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
```

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

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*

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

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

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*

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*

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

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)

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.

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

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

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
```

- Running time as number of operations
  - *n* remains a parameter
    - 1 assignment +
    - n-1 comparisons +
    - up to n-1 assignments
  - Preferred because independent of actual parameters and computer
  - Laborious to estimate
  - Many separate tallies because of many different operations
- Approximate running time
  - Constants do not matter
  - All operations are counted as 1
  - Linear running time, i.e. running time proportional to n, or running time is n
- We will use “approximate” running time, or simply “running time”

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*

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

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	

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

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
```

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

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

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*

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
```

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		

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.

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

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

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

```
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 &lt; 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
```

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		

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		

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		

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		

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: $i + n-i_{ins} = n$*

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

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

Kernel 3x3

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]

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
```

Partial trace

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
```

	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

	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

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

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

Kernel 3x3

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]

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

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
```

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

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

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