1). In the bin packing problem, items of different weights (or sizes) must be packed into a finite number of bins each with the capacity C in a way that minimizes the number of bins used. The decision version of the bin packing problem (deciding if objects will fit into <= k bins) is NP-complete. There is no known polynomial time algorithm to solve the optimization version of the bin packing problem. In this homework you will be examining three greedy approximation algorithms to solve the bin packing problem.

1. Give pseudo code and the running time for each of the approximation algorithms.

**First-Fit algorithm (runtime: O(n^2))**

**There are two loops, with one inside another. The reason why this algorithm is n^2 because the number of bins and the number of items is not independent. Number of bins can be less or equal to the number of items. The outer loop is O(n) whereas the inner loop is theta n.**

First-Fit(capacity, items, item\_list[]){

Create bin\_res array with items size

Assign num\_bins to 0

For I < items from 0, i++{

Make an integer variable named j

For j < num\_bins from 0, j++{

If ( bin\_res at j is >= item\_list at i){

Bin\_res at j gets bin\_res[j] – item\_list[i]

Break out from the current loop

}

}

If(j equals to num\_bins){ //adds a new bin if no bin fits the item

Bin\_res at num\_bins = the capacity – item\_list at i

Num\_bins add 1

}

}

Return num\_bins

}

**Best-Fit Algorithm (runtime: O(n^2))**

**The runtime is n^2 is because it’s like the first fit. The outer loop is O(n) for the number of items and the inner loops are mainly theta (n) for number of bins and the number of candidate bins for item to be packed.**

Best\_fit(capacity, items, item\_list[]){

Create bin\_res array with items size

Assign num\_bins to 0

Create a variable called smallest\_ position

For I < items from 0, i++{

Create a candidate array with num\_bins size

Assign counter to 0

//This loop checks the amount of bins the item can fit into

For k < num\_bins from 0, k++{

If (bin\_res at k is >= item\_list at i){

Candidate array at counter = k

Counter adds 1

}

}

Create a “j” variable

If(counter is >= 2){ //if there are 2 or more bins that can fit the item

Set smallest\_ position to the first candidate

For m < counter from 1, m++{

If(bin\_res at candidate m is <= bin\_res at the smallest \_position{

The smallest position gets candidate m

Break from the current loop

}

}

Bin\_res at the smallest position gets bin\_res at smallest position – item\_list at i

}

Else{

For j < num\_bins from 0, j++{

If ( bin\_res at j is >= item\_list at i){

Bin\_res at j gets bin\_res[j] – item\_list[i]

Break out from the current loop

}

}

}

If(j equals to num\_bins){ //adds a new bin if no bin fits the item

Bin\_res at num\_bins = the capacity – item\_list at i

Num\_bins add 1

}

}

Return num\_bins

}

**First-Fit-Decreasing (runtime: O(n^2))**

**The runtime is also n^2 because we first sort the item list with an ideal algorithm, merge sort, to have a O(nlgn) sorting time. After sorting the item list, we then pass the list to the First-Fit function, which it’s O(n^2). Therefore, this algorithm is O(n^2).**

First\_fit\_dec(capacity, items,item\_ list\_copy[]){

Sort the list by calling merge\_sort(item\_list\_copy, 0, items-1) function

Return from calling the first\_fit(capacity, items, item\_list\_copy) function

}

1. The README.txt and code have been submitted through Teach
2. Randomly generate at least 20 bin packing instances. Summarize the results for each algorithm. Which algorithm performs better? How often?

**Description:**

I first write the test cases as 30 to the random file and by using a for loop, I make a capacity to either 10 or 20; number of items is assigned randomly between 5-25 along with an item list. Inside the for loop, I created another for loop and randomly generate the weight of each item from the list between 1 to the capacity. I then write all of those to the random text file.

**Summary:**

All these results from the three different algorithms are quite similar with the 1-2 difference in the number of bins. Out of 30 test cases, it seems to me that the First Fit algorithm tends to take “more” bins than the other two because there is 1 test case where First Fit algorithm took 1 more bin than the rest. Otherwise, the First Fit and Best Fit algorithm tend to be very similar in number of bins. The best algorithm of these three I believe would be the First Fit Decreasing algorithm because it seems to me that there are couple test cases where it takes the least number of bins compare to the rest. The First Fit algorithm seems to be worst algorithm among the three with the better performance of 0/30 test cases. The Best Fit algorithm seems to be the in the middle between these algorithms with the better performance of 1/30 test cases. The First Fit Decreasing algorithm seems to be the best among these three with the better performance of 8/30 test cases. The rest of the test cases indicating that they all have the same number of bins.

2). Write an integer program for each of the following instances of bin packing and solve with the software of your choice. Submit a copy of the code and interpret the results.

1. Six items S = { 4, 4, 4, 6, 6, 6} and bin capacity of 10

The result I get is 3, which means that the exact number of bins I need for that given number of items at the capacity of 10. I need 3 bins to fit those items.

**CODE**

min S

ST

S >= 1

S - b1 - b2 -b3 - b4 - b5 - b6 = 0

10b1 - 4x11 - 4x12 - 4x13 - 6x14 - 6x15 - 6x16 >= 0

10b2 - 4x21 - 4x22 - 4x23 - 6x24 - 6x25 - 6x26 >= 0

10b3 - 4x31 - 4x32 - 4x33 - 6x34 - 6x35 - 6x36 >= 0

10b4 - 4x41 - 4x42 - 4x43 - 6x44 - 6x45 - 6x46 >= 0

10b5 - 4x51 - 4x52 - 4x53 - 6x54 - 6x55 - 6x56 >= 0

10b6 - 4x61 - 4x62 - 4x63 - 6x64 - 6x65 - 6x66 >= 0

x11 + x21 + x31 + x41 + X51 + x61 = 1

x12 + x22 + x32 + x42 + X52 + x62 = 1

x13 + x23 + x33 + x43 + X53 + x63 = 1

x14 + x24 + x34 + x44 + X54 + x64 = 1

x15 + x25 + x35 + x45 + X55 + x65 = 1

x16 + x26 + x36 + x46 + X56 + x66 = 1

b1 + b2 + b3 + b4 + b5 + b6 >= 1

END

INT b1

INT b2

INT b3

INT b4

INT b5

INT b6

INT x11

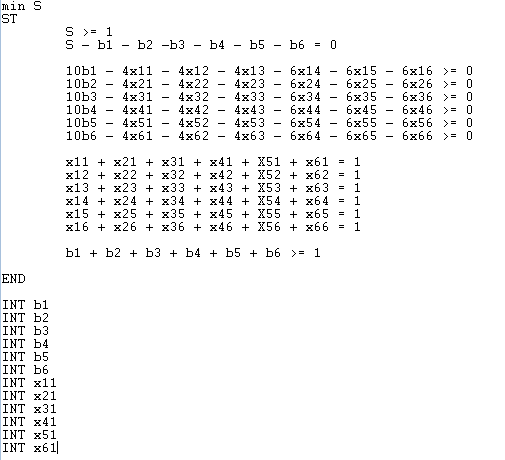
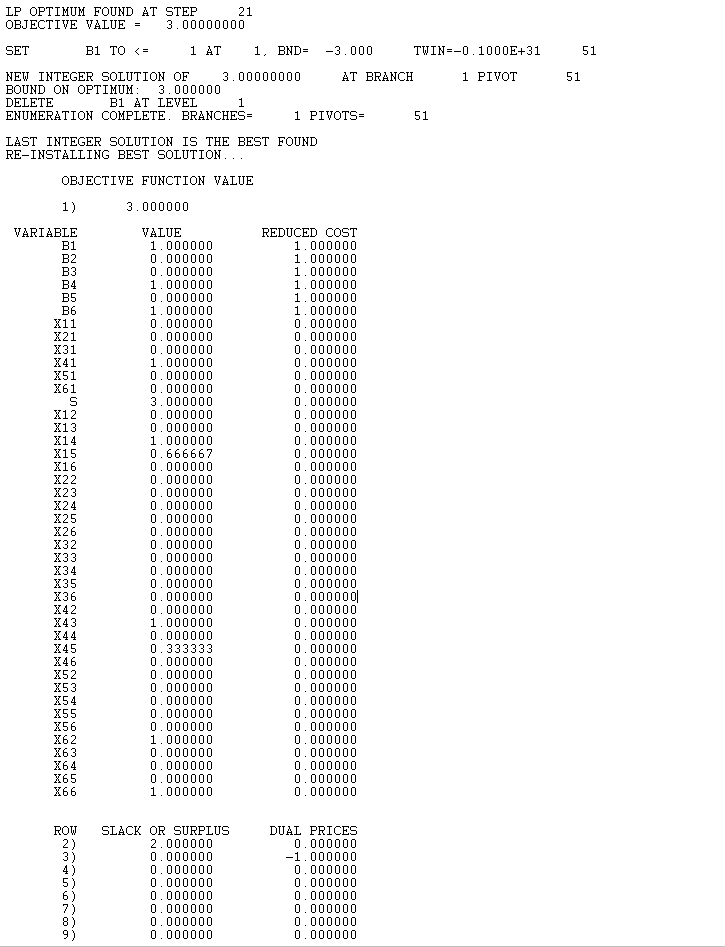
INT x21

INT x31

INT x41

INT x51

INT x61



1. Five items S = { 20, 10, 15, 10, 5} and bin capacity of 20

The result I get is 3 as well, which it also means that I only need 3 exact number of bins to fit all these items, each bin with capacity of 20.

**CODE**

min S

ST

S >= 1

S - b1 - b2 -b3 - b4 - b5 = 0

20b1 - 20x11 - 10x12 - 15x13 - 10x14 - 5x15 >= 0

20b2 - 20x21 - 10x22 - 15x23 - 10x24 - 5x25 >= 0

20b3 - 20x31 - 10x32 - 15x33 - 10x34 - 5x35 >= 0

20b4 - 20x41 - 10x42 - 15x43 - 10x44 - 5x45 >= 0

20b5 - 20x51 - 10x52 - 15x53 - 10x54 - 5x55 >= 0

20b6 - 20x61 - 10x62 - 15x63 - 10x64 - 5x65 >= 0

x11 + x21 + x31 + x41 + X51 = 1

x12 + x22 + x32 + x42 + X52 = 1

x13 + x23 + x33 + x43 + X53 = 1

x14 + x24 + x34 + x44 + X54 = 1

x15 + x25 + x35 + x45 + X55 = 1

x16 + x26 + x36 + x46 + X56 = 1

b1 + b2 + b3 + b4 + b5 >= 1

END

INT b1

INT b2

INT b3

INT b4

INT b5

INT x11

INT x21

INT x31

INT x41

INT x51

