Introduction to Algorithm Design

(Algorithm Correctness)

Algorithm Correctness

When an algorithm is designed it should be analyzed from the following two aspects:

1. Correctness : to verify if the algorithm leads to the solution of the problem or it always produces the expected output for the range of inputs and it eventually terminates.
2. Efficiency: to analyze the performance of the algorithm

Algorithm Correctness Verification:

To verify if the algorithm really solves the problem for which it is designed we can use two strategies :

a) Testing

b) Correctness prove by mathematical induction

Algorithm Correctness

Testing: Try the algorithm on sample inputs

- We need tools to distinguish correct algorithms from incorrect ones
- The best way to prove that an algorithm is incorrect is to produce an instance in which it yields an incorrect answer. Such instances are called **counter examples**.

Good counter examples have two important properties:-

i. Verifiability:- to demonstrate that a particular instance is a counter example to a particular algorithm

ii. Simplicity : once a counter example has been found it is worth simplifying it down to its essence and provide the clear reason exactly why the proposed algorithm fails.

Algorithm Correctness

Question:

Test whether a + b < min(a ,b) correct or not?

Statement: a + b < min(a, b) a<0 and b<0

Let a = -2 b= -4

Then L.H.S := a + b= (-2 +-4)=-6

and R.H.S := min(-2 , -4) = -4 ——— 1:testing by example

Hence L.H.S < R.H.S. -6< -4

Now let a =2 b=4

then L.H.S := a + b= 2+4=6

R.H.S := min(a ,b) = min(2 ,4)=2 ——— 2

Hence L.H.S is not equal to R.H.S

The second case is counter example that shows the given statement is not correct when a ,b is positive integer. Hence we can say the statement is correct when (a<0 and b<0)

Question : Test whether a*b< min(a , b) is correct or not by taking counter example and find out the required constraints to make it correct if not?

Algorithm Correctness

- Failure to find a counter example to a given algorithm does not mean “ it is oblivious that the algorithm is correct. A proof or demonstration of correctness is needed . To proof the correctness we can use mathematical induction is the method of choice.
- Mathematical induction is usually the right way to verify the correctness of recursive and incremental algorithms
- For recursive algorithm we are using method of induction directly to prove the correctness
- For iterative algorithm we are using loop variants and method of induction to prove the correctness.

Algorithm Correctness

Correctness of recursive algorithms:-

To prove correctness of recursive algorithm:

- Prove it by induction on the size of the problem being solved
- Base of recursion is the base of induction
- Need to prove that recursive call are given subproblems that is no infinite recursion.
- Inductive step: assume that the recursive calls work correctly and use this assumption to prove that the current call works correctly.

Question to prove

Example-1

The Fibonacci numbers are a sequence of integers in which the first two elements are 0 & 1, and each following elements are the sum of the two preceding elements:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ..., 233

Algorithm Correctness

Recursive Fibonacci number:

F0 =0 , F1= 1 and for all n >= 2 ,

Fn = Fn-2 + Fn-1

function fib(n)

//comment returns Fn

1. if n = = 0 then return 0;
2. else return(fib(n-1)+fib(n-2))

Claim: For all n >= 0, fib(n) return Fn

Base : for n=0 fib(n) returns 0 F_0 as claimed

for n= 1 fib(n) returns 1 F_1 as claimed

Induction Hypothesis :

suppose that n>= 2 and for all 0<=m <n fib(m) returns F_m

Required to prove:

fib(m+1) returns F_{m+1}

What does fib(m+1) returns ?

fib(m+1) returns

fib(m+1-1) + fib(m+1-2)

= fib(m) + fib(m-1)

= Fm + Fm-1 by induction hypothesis

= Fm+1

Hence by applying induction we are verified the correctness.

Algorithm Correctness

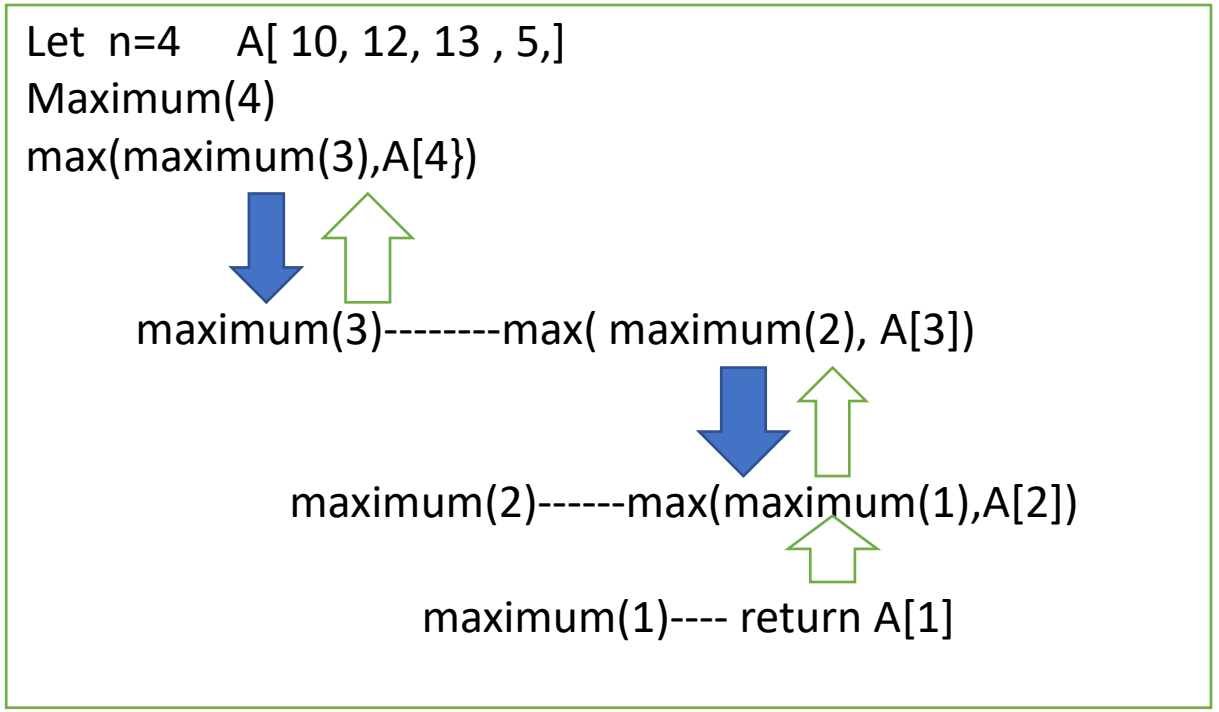
Example-2

Recursive Maximum

function maximum(n)

//comment return max of A[1……n]

1. if n < =1 then return (A[1])
2. else return(max(maximum(n-1),A[n])



Claim:

For all n >=1, maximum(n) returns max{A[1],…….A[n]}

Base: for n=1, maximum(n) returns A[1] as claimed

Induction hypothesis: suppose that n >=1 and maximum(n) returns max{A[1],…….,A[n]}

Required to prove

maximum(n+1)-----returns max{A[1],……..,A[n+1]}

Algorithm Correctness

suppose that n >=1 and isum(A,n) returns sum{A[1],…….,A[n]}
n+1 -------- isum(A,n+1) -- sum(A[1],A[2]….A[n+1])
isum(A,n+1)=sum(A[n+1]+isum(A,n))
=sum(A[n+1],sum(A[1]…A[n])
= sum(A[n+1],A[n]…..A[1])

What does maximum(n+1) returns ?

max(maximum(n),A[n+1])

= max(max{A[1],……,A[n]},A[n+1]) by induction hypothesis

= max{A[1],A[2],……,A[n+1]} proved

Question:1 sum of n numbers

```
function isum(A , n)
    // returns sum of A[1,……,n]
1. if   n<= 1 then return A[1]
2. else  return( sum(A[n] + isum(A,n-1))
```

Question :2 factorial of n

```
function factorial(n)
        // return Fn
  1.  if (n == 0)     return 1
  2  else     return (n * factorial(n - 1))
```

Algorithm Correctness

Correctness of iterative algorithms:

To prove correctness of an iterative algorithm:

- Analysis the algorithm one loop at a time . Starting at the inner loop in case of nested loops.
- For each loop find a loop invariant that remains true each time through the loop and captures the progress made by the loop.
- Prove that the loop invariants holds.
- Use the loop invariants to prove that the algorithm terminates.
- Use the loop invariants to prove that the algorithm computes the correct result.

Algorithm Correctness

Loop Invariant with Examples:

Definition:A loop invariant is a condition [among program variables] that is necessarily true immediately before and immediately after each iteration of a loop. A loop invariant is some predicate (condition) that holds for every iteration of the loop. A loop invariant gives a relationship between variables. If we pick the right invariant, it will help proving the correctness of the program with the loop.

The loop invariant must be true:

- It holds before the loop starts
- It holds after each iteration of the loop
- It holds after the loop terminates

Example:

j := 9; ***i=0 j=9***

```
for( i=0; i<10; i++)
   j--;
```

Initial i=0 j=9

1st iteration i=1 j=8

2nd iteration i=2 j=7……

After loop termination i=9 j=0

i+j=9

Find out the loop variants?

```
x=c; y=0;
While (x>0)
{
   x--;
   y++;
}
```

Entry (before iteration): x=c and y=0

First iteration: x=c-1 y=1-------- x+y=c

Second iteration : x=c-2 y=2-------

x+y=c

……

…

K iteration : x=c-k y=k ---------x+y=c

Final exection: x=0 and y=c x+y=c Loop invariant is x+y=c

Algorithm Correctness

Question: find out the loop invariant?

```
y=0
for(i=0 ; i<=n; i++)
 y+=2^i ;
```

y0 =0 i=0 let n=4

1st iteration: y1=y0+2^0 i=i+1=1

2nd iteration : y2= y1+2^1 i=i+1=2

3rd iteration : y3=y2+2^2 i=3

4th iteration : y4=y3+2^3 i=4

i=0 y0=0=2^0 -1

i=1 y1=1=2^1 -1

i=2 y2=3=2^2 -1

i=3 y3=7=2^3 -1

i=4 y4=15=2^4 -1

$y_i = 2^i - 1$ $i=n$ $y_n = 2^n - 1$

Algorithm Correctness

Properties of good loop invariants:-

1. Initialization: The loop invariants must be true for the first execution of the loop.
2. Maintenance: if the invariant is true before an iteration of the loop it should be true also after the iteration .
3. Termination: when the loop is terminated the invariant should tell us somethings understand the algorithm.

```
function sum(n)
// Input: a non negative integer n
//output: the sum 1+2+3……+n
   sum:=0
   i:=0
  while(i<=n)
// invariant :sum 1+2 …+i-1
            sum := sum + i
            i=i+1
  return sum
```

In this example:

1. The loop invariant holds initially since sum=0 and i=0 , if no element sum is zero
   i =1 sum =0+1
   i=2 sum =1+2
   i= n sum=1+2……+n
2. Invariant holds before ith iteration and also after this iteration since the loop adds i to the sum and increments i by 1
3. When loop terminates sum= 1+2+……+n

Algorithm Correctness

Question:-

```
function max(A)
//input :an array A storing n integers
//output: the largest element in A(max of A[1… n]
 m=A[1]; i:=2
while i<=n   do
      if   A[i]>m then m:=A[i]
           i=i+1
return(m)
```

Consider an array A{7, 5, 3, 10, 2, 6} with 6 elements and we have to find maximum element max in the array.

In the above example after the 3rd iteration of the loop max value is 7, which holds true for the first 3 elements of array A. Here, the loop invariant condition is that max is always maximum among the first i elements of array A.

Algorithm Correctness

Example: Iterative Fibonacci Numbers

```
function fibo(n)
    //comment return Fn
1.      if  n==0 then return 0
2.      else  a:=0; b:=1;i:=2;
3.      while(i<=n) do
4.              c:= a + b; a :=b; b:=c; i=i+1;
5.      return(b)
```

F2=f0+f1 j=1
F3=f1+f2 j=2
Fj=Fj-1 +Fj
Fj+1=Fj+Fj+1

Let n=5 then fib(5) should return F5=5

n=0	0	1	2	3	4	5	6
Fn=0	0	1	1	2	3	5	8
	F0	F1	F2	F3	F4	F5	F6
			j =1	j=2	j=3	j=4	j=5

Facts about the algorithm:

$i_0=2$ | $a_0=0$ | $b_0=1$ | $c_{j+1} = a_j + b_j$

$i_{j+1} = i_j+1$ | $a_{j+1} = b_j$ | $b_{j+1} = c_{j+1}$

Loop invariants

For all natural numbers $j >= 0$, $i_j = j+2$,$a_j=F_j$ and $b_j=F_{j+1}$

Iteration0 j=0	Iteration1 j=1	Iteration2 j=2	Iteration3 j=3	Iteration4 j=4
	i0<=5	**i1<=5**	**i2<=5**	**i3<=5**
	c1=a0+b0=1	**c2=a1+b1=2**	**c3=a2+b2=3**	**c4=a3+b3=5**
a0=0	**a1=b0=1**	**a2=b1=1**	**a3=b2=2**	**a4=b3=3**
b0=1	**b1=c1=1**	**b2=c2=2**	**b3=c3=3**	**b4=c4=5**
i0=2	**i1=i0+1=2+1=3**	**i2=i1+1=3+1=4**	**i3=i2+1=4+1=5**	**i4=i3+1=5+1=6**
i0=j+2	i1=j+2	i2=j+2	i3=j+2	

Algorithm Correctness

Iterative Fibonacci Numbers

Claim: for all natural numbers $j >= 0$, $i_j = j+2$, $a_j = F_j$ and $b_j = F_{j+1}$

Proof by induction on j: The base case j=0 is trivial since $i_0 = 2$ $a_0 = 0 = F_0$, and $b_0 = 1 = F_1$

Now assume that $j >= 0$, $i_j = j+2$, $a_j = F_j$ and $b_j = F_{j+1}$

Required to prove $i_{j+1} = j+3$, $a_{j+1} = F_{j+1}$ and $b_{j+1} = F_{j+2}$

$i_{j+1} = i_j + 1 = (j+2) + 1$ by induction hypothesis $= j+3$

$a_{j+1} = b_j = F_{j+1}$

$b_{j+1} = c_{j+1} = a_j + b_j = F_j + F_{j+1} = F_{j+1}$

Algorithm Correctness

Iterative Maximum:

function max(A)

//input :an array A storing n integers

//output: the largest element in A(max of A[1… n]

```
 m=A[1]; i:=2
while i<=n   do
      if   A[i]>m then m:=A[i]
           i=i+1
return(m)
```

Facts about the algorithm:

m0=A[1] mj+1=max(mj ,A[i_j])

i0=2 i_{j+1} = i_j+1

Loop invariants

For all natural numbers j>=0

mj=max(A[1],A[2]……A[j+1])

i_j=j+2

A[3 ,5 ,9 7, 6]

Iteration no(j)	i	n	A[1,2…n]	Maximum m
0	1	1	A[1]	m=A[1]
1	2	2	A[1,2]	m=max(A[1]A[2])
2	3	3	A[1,2,3]	m=max(A[1],A[2],A[3])
3	4	4	A[1,2,3,4]	m=max(A[1],A[2],A[3],A[4])
4	5	5	A[1,2,3,4,5]	m=max(A[1],A[2],A[3],A[4],A[5])

j>=0 mj=max{A[1],A[2]…….A[j],A[j+1]) ----loop invariant:loop invariant condition is that max is always maximum among the first i elements of array A.

j=0	j=1	j=2	j=3	j=4
m0=A[1]	i0<=5	i1<=5	i2<=5	i3<=5
i0=2	A[2]>m0	A[3]>m1	A[4]>m2	A[5] >m3
	m1=A[2]	m2=A[3]	m3=A[4]	m4=A[5]
	i1=i0+1	i2=i1+1	i3=i2+1	i4=i3+1

Algorithm Correctness

Claim: for all natural numbers $j >= 0$, $i_j = j+2$, $m_j = \max\{A[1],A[2],\ldots\ldots A[j+1]\}$

Proof by induction on j: The base case j=0 is trivial since $m_0 = A[1]$ and $i_0 = 2$

Now assume that $j>=0$, $i_j = j+2$, and $m_j = \max\{A[1],A[2],\ldots..A[j+1]\}$

Required to prove $i_{j+1} = j+3$, and $m_{j+1} = \max\{A[1],A[2],\ldots\ldots A[j+2]\}$

$i_{j+1} = i_j + 1 = (j+2) + 1 = j+3$ by induction hypothesis

$m_{j+1} = \max\{m_j, A[i_j]\}$

$= \max\{m_j, A[j+2]\}$ by induction hypothesis

$= \max\{\max\{A[1],A[2]\ldots\ldots A[j+1]\}, A[j+2]\}$ by induction hypothesis

$= \max\{A[1],A[2],A[3]\ldots\ldots A[j+2]\}$

Algorithm Correctness

```
function factorial(n)
//return Fn
1.  fact=1, i=1
2.  while(i<=n)
3.          fact=fact* i
4.              i=i+1
5.  return(fact)
```

n == 0: the loop repeats 0 times: it computes fact == 1 == 0!
n == 1: the loop repeats 1 time: it computes fact =(m= 1 * 1 == 1! i1=i0+1
n == 2: the loop repeats 2 times: it computes fact == (1 * 1) * 2 == 2! i2= i1+1=3
n == 3: the loop repeats 3 times: it computes fact == (1 * 1 * 2) * 3 == 3! i=3+1=4
n == 4: the loop repeats 4 times: it computes fact == (1 * 1 * 2 * 3) * 4 == 4!
. . .
the loop repeats k times : it computes fact= (1*1*2*3*4*5…..)*k=k! i=k+1
the loop repeats k+1 times: it computes fact == (k!) * (k+1) == (k+1)!

we show that after going through the loop k times, $F_k = k!$ and $i = k + 1$ hold. This is a loop invariant and again we are going to use mathematical induction to prove it

Proof by induction.

Basis Step: n = 1. When n = 1, that is when the loop is entered the first time, F1 = F0* 1 = 1 and $i_1 = i_0 + 1 = 2$.

Induction Hypothesis: For an arbitrary value m of n, $F_m = m!$ and $i_m = m + 1$ hold after going through the loop m times.

Inductive Step: for (m + 1) time, $F_{m+1} = (m+1)!$ and $i_{m+1} = (m+2)$

$$i_{m+1} = i_m + 1 = m + 1 + 1 = m + 2$$

$$F_{m+1} = m!*(m+1) = F_m *(m+1) = (m+1)!$$

producing F = (m + 1)! and i = (m + 1) + 1.

Thus F = k! and i = k + 1 hold for any positive integer k.

Now, when the algorithm stops, i = n + 1. Hence the loop will have been entered n times. Thus F = n! is returned. Hence the algorithm is correct.