Basic algorithms with arrays

Finding minimum in array of integers

```
Input:
        A // array of integers
        n // number of elements in array (array size)
Output:
        Min // value of element with smallest value
Minimum(A, n) // name of algorithm and parameters
        Min = A[0] // initialize minimum as first element
        for i = 1 to n-1 // look at remaining elements
                if A[i] < Min then
                        Min = A[i]
                endif
        endfor
        return Min
endMinimum
```

2

Here is a simple algorithm written in pseudocode.

Remember, pseudocode is a “language” that allows designing and analyzing algorithms, but that is short of a “programming language”. Computers cannot execute pseudocode directly. Pseudocode is the first of the 2 step approach for designing and implementing algorithms.

Notice the main parts of the pseudocode specification of the algorithm: input, output, and actual algorithm consisting of step-by-step instructions. The name of the algorithm is Minimum, and it has two (input) parameters: an array of integers called A, and the length of the array (i.e. the number of elements) called n.

Finding minimum in array of integers

```
Input:
        A // array of integers
        n // number of elements in array (array size)
Output:
        Min // value of element with smallest value
Minimum(A, n) // name of algorithm and parameters
        Min = A[0] // initialize minimum as first element
        for i = 1 to n-1 // look at remaining elements
                if A[i] < Min then
                        Min = A[i]
                endif
        endfor
        return Min
endMinimum
```

// or # introduces comment
what follows is not part of algorithm,
it rather explains algorithm

3

In order to improve algorithm (and code) readability, the algorithm designer (i.e. you) adds comments. Comments are intended to clarify, explain, and are not part of the algorithm instructions. Comments are not executed, they are not instructions.

Finding minimum in array of integers

```
Input:
        A // array of integers
        n // number of elements in array (array size)
Output:
        Min // value of element with smallest value
Minimum(A, n) // name of algorithm and parameters
        Min = A[0] // initialize minimum as first element
        for i = 1 to n-1 // look at remaining elements
                if A[i] < Min then
                        Min = A[i]
                endif
        endfor
        return Min
endMinimum
```

Algorithm is written for arrays of integers in general

- *Not for say an array with elements 0 or 1*
- *Not for say an array of length 10*
- *The size of the array is a variable, called n*
- *The array is a variable called A*
- *Its elements can have any value*
- *A and n are input variables or parameters*

4

As discussed in the previous lecture, the algorithms are written to be as general as possible, in order to be useful in as many circumstances as possible. The algorithm parameters stand for an infinite number of input combinations: arrays with 10, 1000, 10000 elements, with elements with large or small values, etc.

Finding minimum in array of integers

```
Input:
        A // array of integers
        n // number of elements in array (array size)
Output:
        Min // value of element with smallest value
Minimum(A, n) // name of algorithm and parameters
        Min = A[0] // initialize minimum as first element
        for i = 1 to n-1 // look at remaining elements
                if A[i] < Min then
                        Min = A[i]
                endif
        endfor
        return Min
endMinimum
```

The algorithm iterates over array
- *Traverses array, checks all elements*
- *First element doesn't need to be checked*

Minimum updated when smaller element found
- *Minimum is stored in a variable called Min*

Essential to initialize Min, or else first comparison to A[i] doesn't make sense

5

This particular algorithm finds the minimum in an array of numbers (integer numbers).

The output is the minimum over all elements of A, which is called Min, and it is "returned" to the caller of the minimum algorithm. The caller stands here generically for whomever wanted to know the minimum of the array A of length n. This could be a user, and the minimum value is returned through a printout on the screen, or some other algorithm, case in which the minimum algorithm is used as a sub-algorithm.

The strategy is to initialize the minimum (stored in variable Min) to be the first element in the array, and then to traverse the remaining part of the array (i.e. elements index 1 to n-1) to see if there is a smaller element than the current minimum. If such an element is found, Min is updated. Min stores the current minimum, and has the true or final minimum only at the end, because it could be that the last element in the array is the actual minimum.

Notice how the array is traversed with a for loop. The for loop iterates over the array index i. The first value for i is 1, since we do not need to test against the first element of the array.

The decision whether to update Min is done with an if statement. The assignment

Min = A[i] is only executed when A[i] is less than the current minimum. There is no else branch for the if. In other words, if the condition A[i] < Min is not true (i.e. it is false, or A[i] >= Min), nothing happens, no instruction needs to be executed.

Finding minimum in array of integers

```
Input:
        A // array of integers
        n // number of elements in array (array size)
Output:
        Min // value of element with smallest value
Minimum(A, n) // name of algorithm and parameters
        Min = A[0] // initialize minimum as first element
        for i = 1 to n-1 // look at remaining elements
                if A[i] < Min then
                        Min = A[i]
                endif
        endfor
        return Min
endMinimum
```

endfor, endif, endMinimum make algorithm more readable
- They reinforce indentation

6

Notice that indentation and the endif and endfor statements help structure the pseudocode. The if-endif statement is "inside the for loop", meaning that the if statement is executed at every iteration.

Indentation is achieved using multiple tabs.

Finding minimum in array of integers

```
Input:
        A // array of integers
        n // number of elements in array (array size)
Output:
        Min // value of element with smallest value
Minimum(A, n) // name of algorithm and parameters
|       Min = A[0] // initialize minimum as first element
|       for i = 1 to n-1 // look at remaining elements
|       |       if A[i] < Min then
|       |       |       Min = A[i]
|       |       endif
|       endfor
|       return Min
endMinimum
```

endfor, endif, endMinimum make algorithm more readable
- *They reinforce indentation*
- *Black vertical bars could also be used*

One can also use this vertical lines, but they are hard to type and format.

Finding minimum in array of integers

```
Input:
        A // array of integers
        n // number of elements in array (array size)
Output:
        Min // value of element with smallest value
Minimum(A, n) // name of algorithm and parameters
        Min = A[0] // initialize minimum as first element
        for i = 1 to n-1 // look at remaining elements
                if A[i] < Min then
                        Min = A[i]
                endif
        endfor
        return Min
endMinimum
```

The body of the for loop
- One extra tab relative to the for *line*

8

Notice how the body of the for loop is indented with respect to the for statement.

Finding minimum in array of integers

```
Input:
        A // array of integers
        n // number of elements in array (array size)
Output:
        Min // value of element with smallest value
Minimum(A, n) // name of algorithm and parameters
        Min = A[0] // initialize minimum as first element
        for i = 1 to n-1 // look at remaining elements
                if A[i] < Min then
                        Min = A[i]
                endif
        endfor
        return Min
endMinimum
```

Conditional statement

Min is only set to the current element if the current element is smaller than current minimum

9

The instructions to be executed on the then branch of the if are also indented.

Finding minimum in array of integers

```
Input:
        A // array of integers
        n // number of elements in array (array size)
Output:
        Min // value of element with smallest value
Minimum(A, n) // name of algorithm and parameters
        Min = A[0] // initialize minimum as first element
        for i = 1 to n-1 // look at remaining elements
                if A[i] < Min then
                        Min = A[i]
                endif
        endfor
        return Min
endMinimum
```

Once all elements are checked, Min will store minimum Min is returned (output, printed, communicated to whomever wanted to know the minimum of the array A)

10

The return statement has to return the algorithm output. Notice that Min is listed as the algorithm output and it is returned.

iClicker question

```
Input:
        A // array of integers
        n // number of elements in array
Output:
        Min // value of smallest element
Minimum(A, n) // name of algorithm
        Min = A[0] // initialize min as first el.
        for i = 1 to n-1 // look at rem. els.
                if A[i] < Min then
                        Min = A[i]
                endif
        endfor
        return Min
endMinimum
```

If "<" is changed to ">", the algorithm will

A. Compute the maximum but it will be confusing because of the comments and variable names.
B. Crash because Min is assigned larger and larger values.
C. Still return the minimum with the warning that the incorrect comparison is used.
D. Enter an infinite loop since the minimum is not found.
E. Run faster.

11

The return statement has to return the algorithm output. Notice that Min is listed as the algorithm output and it is returned.

Tracing algorithm for testing

```
Input:
        A // array of integers
        n // number of elements in array (array size)
Output:
        Min // value of element with smallest value
Minimum(A, n) // name of algorithm
        Min = A[0] // initialize minimum as first element
        for i = 1 to n-1 // look at remaining elements
                if A[i] < Min then
                        Min = A[i]
                endif
        endfor
        return Min
endMinimum
```

Let's see if the algorithm works for A = {3, 7, 1, 2, -1}, n = 5

We'll trace the algorithm for the given input, keeping track of all variables

n	i	A[i]	Min
5			3
	1	7	
	2	1	1
	3	2	
	4	-1	-1

When a variable changes, write the new value in the next row.

At the end Min is -1 which is correct

12

Although pseudocode cannot be run on a computer, we can still test the algorithm by tracing its execution.

Tracing an algorithm implies choosing an input (actual values, not just variable names) and executing the step by step instructions of the algorithm while keeping track of all variables involved in the algorithm.

Here we chose an array with 5 elements and we keep track of variable values in a table. The table is filled out from top to bottom. Time advances from top to bottom.

The first instruction of the algorithm is Min = A[0], which assigns 3 to variable Min.

Then we enter the for loop. Since n is 5, n-1 is 4, so we have to execute the for loop for i = 1, 2, 3, and 4. The for loop will be executed 4 times.

The first time around i is 1. We evaluate the if condition: is A[1] less than Min? Is 7 less than 3? No, so we do not enter the then branch, we get to endfor, which signals the end of the current iteration. i was 1 now it is 2, and 2 is still less than or equal to n-1, so on to another iteration of the loop. Is A[2] < Min? Yes, so update Min, which is now 1. Another iteration, no update this time since 2 is not less than 1. Final iteration, i is 4, A[i] is -1 which is less than Min, so Min is set to -1.

The for loop is done, the algorithm returns Min, which is -1.

Tracing the algorithm is of course not a proof that the algorithm is correct. We now know that the algorithm works correctly for this particular input, but we cannot extrapolate to “knowing for sure” that the algorithm works correctly for all input. Tracing however increases the our confidence in the correctness of the algorithm. Additional traces would increase our confidence even more.

Algorithm analysis

```
Input:
        A // array of integers
        n // number of elements in array (array size)
Output:
        Min // value of element with smallest value
Minimum(A, n) // name of algorithm
        Min = A[0] // initialize minimum as first element
        for i = 1 to n-1 // look at remaining elements
                if A[i] < Min then
                        Min = A[i]
                endif
        endfor
        return Min
endMinimum
```

- Absolute running time
  - e.g. 1.45s
  - Depends on
    - Actual values for n and A
    - Computer where run
      - Smart phone
      - Old PC
      - New PC
      - Mac
    - Context
      - Computer busy running other programs

13

In addition to correctness, another concern is to estimate the running time.

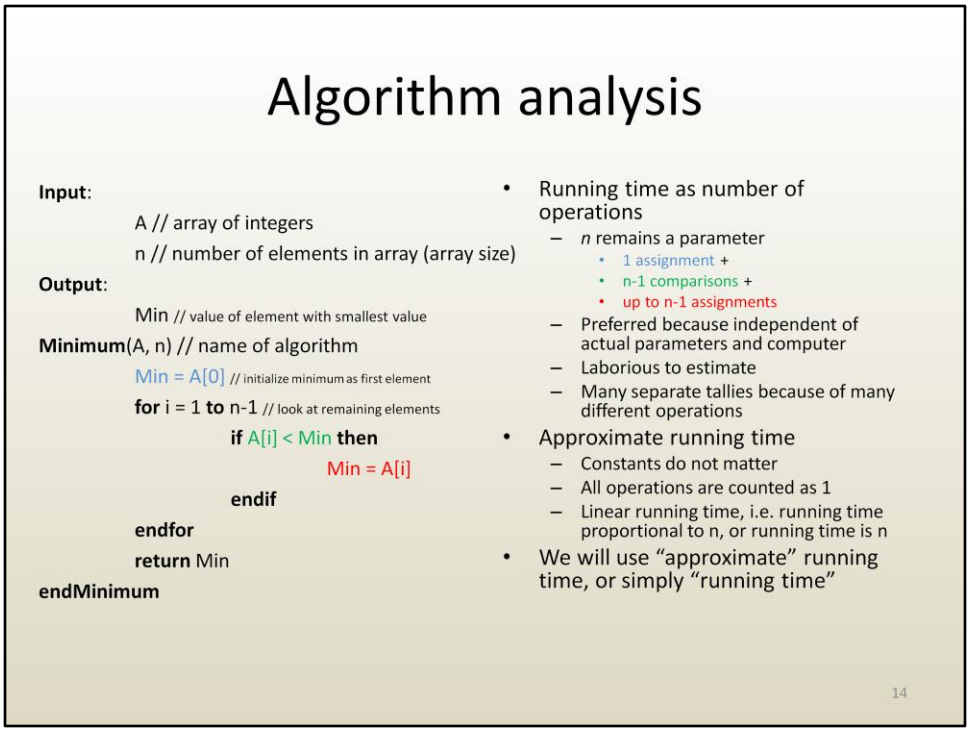
One option is to start a timer when the algorithm starts executing and to stop the timer as soon as the output is available. That could be done and is done. However, this absolute or actual running time of the algorithm is not very informative.

First, one has to run the algorithm on a computer. We only have pseudocode, so that won't work. Getting really good at tracing and measuring the time it takes us to trace the algorithm is also a bad idea.

Second, the absolute running time depends on the actual input. Finding the minimum on a long array will take longer than on a short array. Even when the arrays have the same length, an array sorted from large to small values will take longer to process than an array sorted from small to large, because the then branch of the if is taken all the time in one case, and never in the other.

Third, the absolute running time depends on the computer where the algorithm is run, and on the context (i.e. how busy the computer is doing other things).

What is needed is a way of estimating running time directly from the pseudocode, that is independent of the actual input, and of the actual computer where it is run.



The idea is to count the number of operations, as a function of the algorithm parameters. The parameter that dictates the running time is n, the number of elements in the array.

Although there are many types of instructions (e.g. evaluation of logical expression, assignment), and although different instructions take different times to execute, how long exactly each instruction takes depends on the computer where it is run. To simplify, we count all instructions as 1.

Another approximation is to ignore constants in our running time tally. For example Min = A[0] is done exactly once, and 1 is a constant that can be ignored since the for loop body, which is executed once for every array element, will dominate. Another example of ignoring constants is to always count the body of the for loop as 1. Depending on whether Min is updated or not, the for loop body could cost 1 (for the if) and 1 + 1 (for the if and the assignment), but 2 is only a constant times greater than 1.

Consequently, the approximate running time of the algorithm is n, or linear running time, or proportional to n. This means that if the array size doubles in length, the running time will double.

In conclusion, instead of measuring running time in seconds, we prefer specifying how the running time of the algorithm changes when the size of the input changes.

Finding brightest pixel in BW picture

```
Input:
        I // 2-D array storing pixel intensities
        w // image width (number of pixels in a row)
        h // image height (number of pixels in a column)
Output:
        Max // value of brightest pixel
MaxImage(I, w, h) // name of algorithm
        Max = I[0][0] // initialize maximum as top left pixel
        for i = 0 to h-1 // traverse all rows
                for j = 0 to w-1 // traverse all elements of current row
                        if I[i][j] > Max then
                                Max = I[i][j]
                        endif
                endfor
        endfor
        return Max
endMaxImage
```

Similar to 1-D minimum, but:
- *Now two, nested, for loops*
- *Logic is reversed: update Max when smaller than current pixel*

15

Here is another example of algorithm written in pseudocode.

The input is a digital image, encoded as a two dimensional array. The image size is w x h. The image is black and white meaning that there is a single intensity channel per pixel.

The strategy is similar to finding the 1-D minimum. One change is that we now search for the maximum, so the temporary (i.e. current) maximum is updated when the current element is greater than the maximum. A second change is that traversing the entire image (one has to look at the entire image since the brightest pixel can be anywhere) is done with two nested for loops.

The first for loop says: look at all rows. The second for loops says: for the current row, look at all columns (i.e. elements).

Tracing the algorithm

```
Input:
    I // 2-D array storing pixel intensities
    w // image width
    h // image height
Output:
    Max // value of brightest pixel
MaxImage(I, w, h) // name of algorithm
    Max = I[0][0] // initialize Max
    for i = 0 to h-1 // traverse all rows
        for j = 0 to w-1 // trav. all els.
            if I[i][j] > Max then
                Max = I[i][j]
            endif
        endfor
    endfor
    return Max
endMaxImage
```

- Let's say I is following 3x4 image

100	129	74	200
30	93	233	93
55	23	100	123

- Then the algorithm trace is

w	h	i	j	I[i][j]	Max
4	3				100
		0	0	100	
			1	129	129
			2	74	
		0	3	200	200
		1	0	30	
			1	93	
			2	233	233

16

Let's trace the algorithm for the given image. Notice how, for each value of i, there are 4 values for j. That is because there are 4 elements in each row.

Max is initialized to the first pixel. Notice that the algorithm tests against the first pixel, which is unnecessary, since we know that's not going to provide a larger value. Although the test is redundant, the algorithm is correct. Avoiding that test would require unnecessary work to single that case out.

Tracing the algorithm

```
Input:
    I // 2-D array storing pixel intensities
    w // image width
    h // image height
Output:
    Max // value of brightest pixel
MaxImage(I, w, h) // name of algorithm
    Max = I[0][0] // initialize Max
    for i = 0 to h-1 // traverse all rows
        for j = 0 to w-1 // trav. all els.
            if I[i][j] > Max then
                Max = I[i][j]
            endif
        endfor
    endfor
    return Max
endMaxImage
```

100	129	74	200
30	93	233	93
55	23	100	123

w	h	i	j	I[i][j]	Max
4	3				100
		0	0	100	
			1	129	129
			2	74	
			3	200	200
		1	0	30	
			1	93	
			2	233	233
			3	93	
		2	0	55	
			1	23	
			2	100	
			3	123	

17

Once we are done traversing the entire image, Max will store the brightest pixel. The last update to Max happened “a while ago”, but there was no way to know that there will not be a brighter pixel, so we have to look at the entire image.

Tracing the algorithm

```
Input:
    I // 2-D array storing pixel intensities
    w // image width
    h // image height
Output:
    Max // value of brightest pixel
MaxImage(I, w, h) // name of algorithm
    Max = I[0][0] // initialize Max
    for i = 0 to h-1 // traverse all rows
        for j = 0 to w-1 // trav. all els.
            if I[i][j] > Max then
                Max = I[i][j]
            endif
        endfor
    endfor
    return Max
endMaxImage
```

- Running time
  - Proportional to w*h (i.e. to the number of pixels)
  - Linear with the number of pixels
  - Quadratic with the image width or height
  - The ranges of the nested for loops are multiplied
  - w steps for each h step
  - Indices i and j change like the digits of a 2-digit counter
    - 00, 01, 02, 03, 10, 11, 12, 13, 20, 21, 22, 23

18

The running time is linear with the image area (i.e. the number of pixels) and quadratic with the image width (or height). Since the for loops are nested, the amount of work gets multiplied and not added. “For each value of i, consider all values of j”.

Finding minimum in array of integers—

modification to run on sub-array and to return index

Input:
A // array of integers
n // number of elements in array (array size)
i_0 // index where to start looking for the minimum (first elements ignored)
Output:
i_{min}// index of element with smallest value
MinIndex(A, i_0, n) // name of algorithm and parameters
$i_{min} = i_0$ // index of minimum is i_0
for i = i_0+1 **to** n-1 // look at remaining elements
if A[i] < A[i_{min}] **then** // if current elem. is smaller than curr. min.
i_{min} = i // update index of minimum
endif
endfor
return i_{min}
endMinIndex

19

Let us now modify the algorithm for finding the minimum in an array by adding a parameter that tells the algorithm where to begin looking for the minimum. A second modification is to return the index of the minimum as opposed to the actual minimum. As we will see shortly, this will be useful to achieve more complicated algorithms.

We introduce a parameter i0 that tells the algorithm the index of the first element to be considered when finding the minimum. The modified algorithm finds minimum on the sub-array i0 to n-1. The second modification is to keep track of the index of the minimum imin as opposed to the minimum value.

The modified algorithm is more powerful, more general than the earlier algorithm: one can call the modified algorithm with i0 set to 0 and then all array elements are examined, as before, and, the caller knows the array thus the index returned by the modified algorithm can be used to find the minimum value by indexing in the array.

imin is initalized to i0, since we start at i0 and since we do not want to find the minimum value but rather the index of the minimum value.

The for loop starts at i0+1, since i0 does not need to be compared (the same way A[0] could be skipped in the original algorithm).

When a smaller element is found, imin is update to its index.

After the for loop completes, the algorithm returns imin.

Trace

A = {3, 7, 1, 2, -1}, n = 5, $i_0 = 0$

n	i_0	i	A[i]	i_{min}	A[i_{min}]
5	0			0	3
		1	7		
		2	1	2	1
		3	2		
		4	-1	4	-1

Input:

A // array of integers

n // number of elements in array (array size)

i_0 // index where to start looking for the min.

Output:

i_{min}// index of element with smallest value

MinIndex(A, i_0, n) // name of algorithm and parameters

$i_{min} = i_0$ // index of minimum is i_0

for i = i_0+1 **to** n-1 // look at remaining elements

if A[i] < A[i_{min}] **then** // if current elem. is smaller than curr. min.

i_{min} = i // update index of minimum

endif

endfor

return i_{min}

endMinIndex

20

Here is a trace of the algorithm on the same input as the one used to trace the original algorithm. Same results.

Trace

A = {0, 7, 4, 2, 6}, n = 5, i_0 = 2

n	i_0	i	A[i]	i_{min}	A[i_{min}]
5	2			2	4
		3	2	3	2
		4	6		2

Input:

A // array of integers

n // number of elements in array (array size)

i_0 // index where to start looking for the min.

Output:

i_{min}// index of element with smallest value

MinIndex(A, i_0, n) // name of algorithm and parameters

i_{min} = i_0 // index of minimum is i_0

for i = i_0+1 **to** n-1 // look at remaining elements

if A[i] < A[i_{min}] **then** // if current elem. is smaller than curr. min.

i_{min} = i // update index of minimum

endif

endfor

return i_{min}

endMinIndex

21

Here is a trace where i0 is 2, indicating that one wants the minimum of elements with indices 2, 3, and 4. The imin returned is 3, since A[3] is the smallest of A[2], A[3], and A[4].

Sorting an array of integers

```
Input:
        A // array of integers
        n // number of elements in array (array size)
Output:
        B // array with elements of A, sorted in ascending order
SortMin(A, n) // idea is to repeatedly extract minimum from original array
        for i = 0 to n-1
                B[i] = A[i] // copy array A into B
        endfor
        for i = 0 to n-2
                i_min = MinIndex(B, i, n)
                Swap(B[i], B[i_min])
        endfor
        return B
endSortMin
```

SortMin uses two sub-algorithms
- *MinIndex*
- *Swap*

22

Using the modified minimum finding algorithm one can easily sort an array.

The algorithm is based on the following idea: the first element in the sorted array is the minimum element in the original array, the second element in the sorted array is the minimum among the original elements other than the overall minimum. In other words, the array is sorted by repeatedly extracting the minimum from the original array and setting the sorted array elements, in order.

The algorithm uses two sub-algorithms. The first one, MinIndex, is the modified minimum algorithm discussed earlier. The second one swaps the value of two variables.

Swapping two variables

```
Input:
        A, B // variables whose values are to be swapped
Output:
        A, B // variables with values swapped
Swap(A, B) // idea: think of swapping liquid of two identical jars
        C = A
        A = B
        B = C
        return A, B
endSwap
```

23

Given two variables as input, their values are modified such that the first variable has the original value of the second variable, and the second variable has the original value of the first variable. This is achieved using a temporary variable called C. Temporary or auxiliary variables are variables other than input and output variables.

Trace

```
Input:
        A // array of integers
        n // number of elements in array (array size)
Output:
        B // array with elements of A, sorted in ascending order
SortMin(A, n) // idea is to repeatedly extract minimum from original array
        for i = 0 to n-1
                B[i] = A[i] // copy array A into B
        endfor
        for i = 0 to n-2
                i_min = MinIndex(B, i, n)
                Swap(B[i], B[i_min])
        endfor
        return B
endSortMin
```

A = {3, 7, 1, 2, -1}, n = 5

n	B[0]	B[1]	B[2]	B[3]	B[4]	i	i_{min}
5	3	7	1	2	-1	0	4
	-1	7	1	2	3	1	2
	-1	1	7	2	3	2	3
	-1	1	2	7	3	3	4
	-1	1	2	3	7		

24

Notice that this sorting algorithm leaves the original array unchanged. For this the original array is first copied into a different, output, array B, which is then sorted. Other sorting algorithms might decide to shuffle the elements of the input array. The decision is based on whether the original array is still needed in its initial form or not.

The algorithm proceeds by repeatedly finding the minimum in the part that has not yet been sorted, to build the sorted array. The array B has a part that's already been sorted (left) and a part that's yet to be sorted (right). At the beginning the left part is 0 in length and the right part is n in length. At the end, the sorted part is n in length and the unsorted part is 0 in length, meaning that we are done sorting the array.

Here is a trace. For the first iteration of the for loop, find the index of the minimum element in B, from 0 to 4. That is 4. Then swap element 0 with element 4. Now the minimum is in its place, index 0. The sorted part (red) has one element. 4 elements remain to be sorted (green).

Second iteration, i is 1. Find the index of minimum element in B starting from index 1 to index 4. That is 2. Swap B[1] and B[2].

Third iteration, i is 2. Find the index of minimum element in B from 2 to 4. That is 3.

Swap B[3] and B[3].

Fourth iteration, i is 3. Find the index of minimum element in B from 3 to 4. That is 4.
Swap B[3] and B[4].

No more iteration. The array is sorted.

iClicker question

```
Input:
        A // array of integers
        n // number of elements in array
Output:
        B // array with elements of A, sorted
SortMin(A, n) // idea is to repeatedly extract min
        for i = 0 to n-1
                B[i] = A[i] // copy A into B
        endfor
        for i = 0 to n-2
                i_min = MinIndex(B, i, n)
                Swap(B[i], B[i_min])
        endfor
        return B
endSortMin
```

Why not use instead of MinIndex the original minimum algorithm that returns the minimum value in the array?

A. We could have, but then we needed to swap B[i] and Min.
B. Because we need to know where the minimum is located, and not its value.
C. Both A and B are correct.
D. Neither answer is correct.
E. I give up.

25

Running time

```
Input:
        A // array of integers
        n // number of elements in array (array size)
Output:
        B // array with elements of A, sorted in ascending order
SortMin(A, n) // idea is to repeatedly extract minimum from original array
        for i = 0 to n-1
                B[i] = A[i] // copy array A into B
        endfor
        for i = 0 to n-2
                i_min = MinIndex(B, i, n)
                Swap(B[i], B[i_min])
        endfor
        return B
endSortMin
```

MinIndex takes n-i time
Swap takes constant time
Overall running time
$n+(n-1)+\ldots+2 = n(n+1)/2-1$
Ignoring constants: n^2

26

When analyzing the running time, we look at the algorithm from the beginning to the end.

We notice the first for loop, which has a cost of n (since the body of the for loop implies a constant number of operations and since there are n for loop iterations.

We notice the second for loop. The index goes from 0 to n-2, which also means linear in n since we ignore constants. Then we have to analyze the body of the for loop, which includes two sub-algorithms, so we have to analyze the running time of each of those sub-algorithms. The Swap sub-algorithm takes constant time, which is less than the MinIndex algorithm, thus Swap can be ignored.

MinIndex's running time is proportional to n-i. As the main for loop is executed, i starts at 0 an goes to n-2. Consequently the total amount of work is proportional to the sum of the first n integers, which is proportional to n squared. The running time of this algorithm is quadratic (as opposed to linear). If n doubles, the running time quadruples, i.e. it grows by a factor of four. For large n values, this algorithm is very slow. For example if n is one billion, this algorithm never finishes, whereas a linear algorithm can easily run on one billion elements.

Quadratic running time

- n = 1,000,000 -> running time is 10^{12}
- n = 1,000,000,000 -> running time is 10^{18}
- Quadratic running time algorithms are impractically slow for large inputs
- There are nlogn (n times logarithm of n) sorting algorithms
  - $\log_{10}$ (1,000,000) = 6
  - $\log_{10}$ (1,000,000,000) = 9
  - nlogn running time is comparable to n running time
  - logn is comparable to a constant

27

Fortunately there are sorting algorithms with an nlogn running time, which are practical for input of any size.

Inserting element in sorted array

```
Input:
        A // sorted array of integers (increasing from left to right)
        n // number of elements in array (array size)
        B // integer to be inserted
Output:
        A // sorted array with n+1 elements (original element and B)
InsertSorted(A, n, B)  // idea is to find where to insert B and then to insert B
        i_ins = n
        for i = 0 to n-1
                if B < A[i] then
                        i_ins = i
                        break
                endif
        endfor
        for i = n down to i_ins+1
                A[i] = A[i-1]
        endfor
        A[i_ins] = B
        return A, n+1
endInsertSorted
```

28

Here is another algorithm, which inserts an element into a sorted array. For this example it is assumed that the original array has been preallocated to a large size, and one does not need to allocate a new array.

The strategy is to first find where to insert the element, then to move the following elements one step to the right, and finally to insert the new element in the empty space created.

The place where the element needs to be inserted is found by starting from the first element and looking for an element that is greater than the element we want to insert. There are two scenarios: we find such an element, and, we don't find such an element (i.e. all original elements are smaller than the element we want to insert).

Let's first discuss the first scenario. Once we find an element that is greater than B, we stop looking, not only because we want to avoid redundant work, but also because we want to find the first element greater than B. The early termination of the for loop is achieved with the instruction break, which breaks out of the current loop immediately, regardless of whether we are at the last iteration or not. iins is the index where B has to be inserted.

Let's now discuss the second scenario: B is greater than all elements of A. In this case,

B should be inserted after all elements of A, i.e. index n (since n-1 is the index of the last element of A). To take care of this case, we initialize iins to n. If the then branch of the if statement is never taken, iins stays to its initial value n.

Now we have to move the elements following the insertion points one step to the right. We start from the last element and go left by subtracting the index i, until iins + 1.

Finally, we insert B and we return the modified array.

Inserting element in sorted array

```
Input:
        A // sorted array of integers (increasing from left to right)
        n // number of elements in array (array size)
        B // integer to be inserted
Output:
        A // sorted array with n+1 elements (original element and B)
InsertSorted(A, n, B)  // idea is to find where to insert B and then to insert B
        i_ins = n
        for i = 0 to n-1
                if B < A[i] then
                        i_ins = i
                        break
                endif
        endfor
        for i = n down to i_ins+1
                A[i] = A[i-1]
        endfor
        A[i_ins] = B
        return A, n+1
endInsertSorted
```

A = {1, 3, 7, 10}, n = 4, B = 6

A[0]	A[1]	A[2]	A[3]	A[4]	i	i_{ins}
1	3	7	10			4
					0	
					1	
					2	2
1	3	7	10	10	4	2
1	3	7	7	10	3	2
1	3	6	7	10		

29

Here is a trace. B is 6, which means that it has to be inserted between 3 and 7. This really means at index 2 (old position of 7), which also means that elements 7 and 10 need to be moved one step to the right.

The insertion point is found correctly since B < A[i] is true for the first time when i is 2, thus iins is 2.

Now we move elements 10 and 7, in this order, one step to the right. 10 is moved to index 4, 7 is moved to index 3. Note that it is important to move elements starting from the right. Starting from the left would overwrite data that we need. For example, if we first move 7 right one element, we lose, i.e. overwrite, the old A[3] (10).

Finally B is written at A[2].

Inserting element in sorted array

```
Input:
        A // sorted array of integers (increasing from left to right)
        n // number of elements in array (array size)
        B // integer to be inserted
Output:
        A // sorted array with n+1 elements (original element and B)
InsertSorted(A, n, B)  // idea is to find where to insert B and then to insert B
        i_ins = n
        for i = 0 to n-1
                if B < A[i] then
                        i_ins = i
                        break
                endif
        endfor
        for i = n down to i_ins+1
                A[i] = A[i-1]
        endfor
        A[i_ins] = B
        return A, n+1
endInsertSorted
```

A = {1, 3, 7, 10}, n = 4, B = 3

A[0]	A[1]	A[2]	A[3]	A[4]	i	i_{ins}
1	3	7	10			4
					0	
					1	
					2	2
1	3	7	10	10	4	2
1	3	7	7	10	3	2
1	3	3	7	10		

30

In order to increase our confidence in the correctness of the algorithm, let's try additional input values. When choosing the input values to try, one aims for potentially challenging input values.

What if there is already an element equal to the number we want to insert? There seems to be no problem.

Inserting element in sorted array

```
Input:
        A // sorted array of integers (increasing from left to right)
        n // number of elements in array (array size)
        B // integer to be inserted
Output:
        A // sorted array with n+1 elements (original element and B)
InsertSorted(A, n, B)  // idea is to find where to insert B and then to insert B
        i_ins = n
        for i = 0 to n-1
                if B < A[i] then
                        i_ins = i
                        break
                endif
        endfor
        for i = n down to i_ins+1
                A[i] = A[i-1]
        endfor
        A[i_ins] = B
        return A, n+1
endInsertSorted
```

A = {1, 3, 7, 10}, n = 4, B = 11

A[0]	A[1]	A[2]	A[3]	A[4]	i	i_{ins}
1	3	7	10			4
					0	
					1	
					2	
					3	
					4	
1	3	7	10	11		

31

What if the value we want to insert is greater than all elements in the original array? We considered this “special” case when we designed the algorithm, now let’s see if the algorithm is handling the special case correctly.

For B equals to 11, B is never less than A[i], thus iins remains 4, which is the value to which it was initialized.

Then the second for loop is asked to iterate i from n, which is 4, down to iins+1, which is 5. Since 4 is less than 5, no iteration is performed.

Finally B is inserted at index 4, and the result is correct.

Inserting element in sorted array

```
Input:
        A // sorted array of integers (increasing from left to right)
        n // number of elements in array (array size)
        B // integer to be inserted
Output:
        A // sorted array with n+1 elements (original element and B)
InsertSorted(A, n, B)  // idea is to find where to insert B and then to insert B
        i_ins = n
        for i = 0 to n-1
                if B < A[i] then
                        i_ins = i
                        break
                endif
        endfor
        for i = n down to i_ins+1
                A[i] = A[i-1]
        endfor
        A[i_ins] = B
        return A, n+1
endInsertSorted
```

A = {1, 3, 7, 10}, n = 4, B = 0

A[0]	A[1]	A[2]	A[3]	A[4]	i	i_{ins}
1	3	7	10			4
					0	0
1	3	7	10	10	4	
1	3	7	7	10	3	
1	3	3	7	10	2	
1	1	3	7	10	1	
0	1	3	7	10		

32

What if the number we want to insert is smaller than all elements of A, which means that it is smaller than the first element of A, since A is sorted in ascending order.

This case is also handled correctly. iins is 0, all elements are moved one step right, and the new minimum is inserted on the first position (i.e. index 0).

Running time

```
Input:
        A // sorted array of integers (increasing from left to right)
        n // number of elements in array (array size)
        B // integer to be inserted
Output:
        A // sorted array with n+1 elements (original element and B)
InsertSorted(A, n, B)  // idea is to find where to insert B and then to insert B
        i_ins = n
        for i = 0 to n-1
                if B < A[i] then
                        i_ins = i
                        break
                endif
        endfor
        for i = n down to i_ins+1
                A[i] = A[i-1]
        endfor
        A[i_ins] = B
        return A, n+1
endInsertSorted
```

Linear in n
- *Find insertion point in i_{ins} steps*
- *Move $n-i_{ins}$ elements*
- *Total: $i + n-i_{ins} = n$*

33

The running time is clearly linear. There are 2 for loops, but they are not nested.

iClicker question

```
Input:
        A // sorted array of integers ascending
        n // number of elements in array
        B // integer to be inserted
Output:
        A // sorted array with n+1 elements
InsertSorted(A, n, B)
        i_ins = n
        for i = 0 to n-1
                if B < A[i] then
                        i_ins = i
                        break
                endif
        endfor
        for i = n down to i_ins+1
                A[i] = A[i-1]
        endfor
        A[i_ins] = B
        return A, n+1
endInsertSorted
```

Can the InsertSorted algorithm be used to sort an array?

A. No, it can only insert one element in an existing array.
B. Yes, start with an empty array and keep inserting one element at the time.
C. No, we already have a sorting algorithm.
D. Yes, and it's going to run in linear time.
E. B and D.

2-D convolution

- Given
  - A large 2-D array A of size wxh
  - A small 2-D array (kernel) B of size 2m+1
- C as A "convolved" with B
  - C has the same size as A
  - Element C[i][j] is weighted sum of elements of A in neighborhood (2m+1) x (2m+1) centered at (i, j)
  - Weights are given by kernel B

	0	1	2	3	4	5	6	7	8	9
0	1	3	1	4	6	7	4	3	4	3
1	2	4	8	7	4	3	1	2	4	4
2	2	6	4	6	5	3	3	8	3	9
3	4	5	4	3	6	1	4	7	4	4
4	6	2	5	4	3	2	4	6	2	6
5	7	4	5	2	2	3	5	6	4	6
6	8	3	3	4	4	4	6	5	5	6
7	1	2	6	2	6	5	3	4	2	6
8	1	1	7	1	7	3	2	3	2	5
9	1	2	8	4	2	1	2	2	3	4

Array A of size 10x10

In figure:
w = h = 10
m = 1
C[7][2] = A[6][1]*B[0][0] + A[6][2]*B[0][1] + A[6][3]*B[0][2] +
A[7][1]*B[1][0] + A[7][2]*B[1][1] + A[7][3]*B[1][2] +
A[8][1]*B[2][0] + A[8][2]*B[2][1] + A[8][3]*B[2][2]

	0	1	2
0	2	1	2
1	2	3	2
2	2	3	9

Kernel 3x3

35

Here is another example of a pseudocode algorithm.

The algorithm achieves 2-D convolution, which means processing a 2-D array by replacing its elements with a weighted average of neighboring elements. Convolution is not necessarily restricted to immediate neighbors. The size of the neighborhood is determined by the size of the kernel.

The algorithm has to do the following:
- Traverse the original array
- For each array element, compute the new value as the weighted average by centering the kernel at the current element

2-D convolution

```
Input:
        A // 2-D array
        w,h // array dimension
        B // 2-D kernel
        2m+1 // square kernel of size (2m+1) x (2m+1)
Output:
        C // 2-D array obtained by convolving A with B
Convolve2D(A, w, h, B, m) // idea is to slide kernel over image and compute convolution
        for i = m to h-1-m // do not process border m thick
            for j = m to w-1-m // do not process border m thick
                C[i][j] = 0
                for k = 0 to 2m
                    for l = 0 to 2m
                        C[i][j] = C[i][j] + A[i-m+k][j-m+l]B[k][l]
                    endfor
                endfor
            endfor
        endfor
        return C
endConvolve2D
```

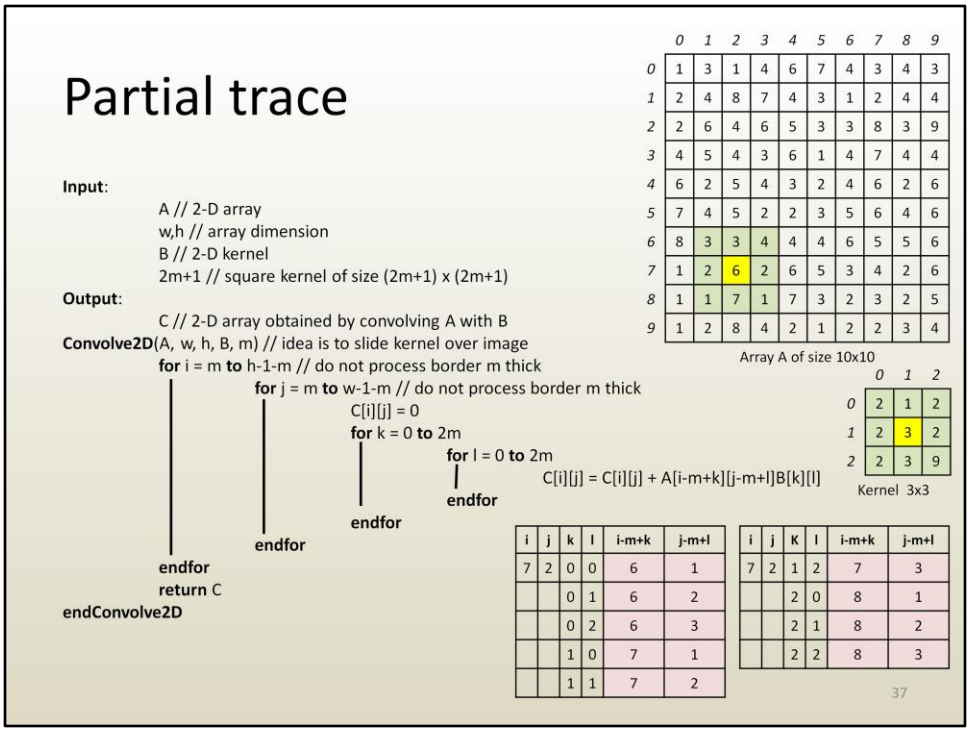
36

Here is the algorithm in pseudocode. The algorithm computes C = A convolved with B.

There are 4 nested loops.

The outer two loops traverse the original array. Notice that a margin of size m is ignored. This is because elements too close to the edge of the original array do not have enough neighbors. In other words they only have a partial neighborhood, and this algorithm has decided to ignore these elements close to the edge.

The new element value is computed as a sum that starts out at 0 and is then incremented by the product between the neighborhood element and the corresponding weight. The sum is computed by the inner two loops, which iterate over the neighborhood (and implicitly the kernel). Notice that the inner loop indices k and l start at 0 and end at 2m, so they can be used directly to index in the 2-D kernel. However, the indices for the neighborhood element have to be computed relative to the top left corner of the neighborhood, which is at [i-m][j-m].



It is impractical to trace the entire algorithm. We would get a good sense of how the algorithm works even if we focus only on one pair of values i and j, here 7 and 2, respectively. This means that we are tracing the inner two loops for the given i and j values.

k and l start at 0, 0, and end at 2, 2, which means that they correctly traverse the kernel; in other words the weights are read correctly from the kernel.

The indices determining the neighborhood element start our at 6, 1 and end at 8, 3, which means that the neighbors to be blended are found correctly.

Partial trace

	0	1	2	3	4	5	6	7	8	9
0	1	3	1	4	6	7	4	3	4	3
1	2	4	8	7	4	3	1	2	4	4
2	2	6	4	6	5	3	3	8	3	9
3	4	5	4	3	6	1	4	7	4	4
4	6	2	5	4	3	2	4	6	2	6
5	7	4	5	2	2	3	5	6	4	6
6	8	3	3	4	4	4	6	5	5	6
7	1	2	6	2	6	5	3	4	2	6
8	1	1	7	1	7	3	2	3	2	5
9	1	2	8	4	2	1	2	2	3	4

Array A of size 10x10

C[7][2] = A[6][1]*B[0][0] + A[6][2]*B[0][1] + A[6][3]*B[0][2] +
A[7][1]*B[1][0] + A[7][2]*B[1][1] + A[7][3]*B[1][2] +
A[8][1]*B[2][0] + A[8][2]*B[2][1] + A[8][3]*B[2][2]

	0	1	2
0	2	1	2
1	2	3	2
2	2	3	9

Kernel 3x3

i	j	k	l	i-m+k	j-m+l
7	2	0	0	6	1
		0	1	6	2
		0	2	6	3
		1	0	7	1
		1	1	7	2

i	j	K	l	i-m+k	j-m+l
7	2	1	2	7	3
		2	0	8	1
		2	1	8	2
		2	2	8	3

38

Element 7, 2 is computed correctly.

Running time: wh(2m+1)(2m+1)

```
Input:
        A // 2-D array
        w,h // array dimension
        B // 2-D kernel
        2m+1 // square kernel of size (2m+1) x (2m+1)
Output:
        C // 2-D array obtained by convolving A with B
Convolve2D(A, w, h, B, m) // idea is to slide kernel over image
        for i = m to h-1-m // do not process border m thick
            for j = m to w-1-m // do not process border m thick
                C[i][j] = 0
                for k = 0 to 2m
                    for l = 0 to 2m
                        C[i][j] = C[i][j] + A[i-m+k][j-m+l]B[k][l]
                    endfor
                endfor
            endfor
        endfor
        return C
endConvolve2D
```

39

In terms of running time, we notice that there are 4 nested loops, that the inner loop body implies a constant amount of work, thus the running time is the product of number of iterations of the 4 loops. This clearly shows that convolution becomes expensive as the kernel size increases.

Blurring kernel

- Symmetrical
- Normalized
  - Sum of kernel entries, a.k.a. weights, is 1
  - To avoid adding or removing energy from image
- Weights fall off away from center
  - More rapid fall off, less blurring
  - All weights equal (no fall-off), maximum blurring

	0	1	2
0	1/16	2/16	1/16
1	2/16	4/16	2/16
2	1/16	2/16	1/16

3x3 blurring kernel

40

The kernel can be specialized according to the desired operation.

For a kernel of a given size, the actual values in the kernel determine the amount of blurring. A kernel with a 1 in the center and 0's anywhere else does not blur at all (the output 2-D array is the same as the input 2-D array). A kernel with all weights equal blurs the most.

It is important that the weights add up to 1, or else the overall average of the 2-D array changes. For example, in the case of 2-D images, kernel weights with a sum greater than 1 would brighten the image.

Edge extraction kernel

- Symmetrical
- Negative and positive weights
- Output is 0 over image regions with constant color
  - Sum of the weights is 0
- Picks up horizontal and vertical edges

	0	1	2
0	0	-1	0
1	-1	4	-1
2	0	-1	0

3x3 edge extraction kernel

41

An edge extraction kernel has weights designed to produce a 0 where the image does not change (i.e. where color is constant) and a large value where the image changes. In this example if the input image is constant, i.e. all pixels in the neighborhood have the same value V, the output value will be V times the sum of the weights in the kernel, which is 0.

iClicker question

In 3-D convolution, the initial array and the kernel are 3-D. If the initial array is a cube of side n and the kernel is a cube of side k, what is the running time of 3-D convolution?

A. $n*k$
B. $n*k^3$
C. n^3*k
D. n^2*k^2
E. n^3*k^3

42