**Title :** Implement a calculator (64 bit Binary Multiplication) application using concurrent lisp.

**Objective :**

Apply concurrent lisp to construct calculator.

**Theory :**

**LISP overview**

John McCarthy invented LISP in 1958, shortly after the development of FORTRAN. It was first implement by Steve Russell on an IBM 704 computer.

It is particularly suitable for Artificial Intelligence programs, as it processes symbolic information effectively.

Common Lisp originated, during the 1980s and 1990s, in an attempt to unify the work of several implementation groups, which were successors to Maclisp like ZetaLisp and NIL (New Implementation of Lisp) etc.

It serves as a common language, which can be easily extended for specific implementation. Programs written in Common LISP do not depend on machine-specific characteristics, such as word length etc.

**Features of Common LISP**

* It is machine-independent
* It uses iterative design methodology, and easy extensibility.
* It allows updating the programs dynamically.
* It provides high level debugging.
* It provides advanced object-oriented programming.
* It provides convenient macro system.
* It provides wide-ranging data types like, objects, structures, lists, vectors, adjustable arrays, hash-tables, and symbols.
* It is expression-based.
* It provides an object-oriented condition system.
* It provides complete I/O library.
* It provides extensive control structures.

**Applications Built in LISP**

Large successful applications built in Lisp.

* Emacs
* G2
* AutoCad
* Igor Engraver
* Yahoo Store

**LISP functions**

This section describes a number of simple operations on lists, i.e., chains of cons cells.

* **cl-caddr** *x*

This function is equivalent to (car (cdr (cdr*x*))). Likewise, this package defines all 24 c*xxx*r functions where *xxx* is up to four ‘a’s and/or ‘d’s. All of these functions are setf-able, and calls to them are expanded inline by the byte-compiler for maximum efficiency.

* **cl-first** *x*

This function is a synonym for (car *x*). Likewise, the functions cl-second, cl-third, ..., through cl-tenth return the given element of the list *x*.

* **cl-rest** *x*

This function is a synonym for (cdr*x*).

* **cl-endp** *x*

Common Lisp defines this function to act like null, but signaling an error if x is neither a nil nor a cons cell. This package simply defines cl-endp as a synonym for null.

* **cl-list-length** *x*

This function returns the length of list *x*, exactly like (length *x*), except that if *x* is a circular list (where the cdr-chain forms a loop rather than terminating with nil), this function returns nil. (The regular length function would get stuck if given a circular list. See also the safe-length function.)

* **cl-list\****arg&rest others*

This function constructs a list of its arguments. The final argument becomes the cdr of the last cell constructed. Thus, (cl-list\* *abc*) is equivalent to (cons *a* (cons *bc*)), and (cl-list\* *ab* nil) is equivalent to (list *ab*).

* **cl-ldiff** *list sublist*

If *sublist* is a sublist of *list*, i.e., is eq to one of the cons cells of *list*, then this function returns a copy of the part of *list* up to but not including *sublist*. For example, (cl-ldiff x (cddr x)) returns the first two elements of the list x. The result is a copy; the original *list* is not modified. If *sublist* is not a sublist of *list*, a copy of the entire *list* is returned.

* **cl-copy-list** *list*

This function returns a copy of the list *list*. It copies dotted lists like (1 2 . 3) correctly.

* **cl-tree-equal** *x y* &key :test :test-not :key

This function compares two trees of cons cells. If *x* and *y* are both cons cells, their cars and cdrs are compared recursively. If neither *x* nor *y* is a cons cell, they are compared by eql, or according to the specified test. The :key function, if specified, is applied to the elements of both trees.

**LISP operators**

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. LISP allows numerous operations on data, supported by various functions, macros and other constructs.

The operations allowed on data could be categorized as:

* Arithmetic Operations
* Comparison Operations
* Logical Operations
* Bitwise Operations

**Arithmetic Operations**

The following table shows all the arithmetic operators supported by LISP.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Adds two operands | (+AB) will give 30 |
| - | Subtracts second operand from the first | (- A B) will give -10 |
| \* | Multiplies both operands | (\* A B) will give 200 |
| / | Divides numerator by de-numerator | (/ B A) will give 2 |
| mod,rem | Modulus Operator and remainder of after an integer division | (mod B A )will give 0 |
| Incf | Increments operator increases integer value by the second argument specified | (incf A 3) will give 13 |
| Decf | Decrements operator decreases integer value by the second argument specified | (decf A 4) will give 9 |

**Comparison Operations**

Following table shows all the relational operators supported by LISP that compares between numbers. However unlike relational operators in other languages, LISP comparison operators may take more than two operands and they work on numbers only.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Checks if the values of the operands are all equal or not, if yes then condition becomes true. | (= A B) is not true. |
| /= | Checks if the values of the operands are all different or not, if values are not equal then condition becomes true. | (/= A B) is true. |
| > | Checks if the values of the operands are monotonically decreasing. | (> A B) is not true. |
| < | Checks if the values of the operands are monotonically increasing. | (< A B) is true. |
| >= | Checks if the value of any left operand is greater than or equal to the value of next right operand, if yes then condition becomes true. | (>= A B) is not true. |
| <= | Checks if the value of any left operand is less than or equal to the value of its right operand, if yes then condition becomes true. | (<= A B) is true. |
| Max | It compares two or more arguments and returns the maximum value. | (max A B) returns 20 |
| Min | It compares two or more arguments and returns the minimum value. | (min A B) returns 20 |

**Logical Operations on Boolean Values**

Common LISP provides three logical operators: **and, or,** and **not** that operates on Boolean values.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| And | It takes any number of arguments. The arguments are evaluated left to right. If all arguments evaluate to non-nil, then the value of the last argument is returned. Otherwise nil is returned. | (and A B) will return NIL. |
| Or | It takes any number of arguments. The arguments are evaluated left to right until one evaluates to non-nil, in such case the argument value is returned, otherwise it returns **nil**. | (or A B) will return 5. |
| Not | It takes one argument and returns **t** if the argument evaluates to **nil.** | (not A) will return T. |

**Mathematical Model:**

**Conclusion :**

Hence in this way we have implemented a calculator (64 bit Binary Multiplication) application using concurrent lisp.

**Title:** Vedic Mathematics method to find square of 2-digit number is used in a distributed programming. Use shared memory and distributed (multi-CPU) programming to complete the task.

**Objective :**

To study of Vedic mathematics to find square of 2-digit number is used in a distributed programming.

**Theory:**

**What is vedic mathematics?**

Vedic Mathematics is the name given to the ancient system of Indian Mathematics which was rediscovered from the Vedas between 1911 and 1918 by Sri BharatiKrsnaTirthaji (1884-1960). According to his research all of mathematics is based on [**sixteen Sutras**](http://www.vedicmaths.org/resources/sutras), or word-formulae. For example, 'Vertically and Crosswise` is one of these Sutras. These formulae describe the way the mind naturally works and are therefore a great help in directing the student to the appropriate method of solution.

Perhaps the most striking feature of the Vedic system is its coherence. Instead of a hotch-potch of unrelated techniques the whole system is beautifully interrelated and unified: the general multiplication method, for example, is easily reversed to allow one-line divisions and the simple squaring method can be reversed to give one-line square roots. And these are all easily understood. This unifying quality is very satisfying, it makes mathematics easy and enjoyable and encourages innovation.

In the Vedic system 'difficult' problems or huge sums can often be solved immediately by the Vedic method. These striking and beautiful methods are just a part of a complete system of athematics which is far more systematic than the modern 'system'. Vedic Mathematics manifests the coherent and unified structure of mathematics and the methods are complementary, direct and easy.

The simplicity of Vedic Mathematics means that calculations can be carried out mentally (though the methods can also be written down). There are many advantages in using a flexible, mental system. Pupils can invent their own methods, they are not limited to the one 'correct' method. This leads to more creative, interested and intelligent pupils.

Interest in the Vedic system is growing in education where mathematics teachers are looking for something better and finding the Vedic system is the answer. Research is being carried out in many areas including the effects of learning Vedic Maths on children; developing new, powerful but easy applications of the Vedic Sutras in geometry, calculus, computing etc. But the real beauty and effectiveness of Vedic Mathematics cannot be fully appreciated without actually practising the system. One can then see that it is perhaps the most refined and efficient mathematical system possible.

[**Sixteen ( 16 ) Sutras of Vedic Mathematics**](http://easyvedicmaths.blogspot.in/2009/04/sixteen-16-sutras-of-vedic-mathematics.html)

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Sutras** | **Meaning** |
| 1. | EkadhikinaPurvena | By one more than the previous one |
| 2. | NikhilamNavatashcaramamDashatah | [All from 9 and the last from 10](http://easyvedicmaths.blogspot.com/2010/07/this-is-explanation-to-one-of-16-sutras.html) |
| 3. | Urdhva-Tiryagbyham | Vertically and crosswise |
| 4. | ParaavartyaYojayet | Transpose and adjust |
| 5. | ShunyamSaamyasamuccaye | When the sum is the same that sum is zero. |
| 6. | (Anurupye) Shunyamanyat | If one is in ratio, the other is zero |
| 7. | Sankalana-vyavakalanabhyam | By addition and by subtraction |
| 8. | Puranapuranabyham | By the completion or non-completion |
| 9. | Chalana-Kalanabyham | Differences and Similarities |
| 10. | Yaavadunam | Whatever the extent of its deficiency |
| 11. | Vyashtisamanstih | Part and Whole |
| 12. | ShesanyankenaCharamena | The remainders by the last digit |
| 13. | Sopaantyadvayamantyam | The ultimate and twice the penultimate |
| 14. | EkanyunenaPurvena | By one less than the previous one |
| 15. | Gunitasamuchyah | The product of the sum is equal to the sum of the product |
| 16. | Gunakasamuchyah | The factors of the sum is equal to the sum of the factors |

**1. EkadhikenaPurvena**

The Sutra (formula) EkadhikenaPūrvena means: “By one more than the previous one”.

* Squares of numbers ending in 5 :

Now we relate the sutra to the ‘squaring of numbers ending in 5’. Consider the example 252

Here the number is 25. We have to find out the square of the number. For the number 25, the last digit is 5 and the 'previous' digit is 2. Hence, 'one more than the previous one', that is, 2+1=3. The Sutra, in this context, gives the procedure to multiply the previous digit 2 by one more than itself, that is, by 3. It becomes the L.H.S (left hand side) of the result, that is, 2 X 3 = 6. The R.H.S (right hand side) of the result is 52, that is 25. Thus 252= 2 X 3 / 25 = 625. In the same way,

352= 3 X (3+1) /25 = 3 X 4/ 25 = 1225;

652= 6 X 7 / 25 = 4225;

1052= 10 X 11/25 = 11025;

1352= 13 X 14/25 = 18225;

**Share Memory:**

In computer programming, shared memory is a method by which program [process](http://whatis.techtarget.com/definition/process)es can exchange data more quickly than by reading and writing using the regular operating system services. For example, a [client](http://searchenterprisedesktop.techtarget.com/definition/client) process may have data to pass to a [server](http://whatis.techtarget.com/definition/server) process that the server process is to modify and return to the client. Ordinarily, this would require the client writing to an output file (using the [buffer](http://searchcio-midmarket.techtarget.com/definition/buffer)s of the operating system) and the server then reading that file as input from the buffers to its own work space. Using a designated area of shared memory, the data can be made directly accessible to both processes without having to use the system services. To put the data in shared memory, the client gets access to shared memory after checking a [semaphore](http://searchenterpriselinux.techtarget.com/definition/semaphore) value, writes the data, and then resets the semaphore to signal to the server (which periodically checks shared memory for possible input) that data is waiting. In turn, the server process writes data back to the shared memory area, using the semaphore to indicate that data is ready to be read.

**Distributed Memory:**

Distributed memory refers to a [multiple-processor computer system](http://en.wikipedia.org/wiki/Multiprocessing) in which each [processor](http://en.wikipedia.org/wiki/Central_processing_unit) has its own private [memory](http://en.wikipedia.org/wiki/Computer_memory). Computational tasks can only operate on local data, and if remote data is required, the computational task must communicate with one or more remote processors. In contrast, a [shared memory](http://en.wikipedia.org/wiki/Shared_memory) multi processor offers a single memory space used by all processors. Processors do not have to be aware where data resides, except that there may be performance penalties, and that race conditions are to be avoided.

**Mathematical Model:**

**Conclusion :**

Hence we have implemented Vedic mathematics to find square of 2-digit number is used in a distributed programming. Use shared memory and distributed (multi-CPU) programming to complete the task.

**Title :** Implement a Parallel ODD-Even Sort algorithm using GPU or ARM equivalent.

**Objective :**

To study Parallel ODD-Even Sort algorithm using GPU or ARM equivalent.

**Theory :**

Parallelism on chip level is the hub for advancements in micro processor architectures for high performance computing. The core-processors, in personal computers, were not sufficient for high data- computation intensive tasks.

As a result modular and specialized hardware in the form of sound cards or graphic accelerators are increasingly present in most personal computers.

Graphics cards or graphics processing units (GPU), introduced primarily for high-end gaming requiring high resolution.

The GPU itself is a multi-core processor having support for thousands of threads running concurrently. GPU's are result of dozens of streaming processors with hundreds of core aligned in a particular way forming a single hardware unit.

Performance evaluation in GFLOPS (Giga Floating Point Operations per Second) shows that GPU's outperforms their CPU counterparts. For example: a high-end Core I7 processor (3.46 GHz) delivers up to a peak of 55.36 GFLOPs1.

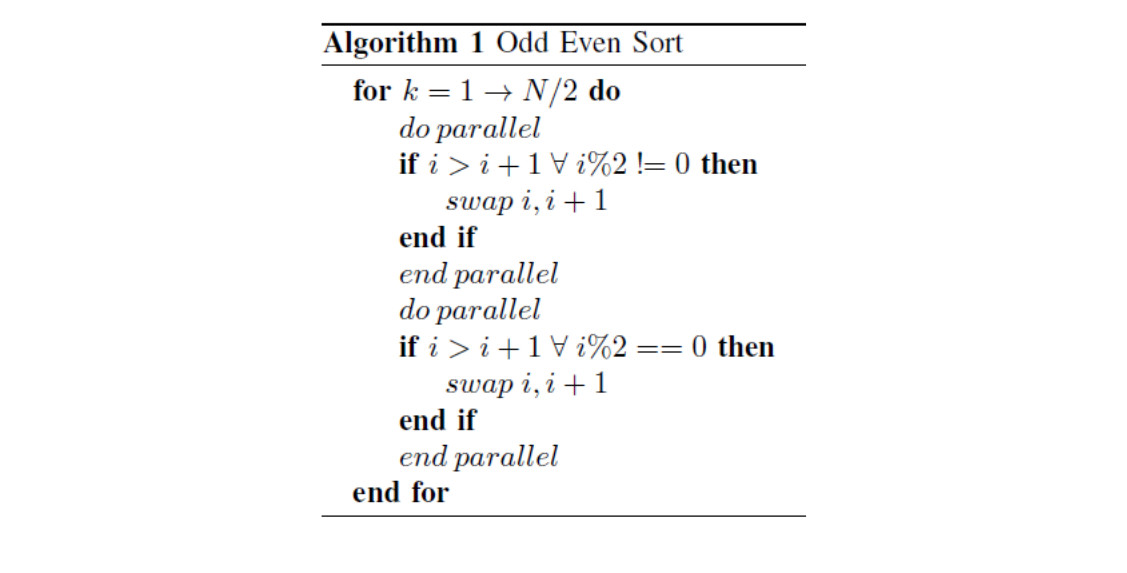
**Parallel sorting algorithms**

Sorting on GPU require transferring data from main memory to on-board GPU global memory. Although on-device bandwidth is in the range of 144Gb/s, thus only those sorting techniques are efficient which require minimum amount of synchronization because the PCI bandwidth is to the range of 2.5Gb/s. i.e., synchronization and memory transfers between CPU and GPU will affect system performance adversely.

Compared to serial sorting algorithms, parallel algorithms are designed requiring high data independence between various elements for achieving better performance. Those techniques which involve large data dependency are categorized as sequential sorting algorithms.

**Odd-Even Sort**

The odd-even sort is a parallel sorting algorithm and is based on bubble-sort technique. Adjacent pairs of items in an array are exchanged if they are found to be out of order. What makes the technique distinct from bubble-sort is the technique of working on disjointed pairs, i.e., by using alternating pairs of odd-even and even-odd elements of the array. The technique works in multiple passes on a queue Q of size N. In each pass, elements at odd-numbered positions perform a comparison check based on bubble-sort, after which elements at even- numbered positions do the same. The maximum number of iterations or passes for odd-even sort s N2. Total running time for this technique is Ο(log 2 N ) . The algorithm works as:-

****

**Mathematical Model:**

**Conclusion :**

Hence we have implemented a Parallel ODD-Even Sort algorithm using GPU or ARM equivalent.

**Title :** Implement *nxn* matrix parallel multiplication using CUDA/OpenCL GPU, use shared memory.

**Objective :**

To study *nxn* matrix parallel multiplication using CUDA/OpenCL GPU, use shared memory.

**Theory :**

CUDA

* It is an extension to the C language that allows GPU code to be written in regular C. The code is either written for the host processor (the CPU) or to the device processor (the GPU).
* The host processor spawns multithread tasks onto the GPU device. The GPU has its own internal scheduler that will then allocate the kernels to avialble GPU hardware.
* A big problem, what fraction of the code can be run in parallel.
* The maximum speedup possible is limited by the quantity of serial code. If you have an infinite amount of processing power and could do the parallel tasks in zero time, you would still be left with the time from the serial code part. NVIDIA is committed to providing support to CUDA.

**Cuda Installation Windows.**

1. System Requirements

Hardware Requirments: Cuda enabled Graphics Card

(Check wheater your System Comes with graphics card)

OS: Windows 7/8

Visual Studio 12 Ultimate Edition

Cuda 6.5 toolkit

2. Install Visual Studio 12 Ultimate edition

(Select Option Visual C++)

3. Install Cuda 6.5 toolkit

4. Launch Visual Studio

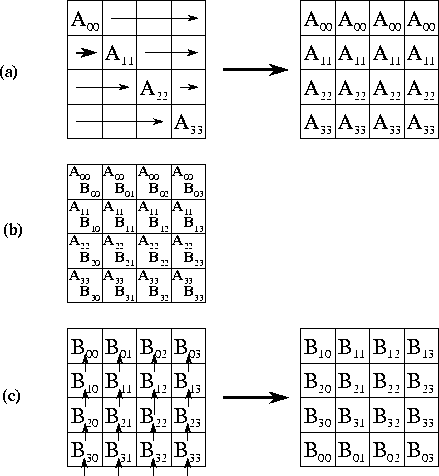
5. \*(first time only) Select Visual C++

6. File -> New Project -> NVIDIA -> Cuda 6.5 (\*Ver) -> Enter Name for project -> OK -> Edit .cu file -> run local debuger.

**Matrix Parellel Multiplication:**

In [mathematics](http://en.wikipedia.org/wiki/Mathematics), **matrix multiplication** is a [binary operation](http://en.wikipedia.org/wiki/Binary_operation) that takes a pair of [matrices](http://en.wikipedia.org/wiki/Matrix_(mathematics)), and produces another matrix. [Numbers](http://en.wikipedia.org/wiki/Number) such as the [real](http://en.wikipedia.org/wiki/Real_number) or [complex numbers](http://en.wikipedia.org/wiki/Complex_number) can be [multiplied](http://en.wikipedia.org/wiki/Multiplication) according to [elementary arithmetic](http://en.wikipedia.org/wiki/Elementary_arithmetic). On the other hand, matrices are *arrays of numbers*, so there is no unique way to define "the" multiplication of matrices. As such, in general the term "matrix multiplication" refers to a number of different ways to multiply matrices. The key features of any matrix multiplication include: the number of rows and columns the original matrices have and specifying how the entries of the matrices generate the new matrix.

Because of the nature of matrix operations and the layout of matrices in memory, it is typically possible to gain substantial performance gains through use of [parallelization](http://en.wikipedia.org/wiki/Parallelization) and [vectorization](http://en.wikipedia.org/wiki/Automatic_vectorization). Several algorithms are possible, among which [divide and conquer](http://en.wikipedia.org/wiki/Divide_and_conquer_algorithms) algorithms based on the [block matrix](http://en.wikipedia.org/wiki/Block_matrix) decomposition



P= n ***Tpar = O(n2)***

Each instance of inner loop is independent and can be done by a separate processor.

**Cost optimal** since O(n3) = n \* O(n2)

P = n2 ***Tpar = O(n)***

One element of C (cij) is assigned to each processor.

**Cost optimal** since O(n3) = n2 x O(n)

P = n3 ***Tpar = O(*log *n)***

n processors compute one element of C (cij) in parallel (O(log n))

**Not cost optimal** since O(n3) < n3 \* O(log n)

***O(*log *n)*** lower bound for parallel matrix multiplication.

We implement a fast matrix multiplication algorithm with asymptotic complexity O(N2.775) for square N×N matrices. In terms of asymptotic complexity, this is the fastest matrix multiplication algorithm implementation to date. However, our performance results show that this algorithm is not practical for the problem sizes that we consider. Overall, we find that Strassen’s algorithm

**Mathematical Model:**

**Conclusion :**

Hence we have implemented *nxn* matrix parallel multiplication using CUDA/OpenCL GPU, use shared memory.

**Title :** Implement a Multi-threading application for echo server using socket programming in JAVA.

**Objective :**

To study a Multi-threading application for echo server using socket programming in JAVA.

**Theory :**

**Socket Programming :**

Socket programming is useful for building client-server applications.

**The server :**

Creates a socket with some port number (>1023):   
*ServerSocketechoServer = new ServerSocket(6789);*

Waits for client connection :   
*Socket clientSocket = echoServer.accept();*

Gets input/output streams :   
*BufferedReader is = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));*  
*PrintStreamos = new PrintStream(clientSocket.getOutputStream());*

Exchanges information with the client :   
*String line = is.readLine();*  
*os.println( "Echo: " + line );*

Clean up :   
*is.close();   os.close();*  
*clientSocket.close();   echoServer.close();*

**The client :**

Creates a socket with the same port number :   
*Socket clientSocket = new Socket( "localhost", 6789 );*

Gets input/output streams.

Exchanges information with the server.

Clean up.

**ECHO Server :**

**What is it and what is it for ?**

An "echo server" is a server that does nothing more than sending back whatever is sent to it. Hence the name : echo What can you use it for ? Whatever you feel like. Practical applications could be network and connectivity testing and troubleshooting. Assume you've build a rather complex network with VLANs and subnets, and really strick firewalls between those subnets, and you're beginning to wonder if a client on one segment of the network will still be able to connect to your web server, database server, ... on some other segment. A ping or a traceroute will establish if the server (IP address) can be reached but does not tell you if an application will be able to connect to the desired port on the server and whether a reply from the server will be able to reach the client again.

This "echo server" can be set up to listen on any desired (tcp) port to simulate whatever application you want to run (eg web server = port 80, Microsoft SQL Server = port 1433, etc). From the client machine, you can then telnet to this port. When a telnet connection has been established, everything you type will be echoed back to your screen, indicating that the telnet client and the echo server can talk to each other : you've established connectivity at the application level.

In a similar way, you can use this echo server to troubleshoot networks, test a firewall (eg "if I have a server listening on port 123, wil my firewall allow connections to it ?) and so on.

**Echo server :**

1. The client reads a line from its standard input and writes that line to the server.

2. The server reads a line from its network input and echoes the line back to the client over the network.

3. The client reads the echoed line from the network and prints it on its standard output.

**Multithreaded Server Advantages :**

The advantages of a multithreaded server compared to a single threaded server are summed up below:

Less time is spent outside the accept() call.

Long running client requests do not block the whole server.

In a single threaded server long running requests may make the server unresponsive for a long period. This is not true for a multithreaded server, unless the long-running request takes up all CPU time and/or network bandwidth.

**Mathematical Model:**

**Conclusion :**

Hence we have implemented a Multi-threading application for echo server using socket programming in JAVA.

**Title :** Implement a Parallel Quick Sort algorithm using NVIDIA GPU or equivalent ARM board.

**Objective :**

To study Parallel Quick Sort algorithm using NVIDIA GPU or equivalent ARM board.

**Theory :**

**Features**

* Similar to mergesort - divide-and-conquer recursive algorithm
* One of the fastest sorting algorithms
* Average running time O(NlogN)
* Worst-case running time O(N2)

**Basic idea**

1. Pick one element in the array, which will be the *pivot*.
2. Make one pass through the array, called a *partition* step, re-arranging the entries so that:
   * + the pivot is in its proper place.
     + entries smaller than the pivot are to the left of the pivot.
     + entries larger than the pivot are to its right.
3. Recursively apply quicksort to the part of the array that is to the left of the pivot,   
   and to the right part of the array.

Here we don't have the merge step, at the end all the elements are in the proper order.

**Algorithm**

**STEP 1**. Choosing the pivot

Choosing the pivot is an essential step.   
Depending on the pivot the algorithm may run very fast, or in quadric time.:

* 1. Some fixed element: e.g. the first, the last, the one in the middle

This is a bad choice - the pivot may turn to be the smallest or the largest element,   
then one of the partitions will be empty.

* 1. Randomly chosen (by random generator ) - still a bad choice.
  2. The median of the array (if the array has N numbers, the median is the [N/2] largest number. This is difficult to compute - increases the complexity.
  3. The median-of-three choice: take the first, the last and the middle element.   
     Choose the median of these three elements.

Example:

8, 3, 25, 6, 10, 17, 1, 2, 18, 5

The first element is 8, the middle - 10, the last - 5.  
The median of [8, 10, 5] is 8

**STEP 2**. Partitioning

Partitioning is illustrated on the above example.

1. The first action is to get the pivot out of the way - swap it with the last element

5, 3, 25, 6, 10, 17, 1, 2, 18, 8

2. We want larger elements to go to the right and smaller elements to go to the left.

Two "fingers" are used to scan the elements from left to right and from right to left:

[5, 3, 25, 6, 10, 17, 1, 2, 18, 8]

^ ^

i j

* While i is to the left of j, we move i right, skipping all the elements   
  less than the pivot. If an element is found greater then the pivot, i stops
* While j is to the right of i, we move j left, skipping all the elements   
  greater than the pivot. If an element is found less then the pivot, j stops
* When both i and j have stopped, the elements are swapped.
* When i and j have crossed, no swap is performed, scanning stops,   
  and the element pointed to by i is swapped with the pivot .

In the example the first swapping will be between 25 and 2, the second between 10 and 1.

3. Restore the pivot.

After restoring the pivot we obtain the following partitioning into three groups:

[5, 3, 2, 6, 1] [ 8 ] [10, 25, 18, 17]

**STEP 3**. Recursively quicksort the left and the right parts

**Mathematical Model:**

**Conclusion :** Hence we have implemented a Parallel Quick Sort algorithm using NVIDIA GPU