Algorithm design

Brute Force

Input: The number files and their bandwidth and the number of servers
Output: The maximum of bandwidths of all servers

m ← number of servers //user input
n ← number of files //user input

F1 ←Arraylist of files objects which has filenumber and bandwidth as attributes
R1 ← Arraylist of Server objects which has resource number and resource bandwidth as attributes

Resource ← Class for servers with attributes server number and bandwidth and their corresponding getter and setter methods and addbandwidth function
Files ← class for files with attributes file number and corresponding bandwidth and their corresponding getter and setter methods

for(i←1;i<=m;i++) // creation m servers
Resource r=**new** Resource(0,i); // Create resource object with resource number i and default bandwidth 0
R1.add(r); // add resource object r to arraylist R1
End for

for(i←0;i<n;i++) {//creation of n files
Files f=**new** Files(*array1*[i],i+1); //Create file object with file number i+1 and file bandwidth stored in array array1 after reading from file
F1.add(f); // add file object f to arraylist F1
End for

Collections.*sort*(F1,**new** Mycomparator()); // sort the Arraylist of files in decreasing order using the comparator class

S1 ← size of Arraylist of files
S2 ← size of Arraylist of servers
if(s1<=s2) // if number of files are less than are equal to number of servers

for(i←0;i<s1;i++)/ Traverse through all the files

Files f=**new** Files(); // Create file object

Resource r=**new** Resource(); // Create server object

```
                f=F1.get(i); // get the ith file object from the arraylist F1
                r=R1.get(i); //  get the ith file object from the arraylist F1
              r.addbandwidth(f.fbandwidth); // add the bandwidth of file to
server and allocate file on that server
                R1.set(i,r); // set server attributes in Arraylist R1
            End for  End if
          else   // if number of files are greater than number of servers
              index = 0; // index to keep track of number of files
              for(i=0;i<s2;i++){ // Allocate one file per server
                Files f=new Files(); // create a file object
                Resource r=new Resource(); // create a server object
                f=F1.get(i); // get the file object from Arraylist F1
                r=R1.get(i); // get the server object from Arraylist R1
                r.addbandwidth(f.fbandwidth); // add the bandwidth of file to
server and allocate file on that server
                  R1.set(i,r); // set server attributes in Arraylist R1
                  index++; // increment the file counter
                  End if
for(i=index;i<s1;i++) // for more number of files

                    Files f=new Files();// create a file object
                      f=F1.get(i); // get the file object from Arraylist F1
ArrayList<Resource> R = (ArrayList<Resource>)R1.clone(); // create a copy of
server arraylist
Collections.sort(R,new Mycomparator1()); // sort in non decreasing order of
server bandwidths
//this for looking for all possible ways  for the solution to the problem by
looking to the bandwidth of each server and check which one has the minimum
bandwidth and add file to that server
 R.get(0).addbandwidth(f.fbandwidth); // select the server with minimum
bandwidth and the file to that server and add the bandwidth
        End else
        End for
```

Collections.*sort*(R1,**new** Mycomparator1()); // finally sort the list of servers in non-decreasing order using the comparator
Optimal← R1.get(s2-1).sbandwidth; // optimal is the max of bandwidth for the given problem

Time and space complexity

The brute force algorithm allocates all the files using a for loop hence the loop of allocation of files has a complexity of O(no of files) and every time before adding the files to servers it checks the server with minimum bandwidth and the complexity of that function is O(no of servers).
Time complexity is O(no of files * no of servers).
Space complexity is O(no of files + no of servers).

Heuristic

Input: The number files and their bandwidth and the number of servers
Output: The maximum of bandwidths of all servers

m ← number of servers //user input
n ← number of files //user input

F1 ←Arraylist of files objects which has filenumber and bandwidth as attributes
R1 ← Arraylist of Server objects which has resource number and resource bandwidth as attributes

Resource ← Class for servers with attributes server number and bandwidth and their corresponding getter and setter methods and addbandwidth function
Files ← class for files with attributes file number and corresponding bandwidth and their corresponding getter and setter methods

for(i←1;i<=m;i++) // creation m servers
Resource r=**new** Resource(0,i); // Create resource object with resource number i and default bandwidth 0
R1.add(r); // add resource object r to arraylist R1
End for

for(i←0;i<n;i++) //creation of n files
Files f=**new** Files(*array1*[i],i+1); //Create file object with file number i+1 and file bandwidth stored in array array1 after reading from file
F1.add(f); End for // add file object f to arraylist F1

Collections.*sort*(F1,**new** Mycomparator()); // sort the Arraylist of files in decreasing order using the comparator class

S1 ← size of Arraylist of files
S2 ← size of Arraylist of servers
for(i←0,j←0;i<=s1&&j<=s2;i++,j++) // loop that that traverses until either number of files or number of servers reaches the maximum

i1=i+1; // counter to keep track of number of files
j1=j+1; // counter to keep track of number of servers
if(i1>s1) // if all files are allocated break out from the allocation loop
break; End if
if(j1>s2)// if all servers are allocated with one file and there are still more files to be allocated then again start from the first server till all the files are allocated.

j=0; End if
Files f=**new** Files(); // create a file object

Resource r=**new** Resource(); // create a server object

f=F1.get(i); // get the file object from Arraylist F1

r=R1.get(j); // get the server object from Arraylist R1

r.addbandwidth(f.fbandwidth); // add the bandwidth of file to server and allocate file on that server

R1.set(j,r); // set server attributes in Arraylist R1

End for

Collections.*sort*(R1,**new** Mycomparator1()); // finally sort the list of servers in non-decreasing order using the comparator

Optimal← R1.get(s2-1).sbandwidth; // optimal is the max of bandwidth for the given problem

Time and space complexity

The heuristic algorithm allocates the files on the servers without checking any condition, therefore there is only one loop that allocates files and runs for no of files number of times.

Time complexity is O(no of files).

Space complexity is O(no of files + no of servers).

Dynamic programming

Input: The number files and their bandwidth and the number of servers
Output: The maximum of bandwidths of all servers

m ← number of servers //user input
n ← number of files //user input

F1 ←Arraylist of files objects which has filenumber and bandwidth as attributes
R1 ← Arraylist of Server objects which has resource number and resource bandwidth as attributes
S1 ← Arraylist of Suboptimalsolution objects which has number of files and sub optimal solution as attributes

Resource ← Class for servers with attributes server number and bandwidth and their corresponding getter and setter methods and addbandwidth function
Files ← class for files with attributes file number and corresponding bandwidth and their corresponding getter and setter methods
Suboptimalsolution ← Class for storing Suboptimalsolution to memorize the solution, which has numberfiles as one attribute that indicates files in solution set and the Suboptimalsolution to store optimal solution which will be in terms of max(B(j))

Bmax ← It is the decision parameter of the dynamic programming algorithm, it decides whether to add the file on to the server or not.
It is the threshold parameter which checks the condition B(j)<= Bmax for each server , which skips the allocation of particular file if its addition causes the bandwidth of the server violates rule B(j)<= Bmax.

for(i←1;i<=m;i++) // creation m servers
Resource r=**new** Resource(0,i); // Create resource object with resource number i and default bandwidth 0
R1.add(r); // add resource object r to arraylist R1
End for
for(i←0;i<n;i++) //creation of n files
Files f=**new** Files(*array1*[i],i+1); //Create file object with file number i+1 and file bandwidth stored in array array1 after reading from file
F1.add(f); End for// add file object f to arraylist F1

Collections.*sort*(F1,**new** Mycomparator()); // sort the Arraylist of files in decreasing order using the comparator class
S1 ← size of Arraylist of files
S2 ← size of Arraylist of servers

if(s1<=s2) // if number of files are less than are equal to number of servers
int l=0; // to keep track number of files in the solution list

for(i←0;i<s1;i++)// Traverse through all the files

Files f=**new** Files(); // Create file object

Resource r=**new** Resource(); // Create server object

f=F1.get(i); // get the ith file object from the arraylist F1

r=R1.get(i); // get the ith file object from the arraylist F1

int k=f.fbandwidth; // get the bandwidth of the file

int k1=r.sbandwidth; // get the bandwidth of the server

int k2=k+k1; // the sum of bandwidth of the file and the server on which it is to be allocated

if(k2<=*Bmax*) // checks whether the sum is less than or equal to the specified Bmax or the decision threshhold

l++; // increments the file counter

r.addbandwidth(f.fbandwidth); // add the bandwidth of file to server and allocate file on that server

R1.set(i,r); // set server attributes in Arraylist R1

Resource g= Collections.*max*(R1,**new** Mycomparator2()); // select the maximum bandwidth among the servers
Suboptimalsolution o=**new** Suboptimalsolution(); // create a object to Suboptimalsolution

o.setNumberoffiles(l); // set number of files in the solution

o.setOptimalsolution(g.getSbandwidth()); // set the optimal solution for the corresponding number of files

S1.add(o); End if // Memorize the suboptimal solution in the arraylist of Suboptimal solutions.
else file is skipped from the allocation and kept in waiting queue
End else
End if
Collections.*sort*(R1,**new** Mycomparator1()); // finally sort the list of servers in non-decreasing order using the comparator

Optimal← R1.get(s2-1).sbandwidth; // optimal is the max of bandwidth for the given problem

End if

else // if number of files are greater than number of servers

index = 0; // index to keep track of number of files

for(i=0;i<s2;i++) // Allocate one file per server

Files f=**new** Files(); // create a file object

Resource r=**new** Resource(); // create a server object

f=F1.get(i); // get the file object from Arraylist F1

r=R1.get(i); // get the server object from Arraylist R1

if(k2<=*Bmax*) // checks whether the sum is less than or equal to the specified Bmax or the decision threshhold

l++; // increments the file counter

r.addbandwidth(f.fbandwidth); // add the bandwidth of file to server and allocate file on that server

R1.set(i,r); // set server attributes in Arraylist R1

Resource g= Collections.*max*(R1,**new** Mycomparator2()); // select the maximum bandwidth among the servers
Suboptimalsolution o=**new** Suboptimalsolution(); // create a object to Suboptimalsolution

o.setNumberoffiles(l); // set number of files in the solution

o.setOptimalsolution(g.getSbandwidth()); // set the optimal solution for the corresponding number of files

S1.add(o); End if// Memorize the suboptimal solution in the arraylist of Suboptimal solutions.
else file is skipped from the allocation and kept in waiting queue End else index++; // increment the file counter

for(i=index;i<s1;i++) // for more number of files

Files f=**new** Files();// create a file object
f=F1.get(i); // get the file object from Arraylist F1

ArrayList<Resource> R = (ArrayList<Resource>)R1.clone(); // create a copy of server arraylist
Collections.*sort*(R,**new** Mycomparator1()); // sort in non decreasing order of server bandwidths
//this for looking for all possible ways for the solution to the problem by looking to the bandwidth of each server and check which one has the minimum bandwidth and add file to that server
if(k2<=*Bmax*) // checks whether the sum is less than or equal to the specified Bmax or the decision threshhold

l++; // increments the file counter

r.addbandwidth(f.fbandwidth); // add the bandwidth of file to server and allocate file on that server

R1.set(i,r); // set server attributes in Arraylist R1

Resource g= Collections.*max*(R1,**new** Mycomparator2()); // select the maximum bandwidth among the servers
Suboptimalsolution o=**new** Suboptimalsolution(); // create a object to Suboptimalsolution

o.setNumberoffiles(l); // set number of files in the solution

o.setOptimalsolution(g.getSbandwidth()); // set the optimal solution for the corresponding number of files

S1.add(o); // Memorize the suboptimal solution in the arraylist of Suboptimal solutions.
else file is skipped from the allocation and kept in waiting queue End if

End for
Collections.*sort*(R1,**new** Mycomparator1()); // finally sort the list of servers in non-decreasing order using the comparator
Optimal← R1.get(s2-1).sbandwidth; // optimal is the max of bandwidth for the given problem

End if

Time and space complexity

The dynamic algorithm allocates all the files using a for loop hence the loop of allocation of files has a complexity of O(no of files) and every time before adding the files to servers it checks the server with minimum bandwidth and the complexity of that function is O(no of servers) and it has one more condition

check because of decision problem and it does not adds to the complexity since its complexity is O(1).
Time complexity is O(no of files * no of servers).
Space complexity is O(no of files + no of servers).

Dynamic programming formula:

The dynamic programming approach used is a bottom up approach since the solution for the approach is started from a single file and continues till all files are allocated.

This decision problem approach or black box algorithm approach for the dynamic programming is enforced using Bmax which is the maximum threshold for the server bandwidth and skips the file from allocation keeps in waiting queue until the required amount of resource is again available.

$$\text{SuboptimalSolution}(j\ldots\ldots.1) = \begin{cases} \max(b_j + \text{SuboptimalSolution}(j\text{-}1\ldots\ldots.1)) & \\ 0 & \text{if no files are there in list} \end{cases}$$

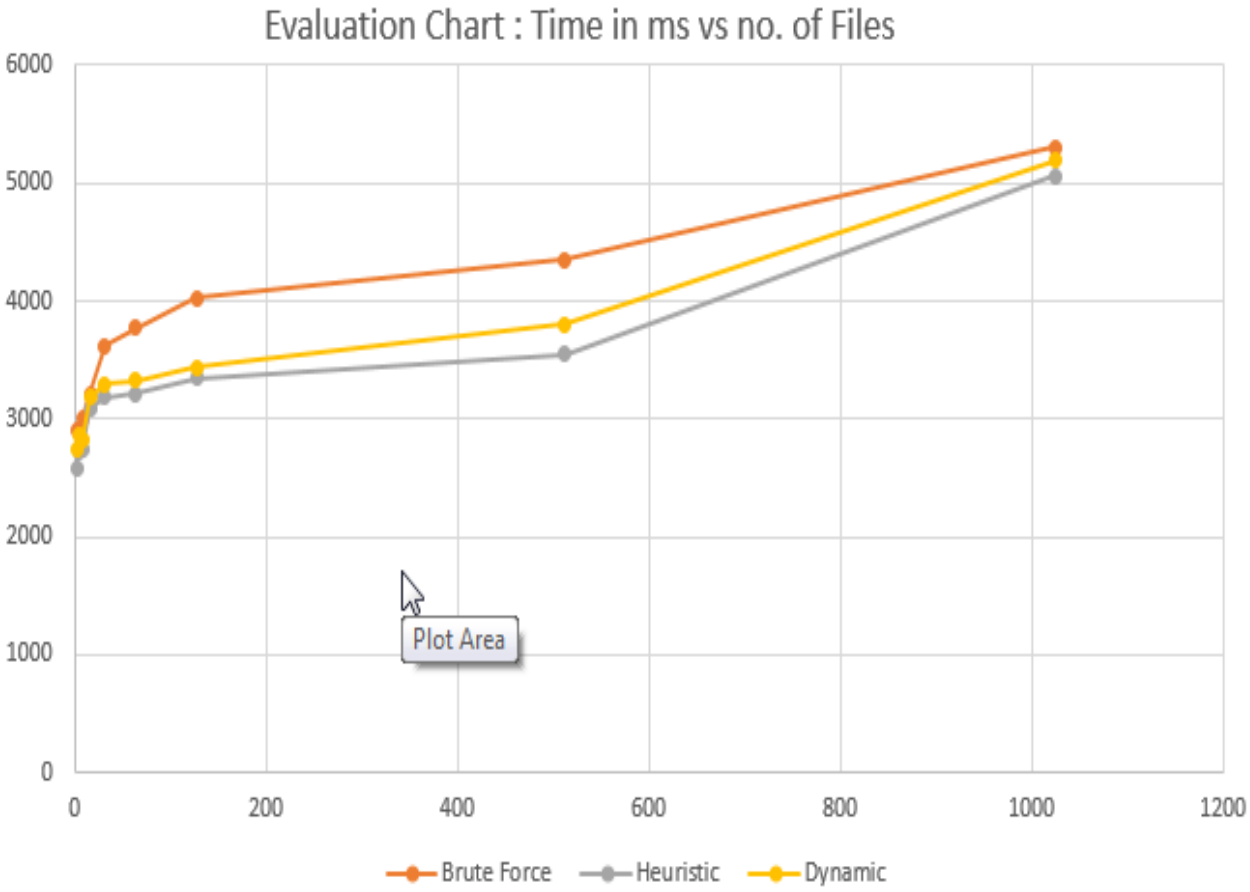
The dynamic programming formula listed above signifies that the optimal solution for the given problem is maximum of the current optimal bandwidth after the allocation of the file and optimal solution of the previous instance of the problem.

The sub-optimal property:

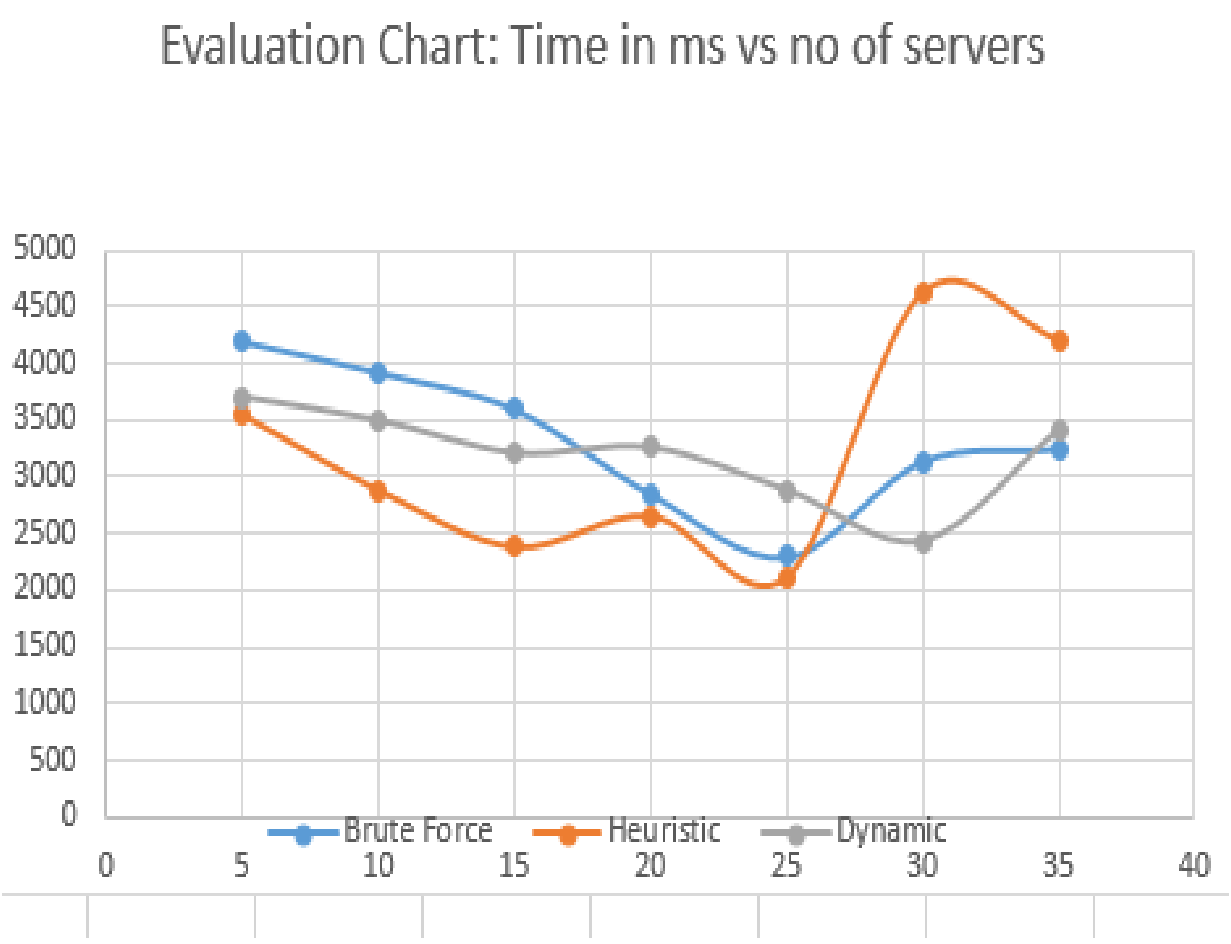
Consider the File sequence 1……..j files and let pi be the optimal solution and let jth file be removed from the optimal solution file set and then also the order of allocation of the files from 1……..j-1 does not changes.

Report

Resources	Files	Brute Force Time in ms	Heuristic Time in ms	Dynamic Time in ms
20	2	2911	2588	2734
20	4	2897	2716	2862
20	8	3001	2739	2818
20	16	3213	3090	3181
20	32	3623	3185	3296
20	64	3767	3214	3329
20	128	4026	3341	3437
20	512	4349	3547	3803
20	1024	5304	5064	5197



Files	Resources	Brute Force Time in ms	Heuristic Time in ms	Dynamic Time in ms
20	5	4188	3538	3702
20	10	3917	2878	3495
20	15	3593	2381	3206
20	20	2852	2655	3255
20	25	2296	2116	2878
20	30	3139	4618	2428
20	35	3247	4201	3413



Analysis based on the execution time

The chart of execution time in mili seconds vs number of files with number of servers being constant with number equal to 20. The heuristic algorithm runs faster and the brute force algorithm is the slowest algorithm.

The heuristic algorithm runs faster than any other algorithm since it does not checks any of the condition and it allocates files sorted in the non-decreasing order of file bandwidths in a sequence to each server. Brute force takes more time since it checks for all possible chances by checking the current bandwidth of each server and allocating the file on the server with the minimum current bandwidth.

The dynamic algorithm's run time is in between heuristic and brute force, the run time of the algorithm mainly depends on the threshold factor Bmax , that checks whether the current servers bandwidth is below the maximum threshold with the addition of current file , if not keeps the file in waiting queue.

Analysis based on the optimal solution

The dynamic programming algorithm gives the optimal solution (min (max B_j)) since there is a decision condition on the addition of files on the servers. The decision condition avoids the accumulation of the files on the single server by maintaining the maximum threshold Bmax on each server and there by keeping min (max B_j) as low as possible.

The brute force gives the next possible optimal solution which is better than heuristic algorithm's optimal solution, since it checks for all possible conditions before allocating the file and allocates the file on server with minimum current bandwidth.

The heuristic algorithm gives the worst optimal solution since there is no allocation policy, the files are allocated on the server in a sequential order.

Quality of optimal solution

The solution for Bruteforce and Dynamic algorithm gives almost same result and more precise optimal solution.

```
<terminated> Bruteforce [Java Application] C:\Program Files\Java\jre7\bin\ja
Enter the number of files
4
Enter the number of servers
2
The optimal solution is terms of max(Bj) 290
```

```
<terminated> Dynamic [Java Application] C:\Program Files\Java\jre7\bin\ja
Enter the number of files
4
Enter the number of servers
2
The optimal solution is terms of max(Bj) 290
```

The solution for Heuristic algorithm has a higher value for optimal solution.

```
Enter the number of files
4
Enter the number of servers
2
The optimal solution is terms of max(Bj) 360
```

Program code:

The Bruteforce.java, Heurisitc.java and Dynamic.java are the java files that corresponds to the bruteforce , heuristic and dynamic algorithm respectively. Files.java and Resource.java are the java files corresponding to files and server objects. Mycomparator , Mycomparator1 and Mycomparator2 are the comparator classes. The Suboptimalsolution.java stores the suboptimal solution as a object.

The number of files and servers can be given by the user through console and the file path is hard codded in the file and the file input.txt which contains file bandwidths one per line and is attached in zip file submitted.

The finalproject1 folder is embedded in the submitted zip folder contains src folder which contains all the java files.