13. Write a program to implement first-fit, best-fit and worst-fit allocation strategies.

A13.

CODE:

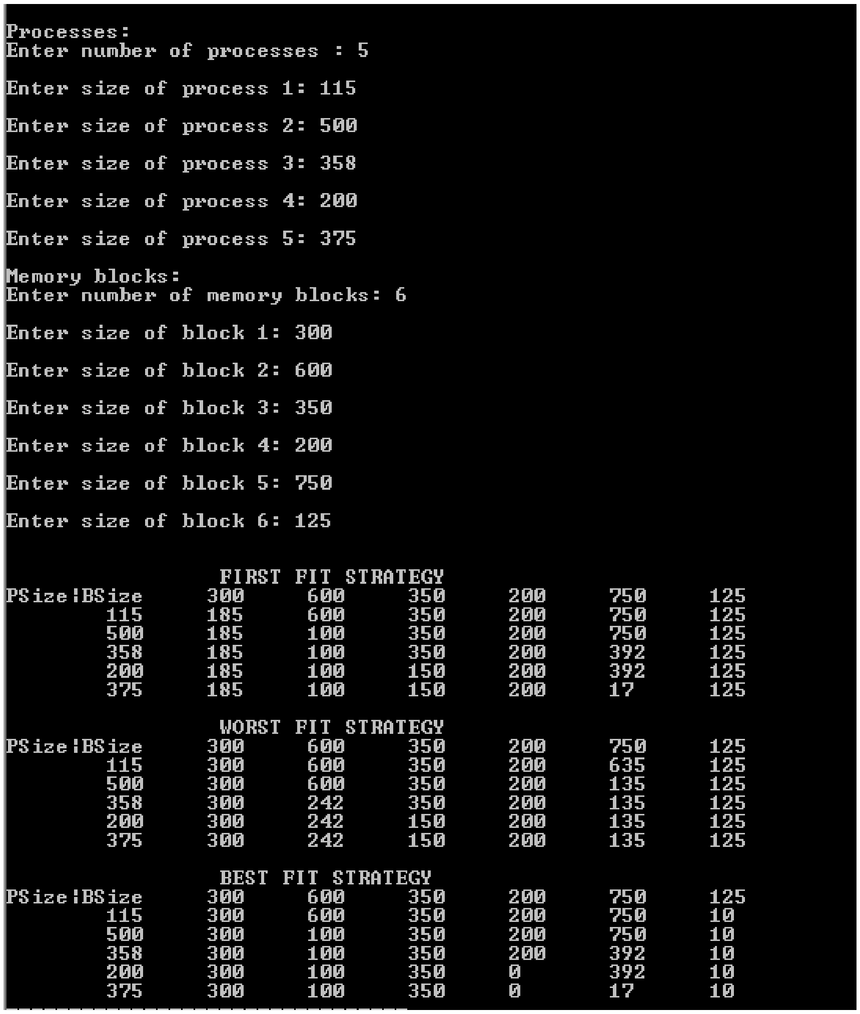
```
#include<iostream>
using namespace std;

struct MemBlock
{
      int bid;
      int size;
      bool free;
};

struct Process
{
      int pid;
      int size;
      int blockid;
};


int MAXP = 10, MAXB = 10;
int num_blocks, num_process;
MemBlock *blocks1, *blocks2, *blocks3;
Process *p1, *p2, *p3;

void entry()
{
      cout<<"\nProcesses: ";
      cout<<"\nEnter number of processes : ";
      cin>>num_process;
      p1 = new Process[num_process];
      p2 = new Process[num_process];
      p3 = new Process[num_process];

      for(int i=0; i<num_process; ++i)
      {
            p1[i].pid=i+1;
            p2[i].pid=i+1;
            p3[i].pid=i+1;
            cout<<"\nEnter size of process "<<i+1<<": ";
            cin>>p1[i].size;
            p2[i].size = p1[i].size;
            p3[i].size = p1[i].size;

            p1[i].blockid = 0;
            p2[i].blockid = 0;
            p3[i].blockid = 0;
      }

      cout<<"\nMemory blocks: ";
      cout<<"\nEnter number of memory blocks: ";
      cin>>num_blocks;
      blocks1 = new MemBlock[num_blocks];
      blocks2 = new MemBlock[num_blocks];
      blocks3 = new MemBlock[num_blocks];

      for(int j=0; j<num_blocks; ++j)
      {
            blocks1[j].bid = j+1;
            blocks1[j].free = true;

            blocks2[j].bid = j+1;
            blocks2[j].free = true;

            blocks3[j].bid = j+1;
            blocks3[j].free = true;

            cout<<"\nEnter size of block "<<j+1<<": ";
            cin>>blocks1[j].size;

            blocks2[j].size = blocks3[j].size = blocks1[j].size;
      }
}

void show_blocksize(MemBlock *b)
{
      for(int i=0; i<num_blocks; ++i)
            cout<<b[i].size<<"\t";
}

void firstfit()
{
      //assign the first sufficient hole
```

```
	cout<<"\nPSize|BSize\t";
	show_blocksize(blocks1);

	for(int i=0; i<num_process; ++i)
	{
		if(p1[i].blockid==0)
		{
			for(int j=0; j<num_blocks; ++j)
			{
				if(blocks1[j].free && blocks1[j].size>=p1[i].size)
				{
					p1[i].blockid = blocks1[j].bid;
					blocks1[j].size -= p1[i].size;

					if(blocks1[j].size == 0)
						blocks1[j].free = false;
					break;
				}
			}
			cout<<"\n\t"<<p1[i].size<<"\t";
			show_blocksize(blocks1);
		}
	}
}

void worstfit()
{
	//assign largest hole
	cout<<"\nPSize|BSize\t";
	show_blocksize(blocks2);

	for(int i=0; i<num_process; ++i)
	{
		if(p2[i].blockid==0)
		{
			int max = 0;
			for(int j=0; j<num_blocks; ++j)
			{
				if(blocks2[j].free  &&  blocks2[j].size>=p2[i].size  &&
blocks2[j].size>blocks2[max].size)
				{
					max=j;
				}
```

```
            }

            if(blocks2[max].size>=p2[i].size)
            {
                p2[i].blockid = blocks2[max].bid;
                blocks2[max].size -= p2[i].size;

                if(blocks2[max].size == 0)
                    blocks2[max].free = false;
            }
            cout<<"\n\t"<<p2[i].size<<"\t";
            show_blocksize(blocks2);
        }
    }
}

void bestfit()
{
    //assign best hole - minimum fragmentation
    cout<<"\nPSize|BSize\t";
    show_blocksize(blocks3);

    for(int i=0; i<num_process; ++i)
    {
        if(p3[i].blockid==0)
        {
            int min = 0, mindiff=99999;

            for(int j=0; j<num_blocks; ++j)
            {
                if(blocks3[j].free && blocks3[j].size>=p3[i].size)
                {
                    int diff = blocks3[j].size - p3[i].size;
                    if(diff<mindiff)
                    {
                        min=j;
                        mindiff=diff;
                    }
                }
            }

            if(blocks3[min].size>=p3[i].size)
            {
```

```
                        p3[i].blockid = blocks3[min].bid;
                        blocks3[min].size -= p3[i].size;

                        if(blocks3[min].size == 0)
                              blocks3[min].free = false;
                  }
                  cout<<"\n\t"<<p3[i].size<<"\t";
                  show_blocksize(blocks3);
            }
      }
}

int main()
{
      entry();
      cout<<"\n\n\t\t FIRST FIT STRATEGY";
      firstfit();

      cout<<"\n\n\t\t WORST FIT STRATEGY";
      worstfit();

      cout<<"\n\n\t\t BEST FIT STRATEGY";
      bestfit();

      return 0;
}
```

OUTPUT:



```
Processes:
Enter number of processes : 5

Enter size of process 1: 115

Enter size of process 2: 500

Enter size of process 3: 358

Enter size of process 4: 200

Enter size of process 5: 375

Memory blocks:
Enter number of memory blocks: 6

Enter size of block 1: 300

Enter size of block 2: 600

Enter size of block 3: 350

Enter size of block 4: 200

Enter size of block 5: 750

Enter size of block 6: 125


                 FIRST FIT STRATEGY
PSize|BSize     300     600     350     200     750     125
        115     185     600     350     200     750     125
        500     185     100     350     200     750     125
        358     185     100     350     200     392     125
        200     185     100     150     200     392     125
        375     185     100     150     200     17      125

                 WORST FIT STRATEGY
PSize|BSize     300     600     350     200     750     125
        115     300     600     350     200     635     125
        500     300     600     350     200     135     125
        358     300     242     350     200     135     125
        200     300     242     150     200     135     125
        375     300     242     150     200     135     125

                 BEST FIT STRATEGY
PSize|BSize     300     600     350     200     750     125
        115     300     600     350     200     750     10
        500     300     100     350     200     750     10
        358     300     100     350     200     392     10
        200     300     100     350     0       392     10
        375     300     100     350     0       17      10
```