COL 100 Lec 19

Imperative model of computation

- State is the key; gets modified by the steps of computation
- Steps of computation: via instructions

Eg:
```
fun foo (a:int, b:int)
{
    c = a + b
    return c
}

x = foo(a0, b0)
```

Assignment Statement

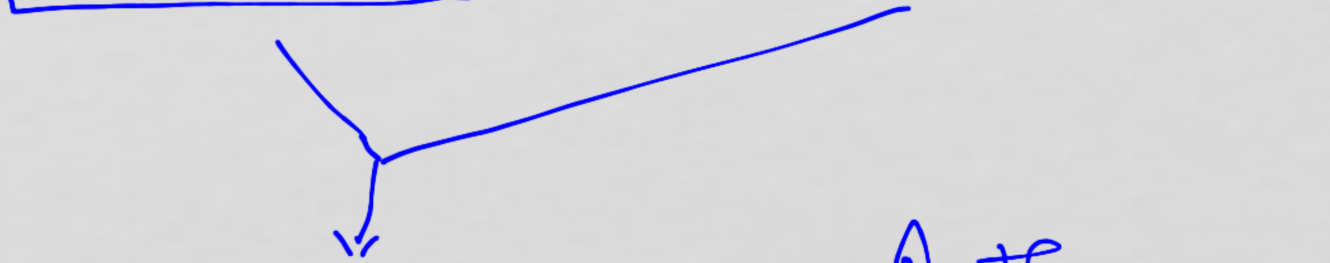
Return statement

State: c = —, a = a_0, b = b_0

State: c = $a_0 + b_0$, a = a_0, b = b_0

State: x = $a_0 + b_0$, a = a_0, b = b_0

- Each statement (or a sequence of statements) has a precondition and postcondition



Logical properties of the states just before and after the statement

Eg:

(SWAP : Precondition $(a == a_0 \wedge b == b_0)$

```
temp = a
a = b
b = temp
```

(Post Condition: $(a == b_0 \wedge b == a_0)$

Other type of statements

- Assert → statements that evaluate a logical condition at some step of the computation

OR

- A logical statement used in documentation of your correct design

Eg:

```
f ( a , b)
{
   a = a+b
   b = a-b
   a = a-b
}
```

Precond^n of the fnc f

$(a == a_0) \wedge (b == b_0)$

$(a == a_0 + b_0) \wedge (b == b_0)$

$(a = a_0 + b_0) \wedge (b == a_0)$

$(a == b_0) \wedge (b == a_0)$

PostCondition of the fnc f

- if then else

if (c) then S_1 → assert : $A \wedge C$

else S_2 → assert : $A \wedge \neg C$

assert(A)

Eg:

→ assert : $(a == a_0) \wedge (b == b_0)$

if (a > b):

→ assert : $(a > b) \wedge$ "

temp = a

a = b

b = temp → assert : $(a == b_0) \wedge (b == a_0)$

→ assert : $(a == b_0 \wedge b == a_0) \vee$

$(a_0 <= b_0 \wedge a == a_0 \wedge b == b_0)$

- While statement

```
while (C):
    S
```

] While cond^n C holds, true repeatedly execute S

How do precond^n & postcond^n play out here?

```
(* assert: I *)
```

→ Also called the loop invariant!

```
while (C):
    (* assert: I ∧ C *)
    S
    (* assert: I *)
```

```
(* assert: I ∧ ¬C)
```

→ outside of the while loop

Eg:

SML

```
fact_iter (n,f,c) =
    if (c = n) then f
    else fact_iter (n, f * (c+1), c+1)
```

Imperative code

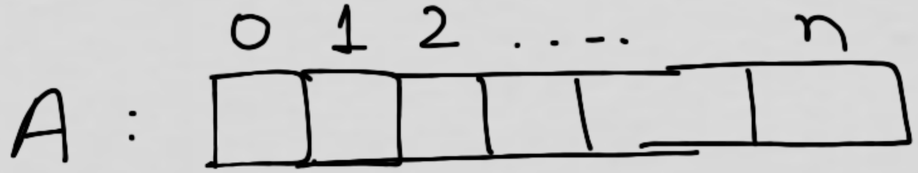
```
fact_iter (n) :
        f = 1
        c = 0
    (* I:
    while ( c != n) :
                c = c + 1
                f = f * c
            (* I:  f = c! ∧
                    0 ≤ c ≤ n *)
    (* f = c! ∧ 0 ≤ c ≤ n ∧
            c == n *)
```

- functions

```
def foo (arglist):
    ⋮
    return V;
```

optional

Arrays



A[i]: i^{th} element of A O(1) operation

Eg: Sequential Search

```
def SeqSearch (A, x, left, right):
    1# assert: A[left,...,right] and
               x is established
    i = left
    found = False
    3# found = false ⟺
              (x ∉ A[left, ..., i-1] ∧ left ≤ i ≤ right+1)
    while ( (not found) and
            (i <= right)) :
        if (A[i] == x):
            found = True
        Else :
            i = i + 1
    2# assert: (found at i) or (x ∉ A[left,...
                                      right])
```

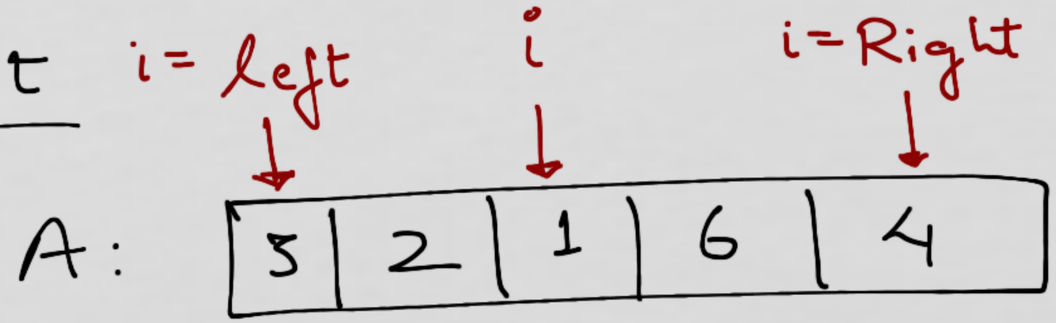
Initially: $i = left \wedge found = false$

finally : $[found = true \wedge left \leq i \leq right] \vee [found = false \wedge i > right]$

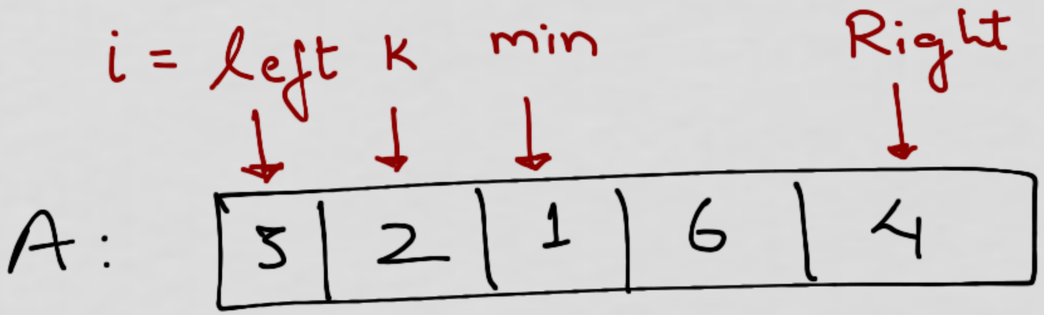
```
if found :
        print i
else :
        print "not found"
```

Eg: Using Arrays & Loops

Selection Sort



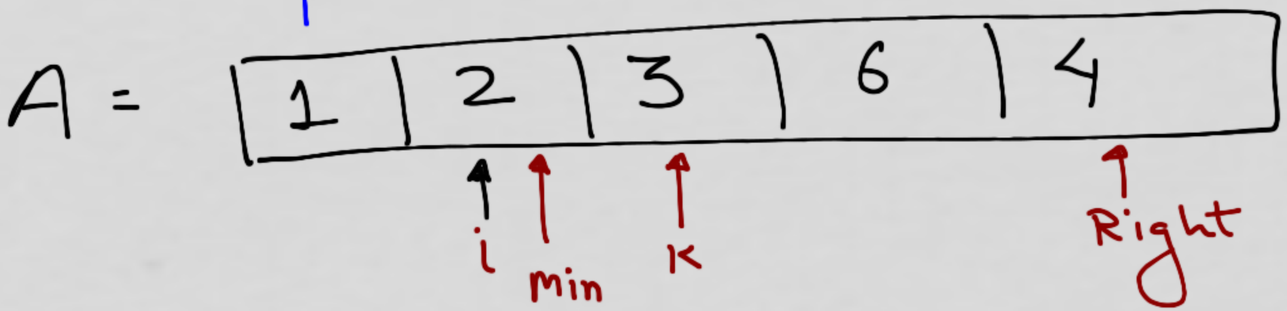
Strategy: Search the minimum value in the array and place it at the start of the array



Initially : i = left, unsorted

Finally : i = right, sorted

Swap A[i] and A[min]



```
def SelSort(A, left, right):
    i = left
    while (i <= right):
        min = i
        k = i+1
        while (k <= right):
            if (A[k] < A[min]):
                min = k
            k = k+1
        A[i], A[min] = A[min], A[i]
        i = i+1
```

assert: A[left, ..., right] is established

assert: A[left, ..., i, i-1] is sorted, left ≤ i ≤ right+1

assert: min s.t. A[min] ≤ all A[i, ..., k-1], i ≤ min ≤ right, i+1 ≤ k ≤ right+1

assert: ∃ min ∈ [i, right] s.t. A[min] ≤ all A[i, ..., right]

assert: A[left, ..., right] is sorted