) + 1$
- Let us try to solve it using the master method

In this case: $a = 1, b = 2$

$\log_b(a) = 0$

$n^{\log_b(a)} = n^0 = 1$

$f(n) = 1$

$\quad = \Theta(n^{\log_b(a)})$

Hence, by case 2 of Master Method

$T(n) = \Theta(1.\lg(n)) = \Theta(\lg(n))$

Recursion: $T(n) = a.T(n/b) + f(n)$

- The Master Method is not applicable if
- $f(n)$ is smaller than $n^{\log_b(a)}$ but not polynomially smaller
- Example: $T(n) = 2T(n/2) + n/\log(n)$
- $f(n)$ is larger than $n^{\log_b(a)}$ but not polynomially larger
- Example: $T(n) = 2T(n/2) + n*\log(n)$

THANK YOU!