Sequential Search

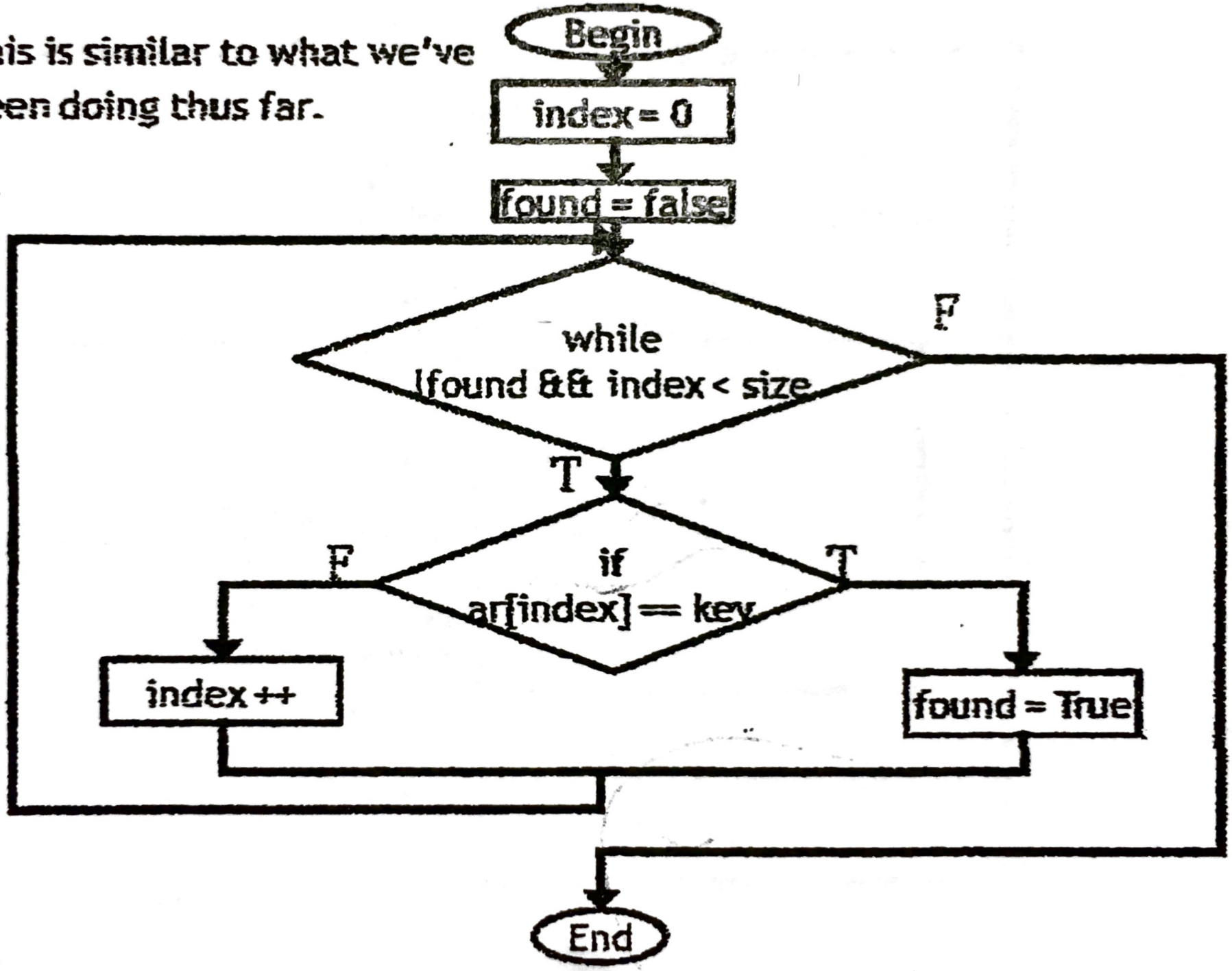
- Search each element until the element is found or until you've completed the array

- Pros:
  - Easy to implement
  - Works well with unsorted arrays

- Cons: Inefficient
  - imagine an array of 100 elements

Sequential Search

This is similar to what we've been doing thus far.



in C++... How does it work?

```
const int AR_SIZE = 5;
int intAr[AR_SIZE] = { 5, 12, 6, 2, 4};
int index=0;
bool found = false;
int ikey = 6;

while (!found  && index < AR_SIZE)
{
    if (intAr[index] == ikey)
    {
      found = true;
    }
    else
    {
      index ++;
    }
}
```

index	found	ikey
0	false	6

intAr

[0]	[1]	[2]	[3]	[4]
5	12	6	4	2

index	found	ikey
0	false	6

intAr

[0]	[1]	[2]	[3]	[4]
5	12	6	4	2

index	found	ikey
1	false	6

intAr

[0]	[1]	[2]	[3]	[4]
5	12	6	4	2

index	found	ikey
2	false	6

intAr

[0]	[1]	[2]	[3]	[4]
5	12	6	4	2

index	found	ikey
2	true	6

intAr

[0]	[1]	[2]	[3]	[4]
5	12	6	4	2

Completes with index = 2 → the location of the key (6)

Sorting

- Is necessary for more efficient searches
  - Like the binary search

- We must first sort the array
  - Not all sorts are created equally

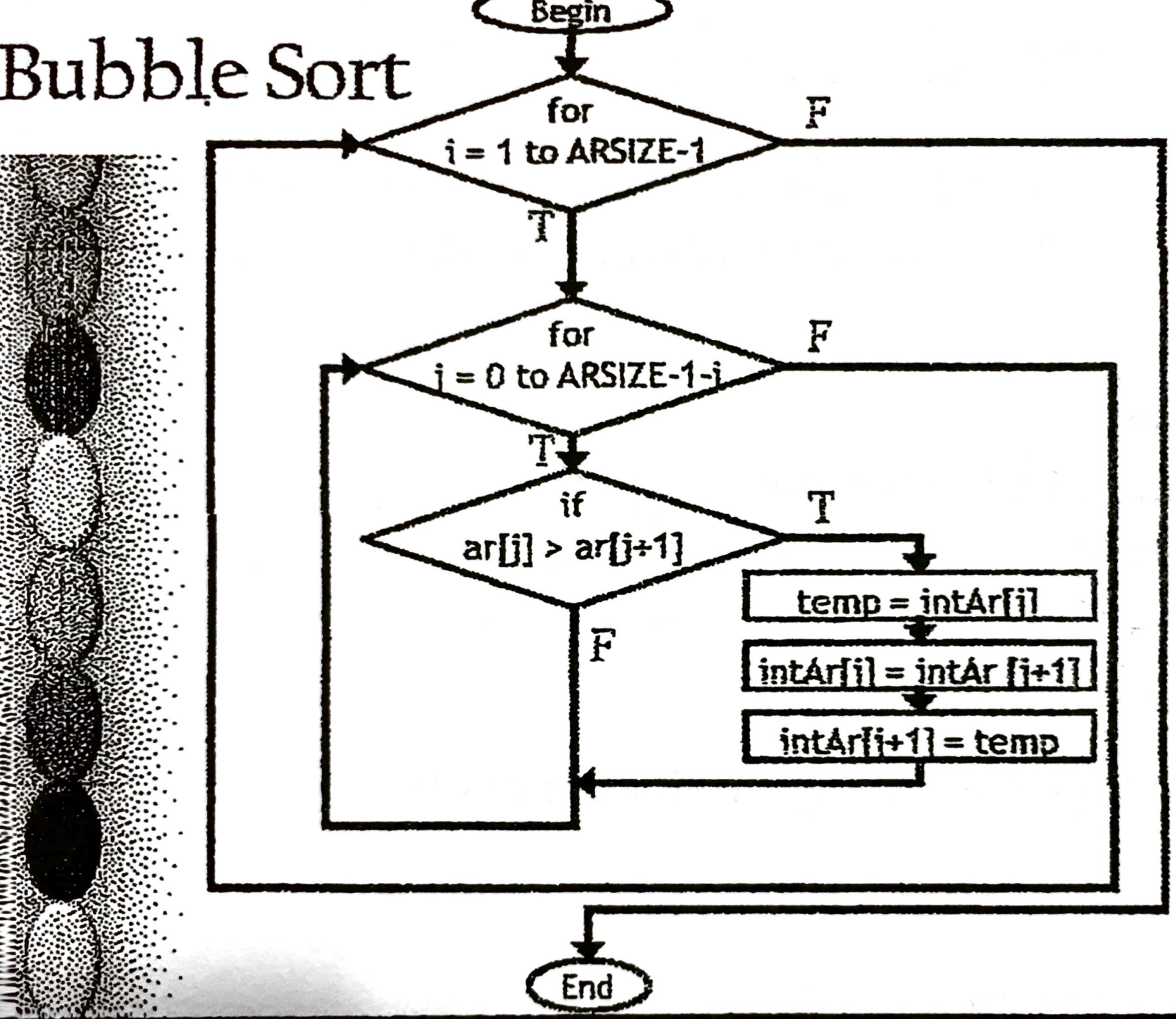
Bubble Sort

- The **bubble sort** gets its name because as elements are sorted they gradually "bubble" (or rise) to their proper positions
  - like bubbles rising in a glass of soda.

- compares **adjacent elements** of an
  - swaps them if they are out of order
  - repeat

- Very easy to implement BUT...
  - very Inefficient → especially if the array is already in order

Bubble Sort


Begin
for
i = 1 to ARSIZE-1
F
T
for
j = 0 to ARSIZE-1-i
F
T
if
ar[j] > ar[j+1]
T
F
temp = intAr[i]
intAr[i] = intAr [i+1]
intAr[i+1] = temp
End

Bubble Sort

```
void BubbleSort(int intAr[])
{
    int temp;
    int i, j;

    for (i = 0 ; i < AR_SIZE - 1; i++)
    {
        for (j = 0; j < AR_SIZE - 1 - i; j++)
        {
            if (intAr[j] > intAr[j+1])
            {
                temp = intAr[j];
                intAr[j]=intAr[j+1];
                intAr[j+1]= temp;
            }
        }
    }
```

intAr	[0]	[1]	[2]	[3]	[4]
	2	4	5	6	12

i	j	temp	AR_SIZE
0	0	4	5
1	1	5	
2	2	6	
3	3	12	
4	4	5	
	0	6	
	1	12	
	2	5	
	3	12	
	0	5	
	1	6	
	2	12	
	0	5	
	1	6	
	0	5	
	1		

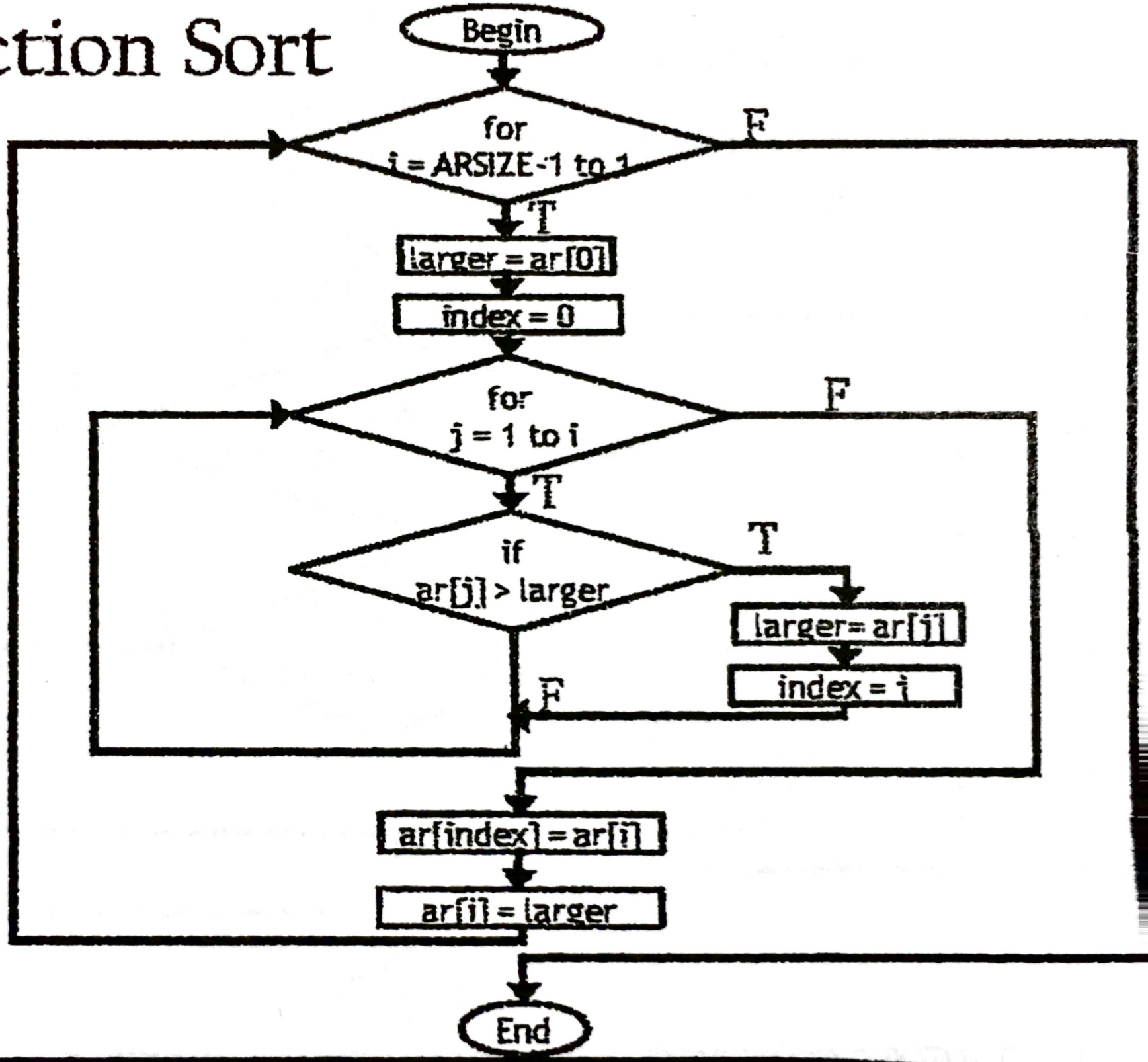
Now that we know the last position is correct we increment i and loop through j again - BUT we don't have to do the last comparison so we loop through j - but 1 less time

Selection Sort

- Finds the largest value and puts it at the end
  - then checks the array - 1 element
  - repeats

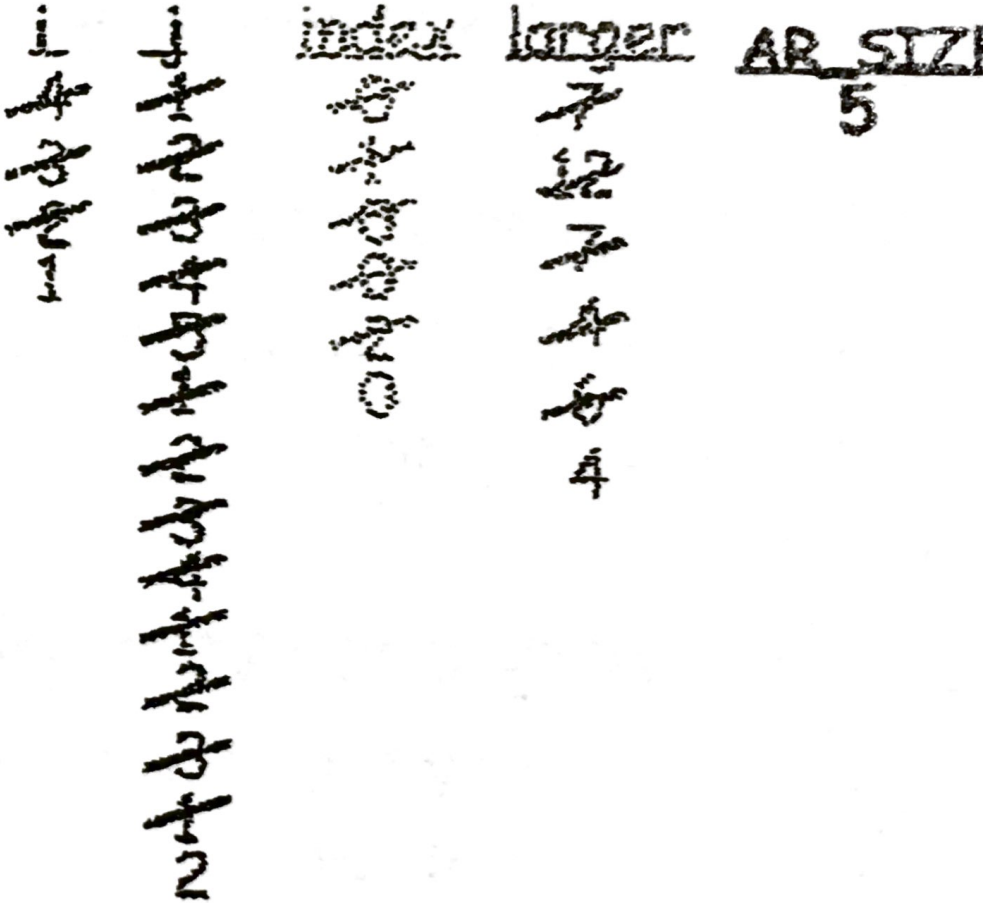
- Easy to implement BUT...
  - Inefficient → imagine 1000 elements
  - Better than the bubble sort

Selection Sort



in C++... How does it work?

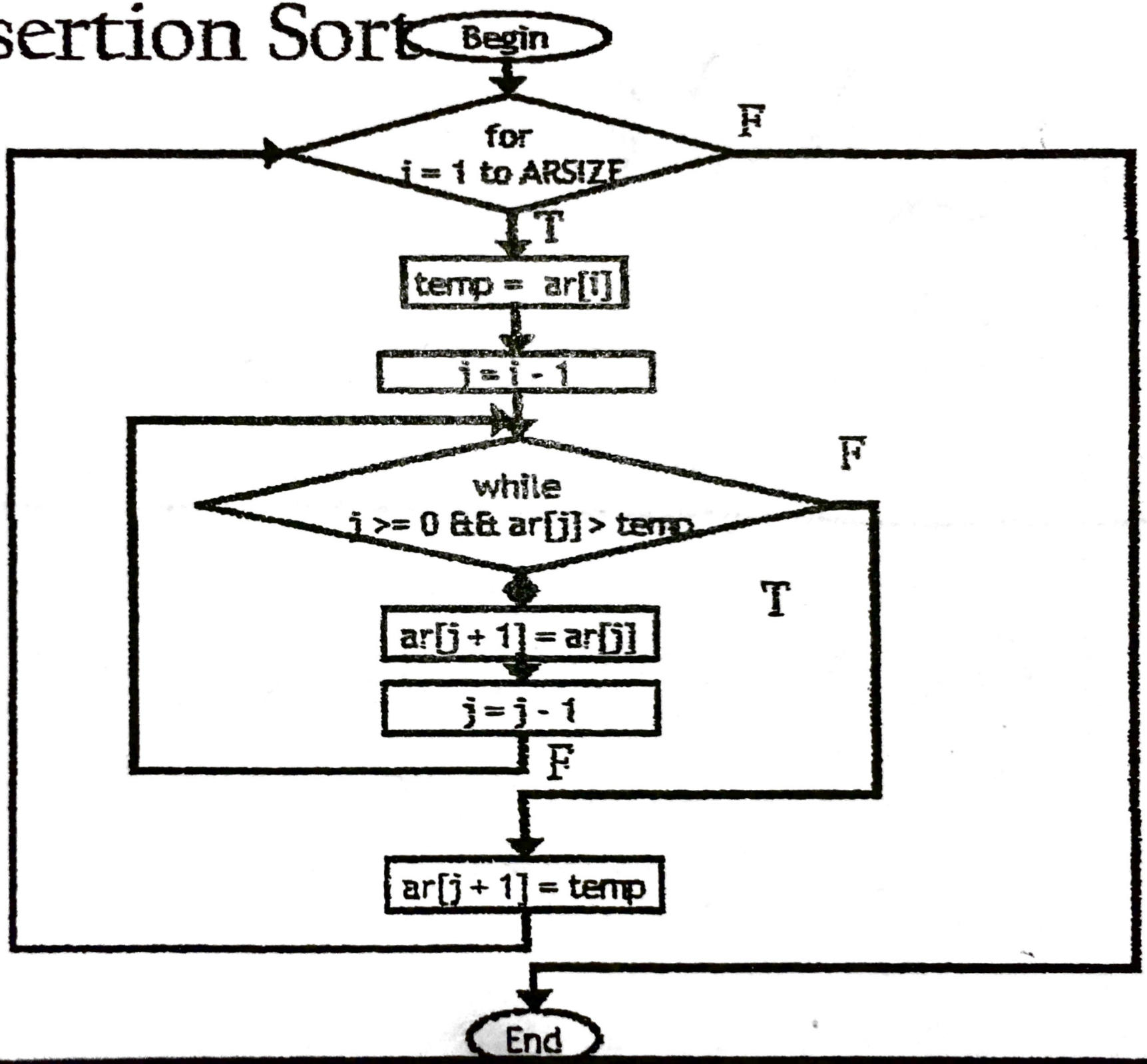
```
const int AR_SIZE = 5;
int intArray[AR_SIZE] = { 7, 12, 6, 4, 2};
for (int i = AR_SIZE-1; i >= 1; --i)
{
    larger = intArray[0];
    index = 0;
    for (int j = 1 ; j <= i; ++j)
    {
        if (intArray[j] > larger)
        {
            larger = intArray[j];
            index=j;
        }
    }
    intArray[index] = intArray[i];
    intArray[i] = larger;
}
```



Insertion Sort

- Passes through the array only once
  - More efficient than the selection sort
- Sorts like you would sort a hand of cards
  - pick up one card
  - put it in the correct location
  - repeat

Insertion Sort



in C++... How does it work?

```
const int AR_SIZE = 5;
int intAr[AR_SIZE] = { 7, 12, 6,14  22};

for (int i = 1; i < AR_SIZE; ++i)
{
      temp = intAr[i];
      j = i - 1;
      while ( j >= 0 && intAr[j] > temp)
      {
            intAr[ j + 1 ] = intAr[j];
            j = j - 1;
      }
      intAr[ j + 1 ] = temp;
}
```

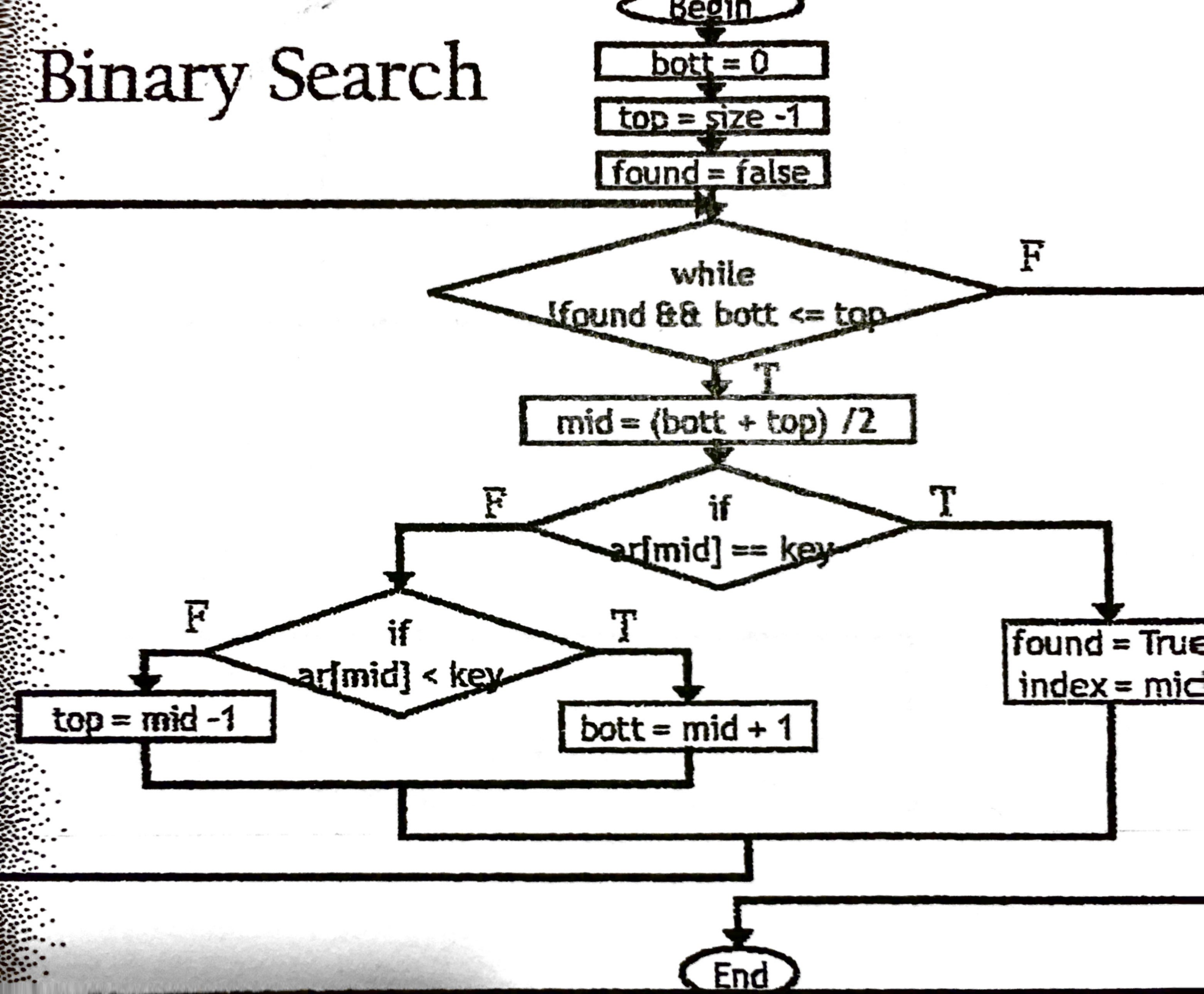
intAr	[0]	[1]	[2]	[3]	[4]
	6	7	12	14	22

i	j	temp	AR_SIZE
1	0	12	5
2	1	6	
3	2	14	
4	3	22	

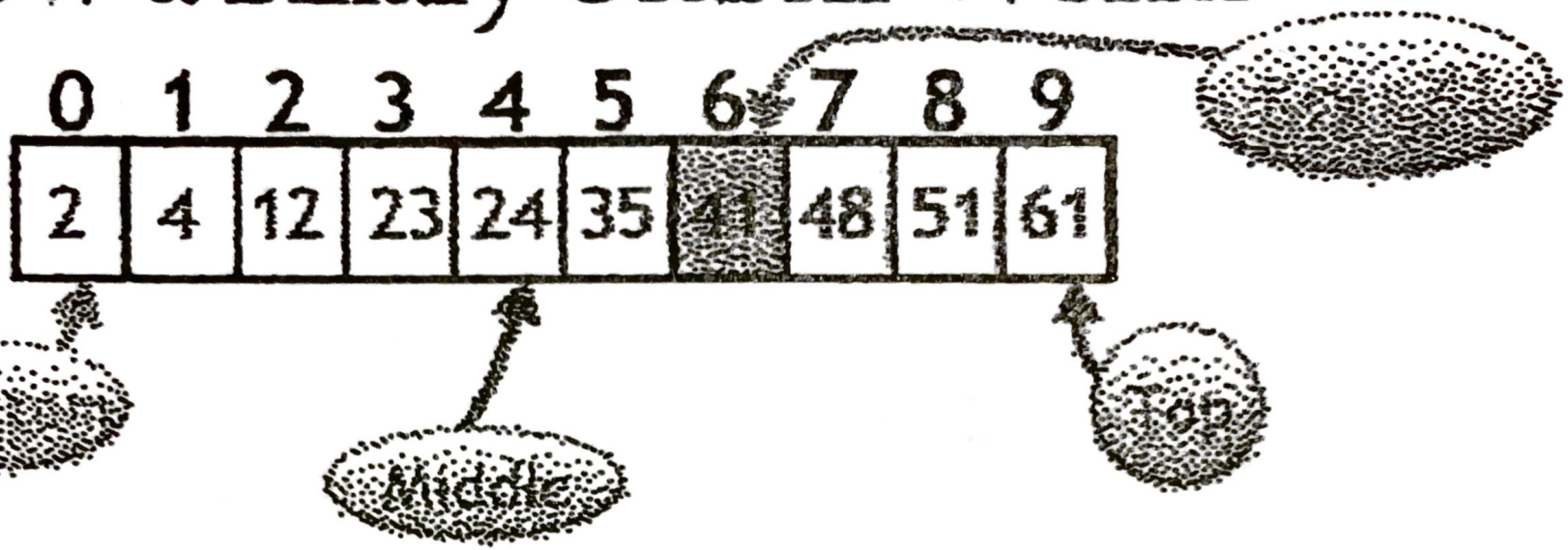
Binary Search

- Now that we know how to sort...
  - We can explore better search routines
- For a Binary Search the array must be in order
- Gets its name because the algorithm continually divides the list into two parts.
- Much more efficient than the Sequential Search

Binary Search



How a Binary Search Works



Keep track of the top, bottom and middle indices

Middle = (Top + Bottom) / 2

CHECK the middle value

IF the middle value is equal to the KEY

we're done!

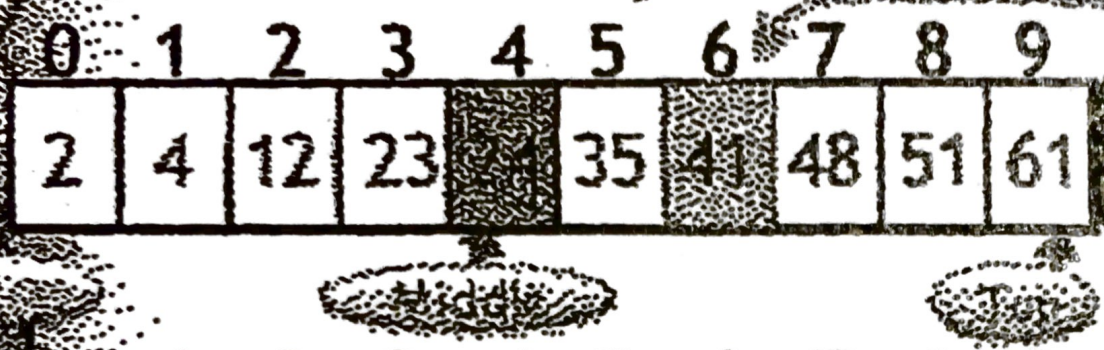
ELSE IF the middle value is less than the K

MOVE the bottom index to middle + 1

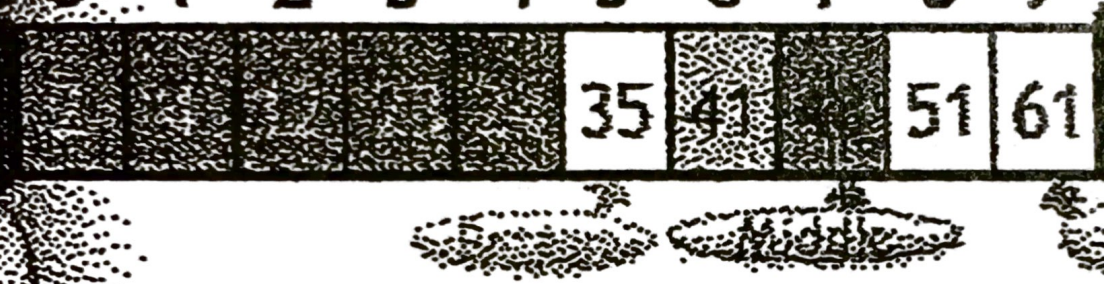
ELSE

MOVE the top index to middle - 1

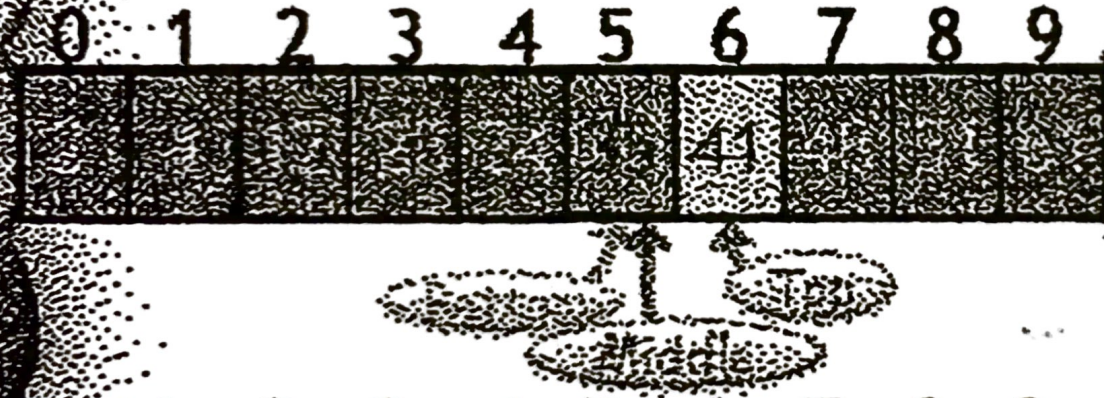
How a Binary Search Works



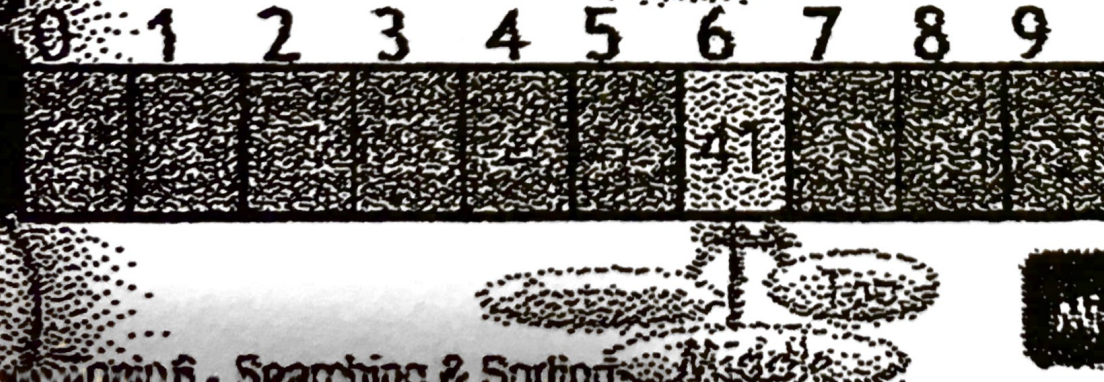
- Set middle index to (top + bottom) / 2
- Check if bottom <= top and if found



- Bottom is <= top and key is not found
- Key is >= Middle so bottom = middle + 1
- middle = (top + bottom / 2)



- Bottom is <= top and key is not found
- Key is < Middle so top = middle - 1
- middle = (top + bottom / 2)



- Bottom is <= top and key is not found
- Key is >= Middle so bottom = middle +
- middle = (top + bottom /2)

- Middle = Key so f

	Bubble	Selection	Insertion
Outer loop	For 0 → ArrSize -1	For ArrSize → 0	For 0 → Arr_size -1
inner loop	For	For	While
Task	Swap element	find lgest put at end	goes through each element are put in corrct order