EECS 3101

Prof. Andy Mirzaian

Iteration
&
Recursion

STUDY MATERIAL:

- **[CLRS]** chapters 2, 4.1-2, 12.1, 31.1-2, 33.4
- Lecture Note 4

TOPICS

- **The algorithm design process:**
  - **Central Tools:** Iteration & Recursion
  - **Partial correctness:** Assertions
  - **Termination:** Measure of Progress

- **Iterative Algorithms:**
  - Loop Invariant
  - Incremental Method

- **Recursive Algorithms:**
  - Recursive Pre- & Post- Condition & strong induction
  - Divide-&-Conquer Method

The Algorithm Design Process

We are given the pleasurable task of designing an algorithm for a computational problem.

- Where should we start?
- What design process should we follow?
- Are we on the right track?
- Is there a systematic way to convince ourselves, our friends, and all skeptics, that our algorithm is correct?
- Is our algorithm guaranteed to terminate on every possible input?
- Is there a better algorithm?
- …

Assertions

- An **assertion** is a true/false logical statement about the state of the variables and data structures of the algorithm.

- Although they appear as inserted comments, **assertions** are **not** descriptions of the code's intended purpose or an explanation of its action.

- Important examples of assertions are:
  - Pre-Condition
  - Post-Condition
  - Loop Invariant

- **What are assertions good for?**
  - They guide us along the design process. They should be with us from the initial design stage; to the middle, keeping us on track; to the end, convincing us that we have a correct algorithm, if we maintain correct assertions.

  - The design process might go through a revise-&-repeat cycle. If our assertions show us we are not quite on the right track, they usually also show us what is lacking and suggest ways of correction!

Algorithm GuessWhat(n)
Pre-Cond: input n is a natural number
Post-Cond: output is ---------- (fill in the blanks)

```
    i ← 0
    j ← 1
    while i < n do
        i ← i +1
        j ← j + j
    end-while
    return j
end
```

§ What is true about values of i, j & n here?
§ Any relationship between them?

Termination:

Measure of Progress:
$MP = n - i.$

Each iteration reduces MP by 1.

Initial MP = n.

Loop exits when $MP \le 0.$

 # iterations = n.

Algorithm GuessWhat(n)
Pre-Cond: input n is a natural number
Post-Cond: output is 2^n (how can we be sure?)

```
    i ← 0
    j ← 1
    loop
        if i ≥ n then exit loop
        i ← i +1
        j ← j + j
    end-loop
    return j
end
```

$1 = 2^0$ & $0 \le n$

Loop-Invariant: $j = 2^i$ & $i \le n$

$j = 2^i$ & $i < n$

$2j = 2^{i+1}$ & $i+1 \le n$

so now: $j' = 2^{i'}$ & $i' \le n$

$j = 2^i$ & $i \le n$ & $i \ge n$

Establish LI (induction Base)

(induction hypothesis)

Maintain LI (induction) & progress towards goal

Exit loop, use LI & reach goal

ACTIONS

Algorithm ALG(Input)

Pre-Cond: Input satisfies problem instance specifications

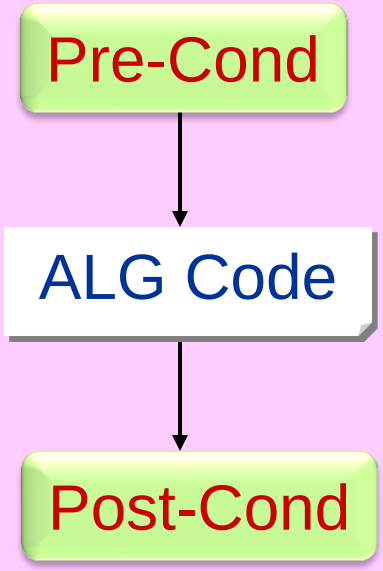
ALG Code

Post-Cond: output satisfies problem solution requirements

end

ASSERTIONS

Pre-Cond & ALG Code ⇒ Post-Cond



Partial Correctness Warranty:

If input satisfies Pre-Cond, and
if ALG is executed on that input, and
if ALG eventually terminates,
then the output is guaranteed to satisfy Post-Cond.

Post-Cond is not guaranteed to hold
if input does not satisfy Pre-Cond, or
if ALG does not terminate on the given input.

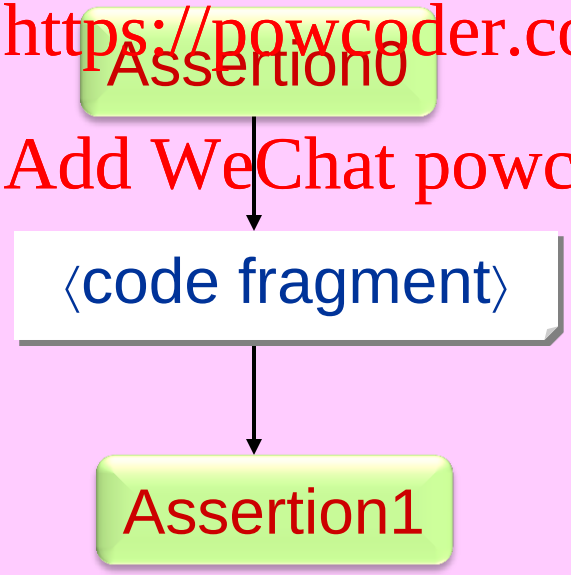
Total Correctness = Partial Correctness + Termination.

Assertion0

⟨code fragment⟩

Assertion1

Assertion0 & ⟨code fragment⟩ ⇒ Assertion1



Sequential code:

Assertion 0

⟨code 1⟩

Assertion 1

⟨code 2⟩

…….

Assertion M-1

⟨code M⟩

Assertion M

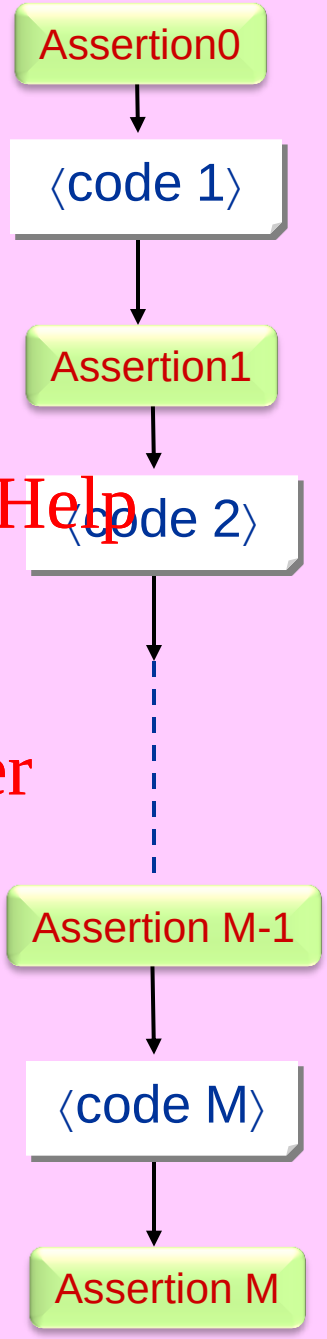
Assertion 0 & ⟨code 1⟩ ⇒ Assertion 1

Assertion 1 & ⟨code 2⟩ ⇒ Assertion 2

Assertion 2 & ⟨code 3⟩ ⇒ Assertion 3

….

Assertion M-1 & ⟨code M⟩ ⇒ Assertion M



If-then-else:

Assertion0

if ⟨cond⟩

then ⟨code+⟩

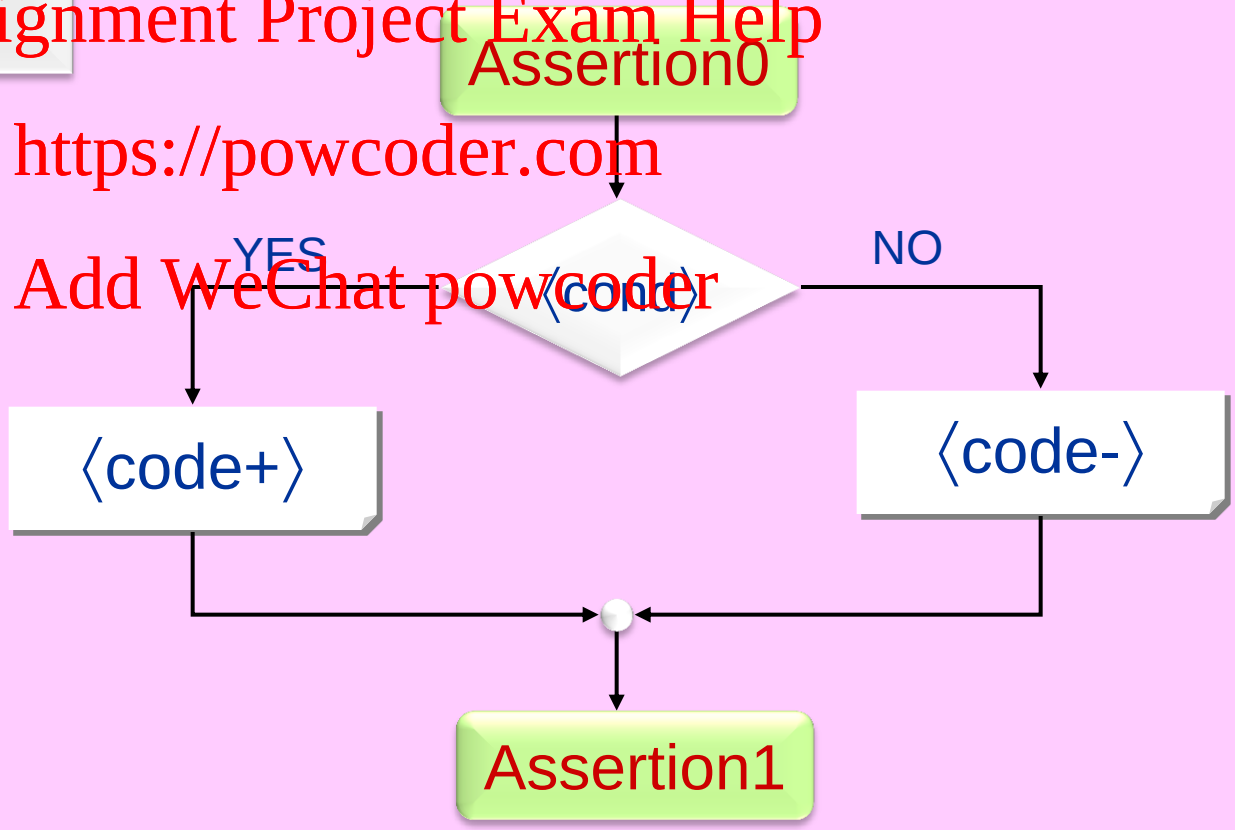
else ⟨code-⟩

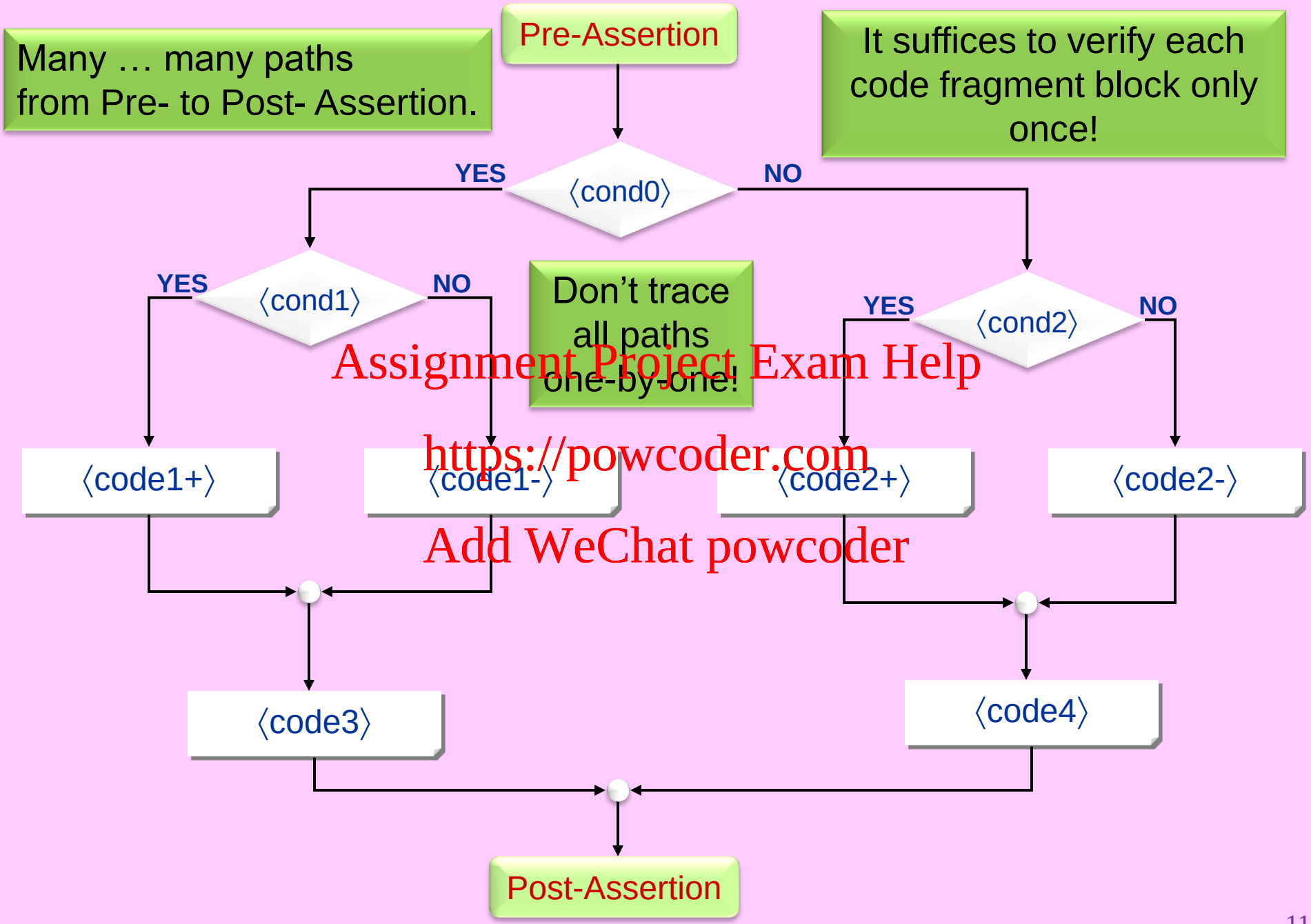
Assertion1

Assertion0 & ⟨cond⟩ & ⟨code+⟩ ⇒ Assertion1

and

Assertion0 & ¬⟨cond⟩ & ⟨code-⟩ ⇒ Assertion1





Loop Invariant

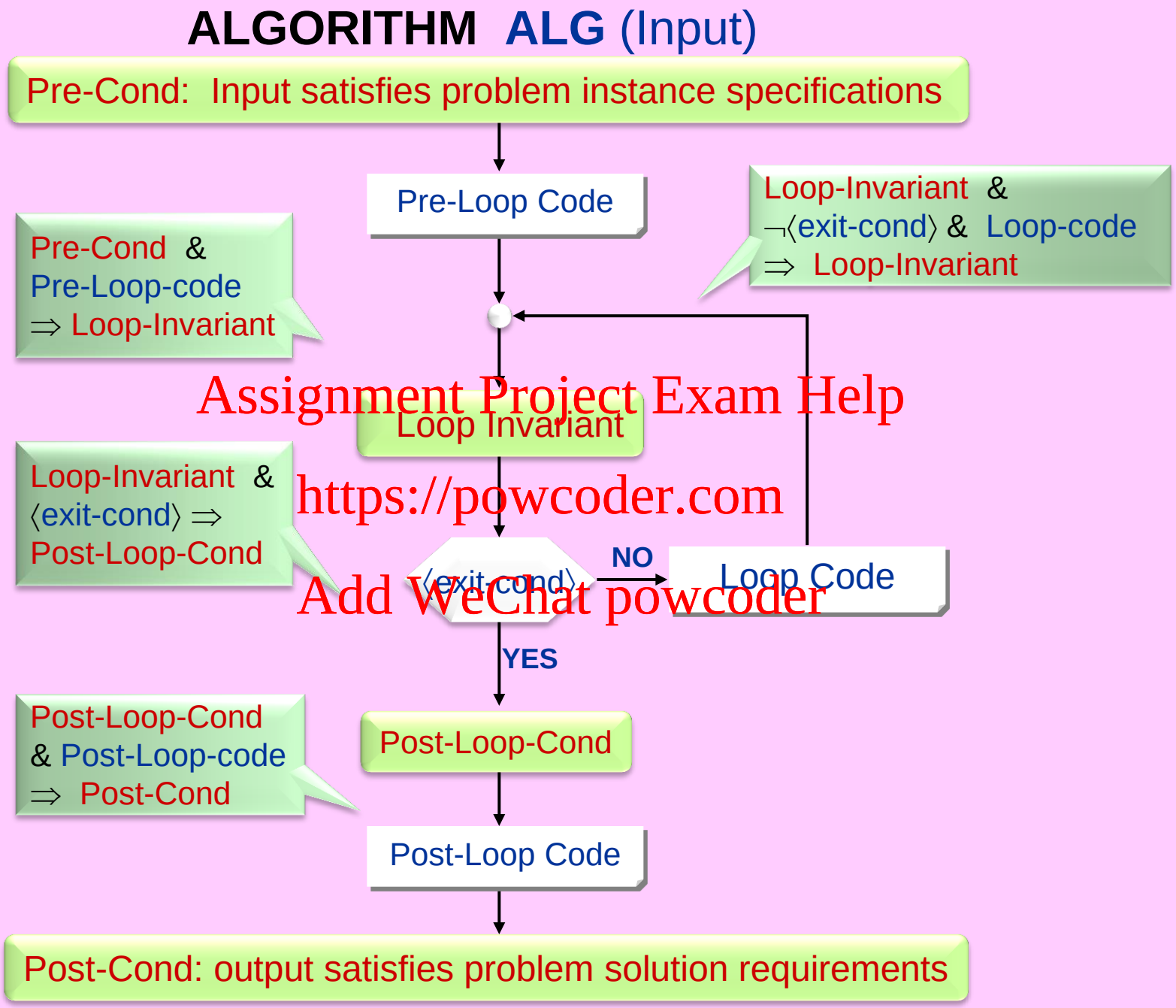
ACTIONS

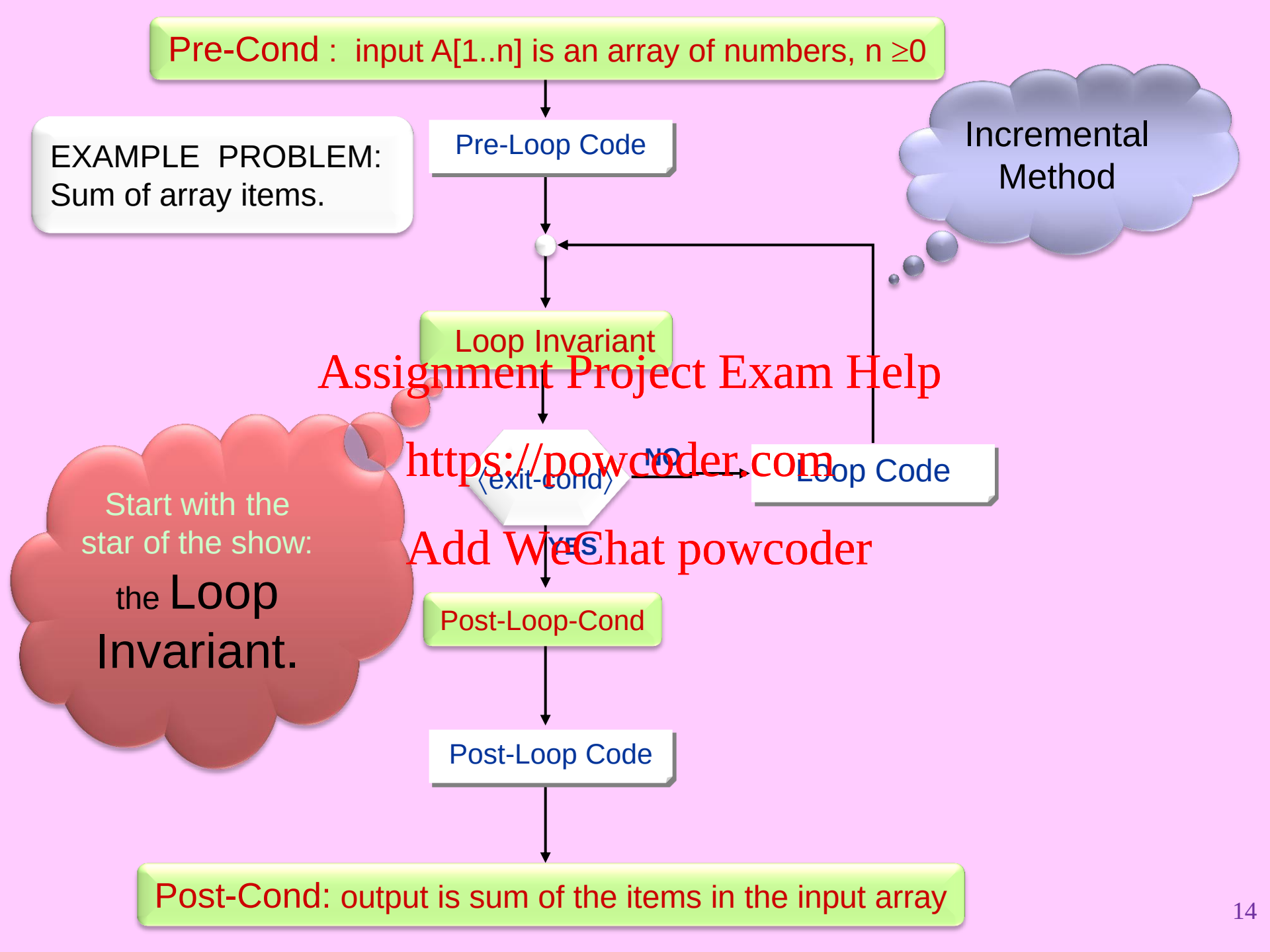
Algorithm ALG(Input)
Pre-Cond: Input satisfies problem instance specifications
    Pre-Loop Code
    Loop:
        Loop Invariant
        if ⟨exit-condition⟩ **then exit loop**
        Loop-Code
    end-loop
    Post-Loop-Cond
    Post-Loop Code
Post-Cond: output satisfies problem solution requirements
end

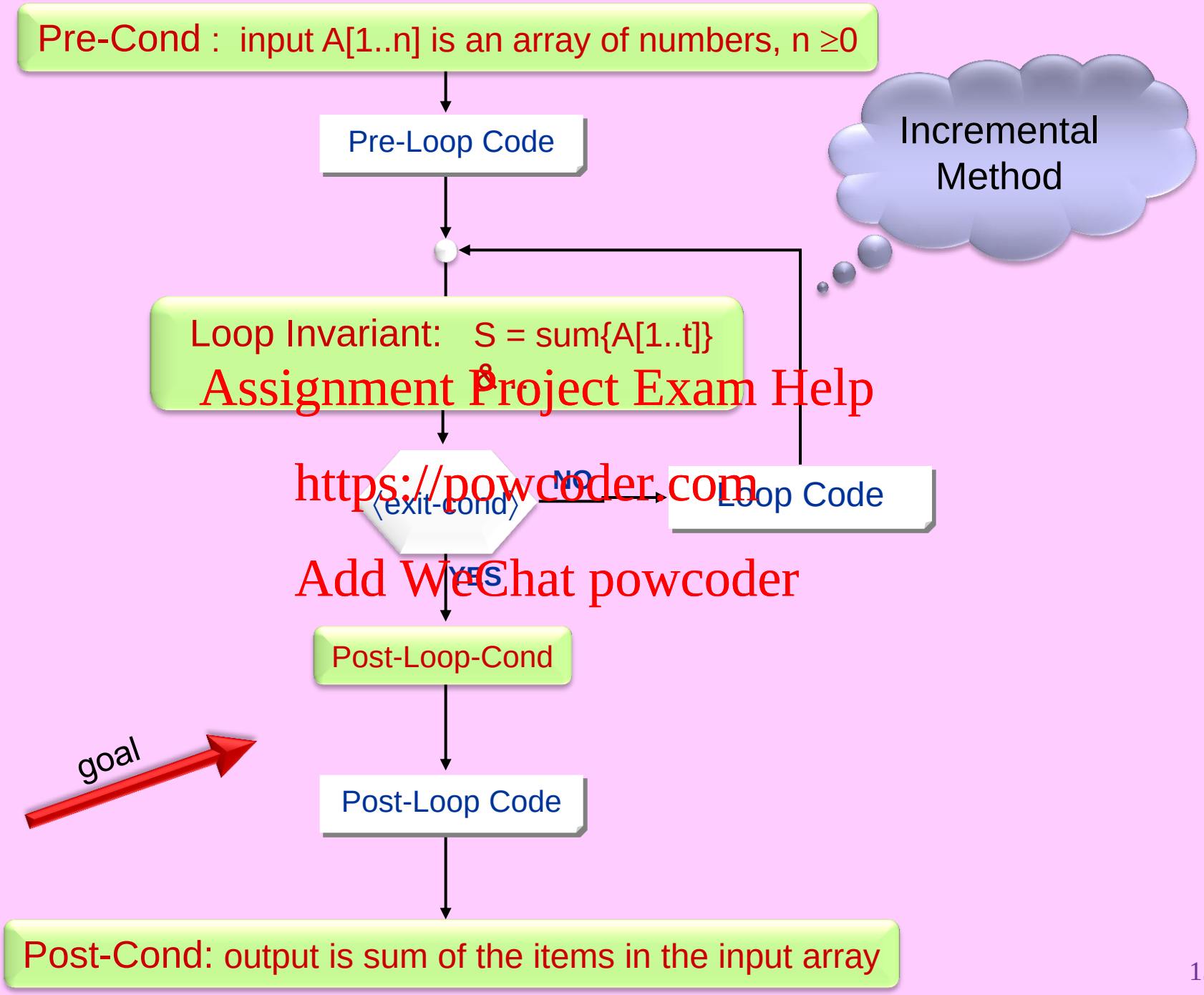
ASSERTIONS

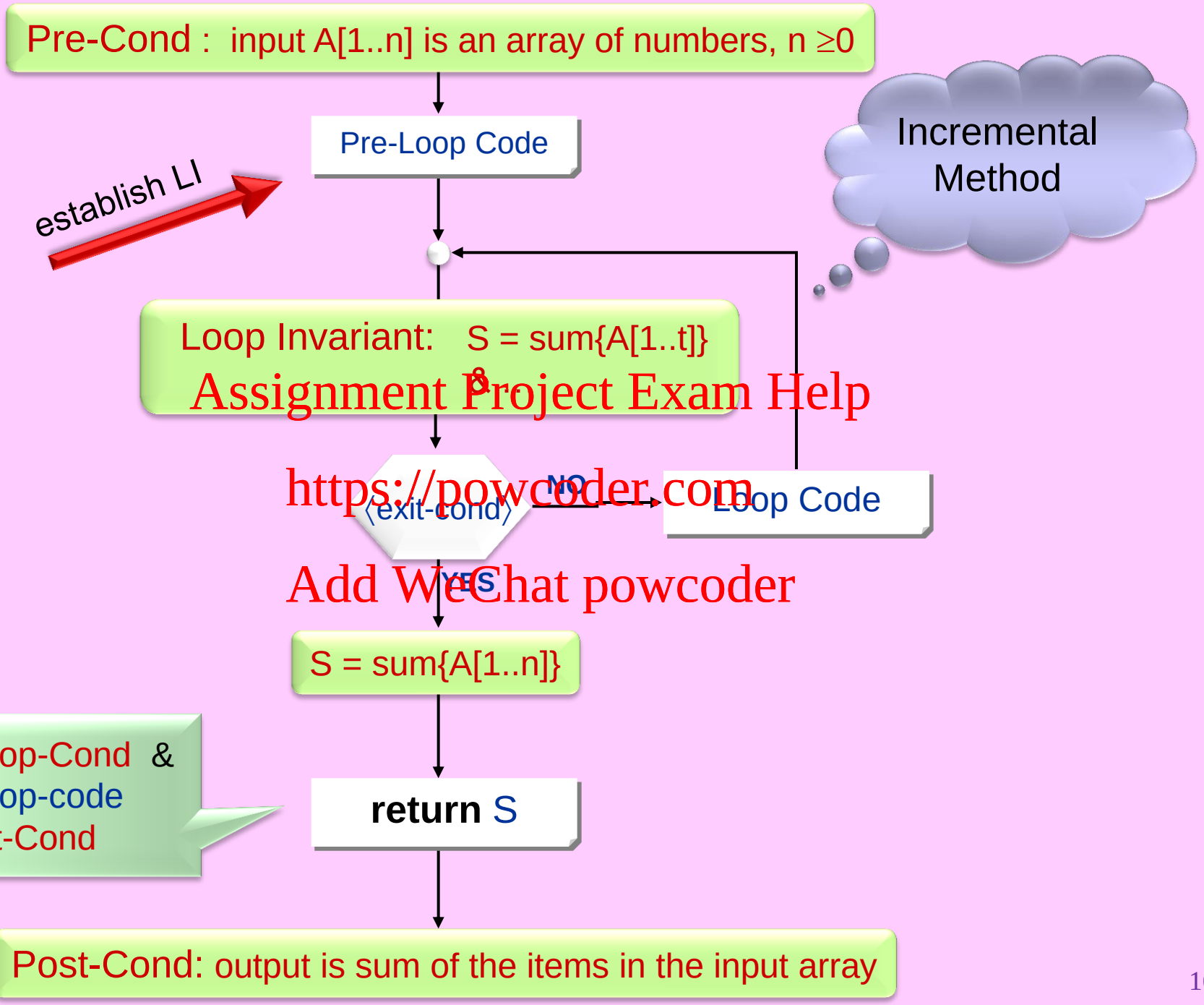
Pre-Cond & Algorithm Code ⇒ Post-Cond

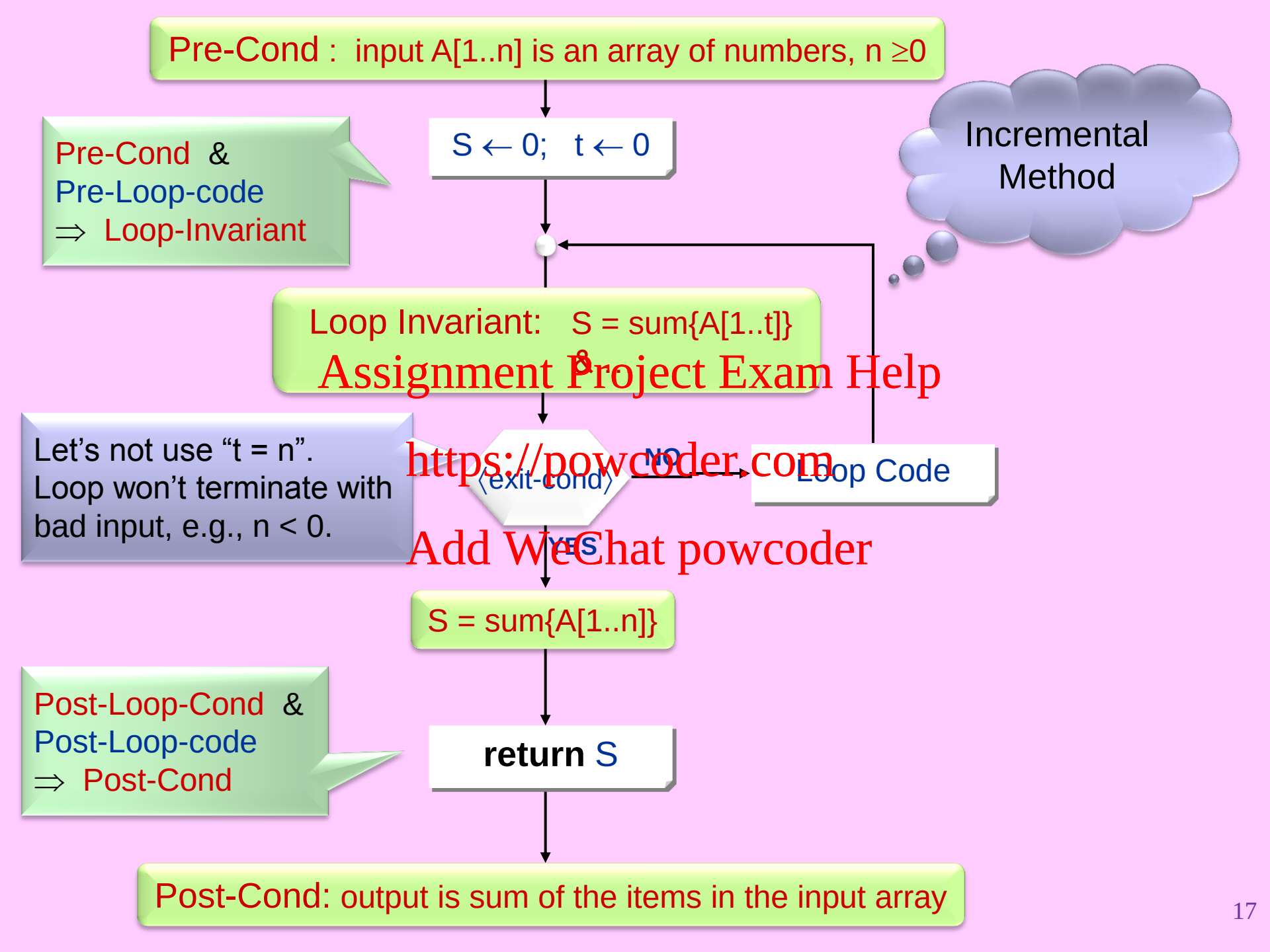
Loop-Invariant ⟵ induction hypothesis
Pre-Cond & Pre-Loop-code ⇒ Loop-Invariant ⟵ induction base
Loop-Invariant & ¬⟨exit-cond⟩ & Loop-code ⇒ Loop-Invariant ⟵ induction
Loop-Invariant & ⟨exit-cond⟩ ⇒ Post-Loop-Cond
Post-Loop-Cond & Post-Loop-code ⇒ Post-Cond ⟵ clean up loose ends

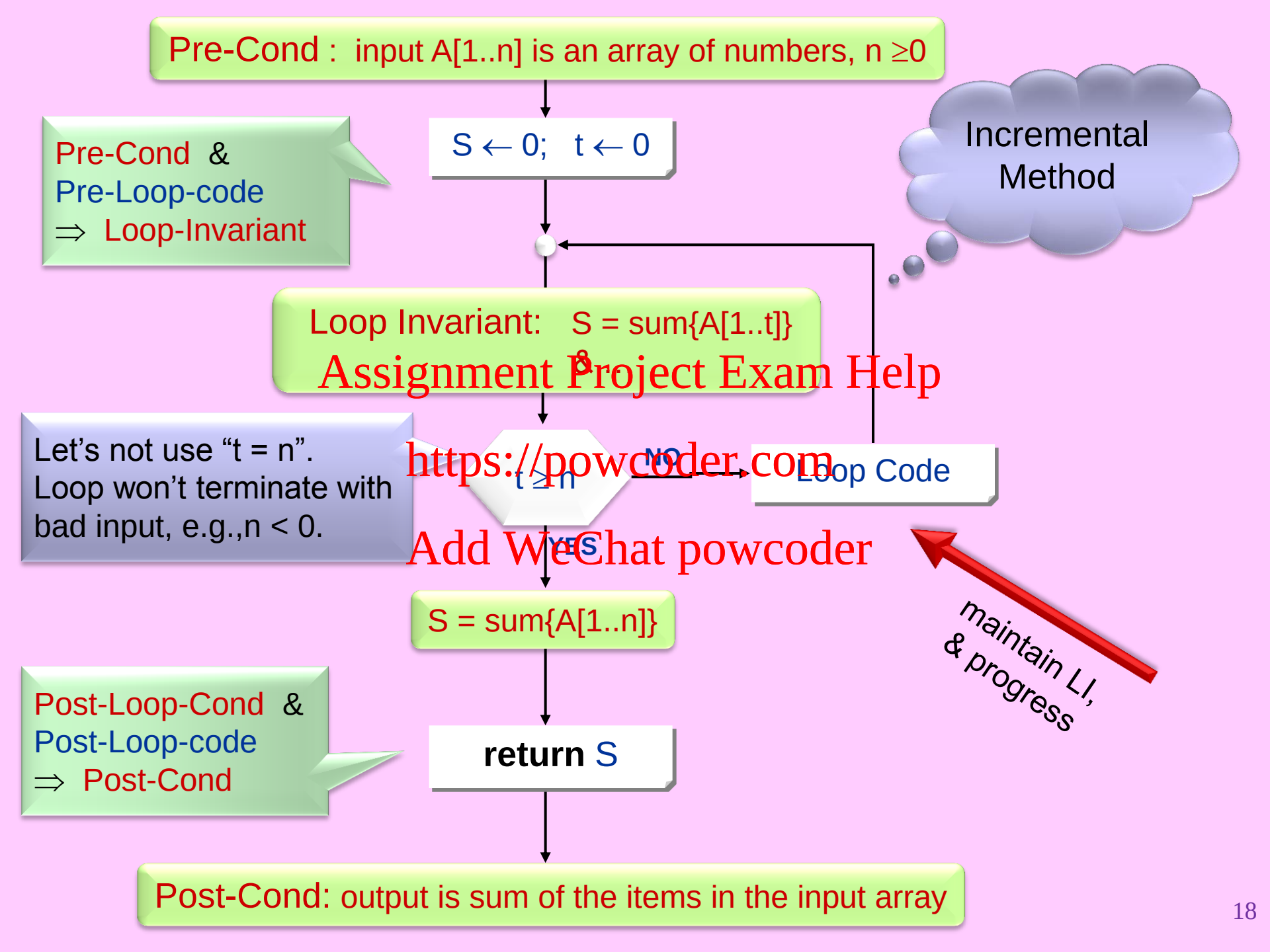


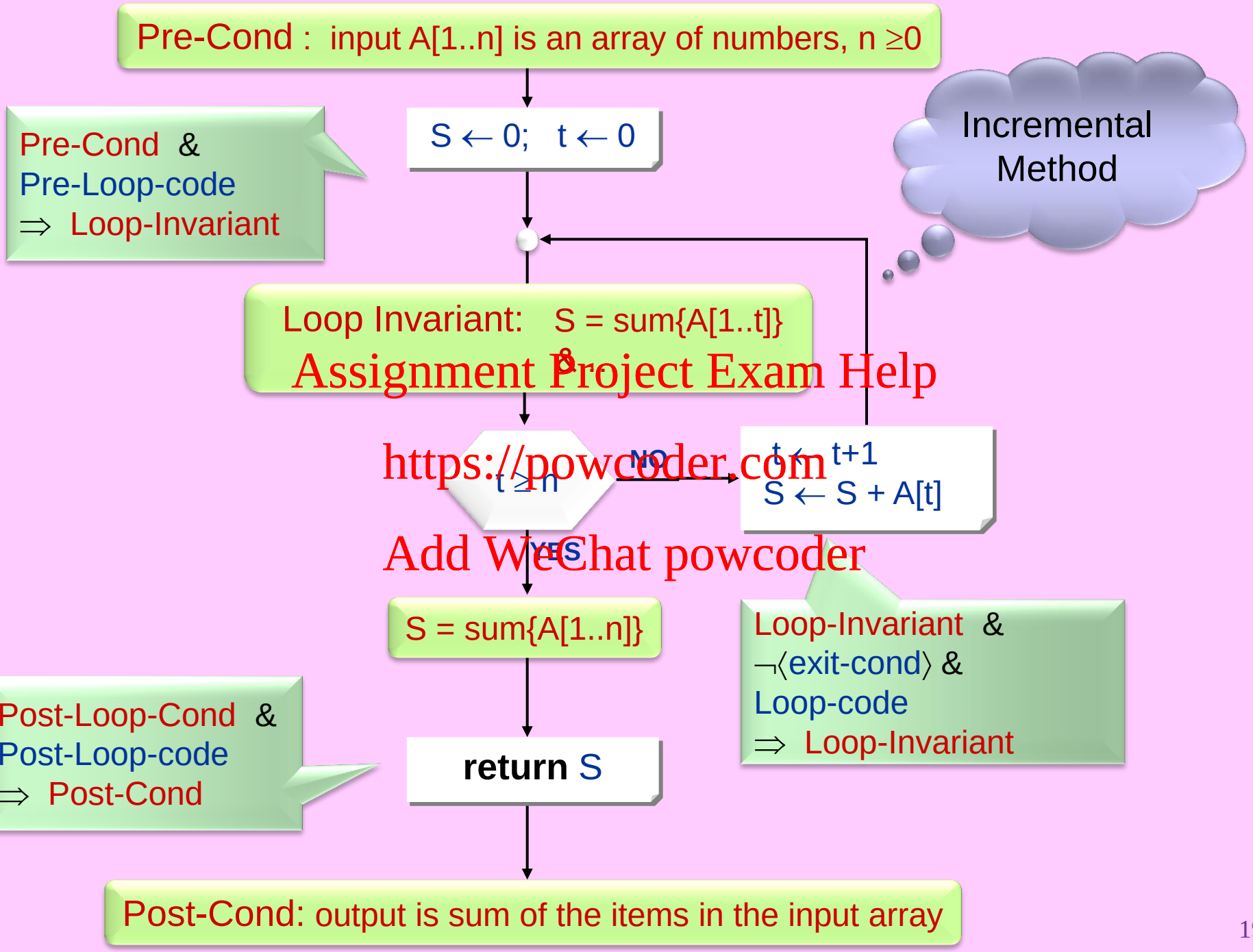


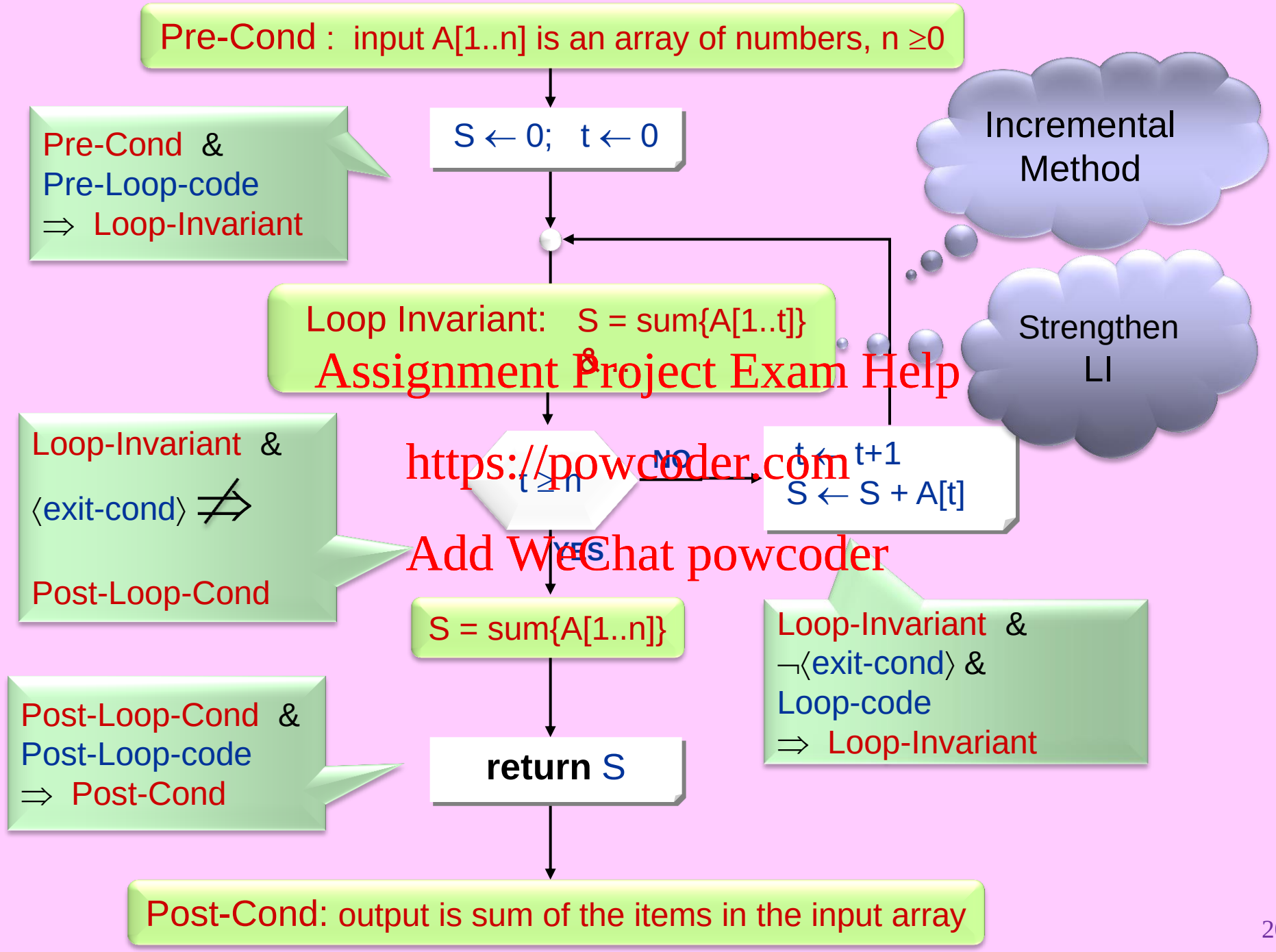

Pre-Cond : input A[1..n] is an array of numbers, n ≥0
Pre-Loop Code
Incremental Method
Loop Invariant: S = sum{A[1..t]}
Assignment Project Exam Help
https://powcoder.com
⟨exit-cond⟩
NO
Loop Code
Add WeChat powcoder
YES
Post-Loop-Cond
goal
Post-Loop Code
Post-Cond: output is sum of the items in the input array

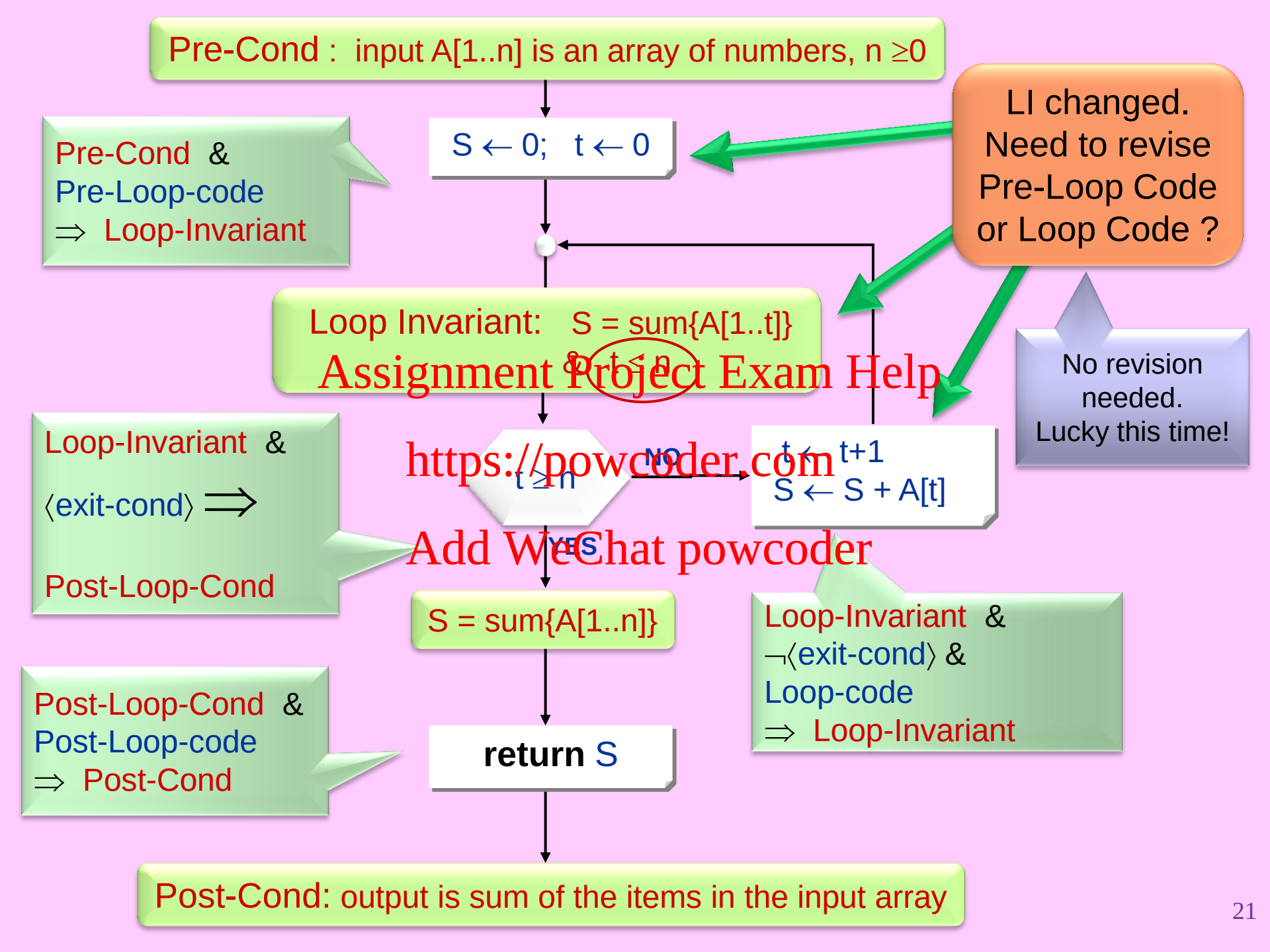

Pre-Cond : input A[1..n] is an array of numbers, n ≥0
Pre-Loop Code
establish LI
Incremental Method
Loop Invariant: S = sum{A[1..t]} & ...
Assignment Project Exam Help
https://powcoder.com
⟨exit-cond⟩
NO
Loop Code
Add WeChat powcoder
YES
S = sum{A[1..n]}
Post-Loop-Cond &
Post-Loop-code
⇒ Post-Cond
return S
Post-Cond: output is sum of the items in the input array

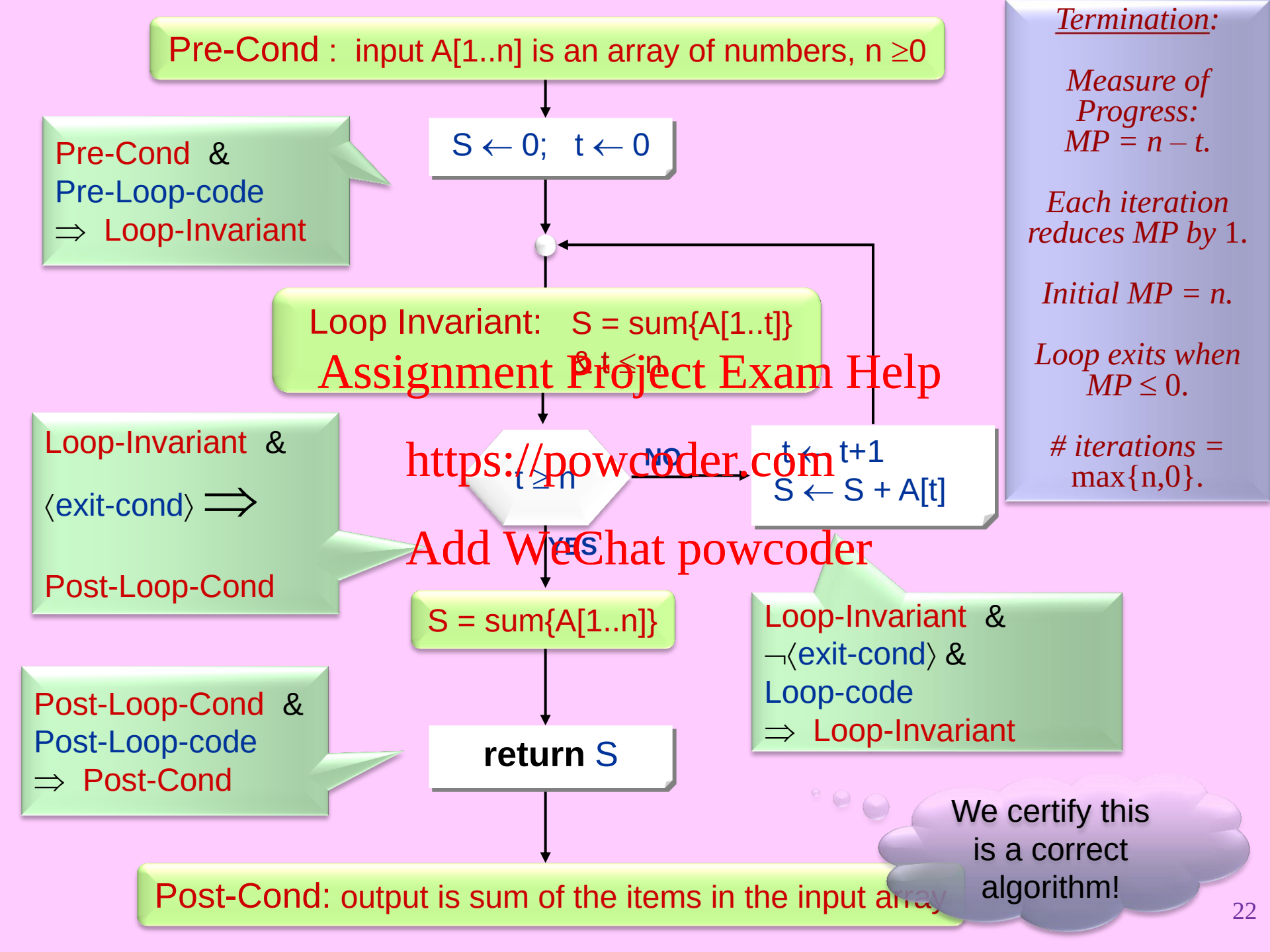







Pre-Cond : input A[1..n] is an array of numbers, n ≥0
S ← 0; t ← 0
Pre-Cond &
Pre-Loop-code
⇒ Loop-Invariant
Incremental Method
Loop Invariant: S = sum{A[1..t]}
&
t ≤ n
Strengthen LI
Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder
NO
t ← t+1
S ← S + A[t]
Loop-Invariant &
⟨exit-cond⟩
Post-Loop-Cond
YES
S = sum{A[1..n]}
Loop-Invariant &
¬⟨exit-cond⟩ &
Loop-code
⇒ Loop-Invariant
Post-Loop-Cond &
Post-Loop-code
⇒ Post-Cond
return S
Post-Cond: output is sum of the items in the input array


Pre-Cond : input A[1..n] is an array of numbers, n ≥0
S ← 0; t ← 0
Pre-Cond &
Pre-Loop-code
⇒ Loop-Invariant
LI changed.
Need to revise
Pre-Loop Code
or Loop Code ?
Loop Invariant: S = sum{A[1..t]}
& t ≤ n
No revision
needed.
Lucky this time!
Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder
t ≥ n
NO
t ← t+1
S ← S + A[t]
YES
Loop-Invariant &
⟨exit-cond⟩ ⇒
Post-Loop-Cond
S = sum{A[1..n]}
Loop-Invariant &
¬⟨exit-cond⟩ &
Loop-code
⇒ Loop-Invariant
Post-Loop-Cond &
Post-Loop-code
⇒ Post-Cond
return S
Post-Cond: output is sum of the items in the input array



ACTIONS

Algorithm SUM(A[1..n])
Pre-Cond: input A[1..n] is an array of numbers, n ≥0
S ← 0; t ← 0
Loop:
Loop Invariant: S = sum{A[1..t]} & t≤n
if t ≥ n **then exit loop**
t ← t+1; S ← S + A[t]
end-loop
S = sum{A[1..n]}
return S
Post-Cond: output is sum of the items in the input array
end

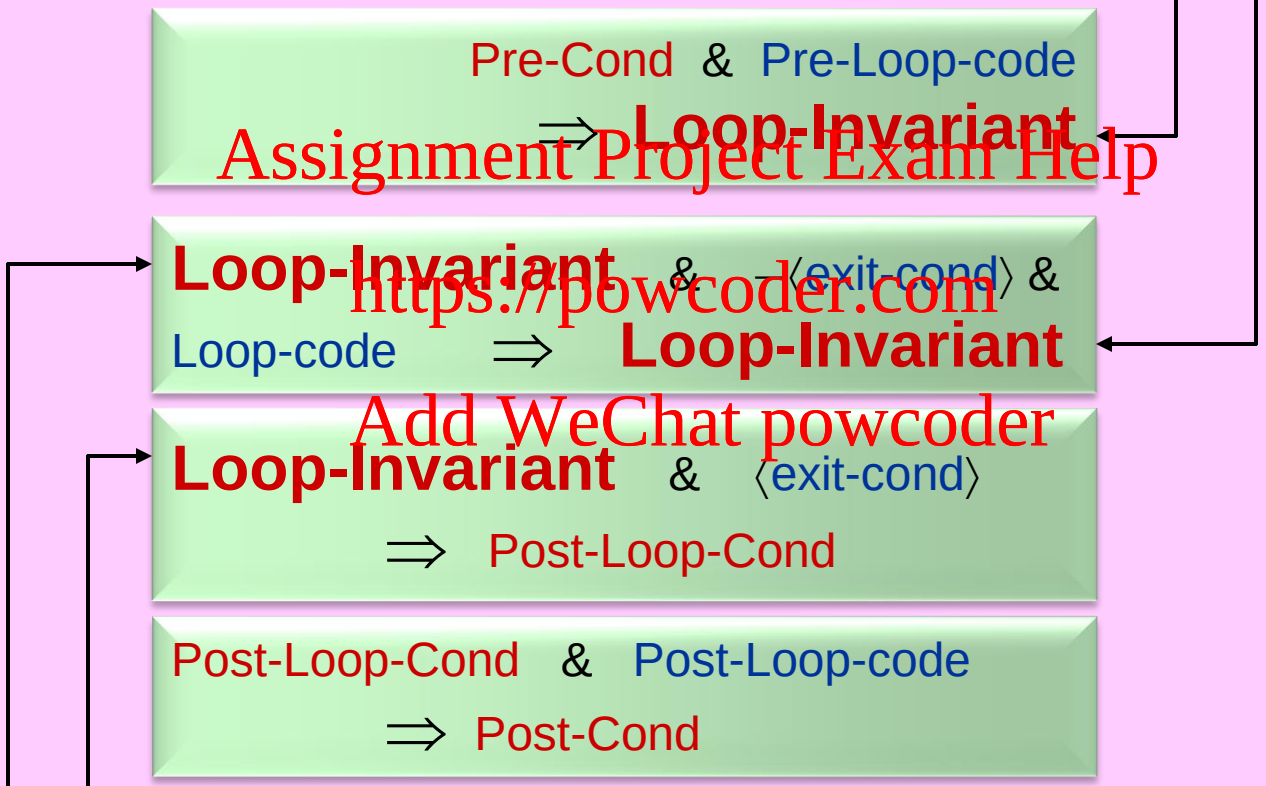
ASSERTIONS

Pre-Cond & Algorithm Code ⇒ Post-Cond

Loop-Invariant ⟵ induction hypothesis
Pre-Cond & Pre-Loop-code ⇒ Loop-Invariant ⟵ induction base
Loop-Invariant & ¬⟨exit-cond⟩ & Loop-code ⇒ Loop-Invariant ⟵ induction
Loop-Invariant & ⟨exit-cond⟩ ⇒ Post-Loop-Cond
Post-Loop-Cond & Post-Loop-code ⇒ Post-Cond ⟵ clean up loose ends

Loop Invariant may need revision

If LI is **too strong**, it may require too much effort to either establish or maintain it.



If LI is **too weak**, it may not imply the induction, or may not imply the desired end goal.

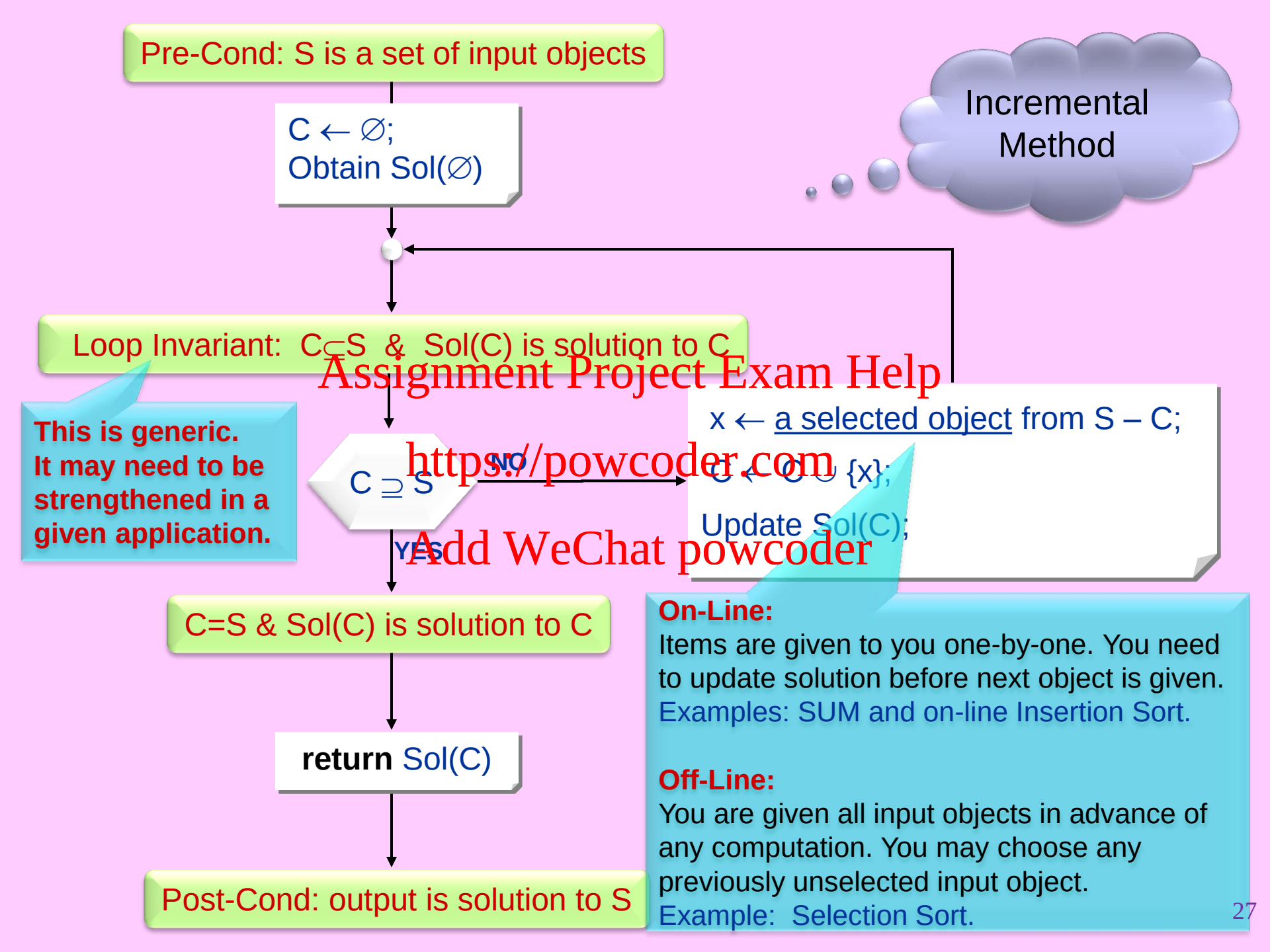
INCREMENTAL ALGORITHMS

Incremental progress over time is an exponential multiplier.
-Michael Erik

Water droplets gathered incrementally bring about a sea.
-Persian proverb

TOPICS

- **Incremental Method & its Loop Invariant**
- **Problems:**
  - **VLSI Chip Testing**
  - **In-place Tri-Partition**
  - **Maximum Sum Sub-Array**
  - **Longest Smooth Sub-Array**



Problem: VLSI Chip Testing

[CLRS Problem 4-5, page 109]

- Pre-Condition:
  Input is a set S of VLSI chips, each chip is either good or bad (unknown to us).
  Strictly more than half of these chips are good. (Note: this implies $S \neq \emptyset$.)

- Post-Condition: Output is one of the good chips from the input.

- Notation: MTHG(S) = "More Than Half of the chips in S are Good".

- Primitive operation: Boolean Test(x,y) on any pair of chips x and y.

$$\text{Test}(x,y) = \begin{cases} \text{True} & \Rightarrow \text{ x \& y are either both good or both bad} \\ \text{False} & \Rightarrow \text{ at least one of x or y is bad} \end{cases}$$

- Use this primitive operation to solve the problem.
- Comment: If good and bad chips are 50-50, Test would be unable to break the good-bad symmetry. If less than 50% are good, that's even worse!

Where to start?

- Brute Force: Compare each chip against every other chip.
  This requires $O(n^2)$ Tests.
  Can we do it faster, incrementally?

- Incremental method could be quite effective! How?
  Suppose we pick a pair of chips x and y from S and invoke Test(x,y).

- If Test(x,y) = False,
  then at least one of x or y is bad (the other is good or bad).
  We can discard both x & y (S ← S – {x,y}, and proceed with what's left).
  We still have MTHG(S). So, S still contains a good chip. There is hope !!!

- If Test(x,y) = True, then what?
  If x & y both happen to be bad, then discarding both is still safe.
  But x & y could both be good. It's unsafe to discard them both.
  Why? You no longer have the guarantee that MTHG(S)!
  You may have inadvertently discarded all the good chips!

- How should we "repair" this shortcoming?
  Strengthen the Loop Invariant. …….. P.T.O.

Strengthen the Loop Invariant

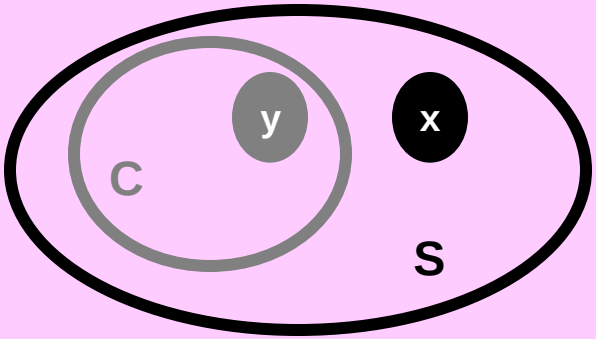
- Notation:

  MTHG(S) = “More Than Half of the chips in S are Good”.
  AllGood(C) = “chips in set C are all good”.
  Uniform(C) = “chips in set C are either all good or all bad”.

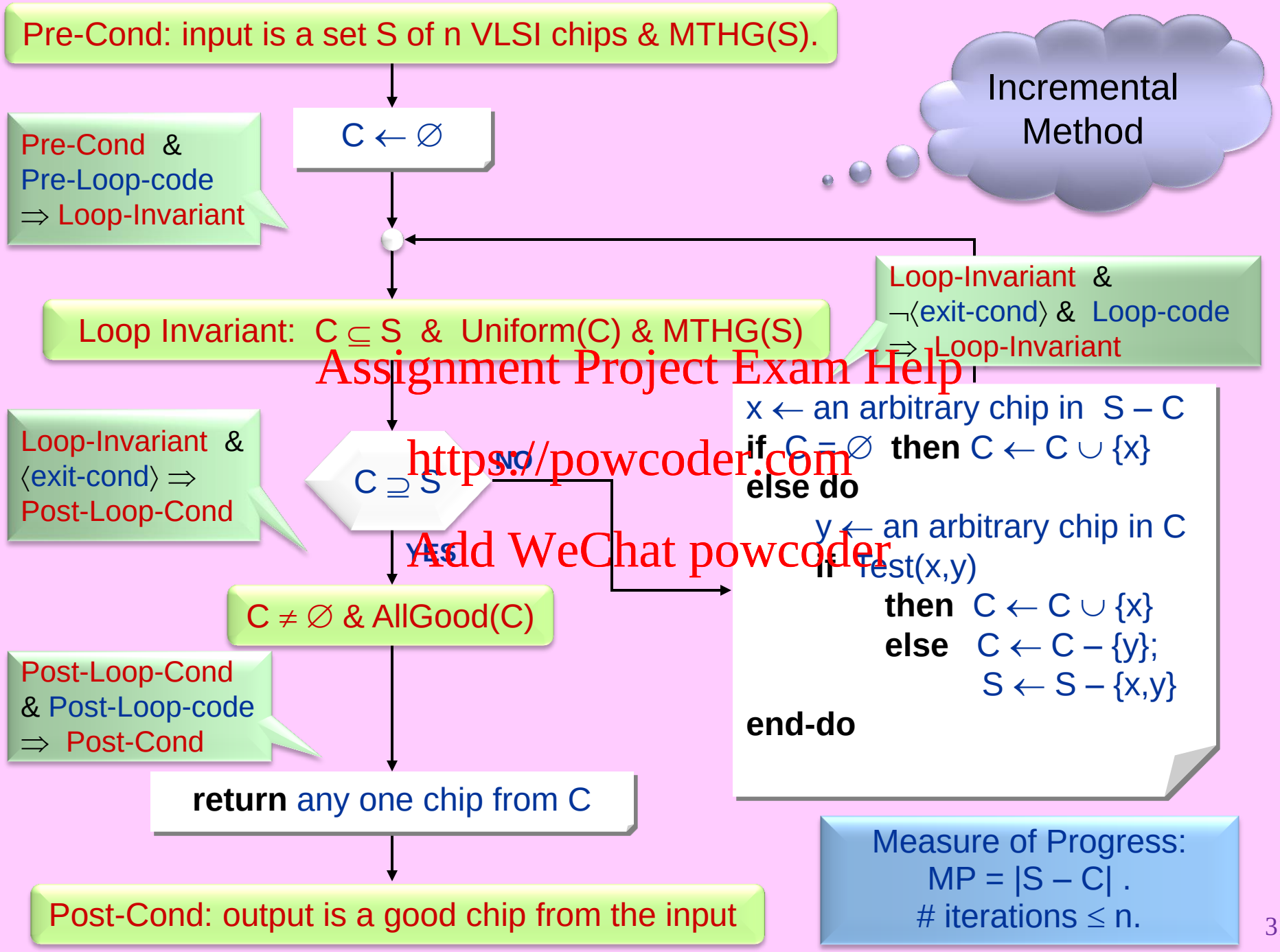
  Note: [MTHG(C) & Uniform(C)] $\Rightarrow$ [AllGood(C) & C$\neq\emptyset$].

- Strengthen Loop Invariant:
  In addition to MTHG(S),
  maintain a candidate subset C of the chips with the guarantee: Uniform(C).

- Now, how can you maintain LI and make progress?
  Idea: pick x$\in$S – C and y$\in$C, then Test(x,y).
  What happens?
  What if S – C is empty? What if C is empty?



non-candidate tested chips “discarded”


Pre-Cond: input is a set S of n VLSI chips & MTHG(S).
Incremental Method
C ← ∅
Pre-Cond & Pre-Loop-code ⇒ Loop-Invariant
Loop Invariant: C ⊆ S & Uniform(C) & MTHG(S)
Loop-Invariant & ¬⟨exit-cond⟩ & Loop-code ⇒ Loop-Invariant
Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder
Loop-Invariant & ⟨exit-cond⟩ ⇒ Post-Loop-Cond
C ⊇ S
NO
YES
x ← an arbitrary chip in S – C
if C = ∅ then C ← C ∪ {x}
else do
y ← an arbitrary chip in C
if Test(x,y)
then C ← C ∪ {x}
else C ← C – {y};
S ← S – {x,y}
end-do
C ≠ ∅ & AllGood(C)
Post-Loop-Cond & Post-Loop-code ⇒ Post-Cond
return any one chip from C
Post-Cond: output is a good chip from the input
Measure of Progress:
MP = |S – C| .
# iterations ≤ n.

Algorithm **VLSIChipTesting**(S) § takes O(n) Tests

Pre-Cond: input is a set S of n VLSI chips & MTHG(S).

```
C ← ∅
Loop:
    Loop Invariant:  C ⊆ S  &  Uniform(C) & MTHG(S)
    if  C = S  then exit loop
    x ← an arbitrary chip in  S – C
    if  C = ∅
        then C ← C ∪ {x}
        else do
            y ← an arbitrary chip in C
            if  Test(x,y)
                then  C ← C ∪ {x}
                else  C ← C – {y},  S ← S – {x,y}
        end-do
end-loop
C ≠ ∅ & AllGood(C)
return any one chip from C
```

Post-Cond: output is a good chip from the input

end

Problem: In-Place Tri-Partition

INPUT: An array A[1..n], where each item A[i] ∈ {**red, green, blue**}, i=1..n.

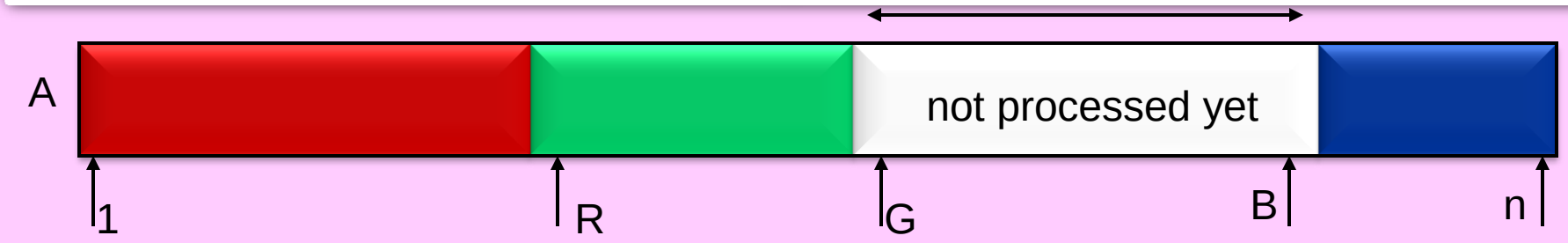
OUTPUT: A permutation of A[1..n] so that all red items appear first, then all green items, then all blue items. Same colour items can appear in any order within their colour group.

Restriction: This has to be done in-place, within array A. We are allowed to use only O(1) additional scalar variables (e.g., array indices). [We cannot use O(n) additional memory cells such as lists.]

Applications: Partitioning in QuickSort & QuickSelect, In-place bucketing, etc.

CLAIM: We can do this partitioning in O(n) time incrementally.

The Loop Invariant



Loop Invariant:

- A[1..R-1] are all reds.
- A[R..G-1] are all greens.
- A[B+1..n] are all blues.
- $1 \le R \le G \le B+1 \le n+1$.

Measure of Progress:
# unprocessed items.
MP = B – G +1.

Exit Condition: G > B.
(i.e., all items processed)

Establish Loop Invariant:
(R, G, B) ← (1, 1, n).
i.e., all n items are unprocessed.

Maintain LI & Make Progress:

Reduce MP while maintaining LI? How?

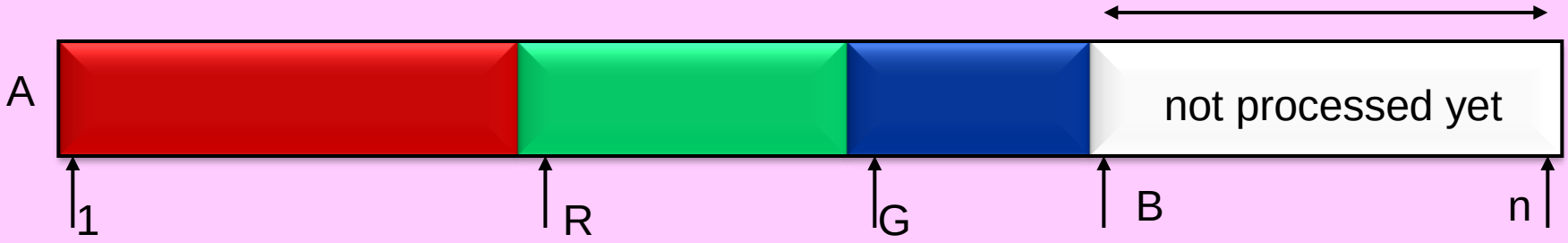
Process A[G]. What's its colour?

If A[G] = **green**
then G++

If A[G] = **blue**
then A[G] ↔ A[B]; B--

If A[G] = **red**
then A[G] ↔ A[R]; R++; G++

Another Loop Invariant



Exercise 1:

Solve the 3-partition problem in-place in O(n) time, using the alternative Loop Invariant shown above.

Exercise 2:

Generalize the 3-partition problem to 4-partition, 5-partition, …

In general, for any integer C, $2 \le C \le n$, show that the C-partition problem can be solved in O(Cn) time, using O(C) extra memory cells.

Therefore, if C is any constant, the C-partition problem can be solved in O(n) time, using O(1) extra memory cells.

Problem: Maximum Sum Sub-Array

- **INPUT:** an array A[1..n] of arbitrary real numbers (positive, zero, negative).
  OUTPUT: a contiguous sub-array, say A[i+1..j], of A[1..n] with max element sum (i.e., A[i+1] + A[i+2] + ⋯ + A[j] is max possible).

- **Example:** [2, 1, -6, -3, 2, -4, **6, -2, -1, 1, 3, -4, 7,** -5, 2, -3, 2, 1].

- **Brute Force Method:** Try every sub-array A[i+1..j], for all $0 \le i \le j \le n$.
  This can be done in $O(n^2)$ time. How?

  We can obtain sum(A[i+1..j]) in O(1) time if we first spend O(n) pre-processing time to obtain PrefixSum[0..n], where
  PrefixSum[t] = sum(A[1..t]), for t=0,1,…,n, (computed incrementally):

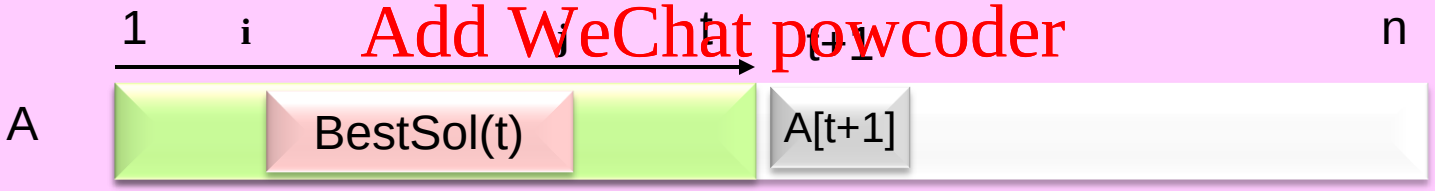
  PrefixSum[0] ← 0; **for** t ← 1..n **do** PrefixSum[t] ←PrefixSum[t-1]+A[t]

  Now, sum(A[i+1..j]) = PrefixSum[j] – PrefixSum[i].

- Can we solve the problem in sub-quadratic time incrementally?
  Let's try …… P.T.O.

Incremental Solution

- A[1..t] = prefix of A considered incrementally so far.
- Suppose maximum sum subarray of A[1..t] is A[i+1..j], $0 \le i \le j \le t \le n$.
  Let's denote that solution by:
  BestSol(t) = A[i+1..j], and SolSum(t) = sum(A[i+1..j]).
- **Loop Invariant:** BestSol(t) = A[i+1..j], $1 \le i \le j \le t$,
  SolSum(t) = sum(A[i+1..j]),
  and ... (?)



BestSol(t) & SolSum(t) & A[t+1] $\xrightarrow{?}$ BestSol(t+1) & SolSum(t+1)

Enough info in Loop Invariant?

Examine the Loop Invariant

LI(t) & A[t+1]

BestSol(t) = A[i+1..j],
SolSum(t) = sum(A[i+1..j]),
A[t+1]

Maintain LI & progress

LI(t+1)

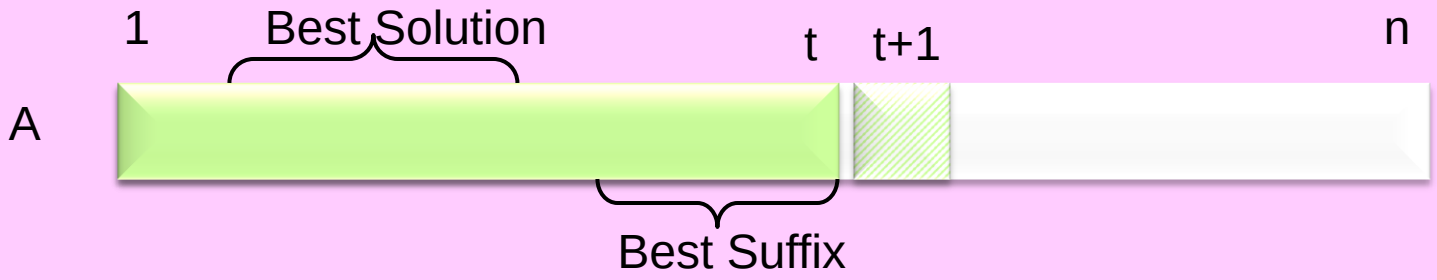
BestSol(t+1) = ?,
SolSum(t+1) = ?

which case are we in?

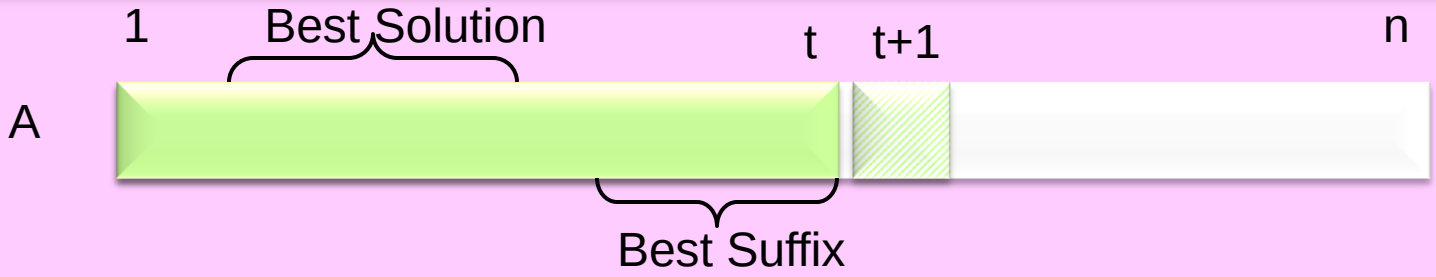
Case 1: A[t+1] is not part of BestSol(t+1):
BestSol(t+1) = BestSol(t) and
SolSum(t+1) = SolSum(t).

Case 2: A[t+1] is part of BestSol(t+1):
BestSol(t+1) = a suffix of A[1..t+1]. Which suffix?

LI is not strong enough. It should also tell us what is the maximum sum suffix of A[1..t] and its element sum.



Revise the Loop Invariant



Loop Invariant: BestSol(t) = A[i+1..j], t≤n, BestSuffix(t) = A[k+1..t],
SolSum(t) = sum(A[i+1..j]) SuffixSum(t) = sum(A[k+1..t])

maintain LI & progress

LI(t) & A[t+1] → LI(t+1)

LI(0) **establish LI**

SuffixSum(t+1) is better of two choices
(the one with bigger sum):

1) A[t+1] excluded:
   BestSuffix(t+1) = A[t+2..t+1] (i.e., nil)
   SuffixSum(t+1) = 0

2) A[t+1] included:
   BestSuffix(t+1) = (BestSuffix(t) , A[t+1])
   SuffixSum(t+1) = SuffixSum(t) + A[t+1]

BestSol(t+1) = BestSol(t) or BestSuffix(t+1),
whichever has bigger sum.

BestSol(0) ← A[1..0] (nil)
SolSum(0) ← 0

BestSuffix(0) ← A[1..0]
SuffixSum(0) ← 0

Pre-Cond: input is an array A[1..n] of arbitrary integers.

(i, j, k, t) ← (0, 0, 0, 0)
SolSum ← SuffixSum ← 0

Incremental Method

Loop Invariant:
BestSol(t) = A[i+1..j],
SolSum = sum(A[i+1..j]),
BestSuffix(t) = A[k+1..t],
SuffixSum = sum(A[k+1..t])

t ≥ n

NO

YES

```
t ← t+1
if  SuffixSum + A[t] > 0
    then  SuffixSum ← SuffixSum + A[t]
    else  SuffixSum ← 0;   k ← t
if  SolSum < SuffixSum
    then  SolSum ← SuffixSum;
          (i, j) ← (k, t)
```

BestSol(n) = A[i+1..j]

return A[i+1..j]

Post-Cond: output is max-sum-subarray of input

MP = n – t.
# iterations ≤ n.

Algorithm **MaxSumSubArray**(A[1..n]) § takes O(n) time

Pre-Cond: input is an array A[1..n] of arbitrary integers.

(i, j, k) ← (0, 0, 0) § t ← 0

SolSum ← SuffixSum ← 0

for t ← 1 .. n **do**

Loop Invariant: BestSol(t) = A[i+1..j],
SolSum = sum(A[i+1..j]),
BestSuffix(t) = A[k+1..t]
SuffixSum = sum(A[k+1..t])

if SuffixSum + A[t] > 0
then SuffixSum ← SuffixSum + A[t]
else SuffixSum ← 0; k ← t
if SolSum < SuffixSum
then SolSum ← SuffixSum;
(i, j) ← (k, t)

end-for

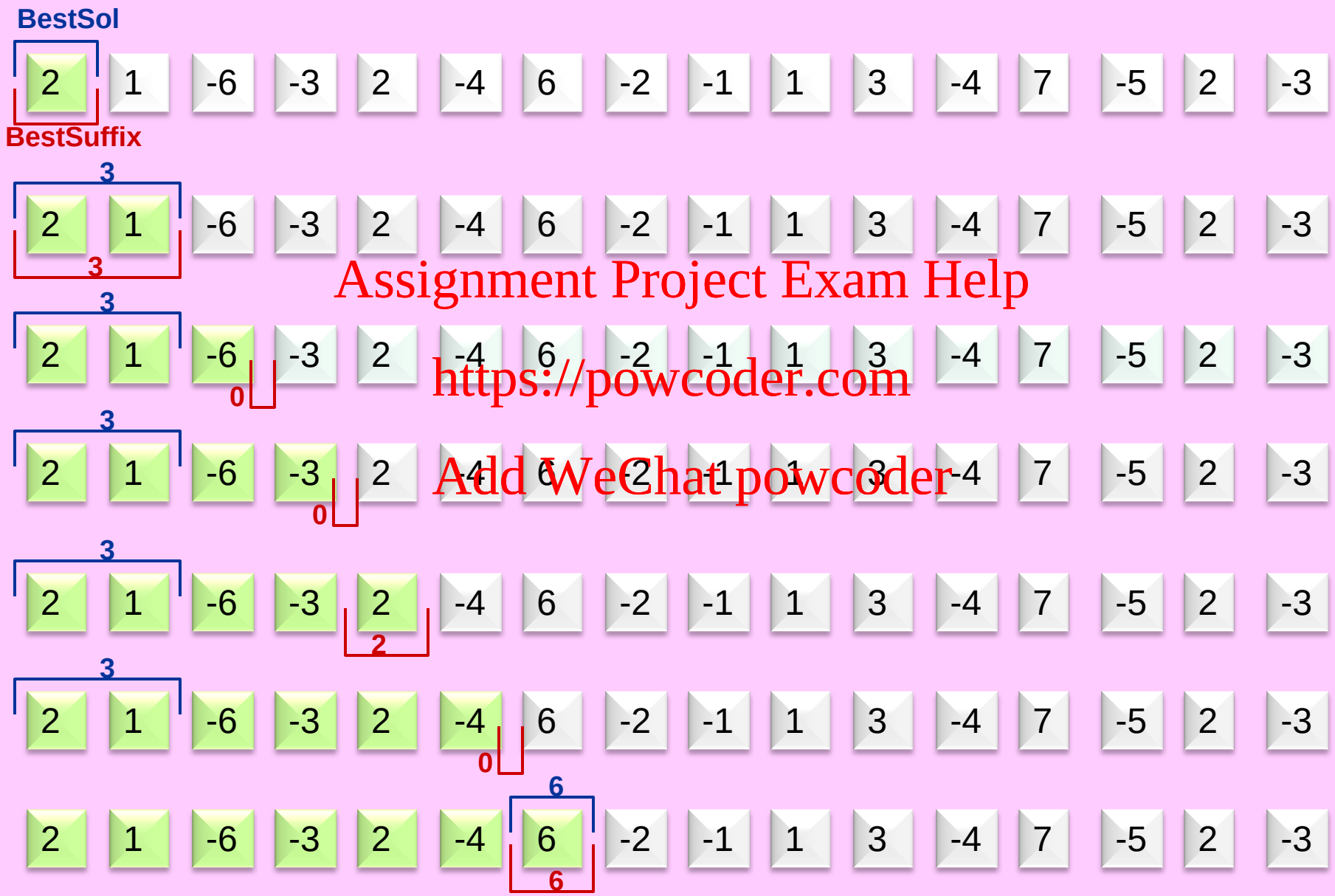
BestSol(n) = A[i+1..j]

return A[i+1..j]

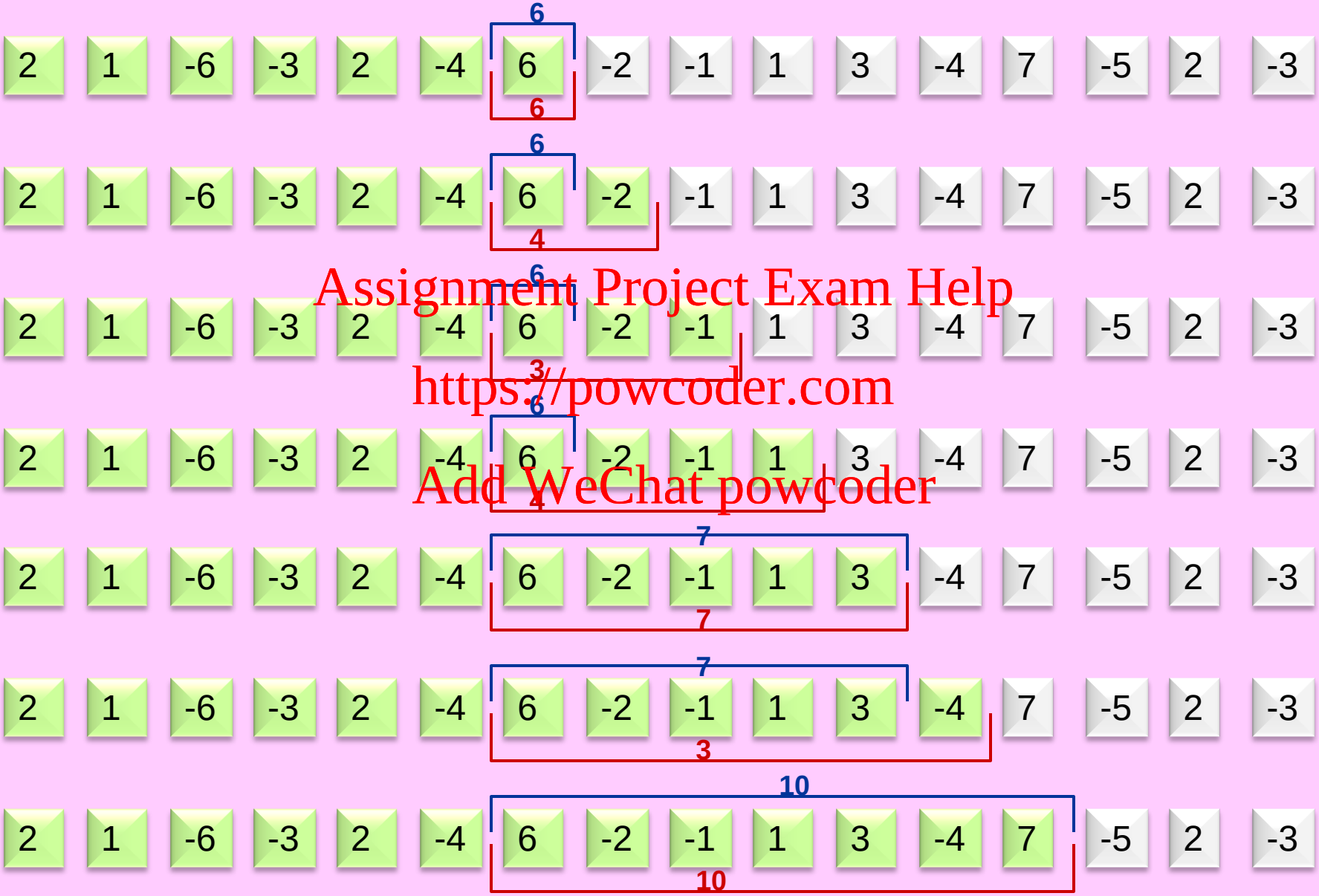
Post-Cond: output is max-sum-subarray of input

end

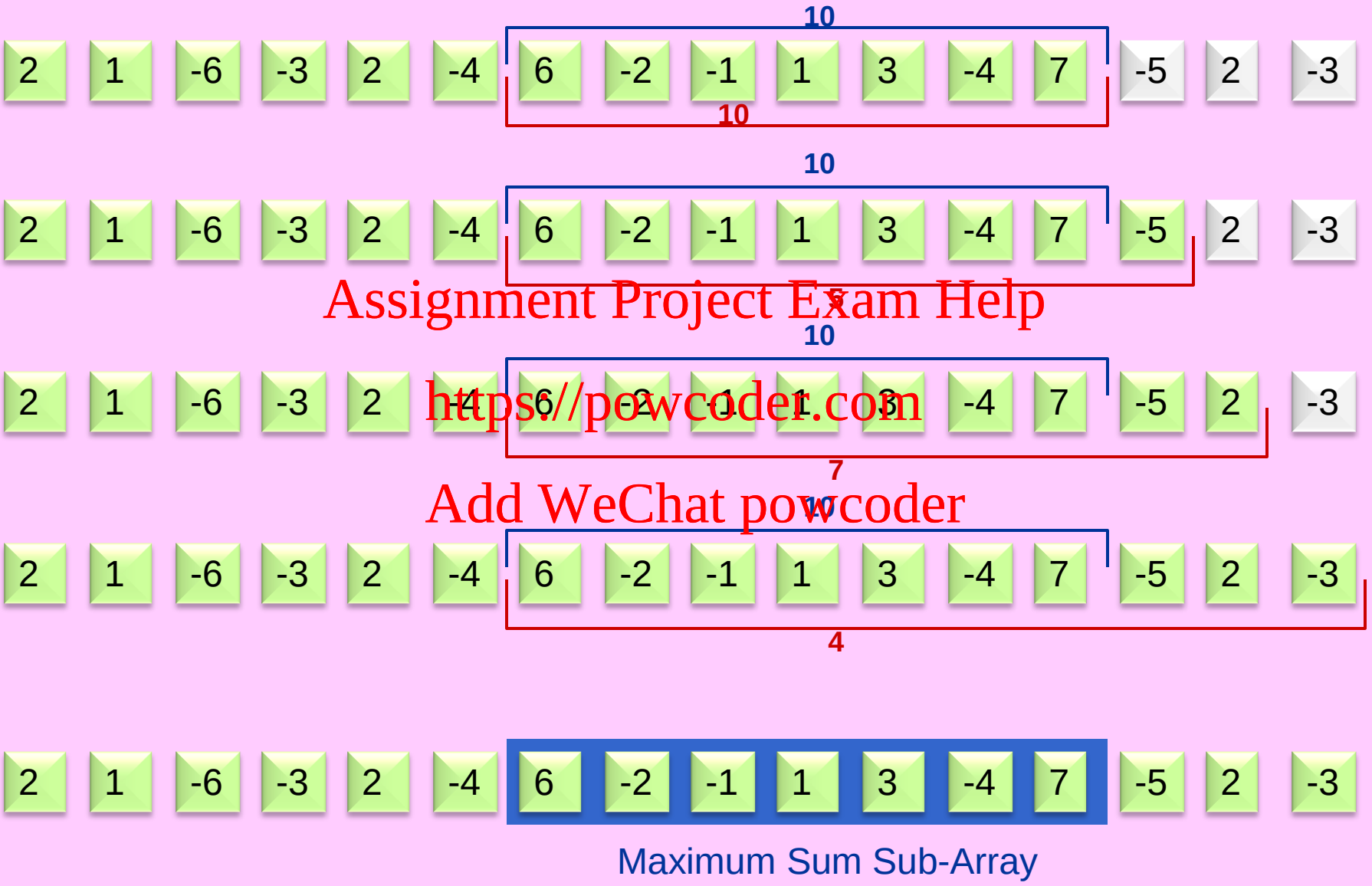
Example run of the algorithm



Example run of the algorithm



Example run of the algorithm

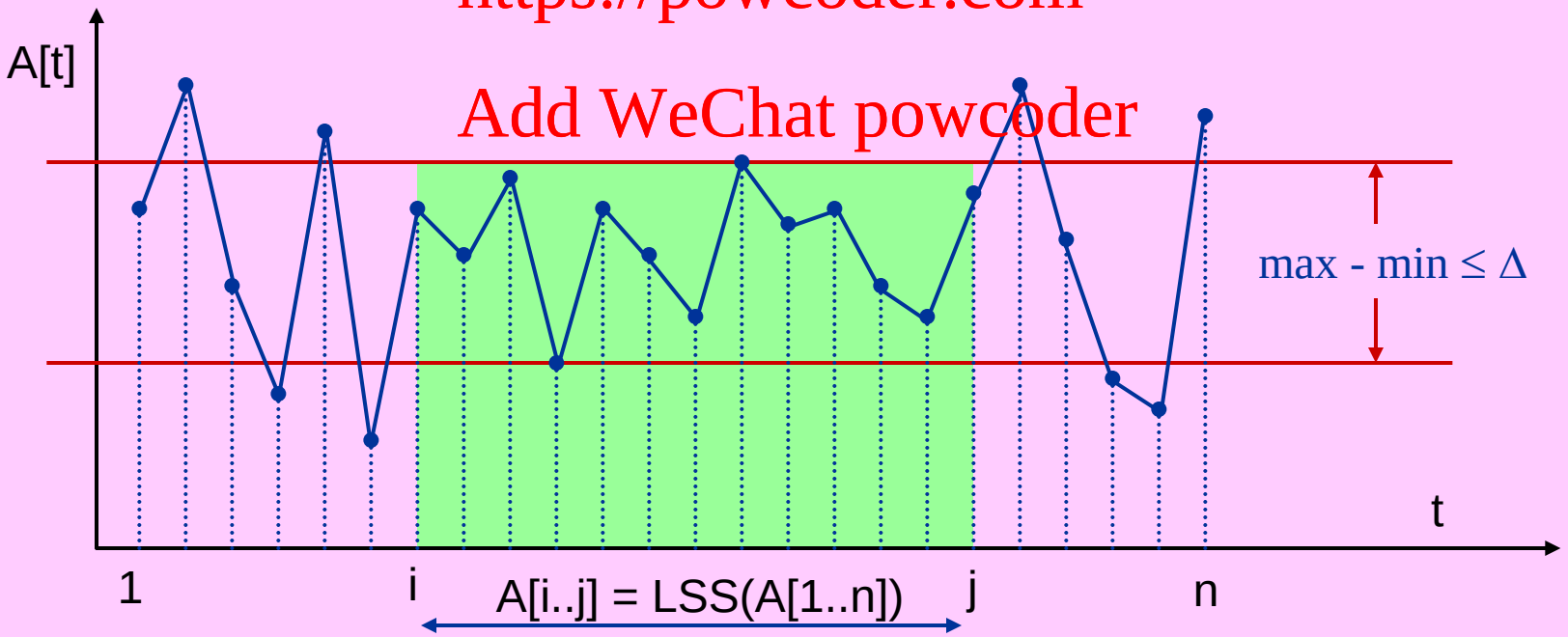


Problem: Longest Smooth Sub-Array

- Consider a real number $\Delta > 0$.
  An array S of numbers is said to be **Δ-smooth,** if no pair of items in S have a difference more than Δ, i.e., $\max(S) - \min(S) \leq \Delta$.
  Example: S = [3, 4, 9, 5, 3, 6] is 7-smooth because $\max(S) - \min(S) = 9-3 \leq 7$.

- **Longest Smooth Subarray (LSS) Problem:**
  INPUT: an array A[1..n] of numbers, and a number $\Delta > 0$.
  OUTPUT: the longest contiguous Δ-smooth subarray of A[1..n].

- **Example:** $\Delta = 8$, A = [4, -3, 7, 9, 12, 5, 9, 10, 6, 1, 6] .



Where to start?

- **Brute Force Method:** Try every subarray A[i..j], for all $1 \le i \le j \le n$.
  This can be done in $O(n^2)$ time. How?

- Can we solve the problem in sub-quadratic time?
  By the Incremental Method?

- Start with the obvious incremental LI and gradually strengthen it till it's right.

  LSS(A[1..t]) denotes the Longest Δ-Smooth Subarray of A[1..t].

- Our Loop Invariant so far includes:
  LI(a): A[i..j] = LSS(A[1..t]), $i \le j \le t \le n$.

- How do we maintain LI(a) going from A[1..t-1] to A[1..t]?

  Like MaxSumSubArray, we see we need the best suffix info too.

- LSX(A[1..t]) denotes the Longest Δ-Smooth Suffix of A[1..t].

Strengthen the Incremental Loop Invariant

- LI(a): A[i..j] = LSS(A[1..t]), $i \le j \le t \le n$.
- LSS(A[1..t]) = LSS(A[1..t-1]) if A[t] is not part of LSS(A[1..t])
  LSS(A[1..t]) = LSX(A[1..t]) if A[t] is part of LSS(A[1..t])
- Strengthen Loop Invariant
  LI(b): A[k..t] = LSX(A[1..t]), $k \le t$.
- How do we maintain LI(b)?
- Suppose LSX(A[1..t-1]) = A[k' ..t-1]. This is Δ-smooth.

  Add A[t] to it to get A[k' ..t].

  If A[k'..t] is Δ-smooth, then LSX(A[1..t]) = A[k' ..t]. Why?

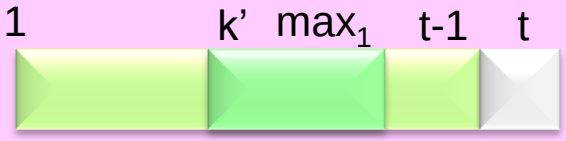
  What if A[k' ..t] is not Δ-smooth?

  …….. P.T.O.

Further Strengthen the Loop Invariant

LI(a): A[i..j] = LSS(A[1..t]), $i \le j \le t \le n$.
LI(b): A[k..t] = LSX(A[1..t]), $k \le t$.

- $A[min_1]$ = rightmost minimum of A[k' ..t-1] &
  $A[max_1]$ = rightmost maximum of A[k' ..t-1].

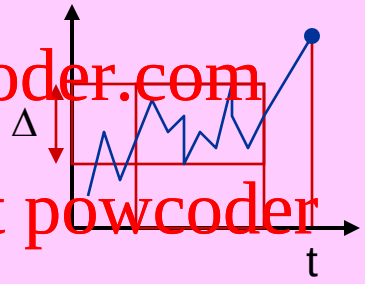


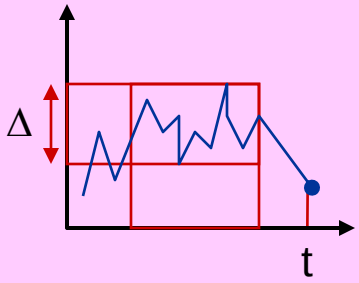
- A[k'..t-1] is Δ-smooth but A[k'..t] is not.
  So, either

$$A[t] - \Delta > A[min_1]$$

or $A[t] + \Delta < A[max_1]$,

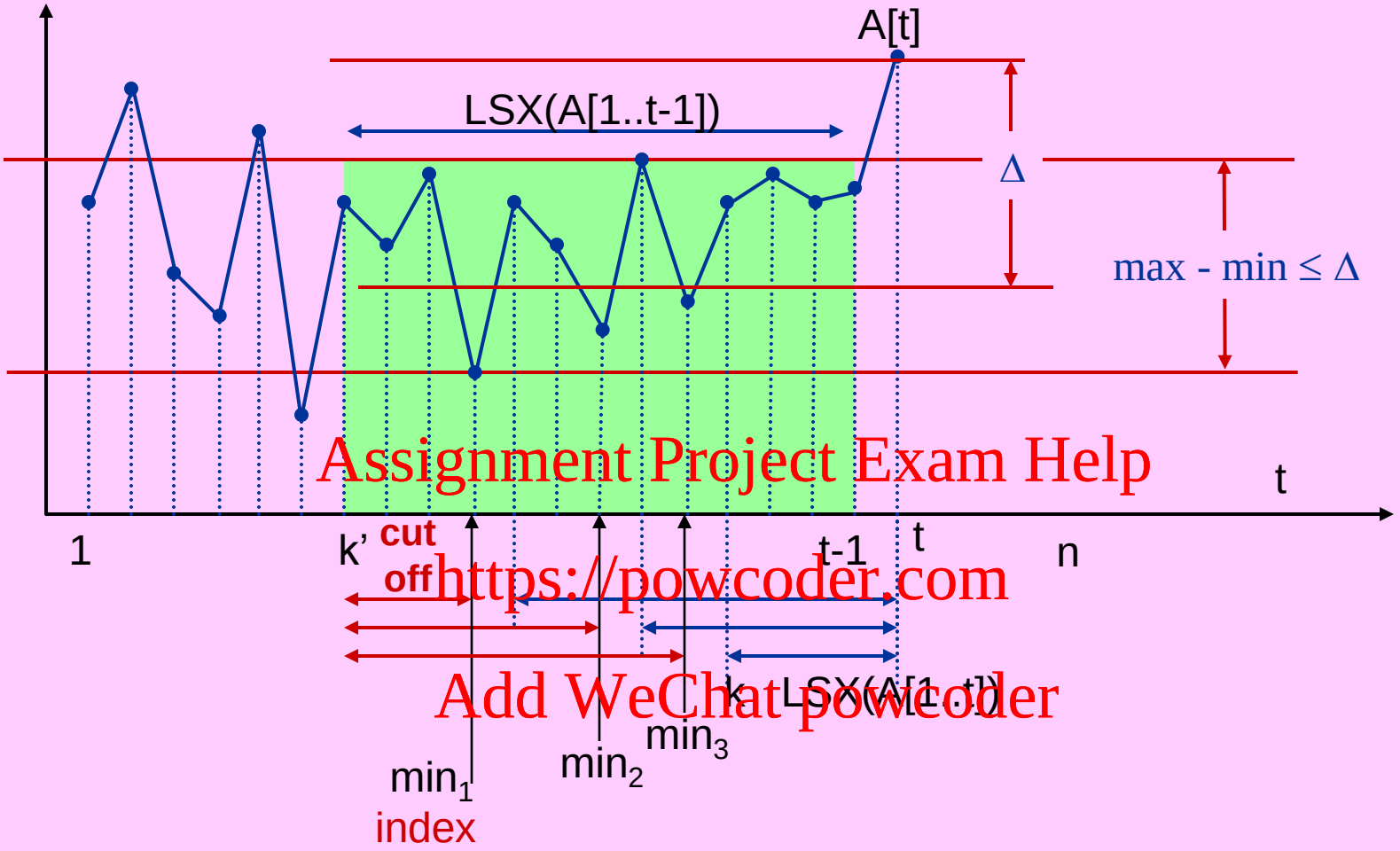
but not both.



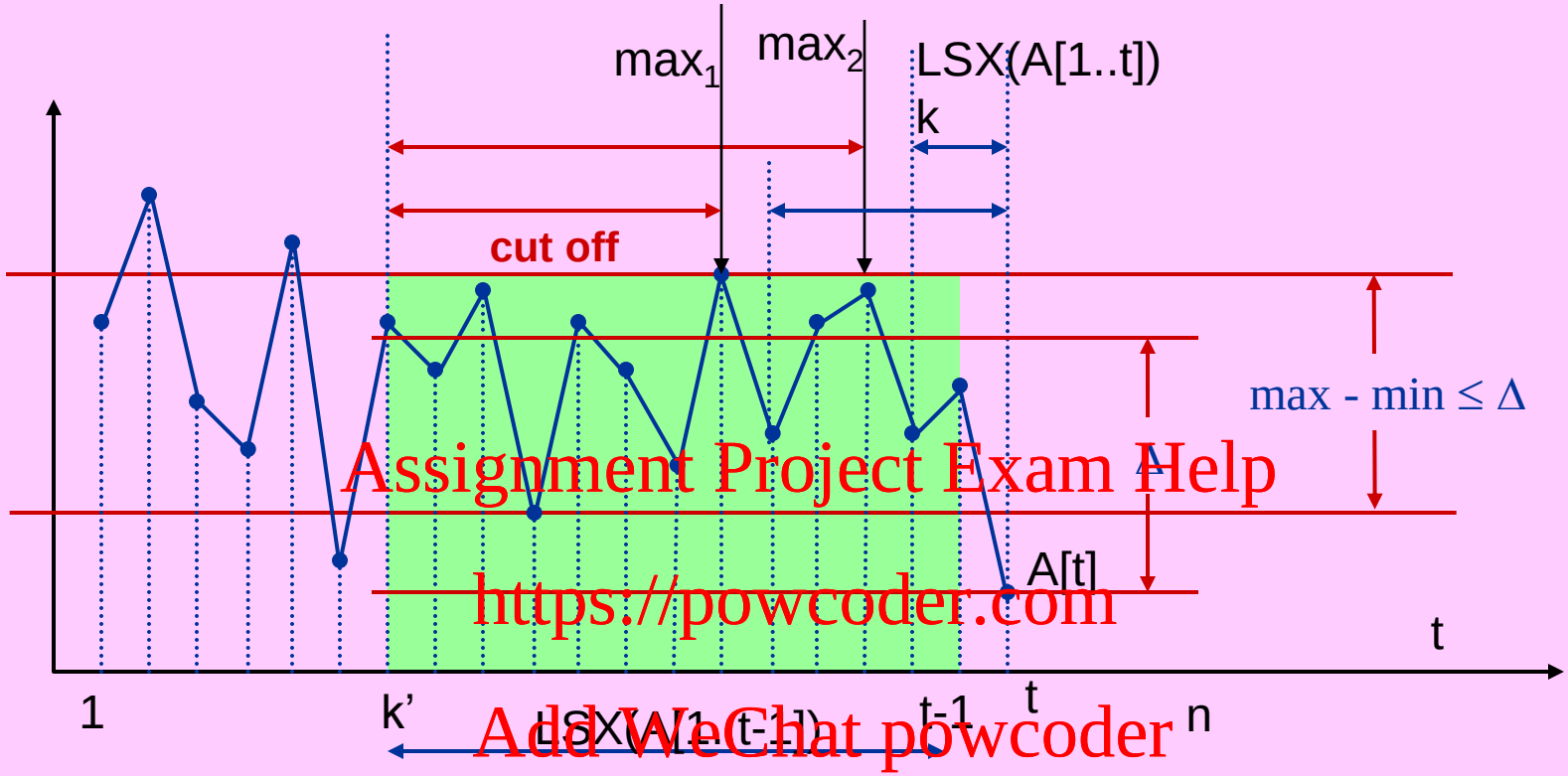


- If $A[t] - \Delta > A[min_1]$, then no part of A[k'.. min_1] can be in LSX(A[1..t]).
  If $A[t] + \Delta < A[max_1]$, then no part of A[k'.. max_1] can be in LSX(A[1..t]).
- To get to k, we upgrade k' to 1+ min_1 or 1+max_1, whichever is appropriate.

But we may not be done yet! …….. P.T.O.



$m \leftarrow 1$
while $A[t] - \Delta > A[min_m]$ **do** $k \leftarrow 1 + min_m$; $m{+}{+}$



$m \leftarrow 1$
while $A[t] + \Delta < A[max_m]$ **do** $k \leftarrow 1 + max_m$; m++

Strengthen Loop Invariant

- Strengthen LI to include these iterated min/max info.
- Two lists: MinL and MaxL.

At the end of iteration t they are:

MaxL = $\langle\ max_1, max_2, \ldots, max_q\ \rangle$

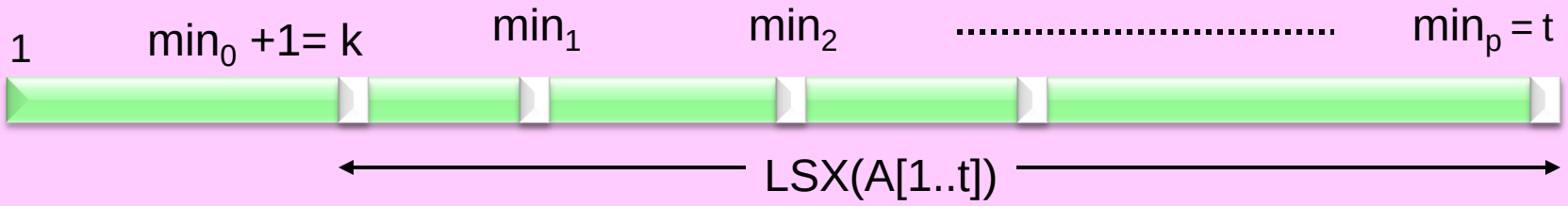
$k\text{-}1 = max_0 < max_1 < max_2 < \ldots max_{q-1} < max_q = t$, and

$A[max_m]$ = rightmost maximum of $A[max_{m-1}+1..t]$, for $m = 1..q$

MinL = $\langle\ min_1, min_2, \ldots, min_p\ \rangle$

$k\text{-}1 = min_0 < min_1 < min_2 < \ldots min_{p-1} < min_p = t$, and

$A[min_m]$ = rightmost minimum of $A[min_{m-1}+1..t]$, for $m = 1..p$.



The final Loop Invariant

LOOP INVARIANT:

LI(a): $A[i..j] = LSS(A[1..t])$, $i \le j \le t \le n$.

LI(b): $A[k..t] = LSX(A[1..t])$, $k \le t$.

LI(c): MinL and MaxL lists satisfy the specifications described below.

$MinL = \langle min_1, min_2, \ldots, min_p \rangle$

$k-1 = min_0 < min_1 < min_2 < \ldots min_{p-1} < min_p = t$, and

$A[min_m]$ = rightmost minimum of $A[min_{m-1}+1..t]$, for m = 1..p.

$MaxL = \langle max_1, max_2, \ldots, max_q \rangle$

$k-1 = max_0 < max_1 < max_2 < \ldots max_{q-1} < max_q = t$, and

$A[max_m]$ = rightmost maximum of $A[max_{m-1}+1..t]$, for m = 1..q.

Operations on lists MinL and MaxL:

- Rear(L) = return rear item in list L (lowest index). Don't alter L.
- Front(L) = return front item in list L (highest index, i.e., t). Don't alter L.
- PopRear(L) = remove and return rear item of L.
- PopFront(L) = remove and return front item of L.
- PushFront(t,L) = push item t to front of list L.
- PushRear not needed.

Maintain LI(b+c) in iteration t-1 to t

1. MinUpgrade: While A[t] - Δ > A[min_m], we need to upgrade lowest index in LSX.
2. MaxUpgrade: While A[t] + Δ < A[max_m], we need to upgrade lowest index in LSX.
3. FrontCleanUp: t is part of the new suffix & must be pushed to the front of both lists. Before doing that, remove front items in MinL (MaxL) that would no longer be rightmost minima (maxima) after t is added.

Procedure MinUpgrade
while MinL ≠ ∅ & A[t] - Δ > A[Rear(MinL)] **do** k ← 1 + PopRear(MinL)
while MaxL ≠ ∅ & Rear(MaxL) < k **do** PopRear(MaxL)

Procedure MaxUpgrade
while MaxL ≠ ∅ & A[t] + Δ < A[Rear(MaxL)] **do** k ← 1 + PopRear(MaxL)
while MinL ≠ ∅ & Rear(MinL) < k **do** PopRear(MinL)

Procedure FrontCleanUp
while MinL ≠ ∅ & A[t] ≤ A[Front(MinL)] **do** PopFront(MinL)
while MaxL ≠ ∅ & A[t] ≥ A[Front(MaxL)] **do** PopFront(MaxL)
PushFront(t, MinL)
PushFront(t, MaxL)

only 2 pushes **all Pops**

Pre-Cond: input is an array A[1..n] of numbers & $\Delta > 0$.

Incremental Method

Pre-Cond & Pre-Loop-code $\Rightarrow$ Loop-Invariant

$(i, j, k, t) \leftarrow (0, 0, 0, 0)$
$(MinL, MaxL) \leftarrow (\varnothing, \varnothing)$

LOOP INVARIANT:
LI(a): A[i..j] = LSS(A[1..t]), $i \le j \le t < n$.
LI(b): A[k..t] = LSX(A[1..t]), $k \le t$.
LI(c): MinL and MaxL as described.

Loop-Invariant & $\neg\langle$exit-cond$\rangle$ & Loop-code $\Rightarrow$ Loop-Invariant

$t \leftarrow t+1$
MinUpgrade
MaxUpgrade
FrontCleanUp
if $j - i < t - k$
then $(i, j) \leftarrow (k, t)$

Nested loops. Quadratic time ???

Loop-Invariant & $\langle$exit-cond$\rangle \Rightarrow$ Post-Loop-Cond

$t \ge n$? **NO** **YES**

LSS(A[1..n]) = A[i..j]

Post-Loop-Cond & Post-Loop-code $\Rightarrow$ Post-Cond

return A[i..j]

MP = 3(n – t) + |MinL| + |MaxL|.

Total iterations in all loops $\le 3n$.

Post-Cond: output is longest Δ-smooth subarray of A[1..n]

Algorithm LongestSmoothSubArray(A[1..n], Δ) § takes O(n) time

Pre-Cond: input is an array A[1..n] of numbers and Δ > 0.

(i, j, k) ← (0, 0, 0); (MinL, MaxL) ← (∅, ∅) § t ← 0

for t ← 1 .. n **do** **Loop Invariant:** LI(a): A[i..j] = LSS(A[1..t]), i ≤ j ≤ t ≤ n.
LI(b): A[k..t] = LSX(A[1..t]), k ≤ t.
LI(c): MinL and MaxL as described.

§ maintain LI(b) and LI(c):

§ MinUpgrade:

while MinL ≠ ∅ & A[t] -Δ > A[Rear(MinL)] **do** k ← 1 + PopRear(MinL)

while MaxL ≠ ∅ & Rear(MaxL) < k **do** PopRear(MaxL)

§ MaxUpgrade:

while MaxL ≠ ∅ & A[t]+ Δ < A[Rear(MaxL)] **do** k ← 1 + PopRear(MaxL)

while MinL ≠ ∅ & Rear(MinL) < k **do** PopRear(MinL)

§ FrontCleanUp:

while MinL ≠ ∅ & A[t] ≤ A[Front(MinL)] **do** PopFront(MinL)

while MaxL ≠ ∅ & A[t] ≥ A[Front(MaxL)] **do** PopFront(MaxL)

PushFront(t, MinL); PushFront(t, MaxL)

§ maintain LI(a):

if j – i < t – k **then** (i, j) ← (k, t)

end-for

LSS(A[1..n]) = A[i..j]

return A[i..j]

Post-Cond: output is longest Δ-smooth subarray of A[1..n].

end

Algorithm **LongestSmoothSubArray**(A[1..n], Δ) § takes O(n) time

```
(i, j, k) ← (0, 0, 0);   (MinL, MaxL) ← (∅, ∅)
for  t ← 1 .. n do
```

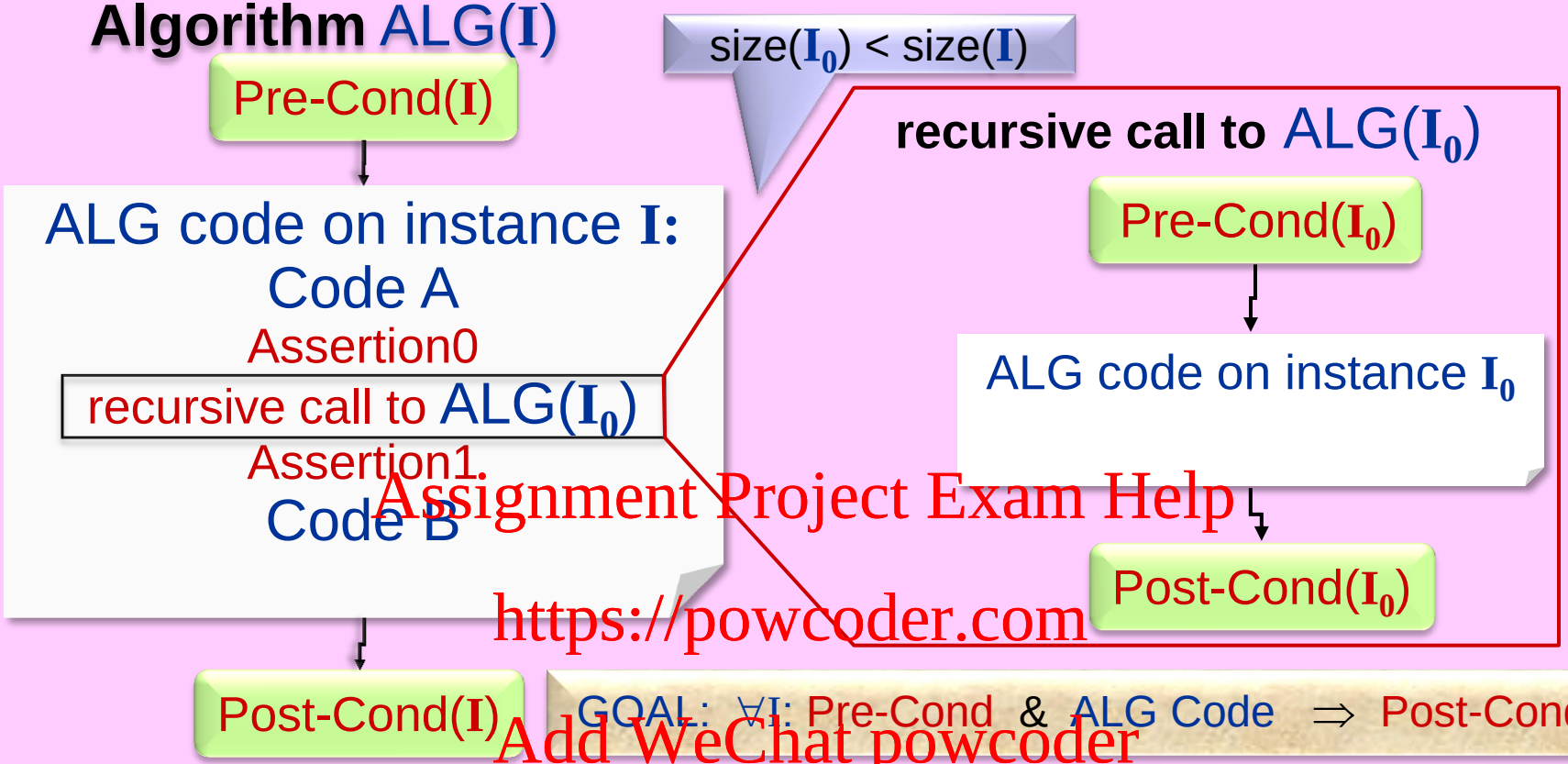
MP = 3(n – t) + |MinL| + |MaxL|

```
    while MinL ≠ ∅  &  A[t] -Δ > A[Rear(MinL)]  do k ←  1 + PopRear(MinL)
    while MaxL ≠ ∅  &  Rear(MaxL) < k  do  PopRear(MaxL)

    while MaxL ≠ ∅  &  A[t]+ Δ < A[Rear(MaxL)]  do k ←  1 + PopRear(MaxL)
    while MinL ≠ ∅  &  Rear(MinL) < k  do  PopRear(MinL)

    while MinL ≠ ∅  & A[t] ≤ A[Front(MinL)]  do  PopFront(MinL)
    while MaxL ≠ ∅  & A[t] ≥ A[Front(MaxL)]  do  PopFront(MaxL)
    PushFront(t, MinL);  PushFront(t, MaxL)

    if  j – i < t – k   then  (i, j) ← (k, t)
end-for

return A[i..j]

end
```

RECURSION

TOPICS

- **Recursive Pre- & Post- Conditions**
- **Correctness through Strong Induction**
- **Divide-&-Conquer:**
  - **Recursive Doubling:**
    - **Exponentiation**
    - **Fibonacci Numbers**
  - **Euclid's GCD algorithm**
  - **Integer & Matrix Multiplication**
  - **Closest Pair of Points in the plane**
  - **Algorithms on Binary Trees**



GOAL: $\forall I$: Pre-Cond & ALG Code $\Rightarrow$ Post-Cond

Assume strong induction hypothesis: Pre-Cond(I_0) & ALG code(I_0) $\Rightarrow$ Post-Cond(I_0)

Prove for an arbitrary instance I:

Pre-Cond & ALG code Base case $\Rightarrow$ Post-Cond $\longleftarrow$ induction base

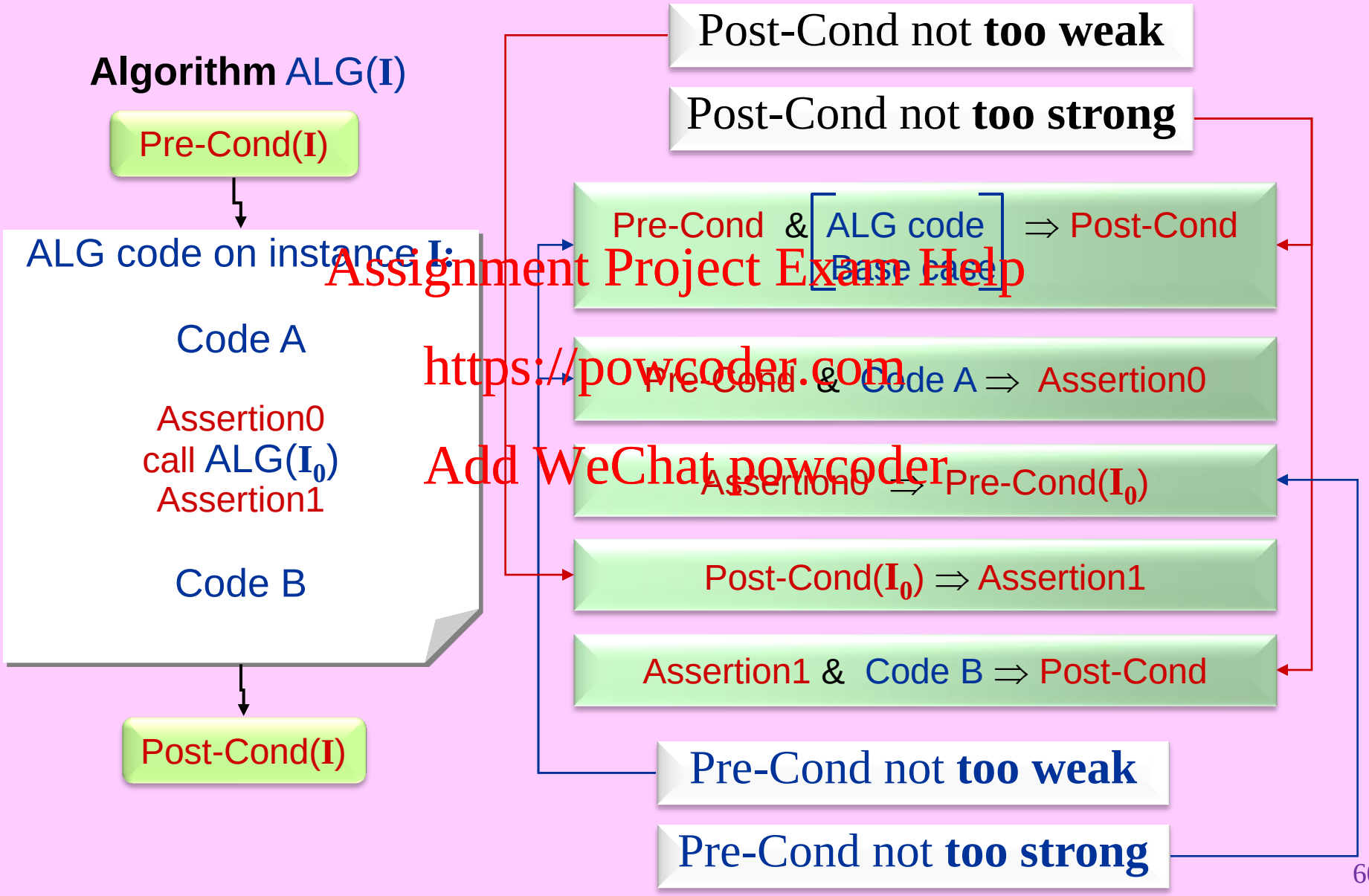
Pre-Cond & Code A $\Rightarrow$ Assertion0

Assertion0 $\Rightarrow$ Pre-Cond(I_0)

Post-Cond(I_0) $\Rightarrow$ Assertion1

Assertion1 & Code B $\Rightarrow$ Post-Cond

Pre-Cond, Post- Cond, or code may need revision



DIVIDE & CONQUER

Divide et impera.
- Niccolò Machiavelli [1469-1527]

The Divide-&-Conquer Method

MergeSort is the prototypical divide-&-conquer algorithm.

Algorithm ALG(I)

Pre-Condition: input is instance I

Post-Condition: output is Sol(I)

Base: **if** |I| = O(1) **then return** Sol(I) by the base method

Divide: Divide I into sub-instances $I_1, I_2, \ldots, I_a$, each of size |I|/b

Conquer: **for** i ← 1 .. a **do** $Sol_i \leftarrow$ **ALG**(I_i)

Combine: Sol ← **Combine**($Sol_1, Sol_2, \ldots, Sol_a$)

Output: **return** Sol

end

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + f(n) & \forall n > \Omega(1) \\ \Theta(1) & \forall n \le O(1) \end{cases}$$

What Number did I guess?

Problem 1:

I guess an integer x between 1 and n. You find x.
You can ask me questions of the form “is x=a?”, “is x > a?”, “is x < a?”, for any number a you choose.
What’s your method? How many questions do you need to ask?

Do binary search on integers between 1 and n. O(log n) questions.

Problem 2:

I guess a positive integer x. You find x.
You can ask me questions of the form “is x=a?”, “is x > a?”, “is x < a?”, for any number a you choose.
What’s your method? How many questions do you need to ask?

Starting from 1, repeatedly double to get an upper-bound n on x.
Then binary search between 1 and n. O(log x) questions.

Recursive Doubling

Example: Given $X \in \Re - \{0\}$ and $n \in \mathcal{N}$, compute X^n using arithmetic.

Method 1: Multiply n copies of X: $X \cdot X \cdot X \cdots X \cdot X \cdot X$.
That takes O(n) arithmetic operations.
That's exponential in the bit length of input n.

Method 2:

Starting with X, repeatedly multiply the result by itself. We get:

$$X,\ X^2,\ X^4,\ X^8,\ X^{16},\ X^{32},\ \ldots$$

Suppose we want to compute X^{13}.

$$13 = 8 + 4 + 1. \quad \text{So, } X^{13} = X^8 \cdot X^4 \cdot X.$$

X^n can be computed this way in O(log n) arithmetic operations. How?
Just figure out n in binary bits.
It's simpler to **look backwards**. Divide n by 2 and let r be the remainder:

$$n = 2 \cdot \left\lfloor \tfrac{n}{2} \right\rfloor + r, \quad r \in \{0,1\}.$$

$$x^n = x^{2\lfloor n/2 \rfloor + r} = \left(x^2\right)^{\lfloor n/2 \rfloor} \cdot x^r = y^m \cdot x^r$$

$$\text{where } y = x \cdot x \text{ and } m = \left\lfloor \tfrac{n}{2} \right\rfloor.$$

X^n by Recursive Doubling

Algorithm **EXP**(X, n)
Pre-Cond: $X \in \Re - \{0\}$, $n \in \mathcal{N}$.
    if n=0 **then return** 1
    $Y \leftarrow X \cdot X$
    $Z \leftarrow EXP(Y, \lfloor n/2 \rfloor)$
    if n mod 2 = 1 **then** $Z \leftarrow Z \cdot X$
    return Z
Post-Cond: returns X^n.
end

$$n = 2 \cdot \left\lfloor \frac{n}{2} \right\rfloor + r, \quad r \in \{0,1\}.$$

$$x^n = x^{2\lfloor n/2 \rfloor + r} = y^{\lfloor n/2 \rfloor} \cdot x^r$$

$$y = x^2$$

$$r = n \bmod 2.$$

$$T(n) = \begin{cases} T(n/2) + \Theta(1) & \forall n > 0 \\ \Theta(1) & n = 0 \end{cases} \Rightarrow \quad T(n) = \Theta(\log n).$$

Fibonacci Numbers

Fibonacci Numbers:

$F_0 = 0,\ F_1 = 1,$ $\qquad F_{n+2} = F_{n+1} + F_n$ for all $n \geq 0$.

n:	0	1	2	3	4	5	6	7	8	9	10	11	12	…
F_n :	0	1	1	2	3	5	8	13	21	34	55	89	144	…

$$x^{n+2} = x^{n+1} + x^n, x \neq 0 \Leftrightarrow x^2 = x + 1.$$

Two roots :

golden ratio $\phi = \frac{1+\sqrt{5}}{2} = 1.61803\cdots,$

and $\hat{\phi} = \frac{1-\sqrt{5}}{2} = -0.61803\cdots.$

$$F_n = \frac{1}{\sqrt{5}}\left(\varphi^n - \hat{\varphi}^n\right)$$

$$F_n = a \cdot \varphi^n + b \cdot \hat{\varphi}^n$$

Now choose a and b to satisfy the boundary conditions $F_0 = 0,\ F_1 = 1.$

$$F_n = \Theta\left(\varphi^n\right)$$

Method 1: the recurrence formula

```
Algorithm Fib(n)
    if n∈{0,1} then return n
    return Fib(n-1) + Fib(n-2)
end
```


$$T(n) = \begin{cases} T(n-1) + T(n-2) + \Theta(1) & n \geq 2 \\ \Theta(1) & n = 0,1 \end{cases}$$

$$\Rightarrow \quad T(n) = \Theta(F_n) = \Theta(\phi^n).$$

[Use guess-&-verify]

Cons: This is <u>double-exponential</u> time in the input n's bit length!

Method 2: incremental

```
Algorithm Fib(n)
    (x,y) ← (0,1)    § (x,y)= (F_0,F_1 )
    for t ← 2 .. n do
        z ← x + y
        x ← y
        y ← z
        Lt:  (x, y)= (F_{t-1}, F_t )
    end-for
    return y
end
```


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

Cons: This is <u>exponential</u> time in the input n's bit length!

Method 3: recursive doubling

$$F_n = \frac{1}{\sqrt{5}}\left(\varphi^n - \hat{\varphi}^n\right)$$

```
Algorithm Fib(n)
        X← EXP( (1 +√5)/2 , n)
        Y← EXP( (1 –√5)/2 , n)
      return (X –Y)/√5
end
```


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

Pros: This is linear in the input n's bit length!

Cons:

- What if we didn't know the formula?
- It uses non-integer real arithmetic & √ even though F_n is always an integer!

Is there a way to do integer-only arithmetic to avoid "round off" errors?

Method 4: A^n recursive doubling

$$\begin{aligned} F_{n+1} &= F_n + F_{n-1} \\ F_n &= F_n \end{aligned}$$

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$$

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix}$$

$$\underbrace{\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}}_{G(n)} = \underbrace{\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix}}_{G(n-1)}$$

$$G(1) = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = A$$

$$G(n) = A \cdot G(n-1) = A^2 \cdot G(n-2) = \cdots = A^{n-1} \cdot G(1) = A^n .$$

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \qquad \forall\, n \geq 1$$

EXP(A , n) on 2-by-2 matrices!
O(log n) time integer-only arithmetic!

Greatest Common Divisor

Input: a, b $\in \mathcal{N}$, not both zero.

Output: Greatest Common Divisor of a and b:
GCD(a,b) = max { d$\in \mathcal{N}$: d|a and d|b }.

d|a means "d divides a", i.e., a is an integer multiple of d.

Elementary School Method:


```
if min{a,b} = 0 then return max{a,b}
for d ← min{a,b} down-to 1 do
        if d|a & d|b then return d
end
```


Running time = O(1 + min{a,b}).

This is exponential time in the # bits in the input.

Try GCD(1000,000,000,001, 1000,000,000,000).

Euclid's GCD Algorithm

FACT 0: d|a & d|b $\Leftrightarrow$ d|GCD(a,b).

FACT 1: b = 0 $\Rightarrow$ GCD(a,b) = a.

Now suppose b ≠ 0. Divide a by b.
Let q = quotient, r = remainder:

$$a = q \cdot b + r, \quad 0 \le r \le b-1.$$

FACT 2: GCD(a,b) = GCD(b,r).

Proof: $\forall$d: d|a & d|b $\Leftrightarrow$ d|b & d|r.

FACT 3:

(a) a < b $\Rightarrow$ q = 0, r = a. So, (b,r) = (b,a).
(b) a = b $\Rightarrow$ q = 1, r = 0. So, (b,r) = (b,0).
(c) a > b $\Rightarrow$ a ≥ b + r > 2r.

$r_0 = a$, $r_1 = b$,
For all i, such that $r_{i+1} \neq 0$:
$r_i = q_i\, r_{i+1} + r_{i+2}$, $\quad 0 \le r_{i+2} \le r_{i+1} - 1$.

Algorithm GCD(a,b)
  if b=0 **then return** a
  return GCD(b, a mod b)
end

$(r_0, r_1) = (a, b)$
(r_1, r_2)
(r_2, r_3)
(r_3, r_4)
…
(r_{k-1}, r_k), $\quad r_i \neq 0, i \le k$
(r_k, r_{k+1}), $\quad r_{k+1} = 0$
r_k = GCD(a,b)

FACT 4:

(a) For all i>0: $r_{i+2} < \frac{1}{2} r_i$.
(b) $k < 2 + 2 \log \min\{a,b\}$.
(c) Running time of GCD
$= O(1 + \log \min\{a,b\})$.

More on GCD

FACT 5: The following are equivalent:

(1) D = GCD(a,b)

(2) D = max {$d \in \mathcal{N}$: d|a and d|b }.

(3) D = min {ax + by > 0 : x & y are integers}.

Proof: (1) $\Leftrightarrow$ (2) by definition. We show (1) $\Leftrightarrow$ (3) as follows:

Let D = GCD(a,b) and D' = min {ax + by > 0 : x & y are integers}.

D|a & D|b $\Rightarrow$ D|(ax+by) $\Rightarrow$ D|D' $\Rightarrow$ D $\leq$ D'.

D $\geq$ D' is implied by the claim below. Therefore, D = D'.

CLAIM: D = GCD(a,b) = ax + by, for some integers x and y.

Proof of Claim: By induction on the recursion depth of Euclid's algorithm.

Basis: If b=0, then GCD(a,b) = a = 1a + 0b.

Induction: If b$\neq$ 0, let a = qb+r.
By induction hypothesis GCD(b, r) = bx'+ ry', for some integers x' & y'.
Therefore, GCD(a,b) = bx'+ry' = bx'+(a-qb)y' = ay'+(x' –qy')b = ax+by
with x = y' and y = x' – qy'.

More on GCD

FACT 5: The following are equivalent:

(1) D = GCD(a,b)

(2) D = max $\{d \in \mathcal{N} : d|a \text{ and } d|b\}$.

(3) D = min $\{ax + by > 0 : x \text{ \& } y \text{ are integers}\}$.

FACT 6: For (possibly negative) integers a and b,
GCD(a,b) = GCD(|a| , |b|).

FACT 7: The equation $ax + by = 1$
with given non-zero integers a and b, has integer solutions for x and y
if and only if a and b are relatively prime, i.e., GCD(a,b) = 1.

FACT 8: The equation $ax + by = c$
with given non-zero integers a, b, c, has integer solutions for x and y
if and only if GCD(a,b)|c.

Exercise:
Show that Euclid's algorithm can be adapted to produce such a solution.

n-bit Integer Addition vs Multiplication

$x = x_{n-1}x_{n-2} \cdots x_1x_0$ compute $x + y$

$y = y_{n-1}y_{n-2} \cdots y_1y_0$ and x * y

FACT 0: The bit-complexity of n-bit addition or multiplication is $\Omega(n)$.

Proof: A correct algorithm must "look" at every input bit.
Suppose on non-zero inputs, input bit b is not looked at by the algorithm.
Adversary gives the algorithm the same input, but with bit b flipped.
Algorithm is oblivious to b, so it will give the same answer.
It can't be correct both times!

Elementary School Addition Algorithm has $O(n)$ bit-complexity:

```
  XXXXXXXXXX
+ XXXXXXXXXX
------------
 XXXXXXXXXXX
```

FACT 1: The bit-complexity of n-bit addition is $\Theta(n)$.

n-bit Integer Multiplication

$x = x_{n-1}x_{n-2} \cdots x_1x_0$ compute $x + y$

$y = y_{n-1}y_{n-2} \cdots y_1y_0$ and x * y

Elementary School Multiplication Algorithm has $O(n^2)$ bit-complexity:

```
   XXXX
 * XXXX
 ------
  XXXXX
 XXXX
XXXX
XXXX
 ------
XXXXXXXX
```


QUESTION: What is the bit-complexity of n-bit multiplication?

n-bit Integer Multiplication by Divide-&-Conquer

$x = x_{n-1}x_{n-2} \cdots x_1x_0$ compute $x + y$

$y = y_{n-1}y_{n-2} \cdots y_1y_0$ and x * y

X = 3141
Y = 5927

X*Y = 18,616,707

X = 3100 + 41 = a • 100 + b
Y = 5900 + 27 = c • 100 + d

X*Y =(a*c) • 10000 + (a*d + b*c) • 100 + b*d

= (31*59) • 10000 + (31*27 + 41*59) • 100 + 41*27

= 1829 • 10000 + (837 + 2419) • 100 + 1107

= 1829 • 10000 + 3256 • 100 + 1107

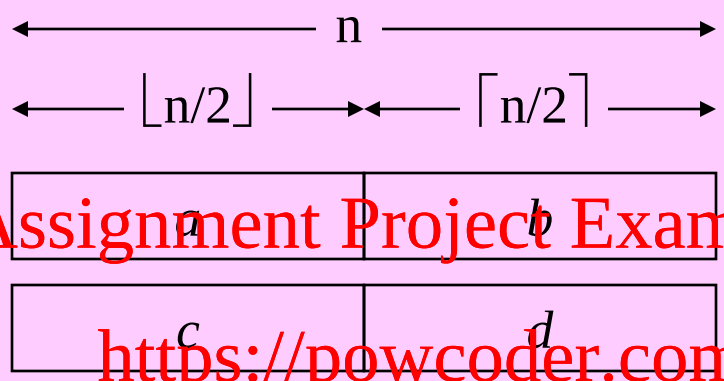
= 18290000 + 325600 + 1107

= 18,616,707

n-bit Integer Multiplication by Divide-&-Conquer

$x = x_{n-1}x_{n-2} \cdots x_1x_0$ compute $x + y$

$y = y_{n-1}y_{n-2} \cdots y_1y_0$ and x * y



$$x * y = (a * c) \cdot 2^{2\lceil n/2 \rceil} + (a * d + c * b) \cdot 2^{\lceil n/2 \rceil} + (b * d)$$

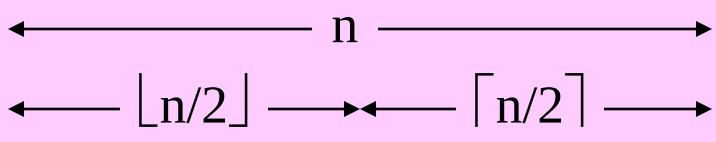
Shift/Add/Subtract $\Theta(n)$ time

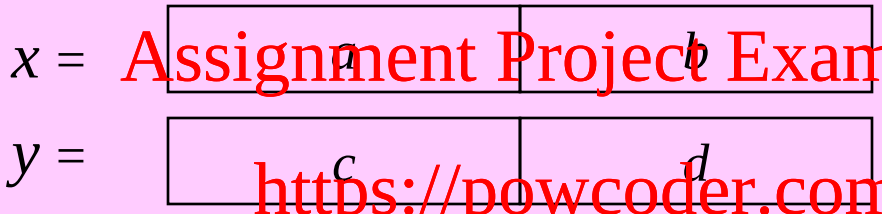
$$T(n) = 4T(n/2) + \Theta(n) \quad \Rightarrow T(n) = \Theta\left(n^{\log_2 4}\right) = \Theta\left(n^2\right)$$

n-bit Integer Multiplication by Divide-&-Conquer

$x = x_{n-1}x_{n-2} \cdots x_1x_0$ compute $x + y$

$y = y_{n-1}y_{n-2} \cdots y_1y_0$ and $x * y$





$$x * y = (a * c) \cdot 2^{2\lceil n/2 \rceil} + (a * d + c * b) \cdot 2^{\lceil n/2 \rceil} + (b * d)$$

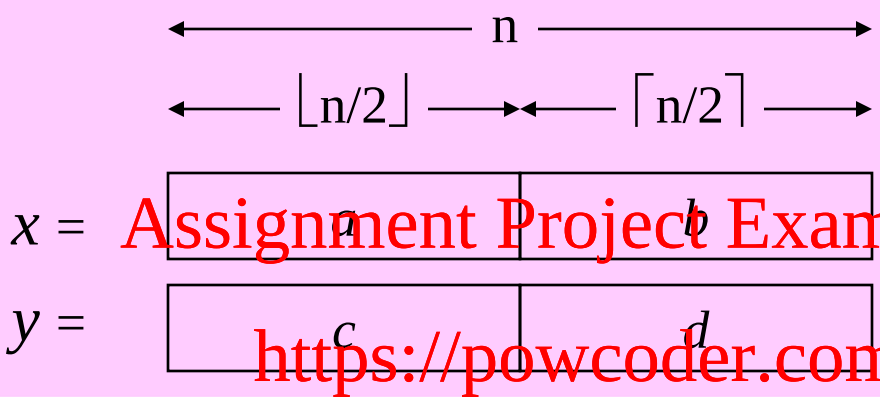
d/Subtract $\Theta(n)$ time

$$(a+b)*(c+d) = (a*c) + (a*d+c*b) + (b*d)$$

n-bit Integer Multiplication by Divide-&-Conquer

$x = x_{n-1}x_{n-2} \cdots x_1x_0$ compute $x + y$

$y = y_{n-1}y_{n-2} \cdots y_1y_0$ and $x * y$



Divide-&-Conquer Multiplication by Karatsuba, Ofman [1962]:

u ← (a + b) * (c + d)

v ← a * c

w ← b * d

xy ← v • $2^{2\lceil n/2 \rceil}$ + (u – v – w) • $2^{\lceil n/2 \rceil}$ + w

return xy

Karatsuba-Ofman's discovery disproved Andrey Kolmogorov's 1956 conjecture that multiplication requires $\Omega(n^2)$ bit operations.

$$T(n) = 3T(n/2) + \Theta(n) \quad \Rightarrow T(n) = \Theta\left(n^{\log_2 3}\right) = \Theta\left(n^{1.585\cdots}\right)$$

n-bit Integer Multiplication by Divide-&-Conquer

$x = x_{n-1}x_{n-2} \cdots x_1x_0$ compute $x + y$

$y = y_{n-1}y_{n-2} \cdots y_1y_0$ and x * y

X = 3141
Y = 5927

X*Y = 18,616,707

X = 3100 + 41 = a • 100 + b
Y = 5900 + 27 = c • 100 + d

u = (a+b)*(c+d) = (31+41)*(59+27) = 72 * 86 = 6,192

v = a*c = 31 * 59 = 1,829

w = b*d = 41 * 27 = 1,107

X*Y = v • 10000 + (u – v – w) • 100 + w

= 1829 • 10000 + (6192 –1829 –1107) • 100 + 1107

= 18,616,707

Complexity of n-bit Integer Multiplication

Known Upper Bounds:

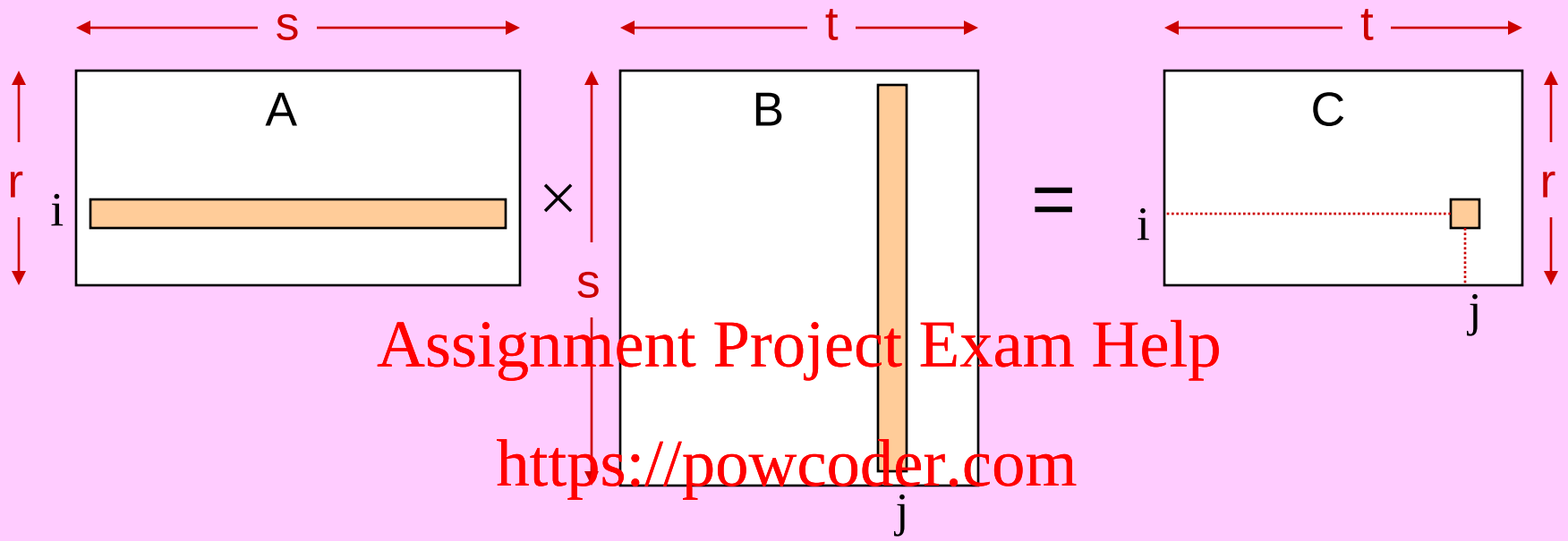
- $O(n^{\log 3}) = O(n^{1.59})$ by divide-&-conquer [Karatsuba-Ofman, 1962]
- $O(n \log n \log \log n)$ by FFT [Schönhage-Strassen, 1971]
- $O(n \log n \; 8^{\log^* n})$ [David Harvey et al., 2016]

Known Lower Bound:

- $\Omega(n \log n / \log \log n)$ [Fischer-Meyer, 1974]

FFT = Fast Fourier Transform [CLRS chapter 30]

Matrix Multiplication

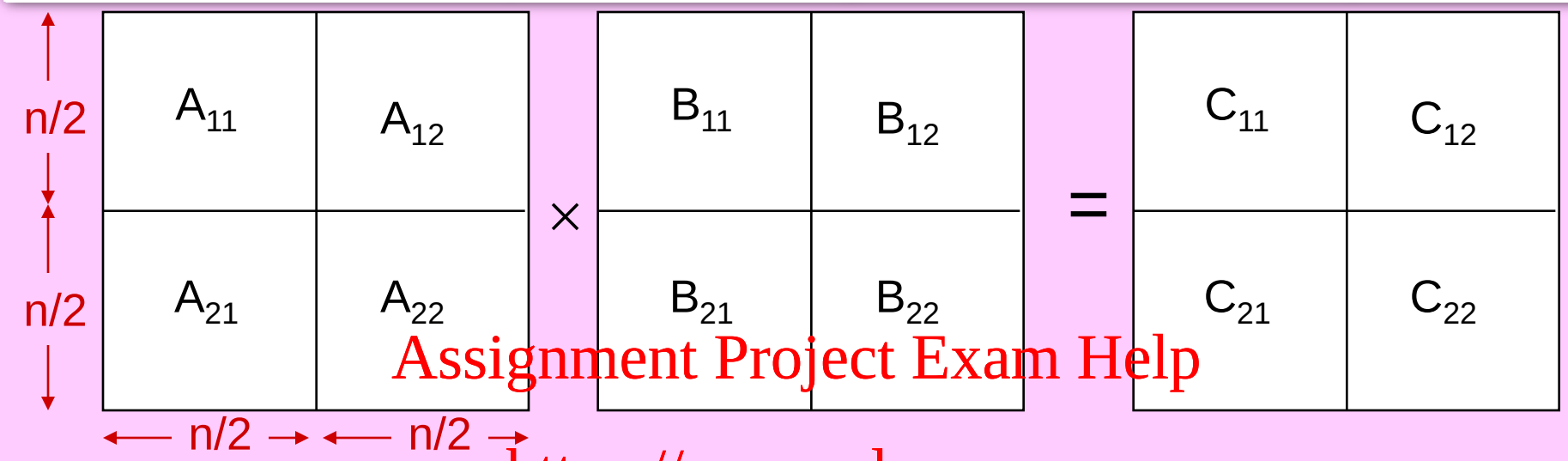


$$C_{ij} = \sum_{k=1}^{s} A_{ik} \cdot B_{kj} \quad \text{for} \quad i = 1..r,\ j = 1..t.$$

$\Theta(rst)$ time.

$\Theta(n^3)$ time, if $r = s = t = n$.

Matrix Multiplication by Divide-&-Conquer



$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \qquad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \qquad C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

$$T(n) = 8T(n/2) + \Theta(n^2) \quad \Rightarrow T(n) = \Theta\left(n^{\log_2 8}\right) = \Theta\left(n^3\right)$$

Strassen's Matrix Multiplication Algorithm

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$m_1 = (A_{11} - A_{22})(B_{21} + B_{22}) \qquad C_{11} = m_1 + m_2 - m_4 + m_6$$
$$m_2 = (A_{11} + A_{22})(B_{11} + B_{22}) \qquad C_{12} = m_4 + m_5$$
$$m_3 = (A_{11} - A_{21})(B_{11} + B_{12}) \qquad C_{21} = m_6 + m_7$$
$$m_4 = (A_{11} + A_{12})B_{22} \qquad C_{22} = m_2 - m_3 + m_5 - m_7$$
$$m_5 = A_{11}(B_{12} - B_{22})$$
$$m_6 = A_{22}(B_{21} - B_{11})$$
$$m_7 = (A_{21} + A_{22})B_{11}$$

Srassen's discovery broke the $\Omega(n^3)$ barrier.

There has been further recent improvements.

$$T(n) = 7T(n/2) + \Theta(n^2) \quad \Rightarrow T(n) = \Theta\left(n^{\log_2 7}\right) = O\left(n^{2.81\ldots}\right)$$

The Closest Pair of Points

Input: A set $P = \{p_1, p_2, \ldots, p_n\}$ of n points in the plane.
Each point is given by its x & y coordinates $p_i=(x_i, y_i)$, i=1..n.

Output: The closest pair of points in P, i.e., the pair (p_i, p_j) in P, $i \neq j$,
with minimum Euclidean distance $d(p_i, p_j) = ((x_i - x_j)^2 + (y_i - y_j)^2)^{1/2}$.

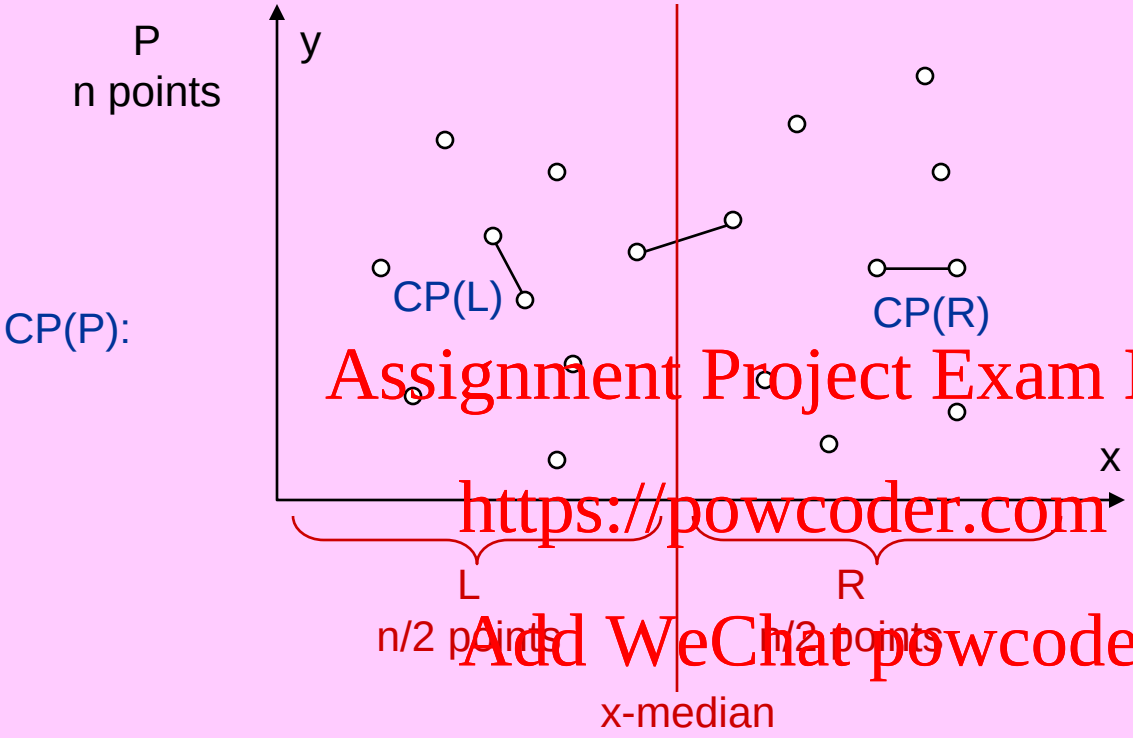
In 1 dimension: **The Min-Gap Problem**.

Known lower bound on time complexity: $\Omega(n \log n)$. Discussed later.
Matching upper bound (by sorting first): $O(n \log n)$.

In 2 dimensions it is at least as hard.

Brute Force: Try every pair. That takes $\Theta(n^2)$ time.
Divide-&-Conquer …… P.T.O.

CP: Divide-&-Conquer



3 possible cases for the closest pair of points:

a) both in the left half L, [recursive call on L]

b) both in the right half R, [recursive call on R]

c) one in L and the other in R. [Combine L & R]

Divide-&-Conquer template

Algorithm ClosestPair(P)
Pre-Condition: P is a finite set of points in the plane
Post-Condition: Output is the closest pair of points in P
1. Pre-Sort points in P on their x-coordinates (lexicographically next on y)
2. **return** CP(P)
end

Procedure CP(P)

Pre-Condition: P is a x-sorted finite set of points
Post-Condition: Output is the closest pair of points in P

3.Base: **if** |P| < 10 **then return** answer by brute-force in Θ(1) time
4.Divide: Partition P at its x-median value into sets L and R, |L|≈ |R| ≈|P|/2
5.Conquer: SolL ← **CP**(L); SolR ← **CP**(R)
6.Combine: Sol ← **MERGE**(SolL , SolR)
7.Output: **return** Sol
end

Divide-&-Conquer template

Algori
Pre-Con
Post-Co
1. Pre
2. **ret**
end

Post-Cond of MERGE must imply Post-Cond of CP(P).

On the other hand,
Post-Cond's of CP(L) and CP(R) must imply
Pre-Cond of MERGE.

Strengthen Post-Cond of CP (CP(L) & CP(R))
to help reduce the burden on MERGE!

Procedure CP(
Pre-Condition: P is finite set of points
Post-Condition: Outp closest pair of points in P
3.Base: **if** |P| < 1 **n return** answer by brute-force in Θ(1) time
4.Divide: Partition P at its x-median value into sets L and R, |L|≈ |R| ≈|P|/2
5.Conquer: SolL ← **CP**(L); SolR ← **CP**(R)
6.Combine: Sol ← **MERGE**(SolL , SolR)
7.Output: **return** Sol
end

Our aim

$T(n) = 2T(n/2) + \Theta(n) \Rightarrow T(n) = \Theta(n \log n)$ time.

Strengthen CP Post-Condition

Procedure CP(P)

Pre-Condition: P is a x-sorted finite set of points

Post-Condition: Output is the closest pair of points in P, and
P is rearranged into y-sorted order.

3.Base: **if** |P| ≤ 10 **then return** answer by brute-force in Θ(1) time

4.Divide: Partition P at its x-median value into sets L and R, |L|≈ |R| ≈|P|/2

§ Now L & R are x-sorted.

5.Conquer: SolL ← **CP**(L); SolR ← **CP**(R)

§ Now L & R are y-sorted.

§ MERGE can y-merge L & R, and …

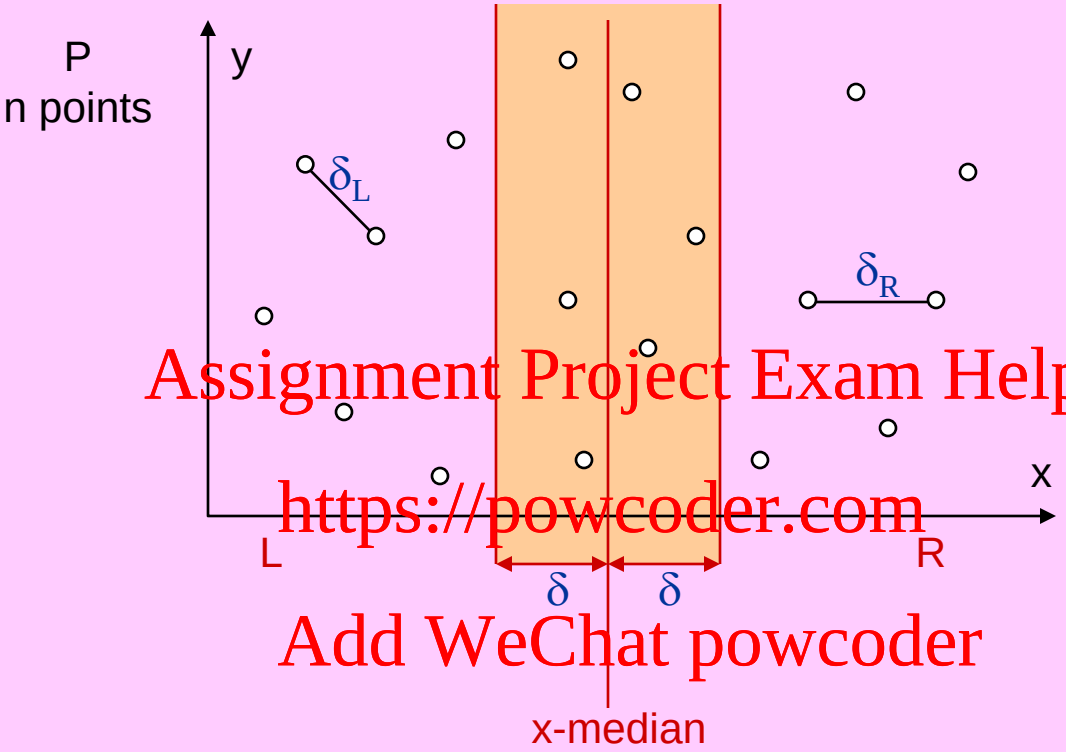
6.Combine: Sol ← **MERGE**(SolL , SolR)

§ Now P = L ∪ R is y-sorted, and …

7.Output: **return** Sol

end

MERGE



MERGE(L, R):

$\delta = \min \{\delta_L , \delta_R\}$

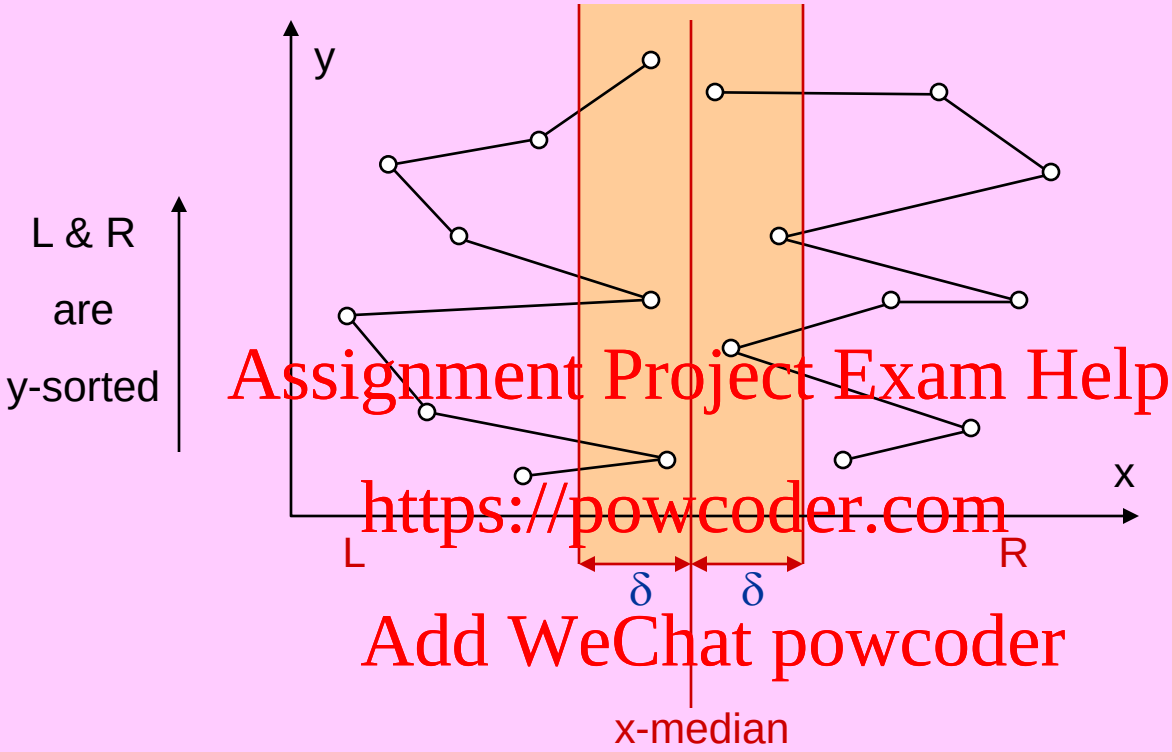
…

…

end

Can we do it in O(n) time?

MERGE



MERGE(L, R):

$\delta = \min \{\delta_L , \delta_R\}$

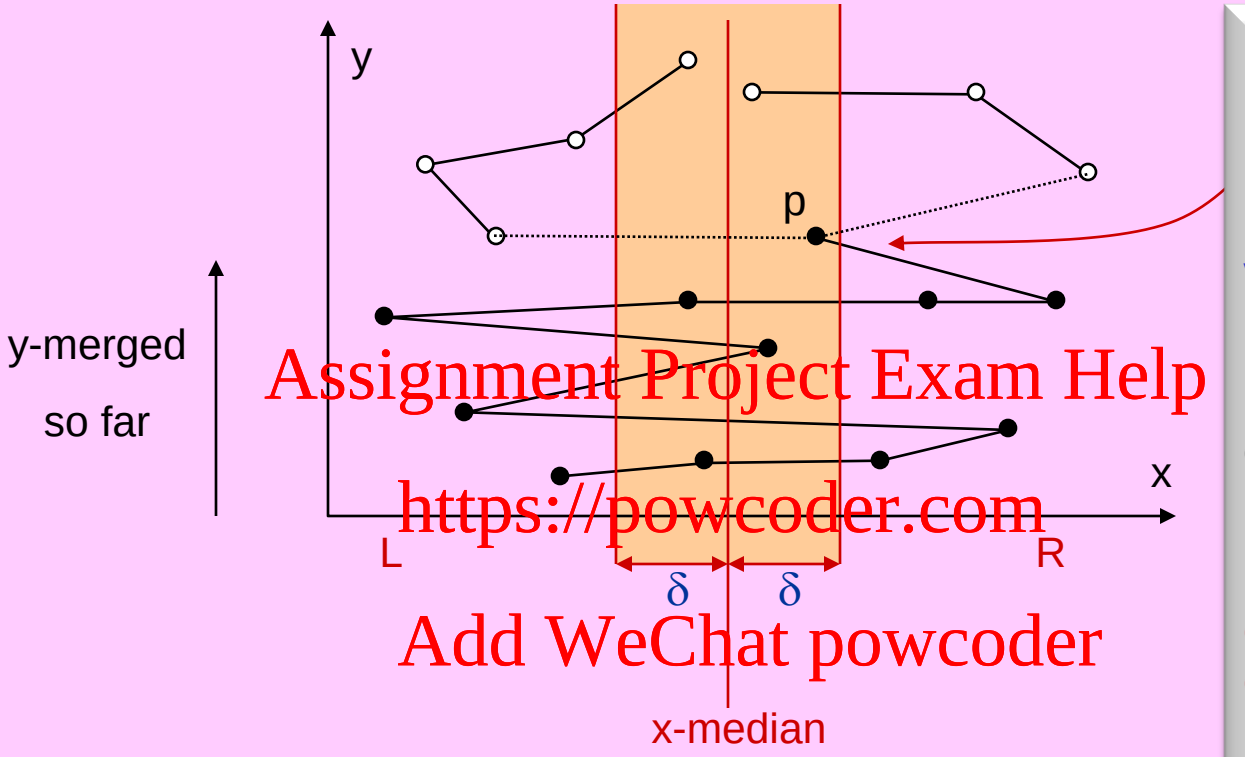
…

…

end

Can we do it in O(n) time?

MERGE



p is the latest point merged so far.

If p is in the 2δ vertical slab:

is p too close to a merged point on the opposite side?

MERGE can't afford checking p against every merged point!

Is there a short-cut?

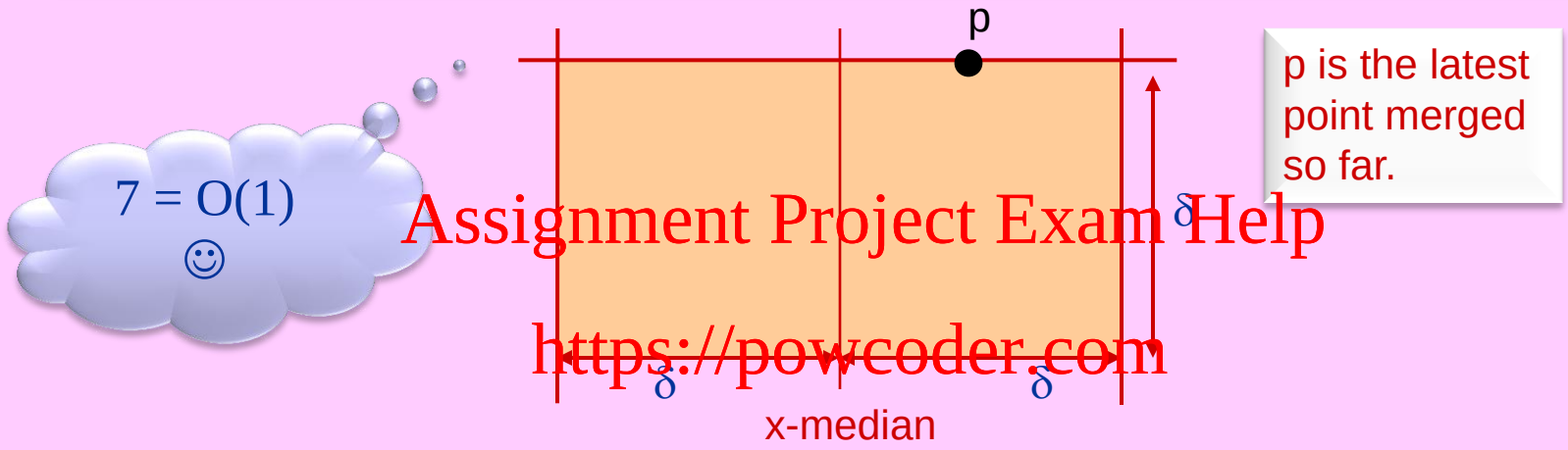
MERGE(L, R):

$\delta = \min \{\delta_L , \delta_R\}$
y-merge L & R
…
end

Can we do it in O(n) time?

MERGE

FACT: There can be at most 7 points (excluding p) in the shaded 2δ-by-δ rectangle shown below. Why?



p is the latest point merged so far.

MERGE:

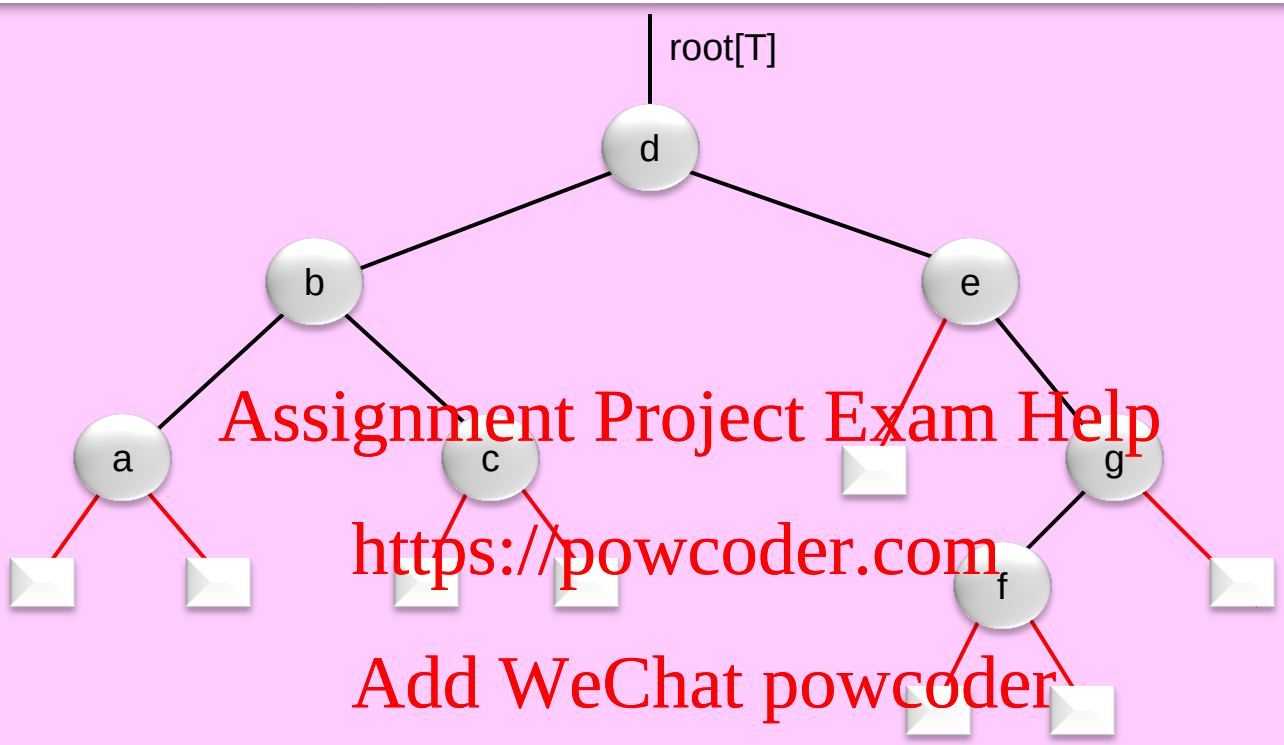
- Maintain the (up to) 7 latest merged points that fall within the 2δ vertical slab.
- **If** next point p being merged falls within this slab
  then compare p against the "7 points"; update closest pair;
  add p to the "7 point" list (remove the now lowest 8th from the list).
- Add p to the merged list and move up to the next point.
- Each point is y-merged in O(1) time.
  MERGE takes O(n) time. Therefore, CP takes O(n log n) time.

Binary Trees

&

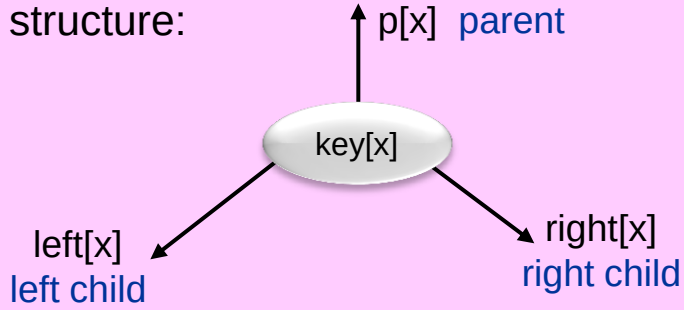
Binary Search Trees

Binary Trees: A Quick Review



external nodes (nil)

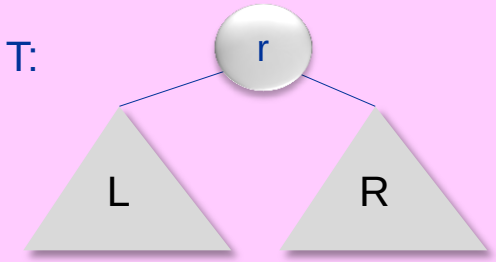
Node x structure:

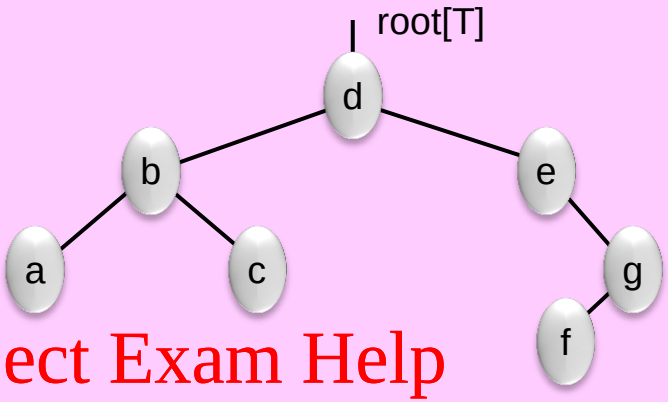


n (internal) nodes
n-1 (internal) edges

n+1 external nodes (nil)
n+1 external edges (nil)

Binary Tree Traversals





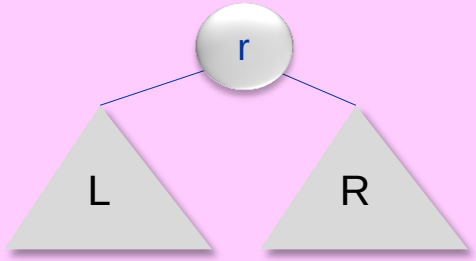
- Inorder(T): Inorder(L); r; Inorder(R). abcdefg
- Preorder(T): r ; Preorder(L); Preorder(R). dbacegf
- Postorder(T): Postorder(L); Postorder(R); r. acbfged

→ graph DFS

- Levelorder(T): non-decreasing depth order (same depth left-to-right) dbeacgf → graph BFS

Traversals in O(n) time

```
procedure Inorder(x)
1.    if  x = nil  then return
2.    Inorder(left[x])
3.    visit(x)
4.    Inorder(right[x])
end
```



Running Time Analysis by Accounting:

Line 1: n+1 external nodes (return), n (internal) nodes (continue).

Line 3: Assume visit takes O(1) time.

Lines 2 & 4: After recursive expansion:

Each node x (internal or external) visited exactly once.

O(1) time execution of lines 1 & 3 charged to node x.

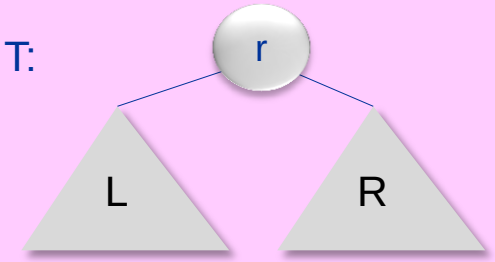
Total n + (n+1) nodes, each charged O(1) time.

Total time = O(2n+1) = O(n).

- Preorder and Postorder are similar and take O(n) time.
- Exercise: Write a simple O(n) time algorithm for Levelorder. [Hint: use a queue.]

Running Time Analysis by Recurrence

$$Time(T) = \begin{cases} Time(L) + Time(R) + 1 & \text{if } T \neq \text{nil} \\ 1 & \text{if } T = \text{nil} \end{cases}$$



CLAIM: Time(T) = 2 |T| + 1.

Proof: By induction on |T|.

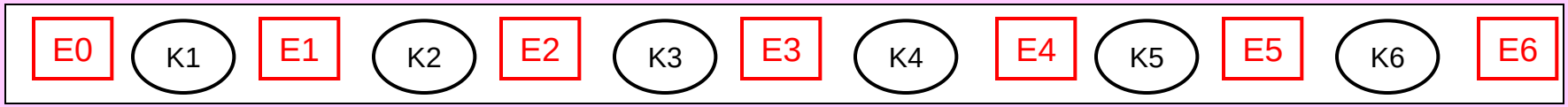
Basis (|T|=0): Time(T) = 1 = 2 |T| + 1.

Induction Step (|T| > 0):

Time(T) = Time(L) + Time(R) + 1 [by the recurrence]
= (2|L|+1) + (2|R|+1) + 1 [by the Induction Hypothesis]
= 2(|L| + |R| + 1) + 1
= 2 |T| + 1.

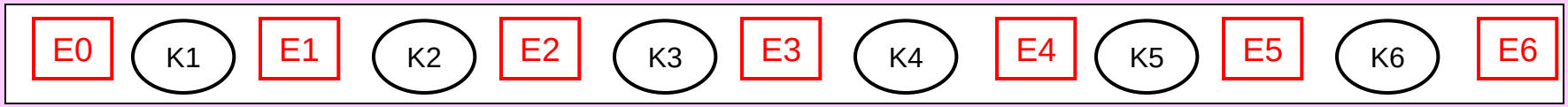
BST from Binary Search on Sorted Array

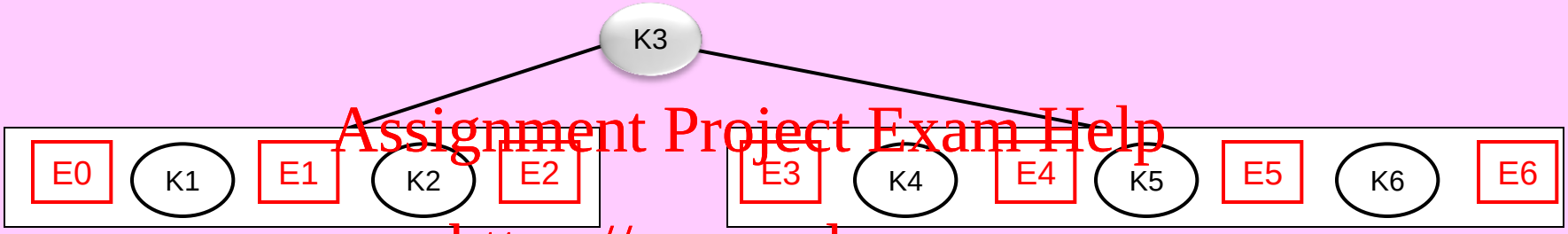
E0 < K1 < E1 < K2 < E2 < K3 < E3 < K4 < E4 < K5 < E5 < K6 < E6 < ••• < Kn < En



BST from Binary Search on Sorted Array

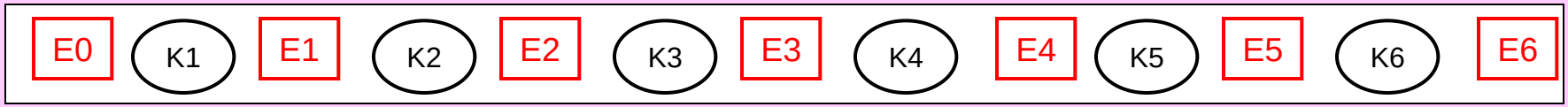
E0 < K1 < E1 < K2 < E2 < K3 < E3 < K4 < E4 < K5 < E5 < K6 < E6 < ••• < Kn < En

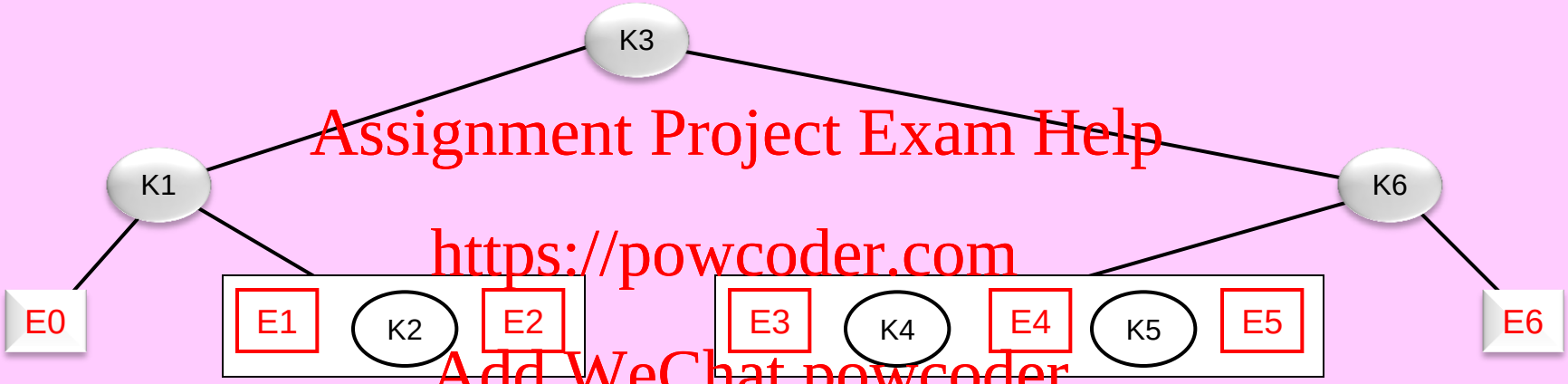




BST from Binary Search on Sorted Array

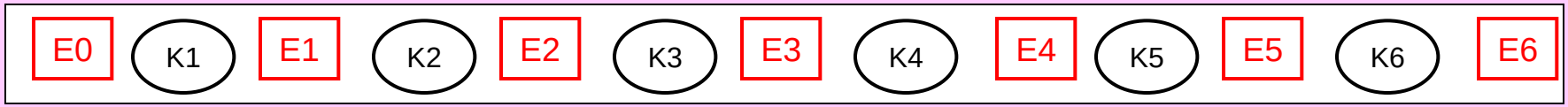
E0 < K1 < E1 < K2 < E2 < K3 < E3 < K4 < E4 < K5 < E5 < K6 < E6 < ••• < Kn < En

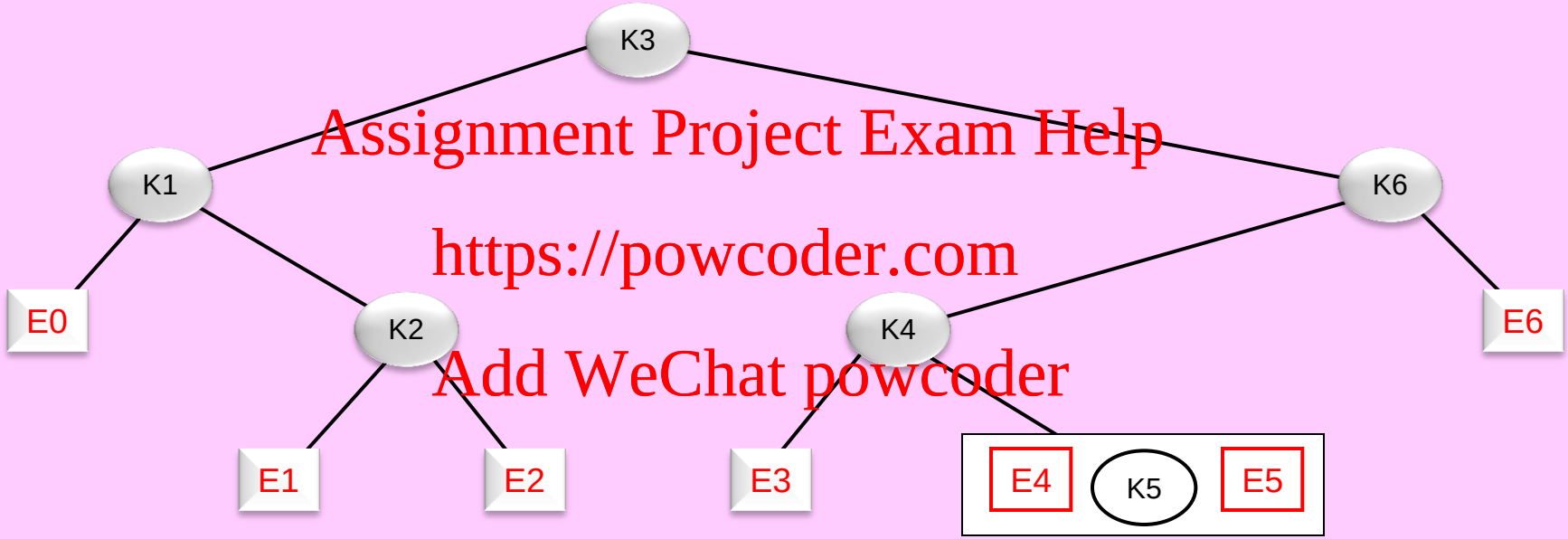




BST from Binary Search on Sorted Array

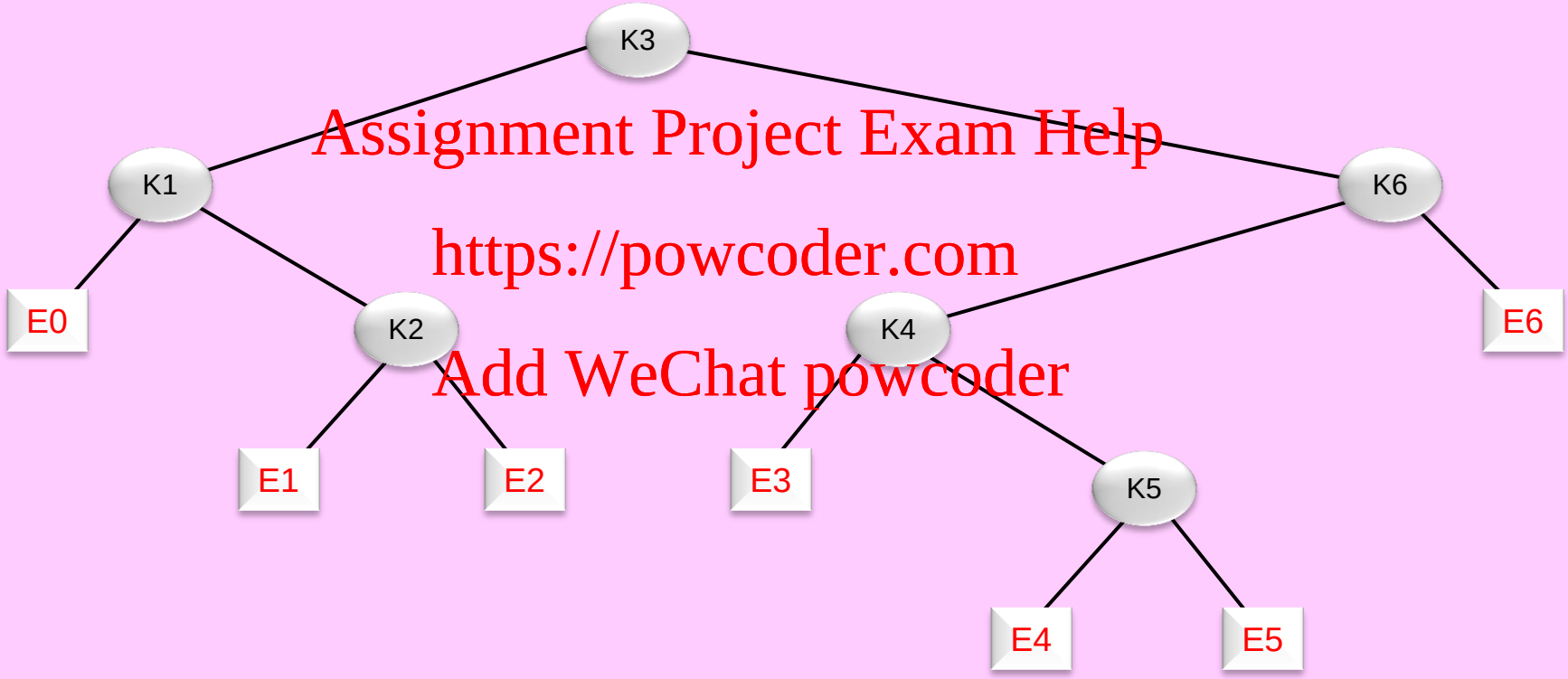
E0 < K1 < E1 < K2 < E2 < K3 < E3 < K4 < E4 < K5 < E5 < K6 < E6 < ••• < Kn < En





BST from Binary Search on Sorted Array

E0 < K1 < E1 < K2 < E2 < K3 < E3 < K4 < E4 < K5 < E5 < K6 < E6 < ••• < Kn < En



SORTED ORDER ≡ BST INORDER

BST Definition

BST is a binary tree T with one distinct key per node such that:

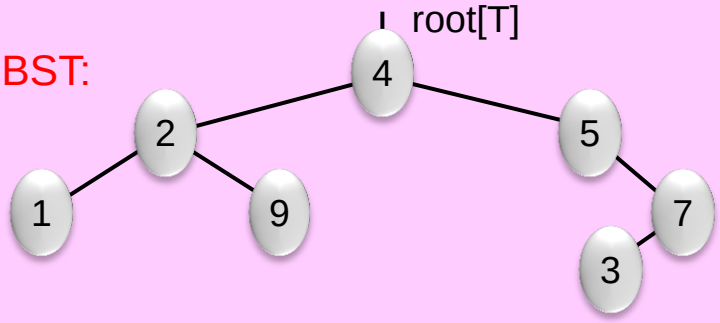
- Inorder node sequence of T encounters keys in sorted order.

- Equivalent definition: For all nodes x & y in T:
  - If x is in the left subtree of y, then key[x] < key[y], and
  - If x is in the right subtree of y, then key[x] > key[y].

- Wrong definition: For all nodes x & y in T:
  - If x is left child of y, then key[x] < key[y], and
  - If x is right child of y, then key[x] > key[y].

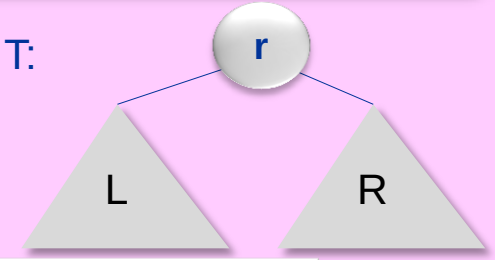
necessary but not sufficient

Not a BST:



BST Testing

Input: A binary tree T with one key per node.
Output: True, if T is a BST; false otherwise.



Method 1: Do an in-order traversal of T and verify that keys appear in increasing order. This can be done in O(n) time, n = |T|.

Observation:
BSTs have recursive structure.

T is a BST $\Rightarrow$ L and R are BSTs

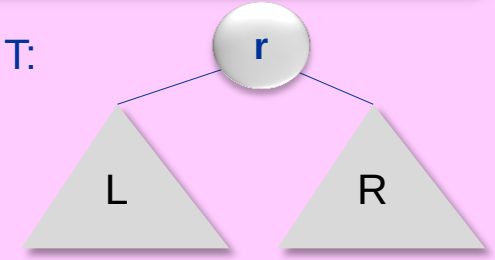
This implication is necessary but not sufficient.
To make it sufficient, strengthen the conclusion:

T is a BST $\Leftrightarrow$ (a) L and R are BSTs, and
(b) max key in L < key[r] < min key in R

Method 2: Do a post-order traversal of T. How?

BST Testing

Input: A binary tree T with one key per node.
Output: True, if T is a BST; false otherwise.



Algorithm CorrectBST(T)
Pre-Condition: T is a binary tree with one key per node
Post-Condition: output is true if T is a BST, false otherwise.
return IsBST(root[T])
end

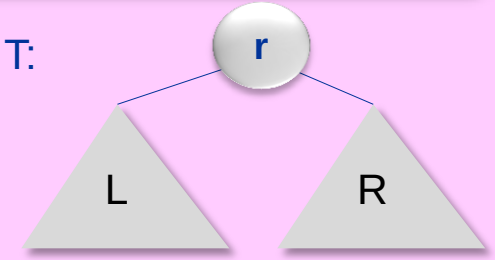
Function IsBST(r)
Pre-Condition: r is pointer to root of a binary tree with one key per node
Post-Condition: output is true if the binary tree is a BST, false otherwise.
Base: **if** r = nil **then return** true
Conquer: SolL ← **IsBST**(left[r]); SolR ← **IsBST**(right[r])
Combine: Sol ← **MERGE**(SolL , SolR) **???**
Output: **return** Sol
end

Two options: (1) strengthen Pre-Cond or (2) strengthen Post-Cond

Strengthen Pre- & Post-Condition

Algorithm **CorrectBST**(T)
Pre-Condition: T is a binary tree with one key per node.
Post-Condition: output is true if T is a BST, false otherwise.
    return **IsBST**(root[T], $-\infty$, $+\infty$)
end

T:



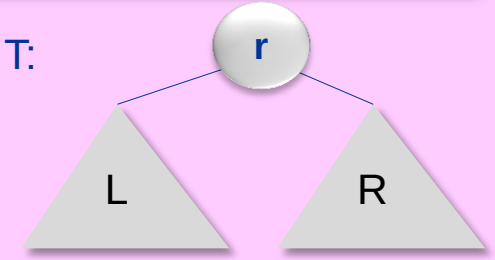
Function **IsBST**(r, minKey, maxKey)
Pre-Condition: r is pointer to root of a binary tree with one key per node,
minKey & maxKey are key values (possibly $\pm\infty$).
Post-Condition: output is true if the binary tree is a BST and
minKey < maxKey and for every node x in that tree:
minKey < key[x] < maxKey; false otherwise.

```
    if   minKey ≥ maxKey then return false
    if   r = nil then return true
    SolL ← IsBST(left[r], minKey,  key[r]);
    SolR ← IsBST(right[r], key[r], maxKey)
    Sol ← SolL & SolR
    return Sol
end
```

Strengthen Post-Condition

Algorithm CorrectBST(T)
Pre-Condition: T is a binary tree with one key per node
Post-Condition: output is true if T is a BST, false otherwise.
    (isBST, minKey, maxKey) ← **BSTtest**(root[T])
    return isBST
end

T:



Function BSTtest(r)
Pre-Condition: r is pointer to root of a binary tree with one key per node
Post-Condition: output is (isBST, minKey, maxKey), where
    isBST = true if the binary tree is a BST, false otherwise, and
    isBST ⇒ minKey = minimum key in the tree rooted at r, and
        maxKey = maximum key in the tree rooted at r
    if r = nil **then return** (true, +∞, –∞) § Note where +∞ & –∞ are
    (isBSTL, minL, maxL) ← **BSTtest**(left[r]);
    (isBSTR, minR, maxR) ← **BSTtest**(right[r])
    isBST← isBSTL & isBSTR & (maxL < key[r] < minR)
    return (isBST, min{ minL, key[r] }, max{ maxR, key[r] })
end

Bibliography

Additional optional sources:

[Edm08] Jeff Edmonds, *"How to Think about Algorithms,"* Cambridge H. Press, 2008.
[Parts of chapters 1-4, 6-10: good coverage of the loop-invariant method.]

[AHU74] Aho, Hopcroft, Ullman, *"The Design and Analysis of Computer Algorithms"*, Addison-Wesley, 1974.
[A classic text book on algorithms, divide-&-conquer, integer & matrix computations, and more.]

[Man89] Manber, *"Introduction to Algorithms: A Creative Approach"*, Addison-Wesley, 1989.
[Contains interesting induction topics and exercise solutions.]

Exercises

INSTRUCTIONS:

In the following problems you are asked to design and analyze efficient iterative or recursive algorithms.

Follow the Loop-Invariant design-&-verification method for iterative algorithms, and the strong induction method for recursive algorithms.

You should clearly state and explain the proof steps as learned in the course.

1. **Double Vision in Sorted Array:**
   Design and analyze an O(n) time **in-place** algorithm for the following problem.
   Input: An array A[1..n] of n positive real numbers sorted in increasing order.
   Output: True if there is a pair of indices i and j such that A[i] = 2A[j] ; false otherwise.

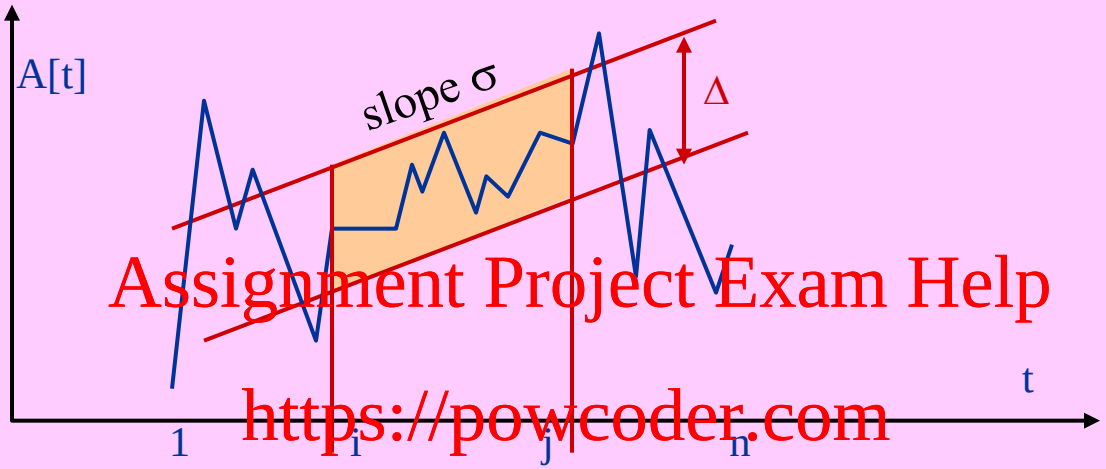
2. **The Longest All-Positive Sub-Array Problem:**
   Input: An array A[1..n] of arbitrary integers.
   Output: The longest contiguous sub-array of A[1..n] with positive entries only.

3. **The Maximum-Sum Monotone Sub-Array Problem:**
   Input: An array A[1..n] of arbitrary positive integers.
   Output: The maximum-element-sum contiguous sub-array of A[1..n] whose entries form a monotone sequence (either ascending or descending).

4. The Longest Δ-Smooth σ-Slope Sub-Array Problem:

Input: An array A[1..n] of arbitrary integers, and two numbers σ, and Δ > 0.

Output: The longest contiguous sub-array, say A[i..j], whose entries fall between two lines of slope σ with vertical separation Δ. (See the illustrative figure below.)



5. The Largest Δ-Flat Subset and Sub-Array Problems:

Let Δ be a positive real number and S be a finite set of real numbers. We say S is **Δ-flat** if

$$\max(S) - \min(S) \leq \Delta \cdot (|S| - 1) .$$

(That is, the average difference between contiguous pairs of elements in the sorted permutation of S is ≤ Δ.)

Design and analyze efficient algorithms for the following problems:

(a) Given a set A of n elements and a Δ > 0, compute the largest Δ-flat subset of A.

(b) Given an array A[1..n] of n elements and a Δ> 0, compute the longest Δ-flat contiguous sub-array of A.

[Note: in part (a) any subset is a potential candidate solution, but in part (b) only those subsets that correspond to contiguous sub-arrays are allowed.

Hint: for part (b) use an incremental method similar to that of the LSS problem we studied in these slides.]

6. Design and analyze efficient algorithms for each of the following problems:

a) **Input:** An array A[1..n], with each element 0 or 1.
Output: the longest contiguous sub-array of A[1..n] that contains at least as many 1's as 0's.

b) **Input:** An array A[1..n], with each element 0 or 1.
Output: The longest contiguous sub-array of A[1..n] with at least twice as many 0's as 1's.

c) **Input:** An array A[1..n], with each element 0 or 1.
Output: The longest contiguous sub-array of A[1..n] such that the number of 0's minus the number of 1's in that sub-array is 2 mod 5.

d) **Input:** An array A[1..n], with each element 0 or 1.
Output: The Length of longest monotone ascending (not necessarily contiguous) sub-array of A[1..n].

e) **Input:** An array A[1..n] of integers in strictly increasing order.
Output: Find an index i such that A[i] = i, or report that no such index exists.

f) **Input:** An array A[1..n] of integer elements, where absolute value of each element is ≤ 5.
Output: Longest contiguous sub-array of the input array, whose sum of elements is non-negative.

7. Inplace Rank Sort: We are given an array A[1..n] of n employee records. Each employee record A[i] consists of two fields: A[i].info and A[i].rank. The rank of each employee is one of 4 possibilities {1,2,3,4}. Design and analyze a linear-time, in-place incremental algorithm to sort the n employee records in ascending order of rank.

8. Generalize 3-Partition to C-Partition: For any fixed integer C, $2 \leq C \leq n$, show that the C-partition problem can be solved in O(Cn) time, using O(C) extra memory cells.

9. 2SUM & 3SUM in a Sorted Array: We are given a sorted array A[1..n] and a target value X.

a) 2SUM: Design & analyze an efficient algorithm to find a pair of array indices (i,j) such that A[i] + A[j] = X, or report that no such pair exists. [Hint: can be done in O(n) time.]

b) 3SUM: Similar to part (a), find a triple array indices (i,j,k) such that A[i] + A[j] + A[k] = X. What is the time complexity of your algorithm?

10. **A Fibonacci Identity:** Let n and m be arbitrary natural numbers.
(a) Prove the identity below by induction on n. (Make sure to cover all required base cases.)
(b) Prove this identity using the matrix formula $G(n) = A^n$ we established on page 70 of these slides.

$$F_{n+m+1} = F_{n+1}F_{m+1} + F_nF_m$$

11. **Euclid & Fibonacci:** Prove that GCD of two Fibonacci numbers is a Fibonacci number.

12. **Integer Linear Equations:** Design and analyze an efficient algorithm that, given integers a, b, c, determines whether the linear equation ax + by = c has an integer solution for variables x and y, and if so it finds one such solution.

13. **Modified Fibonacci Sequence:**
Define $G_0 = 0$, $G_1 = 1$, $G_2 = 2$, $G_{n+3} = 3G_n + 2G_{n+1} + 5G_{n+2}$, for all $n \geq 0$.
Design and analyze an O(log n) time algorithm to compute G_n.

14. **Closest Red-Blue Pair:** We are given a set of n points in the plane, each point is either red or blue. We want to find the closest (red, blue) pair among the input points (if there is any such pair). Design and analyze an efficient divide-&-conquer algorithm for this problem.

15. **Boys & Girls in a Binary Search Tree (BST):**
Each node item in our Binary Search Tree is a record of the form (gender, salary), where gender $\in$ {boy, girl}, and salary is a positive integer which is used as the key. Design & analyze an efficient recursive algorithm to find a pair of opposite sex records with closest salary values.

16. Strengthening Recursion Pre/Post Conditions:

Below we show two algorithms for the following problem:

Given a binary tree T and a height threshold h, return the number of nodes in T of height at least h.

```
Algorithm CountHighNodes1(T, h)
        Count ← CHN1(root[T], h)
        return (Count)
end
Function  CHN1(r, h)
        if   r = nil then return (0)
        LCount ← CHN1(left[r], h)
        RCount ← CHN1(right[r], h)
        Height ← ComputeHeight(r)
        Count ←  LCount + RCount
        if  Height ≥ h then Count ←  Count + 1
        return (Count)
end
Function  ComputeHeight(r)
        if   r = nil then return (-1)
        LHeight ← ComputeHeight(left[r])
        RHeight ← ComputeHeight(right[r])
        return (1 + max(LHeight, RHeight))
end
```

```
Algorithm CountHighNodes2(T, h)
      (Count, RootHeight) ← CHN2(root[T], h)
       return (Count)
end
Function  CHN2(r, h)
        if   r = nil then return (0, -1)
        (LCount, LHeight) ← CHN2(left[r], h)
        (RCount, RHeight) ← CHN2(right[r], h)
        Height ←  1 + max(LHeight, RHeight)
        Count ←  LCount + RCount
        if  Height ≥ h then Count ←  Count + 1
        return (Count, Height)
end
```


a) Give all the Pre- and Post-Conditions, and argue that both algorithms CountHighNodes1 and CountHighNodes2 correctly solve the problem.

b) Analyze the worst-case running times of CountHighNodes1 and CountHighNodes2.

c) Explain the real reason why CountHighNodes1 is less efficient than CountHighNodes2.

17. More Recursion on Binary Trees:

The input is a binary tree T with an integer valued key per node.
Design & analyze efficient recursive algorithms for the following tasks.

[If no solution exists, some special value, say null, may be returned.
If there are several solutions, choose any one of them and return it.]

a) Find the sum of the keys in all the leaves of T.

b) Find a root-to-leaf path with minimum sum key sequence.

c) Find the root of the largest subtree that contains no more than |T|/10 nodes.
[Note: |T| denotes the number of nodes in T and is not part of the input.]

d) Find a node with maximum ***imbalance***, where imbalance of a node is the difference between the heights of its left and right subtrees. [Note: height of an empty tree is –1.]

e) Find a pair of nodes (x, y) in T such that
(i) y is the successor of x in the inorder sequence of nodes of T, and
(ii) | (depth of x) – (depth of y) | is maximum possible.

f) Find a pair of nodes (x,y) in T with longest ***separation***, where the separation between two nodes is the length (number of edges) of the unique path from x to y when thinking of the tree as an undirected graph, i.e., from x up to the lowest common ancestor and back down to y.

[Hint: Look up the previous exercise. Sample Midterm Solutions contain additional examples.
You may need to strengthen the Pre- or Post-Conditions of your recursive routines.
For instance, let the recursive routine return extra info in addition to the required output data.
Then use the returned extra info in the "combine" step of the recursion.]

More Recursion on Binary Trees:

18. The Most Weird Species:

Input: A binary tree T in which each node x contains a field *color*[*x*] which is either *red* or *blue*.

Define **lineage** of a node x to be the set of all those nodes in T that are either ancestor or descendant of x (including x). Define **weirdness** of a node x to be the difference between the number of red nodes and the number of blue nodes in the lineage of x.

Output: A node x in T with maximum weirdness.

19. Richest Heritage:

Input: A binary tree T in which each node x contains a field *worth*[*x*], which is a (positive, zero, or negative) monetary value expressed as a real number.

Define (monetary) **heritage** of a node x to be the total worth of ancestors of x minus the total worth of proper descendants of x.

Output: A node x in T with maximum heritage.

20. Saddle Point:

Input: A binary tree T in which each node x contains a field *value*[*x*] which is a real number.

We say a node x in T is a **saddle point** if x has minimum value among all its ancestors (including x), but it has maximum value among all its descendants (including x).

Output: A saddle point of T if there is one, or null if there isn't.

21. Binary Tree Bisection:

Many divide-&-conquer algorithms that operate on graphs require that the graph be bisected into nearly equal-sized subgraphs by removing a small number of edges. We will discuss graph algorithms later. This problem investigates bisection of binary trees.

a) Prove the following Fact.

Fact: $\forall n \geq 2$: *any n-node binary tree T has a* ***balanced*** *edge: an edge whose removal will separate T into two binary trees, each having at most* $\lceil (2n-1)/3 \rceil$ *nodes.*

b) Design and analyze an efficient algorithm to find a balanced edge of any given binary tree T.
[Hint: O(n) time is possible by a post-order traversal.]

c) Show that the bound in the above Fact is optimal in the worst case by demonstrating that for all $n \geq 2$ there is an n-node binary tree whose most evenly balanced partition upon removal of a single edge results in one of its two parts having size $\lceil (2n-1)/3 \rceil$.

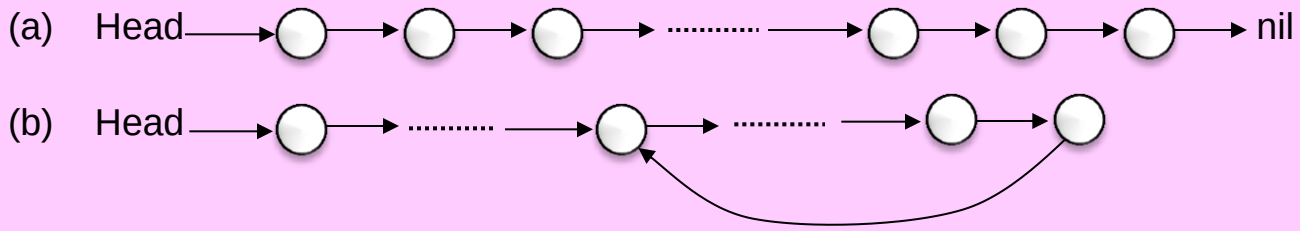
d) Binary Tree Bisection: Using part (a), show that by removing at most O(log n) edges, we can partition the nodes of any n-node binary tree into two sets A and B such that $|A| = \lceil n/2 \rceil$ and $|B| = \lfloor n/2 \rfloor$, and there is no remaining edge with one end vertex in A and the other in B.
For this part you don't need to give a detailed algorithm. A high level description will suffice.

22. Linked-List Correctness Verification (an IBM interview test question):

We are given the head pointer to a read-only linked list. Each node of this list consists of a single field, namely, a pointer to the next node on the list. For the list to have correct structure, each node should point to its successor, except for the last node that points to *nil*, as in figure (a) below. However, due to a possible structural error, the list may look like that of figure (b), where a node points to one of the previous nodes (possibly itself). From that node on, we no longer have access to the rest of the nodes on the list. The task is to verify the structural correctness of the list. If we knew *n*, the number of accessible nodes on the list, we could easily verify the list's structural correctness, namely, starting from the head pointer, we would follow the list nodes for *n* steps and see if we reach a *nil* pointer. However, we have no advance knowledge of the length of the list.

Design & analyze a linear-time in-place algorithm to test the structural correctness of the list. That is, your algorithm should take time linear in the number of accessible nodes, and work in-place, i.e., use only O(1) additional memory cells. So, your algorithm may use a few local scalar variables, but not such things as additional lists or arrays. Also, the nodes on the list do not have any additional fields that you may use for some kind of marking, and your algorithm should not attempt to alter the structure of the list, not even temporarily.

[Hint: use repeated doubling.]

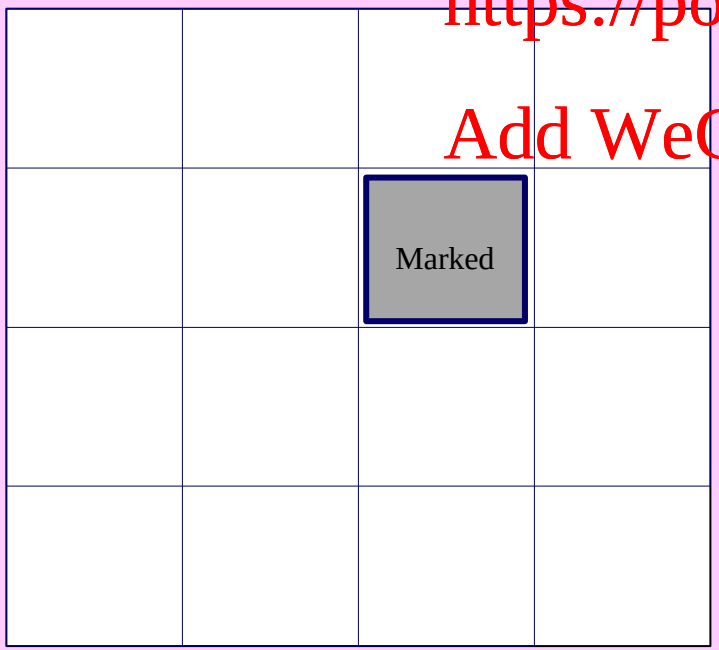


23. A Tiling Problem: The input is a 2^n by 2^n square checkerboard with the unit square at the given coordinates (i,j) marked.
You have a sufficient number of identical tiles of the shape  that can cover 3 board squares.

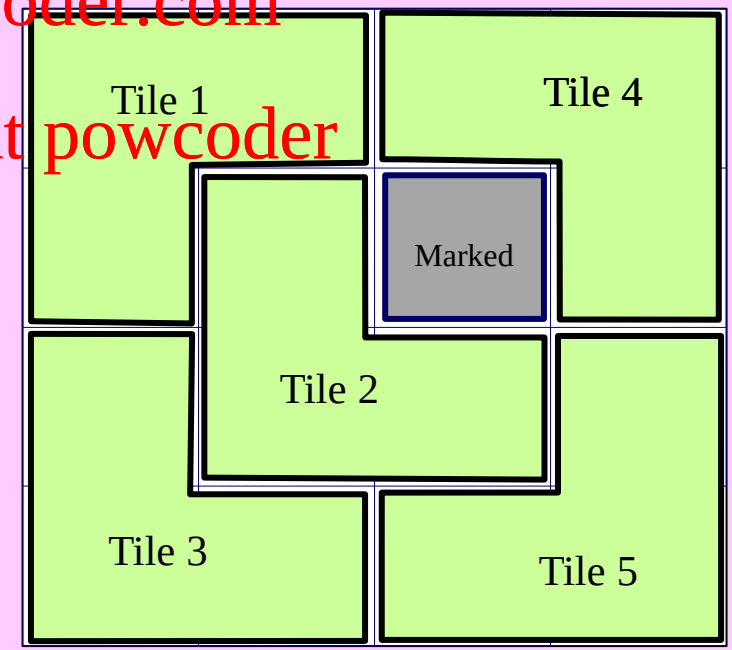
Your task is to place non-overlapping tiles on the board to cover each of the $2^n \times 2^n$ board squares except the marked one. Give a recursive algorithm for this problem.

Further explanation: You need to output the list of tile placements. A tile can cover 3 neighboring squares as mentioned above. So, each tile placement can be given as the set of those 3 squares; each square can be specified by its (x,y) coordinates. See the example below:

Input



Output: Tile Placements



24. **[Goodrich-Tamassia C-1.14. pp. 50-53]**
Suppose you are given a set of small boxes, numbered 1 to *n*, identical in every respect except that each of the first *i* contain a pearl whereas the remaining $n - i$ are empty. You also have two magic wands that can each test if a box is empty or not in a single touch, except that a wand disappears if you test it on an empty box. Show that, without knowing the value of *i*, you can use the two wands to determine all the boxes containing pearls using at most o(n) wand touches. Express, as a function of *n*, the asymptotic number of wand touches needed.

25. **[Goodrich-Tamassia C-1.25. pp. 50-53]**
An evil king has a cellar containing *n* bottles of expensive wine, and his guards have just caught a spy trying to poison the king's wine. Fortunately, the guards caught the spy after he succeeded in poisoning only one bottle. Unfortunately, they don't know which one. To make matters worse, the poison the spy used was very deadly; just one drop diluted even a billion to one will still kill someone. Even so, the poison works slowly; it takes a full month for the person to die. Design a scheme that allows the evil king determine exactly which one of his wine bottles was poisoned in just one month's time while expending at most O(log n) of his taste testers.

26. **[CLRS 2nd edition Problem 4-2, p. 85] Finding the missing integer:**
An array A[1..n] contains all the integers from 0 to n except one. It would be easy to determine the missing integer in O(n) time by using an auxiliary array B[0..n] to record which numbers appear in A. In this problem, however, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is *"fetch the j^{th} bit of A[i],"* which takes constant time. Show that if we use only this operation, we can still determine the missing integer in O(n) time.

27. Polynomial multiplication by divide-&-conquer:

A degree n-1 polynomial $P(x) = \sum_{i=0}^{n-1} p_i x^i = p_0 + p_1 x + p_2 x^2 \ldots + p_{n-1} x^{n-1}$ can be represented by an array $p[0..n-1]$ of its n coefficients. Suppose P(x) and Q(x) are two polynomials of degree n-1, each given by its coefficient array representation. Their product P(x)Q(x) is a polynomial of degree 2(n-1), and hence can be represented by its coefficient array of length 2n-1. The polynomial multiplication problem is to compute the coefficient array of P(x)Q(x), given the coefficient arrays of P(x) and of Q(x).

There is an obvious $\Theta(n^2)$ (i.e., quadratic) time algorithm to solve this problem. However, a method similar to the divide-&-conquer integer multiplication algorithm of Karatsuba-Ofman can achieve sub-quadratic time complexity. Design and analyze one such sub-quadratic algorithm for the polynomial multiplication problem.

28. [CLRS Problem 30-2, p. 921] Toeplitz Matrices:

A **Toeplitz matrix** is an $n \times n$ matrix $A = (a_{ij})$ such that $a_{ij} = a_{i-1,j-1}$ for $i = 2..n$ and $j = 2..n$.

a) Is the sum of two Toeplitz matrices necessarily Toeplitz? What about the product?

b) Describe how to represent a Toeplitz matrix so that you can add two $n \times n$ Toeplitz matrices in $O(n)$ time.

c) Give an $O(n \log n)$ time divide-&-conquer algorithm for multiplying an $n \times n$ Toeplitz matrix by a vector of length n. Use your representation from part (b).
[Hint: you may need to learn how the Fast Fourier Transform (FFT) algorithm works.]

d) Give an efficient algorithm for multiplying two Toeplitz matrices. Analyze its running time.

29. Particle Physics Problem: You have been working with some physicists who need to study, as part of their experimental design, the interaction among large numbers of very small charged particles. Basically, their setup works as follows. They have an inert lattice structure, and they use this for placing charged particles at regular spacing along a straight line. Thus we can model their structure as consisting of the points {1, 2, 3, … , n} on the real line; and at each of these points j, they have a particle with charge q_j. (Each charge can be either positive or negative.)

They want to study the total force on each particle, by measuring it and then comparing it to a computational prediction. This computational predictional part is where they need your help. The total net force on particle j, by Coulomb's Law, is equal to

$$F_j = \sum_{i:i<j} \frac{C\, q_i\, q_j}{(j-i)^2} - \sum_{i:i>j} \frac{C\, q_i\, q_j}{(j-i)^2}.$$

They have written the following simple program to compute F_j for all j.


```
for j ← 1..n do
        F_j ← 0
        for i ← 1 .. j-1 do F_j ← F_j + C q_i q_j / (j-i)^2
        for i ← j+1 .. n do F_j ← F_j – C q_i q_j / (j-i)^2
        output F_j
end
```


It is not hard to see that this algorithm takes $O(n^2)$ time.

The trouble is, for the large values of n they are working with, the program takes several minutes to run. On the other hand, their experimental setup is optimized so that they can throw down n particles, perform the measurements, and be ready to handle n more particles within a few seconds. So they would really like it if there were a way to compute all the forces F_j much more quickly, so as to keep up with the rate of the experiment.

Help them out by designing a divide-&-conquer algorithm that computes all the forces F_j in $O(n \log n)$ time. [Hint: use part (c) of the previous problem.]

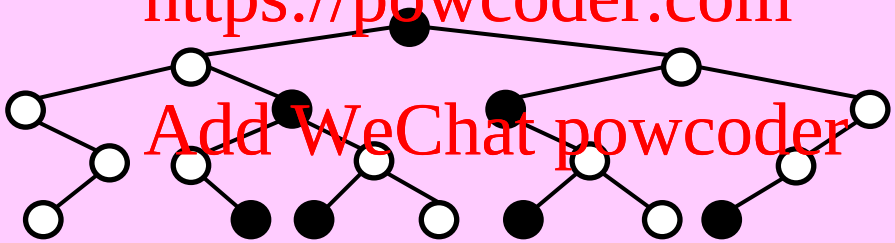
30. **Input:** A binary tree T in which each node x contains a real valued field named $value[x]$. We say a node x in T is **value-balanced** if and only if the average value of ancestors of x (including x) equals the average value of descendants of x (including x).

Output: Among value-balanced nodes in T return the one with minimum depth (break ties arbitrarily). Return nil if T has no value-balanced node.

Note: Apart from the value field and left child and right child pointers, nodes of T do not have any additional storage fields.

Design and analyze an efficient algorithm for this.

31. A node x in a binary tree is said to be **well-oriented** if the path from root to x follows an equal number of left child branches as right child branches. For example, in the binary tree below, all the well-oriented nodes are shown in solid black.



You may assume that for a node x, $left[x]$ is left child of x, $right[x]$ is right child of x. There are no other storage fields in the nodes of the tree.

Design and analyze an efficient algorithm that returns the number of well-oriented nodes in a given binary tree T.

END