flowersayo.log

Recursion vs Iteration

통계 수정 삭제

flowersayo · 방금 전

자료구조

자료구조

▾ 목록 보기 2/2

Recursion

: Method that solves the problem by calling the algorithms (or function) back. A suitable method for circular definition.

Factorial computation

$$n! = \begin{cases} 1 & n = 0 \\ n*(n-1)! & n \geq 1 \end{cases}$$

Fibonacci series

$$fib(n) = \begin{cases} 0 & if \quad n = 0 \\ 1 & if \quad n = 1 \\ fib(n-2) + fib(n-1) & otherwise \end{cases}$$

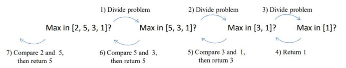
Binomial coefficient

$$_nC_k = \begin{cases} 1 & n = 0 \quad or \quad n = k \\ _{n-1}C_{k-1} + _{n-1}C_k & otherwise \end{cases}$$

Recursion principle

> Divide-and-conquer

: Divide the problem into a set of sub problems. recursively **breaks down a problem** into two or more sub-problems of the same or related type. until these become simple **enough to be solved** directly.



Recursion types

- `tail recursion` : **easily** implemented using iteration
  ex) return n*factorial (n-1);
- `head recursion` : **difficult** to implement using iteration
  ex) return factorial(n-1) * n;
- `multi recursion` : **difficult** to implement using iteration
  ex)

```
function(A, n)
{
//Recursive call of 2.
function(A, n-1)
function(A, n-1)
}
```

Recursion VS Iteration

- `Recursion` : using recursive calls
  pros - Good choice for recursive problems (**easy** to implement)
  cons - overhead of function calls -> usually **slower** execution time

- `Iteration` : using '**for**' or '**while**' loop
  pros - fast execution time
  cons - programming can be often very **difficult** for recursive problems

Then, what is best strategy?
-> It depends on problems.

Factorial

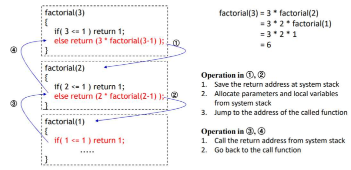
$$n! = \begin{cases} 1 & n = 0 \\ n*(n-1)! & n \geq 1 \end{cases}$$

1. recursive Implimentation of Factorial

```
int factorial_recur(int n) {

	if (n <= 1) return 1;
	else return n * factorial(n - 1);
}
```

> T(N) = T(N-1) + 1 --> Time Complexity : O(N)



2. Iterative Implimentation of Factorial

```
int factorial_iter(int n) {

	int mul = 1;
	for (int i = n; i >= 1; i--) {

		mul *= n; // n* (n-1) ... 2*1
	}
	return mul ;
}
```

> Time Complexity : O(N)

3. compare

Power Computation

: Let's find n-squared value of x : x^n

1. recursive Implimentation of Power Computation

```
double power_iter(double x, int n) { //x^n
```



```
	int i;
	double r = 1.0;
	for (i = 0; i < n; i++) {
		r = r * x;
	}

	return r;
}
```

> Time Complexity : O(N)

2. Iterative Implimentation of Power Computation

```
double power_recur(double x, int n) {

	if (n == 0) return 1;
	else if ((n % 2) == 0)
		return power(x * x, n / 2);
	else
		return x * power(x * x, n-1/ 2);
}
```

> Time Complexity : T(N) = T(2/N) +C -> O(logN)

When n is even

$$power(x,n) = power(x^2, \frac{n}{2})$$

When n is odd

$$power(x,n) = x \cdot power\left(x^2, \frac{n-1}{2}\right) = x \cdot x^{n-1} = x^n$$

3. compare

	slow_power (iteration)	power (recursion)
Time complexity	O(n)	O(log_2n)
Execution time	7.17 sec	0.47 sec

iteration is not good choice😥

Fibonacci Series

- Fibonacci Series

Ex) 0,1,1,2,3,5,8,13,21,...

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-2) + fib(n-1) & otherwise \end{cases}$$

1. recursive Implimentation of Fibonacci Series

```
int fib_recur(int n) {


	if (n == 0) return 0;
	if (n == 1) return 1;
	return fib_recur(n - 1) + fib_recur(n - 2);
}
```

> Time Complexity : T(N) < 2^(N-1) -> O(2^N)

why ? for fib(n), maximum depth of tree is n-1.
and tree must be not filled with leaf node fully.

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

$$\Rightarrow \quad T(n) = \sum_{k=0}^{\lfloor \frac{n-1}{2} \rfloor} \binom{n-k-1}{k} = O(2^n)$$

2. Iterative Implimentation of Fibonacci Series

: to get fib(n), calculating one by one by one from the first two terms .

```
int fib_iter(int n) {

	int temp, current = 1, last = 0; //the first two terms of Fibonacci sequence.

	if (n < 2) return n;
	else {


		for (int i = 1; i < n; i++) {

			temp = current; //keep the last value.

			current = last + current ; //Renewal Nth term
			last = temp; //Save it for the next calculation.
		}


	}

	return current;
}
```

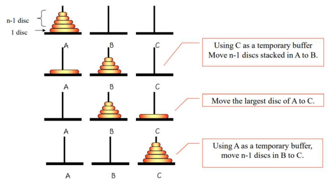
> Time Complexity : O(N)

3. compare

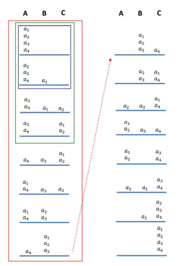
♟ Hanoi tower

: move n disc stacked on rod A to rod C.

For n discs



> Divide and conquer



Hierarchical Structure of Recursion
(Divide-and-Conquer)

Original problem
Move 4 discs a_1, a_2, a_3, a_4 from A to C

: Move 3 discs a_1, a_2, a_3 from A to B

: Move 2 discs a_1, a_2 from A to C

: Move 1 discs a_1 from A to B

Note) $T(n) = 2^n - 1$

1. recursive Implimentation of Hanoi tower

```
void hanoi_tower(int n, char from, char tmp, char to)
{
	if (n == 1) printf("Move 1 disc from % c to % c.\n", from, to);
	else {
		hanoi_tower(n - 1, from, to, tmp);
		printf("Move disc % d from % c to % c.\n", n, from, to);
			hanoi_tower(n - 1, tmp, from, to);
	}
}
```

> Time Complexity : T(N) = 2T(N-1)+1 = 2^N -1 -> O(2^N)

2. Iterative Implimentation of Hanoi tower

3. compare

Binary Search

: when a array of ordered numbers is given, find an index k where ak=b.

1. recursive Implimentation of Binary Search

```
int search_iter(A, b)
for i=1 to n
if(A[i] == b)
k=i;
return k
```

> Time Complexity : O(N)

2. Iterative Implimentation of Binary Search

```
int search_recur(A, b, start, end)
if(start>end) return -1;
int median = (start+end)/2;
if(A[median]<b)
search_recur(A, b, median, end);
else if(A[median]>b)
search_recur(A, b, start, median);
else
return median;
```

> Time Complexity : T(N)=t(N/2)+C -> O(logN)

3. compare

EWHA CSE 20

이전 포스트
Data Structure

0개의 댓글

댓글을 작성하세요

댓글 작성