# First Fit

Code:

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

// Function to allocate memory to
// blocks as per First fit algorithm
void firstFit(int blockSize[], int m, int processSize[], int n)
{
    // Stores block id of the
    // block allocated to a process
    int allocation[n];

    // Initially no block is assigned to any process
    memset(allocation, -1, sizeof(allocation));

    // pick each process and find suitable blocks
    // according to its size ad assign to it
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            if (blockSize[j] >= processSize[i])
            {
                // allocate block j to p[i] process
                allocation[i] = j;

                // Reduce available memory in this block.
                blockSize[j] -= processSize[i];

                break;
            }
        }
    }

    cout << "\nProcess No.\tProcess Size\tBlock no.\n";
    for (int i = 0; i < n; i++)
    {
        cout << " " << i + 1 << "\t\t"
             << processSize[i] << "\t\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";
        cout << endl;
```

```
    }
}

int main()
{
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processSize) / sizeof(processSize[0]);

    firstFit(blockSize, m, processSize, n);

    return 0;
}
```

Output:

```
 Process No.     Process Size    Block no.
  1              212             2
  2              417             5
  3              112             2
  4              426             Not Allocated
```

# Next Fit

Code:

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

// Function to allocate memory to blocks as per Next fit
// algorithm
void NextFit(int blockSize[], int m, int processSize[], int n)
{
    // Stores block id of the block allocated to a
    // process
    int allocation[n], j = 0, t = m - 1;

    // Initially no block is assigned to any process
    memset(allocation, -1, sizeof(allocation));

    // pick each process and find suitable blocks
```

```
    // according to its size ad assign to it
    for(int i = 0; i < n; i++){

        // Do not start from beginning
        while (j < m){
            if(blockSize[j] >= processSize[i]){

                // allocate block j to p[i] process
                allocation[i] = j;

                // Reduce available memory in this block.
                blockSize[j] -= processSize[i];

                // sets a new end point
                t = (j - 1) % m;
                break;
            }
            if (t == j){
                // sets a new end point
                t = (j - 1) % m;
                // breaks the loop after going through all memory block
                break;
            }


            j = (j + 1) % m;
        }
    }

    cout << "\nProcess No.\tProcess Size\tBlock no.\n";
    for (int i = 0; i < n; i++) {
        cout << " " << i + 1 << "\t\t\t\t" << processSize[i]
            << "\t\t\t\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";
        cout << endl;
    }
}

int main()
{
    int blockSize[] = { 5, 10, 20 };
    int processSize[] = { 10, 20, 5 };
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processSize) / sizeof(processSize[0]);
```

```
    NextFit(blockSize, m, processSize, n);

    return 0;
}
```

Output:

| Process No. | Process Size | Block no. |
|---|---|---|
| 1 | 10 | 2 |
| 2 | 20 | 3 |
| 3 | 5 | 1 |

# Best Fit

Code:

```
#include<iostream>
using namespace std;

// Method to allocate memory to blocks as per Best fit algorithm
void bestFit(int blockSize[], int m, int processSize[], int n)
{
    // Stores block id of the block allocated to a process
    int allocation[n];

    // Initially no block is assigned to any process
    for (int i = 0; i < n; i++)
        allocation[i] = -1;

    // pick each process and find suitable blocks
    // according to its size ad assign to it
    for (int i = 0; i < n; i++)
    {
        // Find the best fit block for current process
        int bestIdx = -1;
        for (int j = 0; j < m; j++)
        {
            if (blockSize[j] >= processSize[i])
            {
```

```
                if (bestIdx == -1)
                    bestIdx = j;
                else if (blockSize[bestIdx] > blockSize[j])
                    bestIdx = j;
            }
        }

        // If we could find a block for current process
        if (bestIdx != -1)
        {
            // allocate block j to p[i] process
            allocation[i] = bestIdx;

            // Reduce available memory in this block.
            blockSize[bestIdx] -= processSize[i];
        }
    }

    cout << "\nProcess No.\tProcess Size\tBlock no.\n";
    for (int i = 0; i < n; i++)
    {
        cout << " " << i+1 << "\t\t" << processSize[i] << "\t\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";
        cout << endl;
    }
}

int main()
{
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processSize) / sizeof(processSize[0]);

    bestFit(blockSize, m, processSize, n);

    return 0 ;
}
```

Output:

```
Process No.     Process Size    Block no.
 1              212             4
 2              417             2
 3              112             3
 4              426             5
```

# Worst Fit

Code:

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

// Function to allocate memory to blocks as per worst fit
// algorithm
void worstFit(int blockSize[], int m, int processSize[], int n)
{
    // Stores block id of the block allocated to a
    // process
    int allocation[n];

    // Initially no block is assigned to any process
    memset(allocation, -1, sizeof(allocation));

    // pick each process and find suitable blocks
    // according to its size ad assign to it
    for (int i = 0; i < n; i++)
    {
        // Find the best fit block for current process
        int wstIdx = -1;
        for (int j = 0; j < m; j++)
        {
            if (blockSize[j] >= processSize[i])
            {
                if (wstIdx == -1)
                    wstIdx = j;
                else if (blockSize[wstIdx] < blockSize[j])
                    wstIdx = j;
            }
        }

        // If we could find a block for current process
        if (wstIdx != -1)
        {
```

```
            // allocate block j to p[i] process
            allocation[i] = wstIdx;

            // Reduce available memory in this block.
            blockSize[wstIdx] -= processSize[i];
        }
    }

    cout << "\nProcess No.\tProcess Size\tBlock no.\n";
    for (int i = 0; i < n; i++)
    {
        cout << " " << i + 1 << "\t\t" << processSize[i] << "\t\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";
        cout << endl;
    }
}

int main()
{
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize) / sizeof(blockSize[0]);
    int n = sizeof(processSize) / sizeof(processSize[0]);

    worstFit(blockSize, m, processSize, n);

    return 0;
}
```

Output:

```
 Process No.     Process Size     Block no.
  1              212              5
  2              417              2
  3              112              5
  4              426              Not Allocated
```

# FIFO

Code:

```
#include <bits/stdc++.h>
```

```
using namespace std;

// Method to determine pager faults using FIFO
int getPageFaults(int pages[], int n, int frames)
{
    unordered_set<int> set;
    // The code will store the pages in FIFO technique
    queue<int> indexes;

    // Stating from the first page
    int countPageFaults = 0;

    for (int i = 0; i < n; i++)
    {
        // Checking the capacity to hold more pages
        if (set.size() < frames)
        {
            // if the page is absent, insert it into the set
            // the condition represents page fault
            if (set.find(pages[i]) == set.end())
            {
                set.insert(pages[i]);
                // increment the conter for page fault
                countPageFaults++;

                // Push the current page into the queue
                indexes.push(pages[i]);
            }
        }

        else
        {
            // Check if the page in demand is not already present in
the queue
            if (set.find(pages[i]) == set.end())
            {
                // Remove the first page from the queue
                int val = indexes.front();
                indexes.pop();

                // Pop the index page
                set.erase(val);

                // Push the current page in the queue
                set.insert(pages[i]);
                indexes.push(pages[i]);

                // Increment page faults
```

```
                countPageFaults++;
            }
        }
    }

    return countPageFaults;
}

int main()
{
    int pages[] = {4, 1, 2, 4, 5};
    int n = sizeof(pages) / sizeof(pages[0]);
    int frames = 4;

    cout << "Page Faults: " << getPageFaults(pages, n, frames);

    return 0;
}
```

# LRU

Code:

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

// Function to find page faults using indexes
int pageFaults(int pages[], int n, int capacity)
{

    unordered_set<int> s;

    unordered_map<int, int> indexes;

    // Start from initial page
    int page_faults = 0;
    for (int i = 0; i < n; i++)
    {
        // Check if the set can hold more pages
        if (s.size() < capacity)
        {

            if (s.find(pages[i]) == s.end())
            {
                s.insert(pages[i]);

                // increment page fault
```

```
                page_faults++;
            }

            indexes[pages[i]] = i;
        }

        else
        {
            // Check if current page is not already
            // present in the set
            if (s.find(pages[i]) == s.end())
            {
                // Find the least recently used pages
                // that is present in the set
                int lru = INT_MAX, val;
                for (auto it = s.begin(); it != s.end(); it++)
                {
                    if (indexes[*it] < lru)
                    {
                        lru = indexes[*it];
                        val = *it;
                    }
                }

                // Remove the indexes page
                s.erase(val);

                // insert the current page
                s.insert(pages[i]);

                // Increment page faults
                page_faults++;
            }

            // Update the current page index
            indexes[pages[i]] = i;
        }
    }

    return page_faults;
}

int main()
{
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
    int n = sizeof(pages) / sizeof(pages[0]);
    int capacity = 4;
```

```
    cout << "Number of page Faults using LRU: " << pageFaults(pages, n,
capacity);
    return 0;
}
```

# Optimal

Code:

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

bool search(int key, vector<int> &fr)
{
    for (int i = 0; i < fr.size(); i++)
        if (fr[i] == key)
            return true;
    return false;
}

int predict(int pg[], vector<int> &fr, int pn, int index)
{

    int res = -1, farthest = index;
    for (int i = 0; i < fr.size(); i++)
    {
        int j;
        for (j = index; j < pn; j++)
        {
            if (fr[i] == pg[j])
            {
                if (j > farthest)
                {
                    farthest = j;
                    res = i;
                }
                break;
            }
        }

        if (j == pn)
            return i;
    }

    return (res == -1) ? 0 : res;
}
```

```
void optimalPage(int pg[], int pn, int fn)
{

    vector<int> fr;

    int hit = 0;
    for (int i = 0; i < pn; i++)
    {

        // Page found in a frame : HIT
        if (search(pg[i], fr))
        {
            hit++;
            continue;
        }

        if (fr.size() < fn)
            fr.push_back(pg[i]);

        // Find the page to be replaced.
        else
        {
            int j = predict(pg, fr, pn, i + 1);
            fr[j] = pg[i];
        }
    }
    cout << "No. of hits = " << hit << endl;
    cout << "No. of misses = " << pn - hit << endl;
}

int main()
{
    int pg[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
    int pn = sizeof(pg) / sizeof(pg[0]);
    int fn = 4;
    optimalPage(pg, pn, fn);
    return 0;
}
```

# Buddy System

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

// Size of vector of pairs
```

```
int listSize;

// Global vector of pairs to store
// address ranges available in free list
vector<pair<int, int>> free_list[100000];

// Map used as hash map to store the starting
// address as key and size of allocated segment
// key as value
map<int, int> mp;

void initialize(int sz)
{

    // Maximum number of powers of 2 possible
    int n = ceil(log(sz) / log(2));
    listSize = n + 1;

    for (int i = 0; i <= n; i++)
        free_list[i].clear();

    // Initially whole block of specified
    // size is available
    free_list[n].push_back(make_pair(0, sz - 1));
}

void allocate(int sz)
{

    // Calculate index in free list
    // to search for block if available
    int n = ceil(log(sz) / log(2));

    // Block available
    if (free_list[n].size() > 0)
    {
        pair<int, int> temp = free_list[n][0];

        // Remove block from free list
        free_list[n].erase(free_list[n].begin());
        cout << "Memory from " << temp.first
             << " to " << temp.second << " allocated"
             << "\n";

        // map starting address with
        // size to make deallocating easy
        mp[temp.first] = temp.second -
                         temp.first + 1;
```

```
    }
    else
    {
        int i;
        for (i = n + 1; i < listSize; i++)
        {

            // Find block size greater than request
            if (free_list[i].size() != 0)
                break;
        }

        // If no such block is found
        // i.e., no memory block available
        if (i == listSize)
        {
            cout << "Sorry, failed to allocate memory \n";
        }

        // If found
        else
        {
            pair<int, int> temp;
            temp = free_list[i][0];

            // Remove first block to split it into halves
            free_list[i].erase(free_list[i].begin());
            i--;

            for (; i >= n; i--)
            {

                // Divide block into two halves
                pair<int, int> pair1, pair2;
                pair1 = make_pair(temp.first,
                                  temp.first +
                                      (temp.second -
                                       temp.first) /
                                          2);
                pair2 = make_pair(temp.first +
                                      (temp.second -
                                       temp.first + 1) /
                                          2,
                                  temp.second);

                free_list[i].push_back(pair1);

                // Push them in free list
```

```
                free_list[i].push_back(pair2);
                temp = free_list[i][0];

                // Remove first free block to
                // further split
                free_list[i].erase(free_list[i].begin());
            }
            cout << "Memory from " << temp.first
                 << " to " << temp.second
                 << " allocated"
                 << "\n";

            mp[temp.first] = temp.second -
                                 temp.first + 1;
        }
    }
}

int main()
{

    // Uncomment following code for interactive IO
    /*
    int total,c,req;
    cin>>total;
    initialize(total);
    while(true)
    {
        cin>>req;
        if(req < 0)
            break;
        allocate(req);
    }*/

    initialize(128);
    allocate(32);
    allocate(7);
    allocate(64);
    allocate(56);

    return 0;
}
```

# Address Map

# Conversion of Logical Address to physical Address using Paging

```
#include <bits/stdc++.h>
using namespace std;
int ADDRESSMAP(int C_VA, int arr[], int page_size, int n)
{
    int pte = C_VA / page_size;
    string temp = "";

    if (pte >= n)
    {
        cout << "Page Fault" << endl;
        return -1;
    }
    else
    {
        return ((arr[pte] * page_size) + (C_VA % page_size));
    }
}
int convert(string VA)
{
    int n = VA.length();
    int a = 1;
    int res = 0;
    for (int i = n - 1; i >= 0; i--)
    {
        if (VA[i] == '1')
        {
            res += a * 1;
        }
        a = a * 2;
    }
    return res;
}
int main()
{
    int ptr;
    int page_size;
    string VA;
    int C_VA;
    int arr[100];
    char M[1000][4];
    int VA_SPACE = 100;
    int READ_SPACE = 300;
```

```
    cout << "Enter the size of the page: ";
    cin >> page_size;
    cout << endl;
    int n;
    cout << "Enter the number of entries in the page table:" << endl;
    cin >> n;
    cout << "Enter the contents of the page table";
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    int menu = 1;
    while (1)
    {
        cout << "Numerical Virtual Address: 1" << endl;
        cout << "Binary Virtual Address: 2" << endl;
        cout << "Exit: 3" << endl;
        cin >> menu;
        if (menu == 1)
        {
            cout << "Enter the virtual address:";
            cin >> C_VA;
        }
        else if (menu == 2)
        {
            cout << "Enter the virtual address: ";
            cin >> VA;
            C_VA = convert(VA);
        }
        else
        {
            break;
        }
        cout << "The real address is:" << ADDRESSMAP(C_VA, arr,
page_size, n) << endl;
    }
}
```