Session 5

1. **2 ways to make the architecture of the processor ? And which one is better and why? And the most popular people who use it**

- CISC chips, such as Intel and AMD, have a large number of instructions, of varying length, and some are rarely used. The CISC chip has the overhead of determining whether it has the entire instruction. CISC chips acquire cruft over the years, which slows them down. Since the instruction set is complicated, the chip is more complex and therefore slower in comparison to an equally powerful RISC chip.
- RISC chips, such as Apple Silicon and ARM, have fewer instructions, and all of them have the same length, so there is no overhead in determining whether the chip has the entire instruction. Since the instruction set is simple, the chip is simpler and therefore faster.

- The short answer is that no one architecture is better than the others. Each has its benefits and roadblocks, which make them better suited to particular scenarios. The key is to match up those advantages and roadblocks with the intended application to find the most appropriate instruction set architecture for you.

RISC	**CISC**
Emphasis on software	Emphasis on hardware
Small number of fixed length instructions	Large number of instructions
Simple, standardised instructions	Complex, variable-length instructions
Single clock cycle instructions	Instructions can take several clock cycles
Heavy use of RAM	More efficient use of RAM

- ARM, MIPS, Atmel AVR, PIC, most everything else that is intended for low-power and mobile devices, and IBM's POWER machines are RISC CPUs.
- Many OSs, including Windows and various *nix flavors, have been ported to both RISC and CISC processors

2. **how to use tabulation in recursive?**

3. **disadvantage of memory allocated**

- It requires more execution time due to execution during runtime.
- The compiler does not help with allocation and deallocation. Programmer needs to both allocate and deallocate the memory.

4. Stack vs Heap

- **A stack** is a special area of computer's memory which stores temporary variables created by a function. In stack, variables are declared, stored and initialized during runtime.
  It is a temporary storage memory. When the computing task is complete, the memory of the variable will be automatically erased. The stack section mostly contains methods, local variable, and reference variables
- **The heap** is a memory used by programming languages to store global variables. By default, all global variable are stored in heap memory space. It supports Dynamic memory allocation.
  The heap is not managed automatically for you and is not as tightly managed by the CPU. It is more like a free-floating region of memory.

  Difference between stack and heap:

  - Stack is a linear data structure whereas Heap is a hierarchical data structure.
  - Stack memory will never become fragmented whereas Heap memory can become fragmented as blocks of memory are first allocated and then freed.
  - Stack accesses local variables only while Heap allows you to access variables globally.
  - Stack variables can't be resized whereas Heap variables can be resized.
  - Stack memory is allocated in a contiguous block whereas Heap memory is allocated in any random order.
  - Stack doesn't require to de-allocate variables whereas in Heap de-allocation is needed.
  - Stack allocation and deallocation are done by compiler instructions whereas Heap allocation and deallocation is done by the programmer.

5. C++ framework that used in AI

- ANNetGPGPU - A GPU (CUDA) based Artificial Neural Network library. [LGPL]
- btsk - Game Behavior Tree Starter Kit. [zlib]
- Evolving Objects - A template-based, ANSI-C++ evolutionary computation library which helps you to write your own stochastic optimization algorithms insanely fast. [LGPL]
- frugally-deep - Header-only library for using Keras models in C++. [MIT]
- Genann - Simple neural network library in C. [zlib]
- MXNet - Lightweight, Portable, Flexible Distributed/Mobile Deep Learning with Dynamic, Mutation-aware Dataflow Dep Scheduler; for Python, R, Julia, Scala, Go, Javascript and more website
- PyTorch - Tensors and Dynamic neural networks in Python with strong GPU acceleration. website
- flashlight - Flashlight is a fast, flexible machine learning library written entirely in C++. [BSD]
- Recast/Detour - (3D) Navigation mesh generator and pathfinder, mostly for games. [zlib]

- TensorFlow - An open source software library for numerical computation using data flow graphs [Apache]
- oneDNN - An open-source cross-platform performance library for deep learning applications. [Apache] website
- CNTK - Microsoft Cognitive Toolkit (CNTK), an open source deep-learning toolkit. [Boost]
- tiny-dnn - A header only, dependency-free deep learning framework in C++11. [BSD]
- Veles - Distributed platform for rapid Deep learning application development. [Apache]
- Kaldi - Toolkit for speech recognition. [Apache]
-

6. windows equivalents to cron jobs and how to automate in c++ and python ?

The windows equivalent to a cron job is a scheduled task
Using python:

```
import schedule
import time

def job():
    print("I'm working...")

schedule.every(10).minutes.do(job)
schedule.every().hour.do(job)
schedule.every().day.at("10:30").do(job)

while 1:
    schedule.run_pending()
    time.sleep(1)
```

7. What is polling in cronjobs?

8. How to update a queue priority

To update priority in Priority Queue, get the index of the element that you want to update the priority of and assign a new key to the element.
You can change the value of the element as well. Check out the code below:

```
import heapq

hq = []

heapq.heappush(hq, (3, "Jean"))

heapq.heappush(hq, (2, "Eric"))

heapq.heappush(hq, (4, "Monica"))

heapq.heappush(hq, (1, "Joey"))

print(hq)

hq[1] = (6, 'Eric')

print(hq)

heapq.heapify(hq)

print(hq)
```

9. DFS BFS algorithm in C++

Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array. A graph can have more than one DFS traversal.
Code:

```
#include <bits/stdc++.h>
using namespace std;
 class Graph {
public:
    map<int, bool> visited;
    map<int, list<int> > adj;
    void addEdge(int v, int w);
    void DFS(int v);
};

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}
void Graph::DFS(int v)
{

    visited[v] = true;
```

```
    cout << v << " ";
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFS(*i);
}
int main()
{
    Graph g;
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);
  cout << "Following is Depth First Traversal"
          " (starting from vertex 2) \n";
     g.DFS(2);

    return 0;
}
```

Breadth-First Traversal (or Search) for a graph is similar to Breadth-First Traversal of a tree.

The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

- Visited and
- Not visited.

A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a queue data structure for traversal.

Code:

```
#include<bits/stdc++.h>
using namespace std;

class Graph
{
```

```
    int V;    // No. of vertices

    vector<list<int>> adj;
public:
    Graph(int V);  // Constructor

    void addEdge(int v, int w);
     void BFS(int s);
};

Graph::Graph(int V)
{
    this->V = V;
    adj.resize(V);
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

void Graph::BFS(int s)
{
    vector<bool> visited;
    visited.resize(V,false);
     list<int> queue;

    visited[s] = true;
    queue.push_back(s);

    while(!queue.empty())
    {
        s = queue.front();
        cout << s << " ";
        queue.pop_front();

        for (auto adjecent: adj[s])
        {
            if (!visited[adjecent])
            {
```

```
                visited[adjecent] = true;
                queue.push_back(adjecent);
            }
        }
    }
}

int main()
{
    // Create a graph given in the above diagram
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 2);
    g.addEdge(1, 2);
    g.addEdge(2, 0);
    g.addEdge(2, 3);
    g.addEdge(3, 3);

    cout << "Following is Breadth First Traversal "
         << "(starting from vertex 2) \n";
    g.BFS(2);

    return 0;
}
```
