More

I, ME AND MYSELF !!!

THURSDAY, DECEMBER 10, 2009

Attacking Recursions

Some general approach for solving recursive problems:

If anyone want to read an awesome super cool tutorial on recursion in BANGLA... read from Fahim Vai's page

#1: Forget about what you need to do, just think about any input, for which you know what your function should output, as you know this step, you can build up a solution for your problem. Suppose you have a function which solves a task, and you know, this task is somehow related to another similar task. So you just keep calling that function again and again thinking that, the function I'm calling will solve the problem anyhow, and the function will also say, I'll solve this problem, if you give me the result for another sub-problem first!!! Well, then you'll say, "So, why don't you use your twin-brother to solve that part?" and so on... The following example will show you how to start writing a recursive function.

Example: Think about computing n! recursively. I still don't know what and how my function will do this. And I also don't know, like what could be 5! or 7!... But nothing to worry about, I know that 0! = 1! = 1. And I also know that, n! = n(n-1)!. So I will think like that, "I will get n! from a function F, if some one gives me (n-1)!, then I'll multiply it with n and produce results". And, thus, F is the function for computing n!, so why don't I use again to get (n-1)! ? And when F tries to find out what could be the value of (n-1)!, it also stumbles at the same point, it wonders what would be the value of (n-2)!... then we tell him to use F again to get this... and this thing keeps going as long as we don't know what F actually should return for any k. In case of k = 1, as we know now F can return 1 for k = 1, and it don't need to call another F to solve anything first...

```
int factorial(int n) {
    // I know this, so I don't want my function to go any further...
    if(n==0) return 1;
    // don't bother what to do, just reuse the function...
    else return n*factorial(n-1);
}
```

#2: What ever you can do with loops, can be done with recursions as well. A simple technique for converting recursions and loops is shown below:

Forward:
for loop:

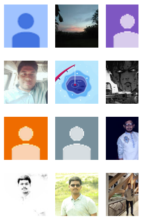
```
for(int i = 0; i < n; i++) {
    // do whatever needed
}
```

Equivalent recursion:

```
void FOR(int i, int n) {
    if(i==n) return; // terminates
    // do whatever needed
    FOR(i+1, n); // go to next step
}
```

Backward:
for loop:

Followers (669)



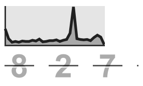
Follow

SUBSCRIBE

Posts

Comments

BLOG HITS



BLOG ARCHIV

- ► 2015 (4)
- ► 2014 (6)
- ► 2013 (19)
- ► 2012 (14)
- ► 2011 (15)
- ► 2010 (33)
- ▼ 2009 (27)
  - ▼ December
    - Happy New
    - Dijkstra's A
    - Linked List
    - Bitwise ope
    - Bitwise ope
    - Bitwise ope
    - Power Seri
    - Gauss–Jor
    - Attacking R
    - Drawing Re
    - Practice Re
  - ► November
  - ► October (1
  - ► Septembe
  - ► August (1)
  - ► July (8)

ABOUT ME

Zobayer Has

View my complet

```
for(int i = n-1; i >= 0; i -= 1) {
    // do whatever needed
}
```

Equivalent recursion:

```
void ROF(int i, int n) {
    if(i==n) return; // terminates
    ROF(i+1, n); // keep going to the last
    // do whatever needed when returning from prev steps
}
```

Well, you may wonder how this is backward loop? But just think of its execution cycle, just after entering the function, it is calling itself again incrementing the value of *i*, and the execution routine that you have written under the function call, was paused there. From the new function it enters, it works the same way, call itself again before executing anything...Thus when you have reached the limiting condition, or the base condition, then the function stops recursion and starts returning, and the whole process can be shown as below...let n=5, and we want to print 5 4 3 2 1...code for this might be:

```
void function(int i, int n) {
    if(i<=n) {
        function(i+1, n);
        printf("%d ", i);
    }
}
```

Explanation might look like this:

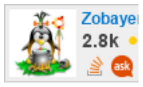
```
01|call function$_1$ with i=1
02|    call function$_2$ with i=2
03|        call function$_3$ with i=3
04|            call function$_4$ with i=4
05|                call function$_5$ with i=5
06|                    call function$_6$ with i=6
07|                        i breaks condition, no more calls
08|                    return to function$_5$
09|                    print 5
10|                return to function$_4$
11|                print 4
12|            return to function$_3$
13|            print 3
14|        return to function$_2$
15|        print 2
16|    return to function$_1$
17|    print 1
18|return to main, done!
```

Left side number shows the execution steps, so from the above program, we get, 5 4 3 2 1. So indeed it ran on reverse direction...

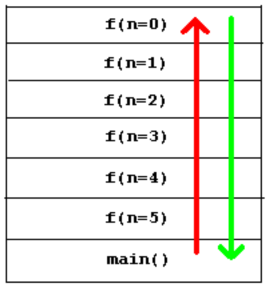
#3: Use the advantage of call stack. When you call functions recursively, it stays in the memory as the following picture demonstrates.

```
int f(int n) {
    if(n==0) return 1;
```

STACK OVERF



```
    return n*f(n-1);
}
```



🟩 returning
🟥 calling

You know about stack, in a stack, you cannot remove any item unless it is the topmost. So you can consider the calling of a recursive function as a stack, where, you can't remove the memory used by f(n=3) before removing f(n=2) and so so... So, you can easily see that, the functions will hold all their variables and values until it is released. This actually serves the purpose of using an array.

#4: Be careful while using recursions. From a programming contest aspects, recursions are always best to avoid. As you've seen above, most recursions can be done using loops somehow. Recursions have a great deal of drawbacks and it most of the time extends the execution time of your program. Though recursions are very very easy to understand and they are like the first idea in many problems that pops out in mind first... they still bear the risks of exceeding memory and time limits.

Generally, in loops, all the variables are loaded at the same time which causes it a very low memory consumption and faster access to the instructions. But whenever we use recursions, each function is allotted a space at the moment it is called which requires much more time and all the internal piece of codes stored again which keeps the memory consumption rising up. As your compiler allows you a specific amount of memory (generally 32 MB) to use, you may overrun the stack limit by excessive recursive calls which causes a Stack Overflow error (a Runtime Error).

So, please think a while before writing recursions whether your program is capable of running under the imposed time and memory constraints. Generally recursions in O(lg n) are safe to use, also we may go to a O(n) recursion if n is pretty small.

#5: If the recursion tree has some overlapping branches, most of the times, what we do, is to store already computed values, so, when we meet any function which was called before, we may stop branching again and use previously computed values, which is a common technique knows as Dynamic Programming (DP), we will talk about that later as that is pretty advanced.

These techniques will be used in almost all problems when writing recursive solution. Just don't forget the definition of recursion:
Definition of recursion = See the Definition of recursion

Now try this

Posted by Zobayer Hasan at 11:58 AM

37 comments:

Shafaet April 9, 2010 at 1:54 PM

Thanks,this helped a lot.
Shafaet
16th Batch,CseDU

Reply

Bidhan May 3, 2010 at 7:25 PM

I found this blog 'accidentally' on google..
Great Blog!
It will be looking for more articles on Recursion :)

CSE DU, 16th

Reply

Anonymous January 15, 2011 at 8:03 AM

very very nice.........thank u

Reply

zahangir_cuet July 26, 2011 at 4:11 AM

a very good learning blog,,,,many many thanks....

Reply

Anonymous October 16, 2012 at 10:15 PM

many many thanks to you, again find a way to learn

Reply

Mishuk April 24, 2013 at 11:14 PM

It was awesome!! Helped me a lot!! Your students are lucky!!

Reply

Anonymous October 15, 2013 at 11:24 PM

ধন্যবাদ !

Reply

Unknown November 20, 2013 at 3:55 PM

Will try to make good use of it bro....

Reply

Replies

Unknown December 13, 2015 at 10:40 PM

:P

LO-49 May 1, 2016 at 6:36 PM

Thanks Vai!!! :)

Reply

Unknown January 13, 2014 at 10:41 PM

It's really helpful.
Thnax vaiya....:)

Reply

Anowar Hossain Anu January 15, 2014 at 12:52 AM

many many thanks vaiya

Reply

Anowar Hossain Anu January 15, 2014 at 12:55 AM

many many thanks vaiya

Reply

Unknown December 5, 2014 at 8:01 PM

It's really helpful....thnx a lot vaia.... :)

Reply

Unknown July 9, 2015 at 4:02 PM

nice blog....thanx vaiya.

Reply

Unknown July 9, 2015 at 4:03 PM

nice blog...thanx vaiya

Reply

মুহাম্মদ শহীদ উল্লাহ October 21, 2015 at 8:07 PM

Thanks a lot vaiya

Reply

joy November 2, 2015 at 12:17 AM

thanks bro

Reply

joy November 2, 2015 at 12:17 AM

thanks bro

Reply

Unknown February 5, 2016 at 5:19 PM

খুব ভালোলাগল , চালিয়ে যান

Reply

Anonymous March 24, 2016 at 12:29 AM

pretty good article!

Reply

wetret April 19, 2016 at 3:42 PM

Great!!!
very helpful blog.Thank you vai :)

Reply

wetret April 19, 2016 at 3:47 PM

Great!!!
very helpful blog.Thank you vai :)

Reply

Replies

Sifatul Islam October 9, 2016 at 11:29 PM

:D Indeed !!!

Reply

LO-49 May 1, 2016 at 6:36 PM

Thanks Vai!!! :)

Reply

Anonymous October 11, 2016 at 12:17 AM

awesome bro .thnx

Reply

Anonymous April 17, 2017 at 10:30 AM

HI Bhai,
You have written nicely. I have a question. printf("%d ", i); line is executing after returning i=6, I mean when the break condition comes. Then is printf("%d ", i); line storing in some where? There might be 1000 line code after function(i+1, n); line. How these line execute after reaching i=6.

Reply

Replies

Zobayer Hasan April 30, 2017 at 10:39 AM

Whenever you call fucntionA from functionB, a new entry for function B is added in the call stack and the last line that you executed in functionA is also remembered. So once you finish executing functionB, you comeback to the next line you executed last in funcitonA.

If you study assembly language, it will be very clear.

Reply

Anonymous April 17, 2017 at 10:30 AM

HI Bhai,
You have written nicely. I have a question. printf("%d ", i); line is executing after returning i=6, I mean when the break condition comes. Then is printf("%d ", i); line storing in some where? There might be 1000 line code after function(i+1, n); line. How these line execute after reaching i=6.

Reply

SH ADNAN July 11, 2017 at 2:58 PM

thank you so much for awesome artificial written

Reply

Unknown November 7, 2017 at 6:48 PM

thank you brother...

Reply

Anonymous November 28, 2017 at 11:27 PM

Nice post. I was checking continuously this blog and I'm impressed!
Very useful information specially the last part :
) I care for such information a lot. I was looking for this
certain information for a long time. Thank you and
best of luck.

Reply

Anonymous December 22, 2017 at 11:52 PM

Thanks for this beautiful article. Awesome really.

Reply

Mohammad Mejbah January 31, 2018 at 2:52 PM

Really nice and learning post ever..

Reply

Anonymous March 26, 2018 at 9:00 PM

very helpful..thanks....

Reply

Unknown May 20, 2018 at 1:44 AM

thanks vaiya
I felt, recursion is critical function in before.
But after read this articles , i am really thank to u.
now i feel better about recursion function.

Reply

Replies

Zobayer Hasan May 27, 2018 at 12:04 PM

Glad that it helped.

Reply

Enter Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

CATAGORIES

academic study (23) access modifiers (1) ajax (1) algorithm (53) analysis (7) apache (1) backtrack (1) bash (1) beginner (17) bfs (2) bigint (1) binary indexed tree (2) binary tree (1) bit (1
blogger (5) bpm (3) brainfuck (1) brute force (1) bst (1) c (5) c++ (41) changes (1) character device driver (1) chat (3) client (3) combinatorics (2) command prompt (1) common (1) comparator (1) compression (1)
(3) confusion (1) connected component (1) console (1) constructible polygon (1) contest (12) crc (1) cse (5) css (1) customize (1) data mining (2) data structure (16) database (3) DCEPC206 (1) decoding (1)
dfs (1) disjoint set (1) divide and conquer (3) dp (4) driver (1) dual boot (1) dynamic programming (10) encoding (1) encryption (1) error (2) esoteric language (2) euler circuit (3) euler path (2) euler phi (1)
expression evaluation (1) extended euclid (1) facebook (3) factorization (2) fibonacci (1) fix time (1) function (1) funny (15) gcd (2) geometry (6) git (3) github (2) gns3 (2) graph (9) GUANGGUN (1) hacker
hiding window (1) hints (16) holi (1) hopcroft karp (1) huffman (1) incorrect clock (1) inner class (1) instance (1) jar (1) java (8) javascript (2) jdbc (1) jquery (1) kernel programming (2) lab (5) lazy propagation (1)
linux (6) ls (1) makefile (1) math (21) matrix (3) matrix algebra (2) matrix exponentiation (2) matrix multiplication (2) maxflow (2) maximum bipartite matching (3) maximum flow (2) merge sort (3) mista
(2) module compiling (2) multichat (3) mysql (1) networking (2) number system (1) number theory (8) online judge (4) operating system (1) os (1) other (8) parallel programming (3) pattern (1) phi (1) poj (1)
practice (2) primes (5) primit (1) priority queue (1) problem (17) problem classifier (4) problem solving (57) problems solving (1) programming (70) pruning (1) pthreadec
qualification round (1) queen (1) queue (1) range maximum query (1) recursion (6) reflection (1) repository (4) rip (1) rmq (1) sample (1) segment tree (2) server (3) shell (1) shell script (1) sieve (4) simulation (3) sc
(3) spacing (1) sphere online judge (28) spoj (29) static routing (1) syntax highlighting (1) system programming (4) table tag (1) tc (1) template (4) thread (3) time mismatch (1) time setting (1) topcoder
training (6) tree (3) tutorial (10) ubuntu (1) usaco (2) uva (5) uva online judge (5) vector (1) version control (1) web server (1) windows (3)

I am only one, but still I am one.
I cannot do everything, but still I can do something.
And because I cannot do everything I will not refuse to do the something that I can do.
{Helen Keller}