n: number of instances in the problem file

load the instances to Ins(i) (i = 0, 1, 2 ... n)

for each instance Ins(i)

memeticAlgorithm(Ins(i)){

    init\_population(Ins(i), parent\_pop); // initialize instance to a population

    while(iteration < itermax && time\_spent < MAX\_TIME){

        selection(mating\_pool, parent\_pop);  //tournament selection

        crossover(mating\_pool);  //uniform crossover

        mutation(mating\_pool);

        feasibility\_repair(mating\_pool);   //keep droping the item with minimal price until the

                                          //objective is within the constrain in all dimensions

        varaible\_neighbourhood\_search(mating\_pool);     //use pair swap and 1-2 swap

        replacement(mating\_pool, parent\_pop);   //replace top 50 best to population

    }

    update\_best\_solution(parent\_pop);   //update best solution

}

output\_solution(best\_sln, out\_file);    //output the best solution

free\_operaton();                        //free the memory

init\_population(Ins(i), parent\_pop){

    for each individual in parent\_pop{

        //initialize all item as unpacked

        //randomly pack item into knapsack until the it violate the capacity

    }

}

selection(mating\_pool, parent\_pop){

    for each individual in mating\_pool{

        //initialize this individual

        //select several candidates from parent\_pop

        //packed the best candidate into mating\_pool

    }

}

crossover(mating\_pool){

    //use uniform crossover

    //crossover between the head and tale individual

}

mutation(mating\_pool){

    for each individual in mating\_pool{

        //mutate each chromosome with MUTATION\_RATE

    }

}

feasibility\_repair(mating\_pool){

    for each individual in mating\_pool{

        while(individual is not feasible){

            //find the packed item with minimal price

            //remove the item

            //evalutate the individual whether it violates the constrain

        }

    }

}

varaible\_neighbourhood\_search(mating\_pool){

    for each individual in mating\_pool{

        initialize current\_solution

        while(neighbourhood < K){       //K: index of neibourhood

            neibourhood\_solution = best\_descent\_vns(neibourhood\_index, current\_solution)

            if(neibourhood\_solution > current\_solution){       //jump back to first neibourhood

                current\_solution = neibourhood\_index;

                neibourhood\_index = 1;

            }

            else{       //go to next neibourhood

                neibourhood\_index++;

            }

        }

        individual = current\_solution

    }

}

best\_descent\_vns(neibourhood\_index, current\_solution){

    if(neibourhood\_index == 1){     //pair swap

        //divide the packed and unpacked items into two lists storing it index for reducing the time complexity

        //travas two list to select two items to swap

        //if the swap is better then record this move

        //if the better swap is greater than the best swap, apply this swap

    }

    else if(neibourhood\_index == 2){    //1-2 swap

        //divide the packed and unpacked items into two lists storing it index for reducing the time complexity

        //apply 1-2 swap for 10,000 times to reduce the running time in one iteration

        //if the swap is better then record this move

        //if the better swap is greater than the best swap, apply this swap

    }

}

replacement(mating\_pool, parent\_pop){

    //joint mating\_pool and parent\_pop togeter

    //quick sort the mixed pool

    //select the top 50 solution as the parent\_pop of next generation

}