Problem 1 (Continuous knapsack):-

The greedy Strategy for continuous knapsack follows below steps:-

(1) We calculate ratio value/weight ($c_i = \frac{v_i}{w_i}$) for each item from 1, 2 - - - n.

(2) Then, Sort the items on the basis of the ratio in decreasing order. Consider the sorted item sequence is 1, 2, - - - - n & the corresponding ratios & weights are $c_1 \ldots c_n$ & $w_1 \ldots w_n$ respectively.

So,

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}$$

(3) Initially we have weight capacity K & let current capacity y.

In Every iteration we select item 'i' from the indexes 1, 2 - - - n.

This i^{th} item present in the top of the list.

(i) If $y \geq w_i$, we can add whole item i & update the current weight $y = y - w_i$

(ii) if $y < w_i$, we need to take a fraction of item i. This means the fraction value is $f = \frac{y}{w_i}$ (v_i) of the item i.

fraction = $\frac{\text{value}_i * \text{remaining capacity}}{\text{weight}_i}$

Time analysis:-

(1) step 1 takes $O(n)$ times.

(2) step 2 takes $O(n \log n)$ times

(3) step 3 takes $O(n)$ times

Thus overall runtime is $O(n \log n)$.

Correctness:-

We define a ordered set $\{a_1, a_2, a_3 \dots a_n\}$

Such that $\frac{v_{ai}}{w_{ai}} > \frac{v_{ai+1}}{w_{ai+1}}$ where $1 < i < n$.

for ordered set $\{a_1, a_2 \dots a_n\}$ we define $\{a_1, a_2 \dots a_k\}$ as a subset of it where $k \leq n$.

Lemma:- Suppose A is a subset of an optimal solution. Let $c_i = \frac{v_0}{w_i}$ be the task that has the highest value/weight ratio among the items that are not in A.
Let A' be the subset obtained by appending c_i to end of A. Then A' is also a subset of an optimal solution.

Proof: Since A is a subset of an optimal Solution. Let A^* be the optimal solution that contain A as a subset of A^*.

Let C_i be the task with highest value/weight ratio among the leftover items means $A^* - A$.
By attending C_i to A, we get another solution A'.

There are two cases

(i) If whole of i is also contained in A^*, then A' & A^* agree, the lemma holds.

(ii) If A^* does not take whole unit of i. Then we first take away the whole weight of w_i from A^* & replace it with whole weight of w_i. The capacity of A^* does not exceed K.

This yields to a new solution A^4.
A^4 will be as good as A^* because item i had the maximum value by weight ratio. & by replacing it in A^* we make sure that the capacity K does not exceed & the value of knapsack is maximum.

If A^* takes fraction f of item i & A' also take the same f of item i, the lemma holds.

If A^* takes less than f part of item i (we can not take larger part because of capacity)

To exchange, we first take $f \times w_i$ away from A^* & replace it with f unit of item i. This will lead to a new solution A^q. The weight of A^q do not exceed the capacity k & also it is as good as A^* because items had the best value/weight ratio value & we took the maximum part that we could accomodate, this lead to another optimal solution. Thus A^q is not any worse then that A^*.

Theorem: Initially A is empty which is a subset of optimal solution

Let All items are sorted by ratio of value/weight in decreasing order.

Since
By lemma & induction on the number of iteration when the function terminates, A is a subset of an optimal solution.

Since there are no more items i that can be placed because we have filled the knapsack upto its weight capacity k. no other solution can contain A, thus A is optimal solution itself.

Problem 2 :- (minimizing average completion time) :-

Our goal is to minimize completion time of the schedule such that $\left[\frac{1}{n}\sum_{1 \le i \le n} c_i\right]$ should be minimum.

Greedy approach follows below steps.

(a) We have given with set of n tasks denoted by y_i where i ranging from 1 to n. We are also given the completion time for each task t_i where i ranging from 1 to n.

(1) Merge sort the n tasks in order of increasing execution time such that

$$t_1 \le t_2 \le \cdots \le t_n.$$

(2) Then we will schedule the tasks based on their sorted execution time.

$$A := [y_1, y_2 \cdots y_n]$$

Time analysis :-

(1) Step (1) takes $O(n \log n)$ time.
(2) Step (2) take $O(n)$ time.

Thus overall runtime is $O(n \log n)$.

Correctness:-

Containment relationship: the schedule sequence is defined as $\{y_1, y_2, \ldots y_m\}$ where y_i should be scheduled only after completion of y_{i-1}.

where $1 \leq i \leq m$. for schedule sequence $\{y_1, y_2 \ldots y_n\}$, we define $\{y_1, y_2 \ldots y_k\}$ as a prefix-subsequence of it where $k \leq n$.

Lemma:- Suppose A is a prefix-subsequence of an optimal schedule sequence. Let y_i be the task that has the shortest execution time among those tasks that are not in A. Let A' be the prefix-subsequence by appending y_i to the end of A. Then A' is also a prefix-subsequence of an optimal schedule solution

Proof:- Since A is a prefix-subsequence of an optimal schedule solution sequence, Let A^* be the optimal schedule sequence that contains A as a prefix-subsequence:

Let y_i be the task that has minimum execution time among those tasks in $A^* - A$. By appending y_i to the end of the A, we get another prefix-subsequence A'. There are two cases:-

(1) A' is a prefix-subsequence of A^*, the lemma holds.
(2) A' is not a prefix subsequence of A^*

$y_1\ y_2 \ldots y_i \mid y_m \ldots$ A^* schedule
$y_1\ y_2 \ldots y_i \mid y_j \ldots$ A schedule
A^* & A' agreed

Let J_m be the time task that appear first in the subsequence $A^* - A$. In this case we must have,

$$t_m = t_i .$$

This can be prooved by Proof of contradiction.

Suppose $t_m \neq t_i$, since J_i is the task with shortest execution time in $A^* - A$, we must have $t_m > t_i$.

Let the last task in A be J_l.

Now A^* continue from A as $J_m, J_{c_1}, J_{c_2} \ldots J_{c_s}, J_i, J_{d_1} \ldots, J_{d_r}$. such that s number of tasks are assigned between J_m & J_i and r tasks scheduled after J_l

we have

$C_m \geq$ (Completion execution time upto l + time required for task m)

$$C_m \geq C_l + t_m .$$

C_{ck} = Completion time for task scheduled till tasks m + Completion time from m upto task i

$$\geq C_m + \sum_{1 \leq k \leq s} t_{ci}$$

C_i = [Execution time upto task m + execution time of task from m upto task i + execution time of i]

$$C_i = C_m + \sum_{m \le i \le s} t_{ci} + t_i$$

C_{dk} = [execution time upto task i + Execution time of tasks after i]

$$C_{dk} \ge C_i + \sum_{i \le j \le k} t_{dj}$$

Suppose we were to exchange the position of y_i & y_m, so our solution would look like similar $y_1, y_2 \dots y_{i-1} y_m, y_{c_1}, y_{c_2} \dots y_{c_s} y_j y_{d_1} \dots y_{d_r}$. Lets say this sequence is A''.

C''_{ck} is the total time of tasks from y_i & y_j

C''_{dk} is the total execution time of tasks from y_j

we get $C''_i = C_i + t_i$

$$C''_{ck} = C''_i + \sum_{i \le i \le s} t_{ci}$$

$$C'_m = C'_i + \sum_{1 \le i \le s} t_{ci} + t_m$$

$$C'_{dk} = C'_m + \sum_{1 \le i \le s} t_{di}$$

Lets compare the equations.

$$\therefore C_m - C'_i = (C_l + t_m) - (C_l + t_i)$$

$$\Rightarrow t_m - t_i$$

$$> 0$$

$$\therefore C_{Ck} - C'_{Ck} = \left[C_m + \sum_{1 \le i \le s} t_{ci} \right] - \left[C_i' + \sum_{1 \le i \le s} t_{ci} \right]$$

$$\Rightarrow C_m - C_i'$$

$$> 0$$

$$\therefore C_i - C'_m = C_m - C'_i + t_i - t_m$$

$$\Rightarrow t_m - t_i + t_i - t_m$$

$$\Rightarrow 0$$

Similarly $\because C_{yk} - C'_{yk} = C_i - C'_m = 0$

This allows us to conclude that sum of completion time A^4 is not smaller than the completion time of A^4 which means A^4 is a more optimal solution. This contradicts our fact that A^4 is an optimal solution. Hence we conclude that $t_i = t_m$.

So, we get another optimal schedule sequence by exchanging the order of q_i & y_m because $t_i = t_m$. the exchange will not affect overall average completion time. & new optimal schedule sequence has A' as a Prefix-subsequence The lemma still holds.

Theorem:-

(1) Initially, A is a empty sequence, which is always a prefix subsequence of any schedule sequence.

(2) All n tasks $\{y_1, y_2 \cdots y_n\}$ have been sorted in order of increasing execution time. By lemma $\{q_i\}$ is a prefix-subsequence of an optimal schedule sequence.

(3) By Induction of no. of iteration when the algorithm terminates, A is still a prefix subsequence of an optimal solution.

(4) When the algorithm terminates, A would contain all n tasks, no other schedule sequence can properly contain A, Thus A is the optimal schedule sequence.

Problem 2

(b) for this problem we can use shortest remaining time policy to find an optimal schedule. It uses a min-heap to organize the released but not completed tasks where heap ordering is based on their remaining execution times. A scheduling decision is made when a task is completed or when a new task is released.

Problem 3 (Deleting the larger half):-

We will use an unsorted array A to implement for the given two operation.

Let n be the size of Array A. Initially n = 0.

Pseudo code for both operation.

Insertion(item,

A[n+1] := item

Insert(x, S)
n := n+1
A[n] := x

DeleteLargerHalf(S)
→ find median using linear-time median algorithm on Array A.
→ we then use this median to partition the array around the median.
→ Then delete the bigger half of the Partitioned array A.
→ then $n := n - \lceil n/2 \rceil$, resize the size of A.

Worst case time analysis:-
Insert takes constant time while DeleteLargeHalf takes O(n) time since finding the median takes linear time, so do Partitioning & removing the elements.
So total time taken to perform the DeleteLarger-Half is O(n) time.

Amortized Analysis :-

The following table shows the true & amortized cost.

Operation	true time (t_i)	Amortized cost (a_i)
Insert	2	6 } O(1) amortized time
DeleteLargerHalf	{ no. of elements to be discarded }	0

True time for insert takes a cost of two, one for incrementing n & second one assigning x to A[n].
DeleteLargerHalf would take linear time n to find median, linear time(n) to partition

Insert takes 2 instruction, so it uses 2 unit cost but it gets 6 units, we Conceptually store the remaining 4 unit of credit on the element we just inserted into the Array.

DeleteLargerHalf uses 2 unit of credit stored on each element, 1 unit for median finding & 1 unit for Partitioning.
While deleting the larger half we redistribute the 2 units of credit on the remaining elements.
This means that there are always 4 units of credit on each item.

So, we always have enough credit to pay for future delete_larger_half operation as well.
Each element in the array we will have 4 unit of credit on the element & size of an array is always non-negative. we make sure that the amount of credit is always enough.

Thus we proved that we take O(1) amortized time cost for each operation.

Problem 4 (Binary search with insertion):-

(a) We can implement find by doing binary search on each array in D because arrays are sorted & full. By using Binary search, we can perform find operation efficiently.

Pseudo code:-

```
find (k, D)
  for i := 1 to l (length of D)
    if bit i = 1 in binary representation of n:
      search key k using binary search.
      if k is found, return associated item with k,
      else continue to next array.
```

Time analysis:-

The worst case would be of all the arrays from $A_1, A_2, A_3 \ldots A_l$ are full where $l = \lceil \log(n+1) \rceil$,

Then total time taken is,

$$T(n) = \theta\left(\sum_{i=1}^{l} \log\left(2^{i-1}\right)\right) \quad \{\text{each array has length } 2^{i-1}\}$$

$$\Rightarrow \theta\left(\sum_{i=1}^{l} (i-1)\right)$$

$$\Rightarrow \theta\left(\frac{l(l-1)}{2}\right)$$

$$\Rightarrow \theta(l^2)$$

$$\Rightarrow \theta\left(\lceil \log(n+1) \rceil^2\right) = \theta(\log^2 n)$$

Thus the Worst case analysis time for the find operation is $O(\log^2 n)$.

(b) The Insert operation can be implemented by creating new sorted array B_1 with size 1 that contains the key value pair, (k, x) to be inserted.

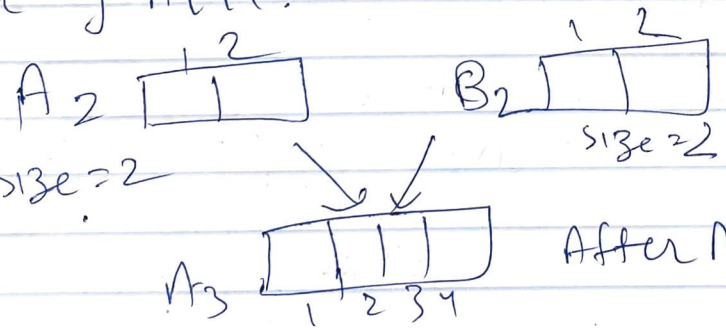
We first check A_1, we do this by checking the i bit in the binary representation in n, if its empty we just replace A_1 with B_1.

If A_1 is not empty, we merge sort A_1 & B_1 into a new sorted array B_2. Then check if A_2 is empty.

If it is empty, we just replace it with B_2

If not we merge sort A_2 & B_2 & continue to check for the empty A_i Array in D.

A_i is size of 2^{i-1} & the array it is merged with also the same size 2^{i-1}, resulting in new array of size 2^i which is exactly the size of A_{i+1}.



size(A_3) = 4

Time analysis :-

merging two sorted array of the size n into one sorted array takes 2n time. The worst case is when all the array from $A_1, A_2 \ldots A_{l-1}$ are full so we have merge sort all of the them into B into A_L.

This takes :-

$$T(n) = 2\left(\sum_{i=1}^{l-1} 2^{i-1}\right)$$

$$\Rightarrow 2(2^{l-1} - 1)$$

$$\Rightarrow 2^l - 2$$

$$\Rightarrow 2^{\lceil \log(n+1) \rceil} - 2$$

$$\Rightarrow \Theta(n)$$ (using exponential property)

The worst case insertion takes $\Theta(n)$.

Amortized Analysis :- using aggregate method.

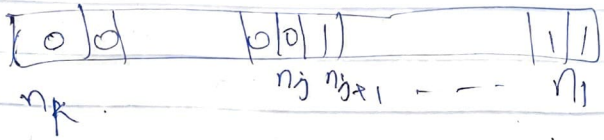
The aggregate method compute the total cost for a sequence of n Insert operation starting with the empty data structure D_0.

The Binary representation of D_i with n element is going to be

$$n_k, n_{k-1} \ldots n_1$$

Let us assume that j be the position of the right most 0 bit & all other bits to right of n_j are 1

which means their corresponding arrays are full.



Thus the total cost of inserting the new element would be

$$T(n) = 2 \cdot \left(\sum_{i=1}^{j-1} 2^{i-1} \right)$$

$$\Rightarrow 2(2^{j-1} - 1)$$

$$\Rightarrow 2^j - 2$$

$$\Rightarrow O(2^j)$$

Binary representation of initial insertions:-

$D_1 = 00\text{---}0001$

$D_2 = 00\text{---}0010$

$D_3 = 00\text{---}0011$

$D_4 = 00\text{---}0100$

$D_5 = 00\text{---}0101$

→ flip in every $2^0 = 1$ iteration

↳ flip in every $2^1 = 2$ iteration

Thus for each value of j, insertion will need atmost $\left\lceil \frac{n}{2^{j-1}} \right\rceil$ insertions.

So; total cost of n operations.

$$T(n) = O\left(\sum_{j=1}^{l} \left\lceil \frac{n}{2^{j-1}} \right\rceil 2^j\right) \quad \left(l = \lceil \log(n+1) \rceil\right)$$

$$\Rightarrow O(n \log n)$$

Therefore the amortized cost for n Insertions operation is $O(n \log n)$ & for each insertion it will $O(\log n)$ time.