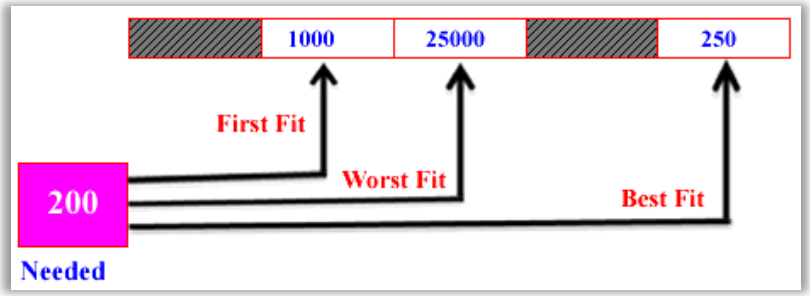
5rName:	Prerna Sunil Jadhav
Sap Id:	60004220127
Class:	S. Y. B.Tech (Computer Engineering)
Course:	Operating System Laboratory
Course Code:	DJ19CEL403
Experiment No.:	08

AIM: **MEMORY ALLOCATION TECHNIQUES (FIRST FIT, BEST FIT, WORST FIT)**

THEORY:

- Processes that have been assigned continuous blocks of memory will fill the main memory at any given time. However, when a process completes, it leaves behind an empty block known as a hole. This space could also be used for a new process.
- Hence, the main memory consists of processes and holes, and any one of these holes can be allotted to a new incoming process.
- We have three strategies to allot a hole to an incoming process:
- First-Fit
    - This is a very basic strategy in which we start from the beginning and allot the first hole, which is big enough as per the requirements of the process.
    - The first-fit strategy can also be implemented in a way where we can start our search for the first-fit hole from the place we left off last time.
- Best-Fit
    - This is a greedy strategy that aims to reduce any memory wasted because of internal fragmentation in the case of static partitioning, and hence we allot that hole to the process, which is the smallest hole that fits the requirements of the process.
    - Hence, we need to first sort the holes according to their sizes and pick the best fit for the process without wasting memory.



- Worst-Fit
    - This strategy is the opposite of the Best-Fit strategy.
    - We sort the holes according to their sizes and choose the largest hole to be allotted to the incoming process.
    - The idea behind this allocation is that as the process is allotted a large hole, it will have a lot of space left behind as internal fragmentation.
    - Hence, this will create a hole that will be large enough to accommodate a few other processes.

FIRST FIT

CODE:

```
class FirstFit
{
    static void firstFit(int blockSize[], int m, int processSize[], int n)
    {
        int allocation[] = new int[n];

        for (int i = 0; i < allocation.length; i++)
            allocation[i] = -1;

        for (int i = 0; i < n; i++){
            for (int j = 0; j < m; j++){
                if (blockSize[j] >= processSize[i]){
                    allocation[i] = j;
                    blockSize[j] -= processSize[i];
                    break;
                }
            }
        }

        System.out.println("\nProcess No.\tProcess Size\tBlock no.");

        for (int i = 0; i < n; i++) {
            System.out.print(" " + (i+1) + "\t\t" + processSize[i] + "\t\t");
            if (allocation[i] != -1)
                System.out.print(allocation[i] + 1);
            else
                System.out.print("Not Allocated");
            System.out.println();
        }
    }

    // Driver Code
    public static void main(String[] args){

        int blockSize[] = {100, 500, 200, 300, 600};
        int processSize[] = {212, 417, 112, 426};
        int m = blockSize.length;
        int n = processSize.length;

        firstFit(blockSize, m, processSize, n);
    }
}
```

OUTPUT:

```
068e57ab59d6ffa050515e89af29e69\redhat.java\jdt_ws\Code_68cc0323\bin' 'FirstFit'

Process No.     Process Size    Block no.
 1              212             2
 2              417             5
 3              112             2
 4              426             Not Allocated
PS C:\Users\Jadhav\Desktop\BTech\4th sem\OS\Prac\Code>
```

BEST FIT

CODE:

```
public class BestFit
{
    static void bestFit(int blockSize[], int m, int processSize[],int n)
    {
        int allocation[] = new int[n];
        for (int i = 0; i < allocation.length; i++)
            allocation[i] = -1;
        for (int i=0; i<n; i++)
        {
            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 (bestIdx != -1)
            {
                allocation[i] = bestIdx;
                blockSize[bestIdx] -= processSize[i];
            }
        }

        System.out.println("\nProcess No.\tProcess Size\tBlock no.");
        for (int i = 0; i < n; i++)
        {
            System.out.print(" " + (i+1) + "\t\t" + processSize[i] + "\t\t");
            if (allocation[i] != -1)
```

```
                System.out.print(allocation[i] + 1);
            else
                System.out.print("Not Allocated");
            System.out.println();
        }
    }

    public static void main(String[] args)
    {
        int blockSize[] = {100, 500, 200, 300, 600};
        int processSize[] = {212, 497, 112, 126};
        int m = blockSize.length;
        int n = processSize.length;

        bestFit(blockSize, m, processSize, n);
    }
}
```

OUTPUT:

```
29e69\redhat.java\jdt_ws\Code_68cc0323\bin' 'BestFit'

Process No.     Process Size    Block no.
 1              212             4
 2              497             2
 3              112             3
 4              126             5
PS C:\Users\Jadhav\Desktop\BTech\4th sem\OS\Prac\Code>
```

WORST FIT

CODE:

```
public class WorstFit
{
    static void worstFit(int blockSize[], int m, int processSize[], int n)
    {
        int allocation[] = new int[n];

        for (int i = 0; i < allocation.length; i++)
            allocation[i] = -1;

        for (int i=0; i<n; i++)
        {
            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 (wstIdx != -1)
            {
                allocation[i] = wstIdx;
                blockSize[wstIdx] -= processSize[i];
            }
        }

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

    public static void main(String[] args)
    {
        int blockSize[] = {10, 50, 20, 30, 60};
        int processSize[] = {22, 47, 11, 46};
        int m = blockSize.length;
        int n = processSize.length;

        worstFit(blockSize, m, processSize, n);
    }
}
```

OUTPUT:

```
jdt_ws\Code_68cc0323\bin' 'WorstFit'

Process No.     Process Size    Block no.
 1              22              5
 2              47              2
 3              11              5
 4              46              Not Allocated
PS C:\Users\Jadhav\Desktop\BTech\4th sem\OS\Prac\Code>
```

CONCLUSION:

- First Fit
  - Advantage
    Fastest algorithm because it searches as little as possible.
  - Disadvantage
    The remaining unused memory areas left after allocation become waste if it is too smaller. Thus request for larger memory requirement cannot be accomplished.
- Best Fit
  - Advantage
    Memory utilization is much better than first fit as it searches the smallest free partition first available.
  - Disadvantage
    It is slower and may even tend to fill up memory with tiny useless holes.
- Worst fit
  - Advantage
    Reduces the rate of production of small gaps.
  - Disadvantage
    If a process requiring larger memory arrives at a later stage then it cannot be accommodated as the largest hole is already split and occupied.
- PROBLEM:
  Given five memory partitions of 100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)?
  Which algorithm makes the most efficient use of memory?
- ANSWER:
  First-fit:
  212K is put in 500K partition
  417K is put in 600K partition
  112K is put in 288K partition (new partition 288K=500K-212K)
  426K must wait
  Best-fit:
  212K is put in 300K partition
  417K is put in 500K partition
  112K is put in 200K partition
  426K is put in 600K partition
  Worst-fit:
  212K is put in 600K partition
  417K is put in 500K partition
  112K is put in 388K partition
  426K must wait
  In this example, best-fit turns out to be the best.