ALGORITHMS. BASICS OF ALGORITHM DEVELOPMENT.

Author:Snassapin Temirlan
T-201.

BASIC ALGORITHMS

Text on Image technique block diagram:

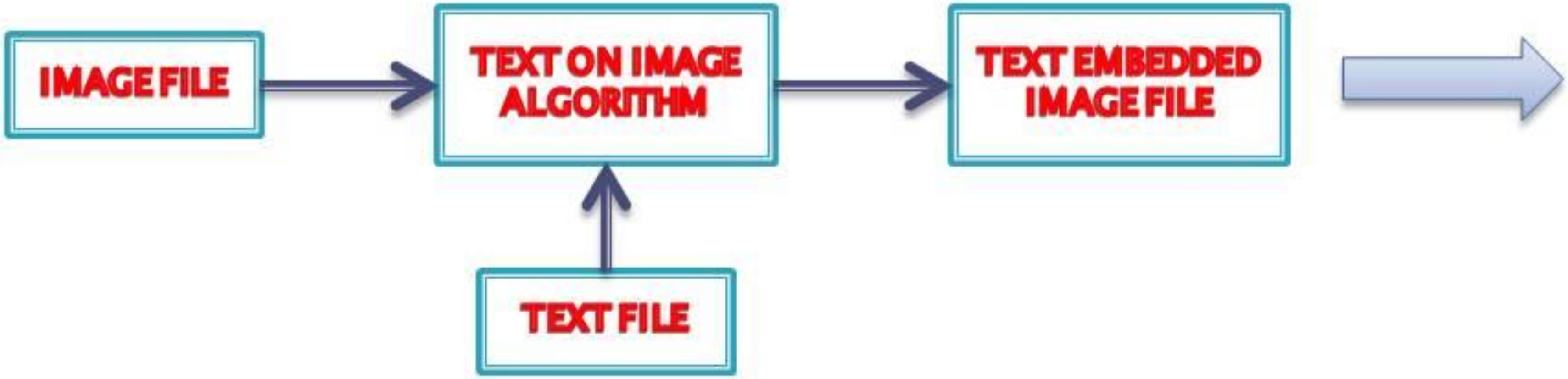
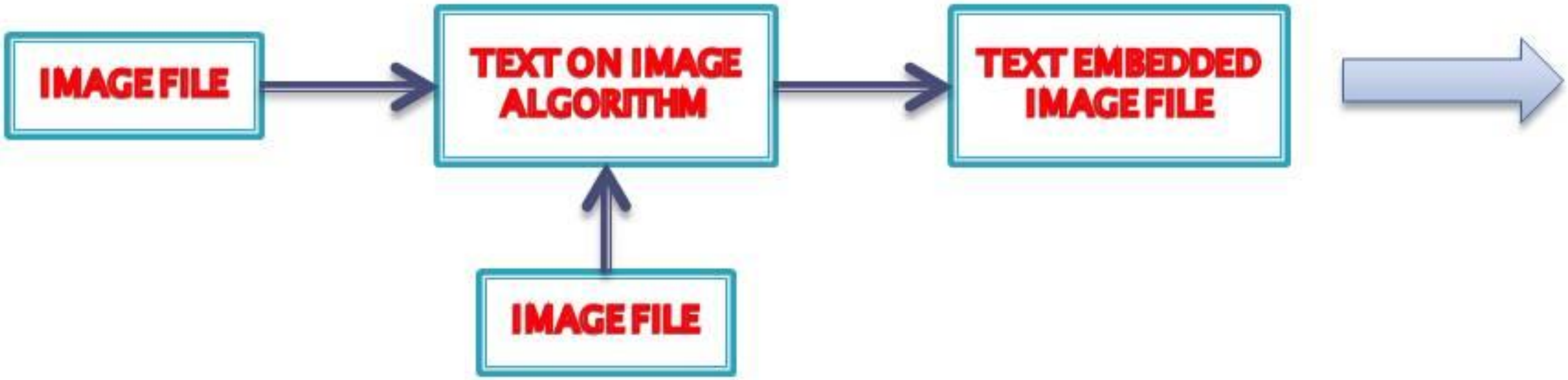


Image on Image technique block diagram:



What is an Algorithm?

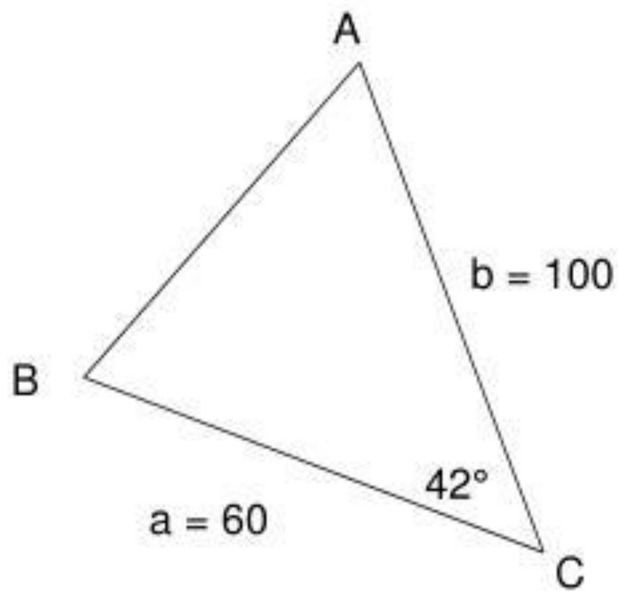
- An algorithm is a high-level set of clear, step-by-step actions taken in order to solve a problem, frequently expressed in English or pseudo code.
- Pseudo code – a way to express program-specific explanations while still in English and at a very high level
- Examples of Algorithms:
  - Computing the remaining angles and side in an SAS Triangle
  - Computing an integral using rectangle approximation method (RAM)

Why are algorithms important?

- Algorithms provide a means of expressing the problem to be solved, the solution to that problem, and a general step-by-step way to reach the solution
- Algorithms are expressed in pseudo-code, and can then be easily written in a programming language and verified for correctness.

Example: Triangulation with SAS

- Let's assume we have a triangle and we wish to compute the missing values:



- **Start with the mathematical computation if we were to do it manually:**
  - $c = A^2 + B^2 - 2 \text{ ab} \cos C$
  - $\sin A = (a \sin C) / c$

    $A = \text{asin}(a (\sin C) / c)$
  - $B = 180 - A - C$ (iff A and C are in degrees)
- We're also assuming that angles are in degrees

SAS: From Math to Pseudocode

- Now that we have all the math done, we can develop the algorithm's pseudo code:
  - Get the sides and their enclosing angle from the user (in degrees)
  - Run the computation from the previous slide
  - Convert angle A to degrees prior to computing angle B
  - Display results to the user

Introduction to Algorithm Development

- We'll get to Matrix Multiplication in a minute.
- Developing this algorithm will help you practice seeing how to take a problem, find a solution, develop an algorithm, flush out the algorithm, and then finally implementing it.
- Keep in mind that "hiding the work" is important – this is crucial to modular design

Matrix Multiplication: Initial Design

- Break down the math:
  - [A] x [B] = [C] ➔ for each element in [C], dot product the rows of [A] with the columns of [B] to get the first value in [C].
  - [A] is an **m** x **n** matrix and [B] is an **n** x **p** matrix
  - [C] is an **m** x **p** matrix

For each row **i** in [A]

  For each column **j** in [B]

    C[**i**, **j**] = **i** · **j**

Matrix Multiplication: Refined Design

- Now we can pull it all together:
- Assumptions:
  - Every element of C is initialized to 0.

For each row **i** in [A]

  For each column **j** in [B]

    For each column **k** in [A]

      C[**i**, **j**] += A[**i**, **k**] x B[**k**, **j**]

Matrix Multiplication: Accessing a Dynamically Allocated 2D Array

- Since C does not implement accessors [i, j] for 2D arrays allocated dynamically, we must implement it.
- Below is a 3 x 4 matrix with 2D and 1D coordinates overlayed.

0 0,0	1 0,1	2 0,2	3 0,3
4 1,0	5 1,1	6 1,2	7 1,3
8 2,0	9 2,1	10 2,2	11 2,3

- Examples:
  - (0, 0) maps onto 0
  - (1, 0) maps onto 4
  - (2, 1) maps onto 9
- Calculation:
  - 0 * 4 + 0 = 0
  - 1 * 4 + 0 = 4
  - 2 * 4 + 1 = 11
- From these values, we can derive that:

C[**i**, **j**] = C[**i** * **numCols** + **j**]

Matrix Multiplication: Accessing a Dynamically Allocated 2D Array

- Code for the “at” function:

```
int at(int i, int j, int numCols)
{
  return i * numCols + j;
}
```

Matrix Multiplication: Wrapping Up Pseudo Code

- This pseudo code can now be written into C code that takes:
  - 2 Pointers to an array of doubles: [A], [B]
  - Numbers of rows & columns of each (**m, n, p**)
- And allocates memory with dynamic memory allocation to return [C], an **m** x **p** matrix that is the result of [A] x [B]
- Now, any time we need to multiply any matrices, we can use and reuse this module.

Matrix Multiplication: The Code

```
double* mult(double *pA, int numRowsA, int numColsA,
             double *pB, int numRowsB, int numColsB)
{
  // allocate memory, set pC to 0
  double *pC = malloc(numRowsA * numColsB * sizeof(double));
  memset(pC, 0, numRowsA * numColsB * sizeof(double));

  for (i = 0; i < numRowsA; i++)
     for (j = 0; j < numColsB; j++)
        for (k = 0; k < numColsA; i++)
        {
           pC[at(i, j, numColsB)] =
               A[at(i, k, numColsA)] * B[at(k, j, numColsB)];
        }
  return pC;
}
```

Flushing Out the Code

- There are two more steps to algorithm development when you use functions:
  - Error handling – what if your inputs are bad?
    - Pointers can be NULL
    - What do you get if you multiply a 2 x 3 matrix by a 5 x 7? You can't!
  - This brings us to defining **requirements** for each function. If those requirements aren't met, we return an error value, in this case the NULL pointer.

Matrix Multiplication: Requirements

- Neither matrix can be NULL
- If [A]'s dimensions are **m x n** and [B]'s dimensions are NOT **n x p**, bail out because we cannot compute [A] x [B]
- If malloc() fails to allocate memory, bail out

Matrix Multiplication: Final Code

```
double* mult(double *pA, int numRowsA, int numColsA,
             double *pB, int numRowsB, int numColsB)
{
   // (m by n) x (p by q) -> can't multiply -> bail out
   if (numColsA != numRowsB) return NULL;

   double *pC = malloc(numRowsA * numColsB * sizeof(double));

   // no memory, bad matrices -> bail out
   if (pA == NULL || pB == NULL || pC == NULL) return NULL;
   memset(pC, 0, numRowsA * numColsB * sizeof(double));

   for (i = 0; i < numRowsA; i++)
      for (j = 0; j < numColsB; j++)
           for (k = 0; k < numColsA; i++)
           {
              pC[at(i, j, numColsB)] =
                   A[at(i, k, numColsA)] * B[at(k, j, numColsB)];
           }
   return pC;
}
```