PART A:

A1

```
// q <- the first item in Queue

Mark(stack)
    Initialise Queue Q_be
    for all stack entries vol do
        Q_be.enqueue(vol)
    while not Q_be.isEmpty() do
        q.colour = black
        if q.left is not null && q.left.colour is white do
            Q_be.enqueue(q.left)
        if q.right is not null && q.right.colour is white do
            Q_be.enqueue(q.right)
        Q_be.dequeue(q)
```

A2

```
Sweep(heap)
    for all heap cells c do
        if obj(c).colour == white do
            reclaim(c)
        else
            obj(c).colour = white
```

A3

The tight asymptotic upper bound of Mark() is O(m+n).
It loops through the whole stack, and at the worst case every heap cell is enqueued.
The tight asymptotic upper bound of Sweep() is O(n). It loops through the whole heap.
Hence the tight asymptotic upper bound of Mark-Sweep Algorithm is O(m+2n) which implies O(m+n).

A4

The best scnario is that the stack entries don't reference to anything,
hence Mark() takes Ω(m) time to complete, and Sweep() takes Ω(n) time to complete as it loops through all the heap cells anyway.
Thus, the tight asymptotic lower bound of Mark-Sweep Algorithm is Ω(m+n).
Since the upper bound is O(m+n) and the lower bound is Ω(m+n),
they allow for a 0 bound of 0(m+n).

A5(a)

The Mark() Algorithm enqueues each reference at most once because
during each while iteration, the left and right nodes are checked
if they are white(unvisited) or black(visited) by the Mark() Algorithm.
Only white vertices are enqueued. The initial stack entries were all not visited when they are enqueued.

A5(b)

Mark():
- Initialise Queue Q_be takes 1 execution.

- enqueue each stack entry takes m execution.

- within the while loop:
    not Q_be.isEmpty() takes 1 execution
    q.colour = black takes 1 execution
    q.left is not null && q.left.colour is white takes 2 execution
    Q_be.enqueue(q.left) takes 1 execution
    q.right is not null && q.right.colour is white takes 2 execution
    Q_be.enqueue(q.right) takes 1 execution
    Q_be.dequeue(q) takes 1 execution

- For exiting the while loop, not Q_be.isEmpty() takes 1 execution

In total, it takes 1+m+(1+1+2+1+2+1+1)n+1, or
m+9n+2 line executions to complete Mark().

Sweep():
- with in the for loop:
    if obj(c).colour == white takes 1 executions
    No matter the outcome of the if statement,
    it takes 1 execution to complete either clauses.
    (reclaim(c) or obj(c).colour = white)
In total, (1+1)n or 2n executions are made to complete Sweep().

The entire Mark-Sweep process takes m+9n+2+2n or m+11n+2 line
executions to complete. Hence, the upper bound is O(m+11n+2).

A5(c)

Assuming m is a constant, we want to show
    $\exists C. \exists N. \forall n \geq N. (m+11n+2) \leq C(n+m)$
Since m must be a non-negative integer,
    $m + 11 + 2 \leq 13 + 13m \rightarrow m \leq 13m \rightarrow 1 \leq 13$
    C = 13, N = 1 would satisfy the condition.

Assuming n is a constant, we want to show
    $\exists C. \exists M. \forall m \geq M. (m+11n+2) \leq C(n+m)$
Since n must be a non-negative integer,
    $1 + 11n + 2 \leq 11n + 11 \rightarrow 3 \leq 11$
    C = 11, M = 1 would satisfy the condition.

Hence, whenever $m \geq 1$ or $n \geq 1$ and $C \geq 13$, $f(m,n) \leq Cg(m,n)$
where f(m,n)=m+11n+2 and g(m,n)=n+m.
Thus, O(m+11n+2) satisfy the claimed O(n+m) bound.

A6a

Since P only works with a maximum of 10 stack entries and
the number of Vertex objects reachable from the stack entries
at any given time never rises above n/2, in the worst case all
10 stack entries would have been used as well as n/2 reachable
objects. Once the memory becomes full (Count=n), the user program
would be suspended till recycling has been done.

In this case, the all memory would have been used and n/2
unreachable objects would need to be recycled.

Mark() takes m+9n+2 line executions given m = 10, n = n/2 -> 4.5n+12

Sweep takes 2n line executions where n = n

Mark-Sweep() = O(6.5n+12)

to justify this upper bound, we want to show
$\exists C. \exists N. \forall n \geq N. (6.5n+12) \leq C(n+m)$
In this case, there can only be a maximum of 10 stack entries
hence we want to show
$\exists C. \exists N. \forall n \geq N. (6.5n+12) \leq C(n+10)$
$6.5 + 12 \leq 45.5 + 70 \rightarrow 18.5 \leq 115.5$
C = 7, N = 1 would satisfy the condition
Hence, O(7n+12) would be a valid asymptotic upper bound for
the total time spent on memory recycling.

A6b

The average time spent on memory recycling over p program actions
would be O((7n+12)/p).

to justify this upper bound, we want to show
$\exists C. \exists N. \forall n \geq N. ((7n+12)/p) \leq C(n+10)$
In this case, p would be considered a positive integer constant where
$p \geq 1$, since only positive amount of program actions can be done.

Since p would be at least 1,
$19/p \leq 77$
C = 7, N = 1 would satisfy the condition
Hence, O((7n+12)/p) would be a valid asymptotic upper bound for the
average time spent on memory recycling per program action.

A7

In this case, m = 10 and n = rn where $0<r<1$.
The time complexity would be:

Mark() line executions given m = 10, n = rn -> 10+9rn+2

Sweep() line executions -> 2n

Mark-Sweep() = O((9r+2)n + 12)

The average time spent on memory recycling per program action would be:
$O((9r+2)n + 12)/p) \rightarrow O((9r+2)/p)n + 12/p)$
since r, p, and m are constants, and 12/p isn't asymptoticly significant,
it would be a valid upper bound in this case relative to O(n).

As r approaches 1 from 0, the time spent on memory recycling per program
action can slow down as much as 5.5 times.

A8

Firstly, up till the while loop, as the stack entries are being added
to the queue, the maximum length of the queue before the while loop
at most can only be m, hence:

$Q \leq m$ + number of blacks not in queue

where Q = the queue before while loop iterations.

which hold at all times up to the start of the main loop.

After entering the while loop, while processing the current queue entry, there could be multiple cases:

- case 1: both s.left and s.right are invalid/null

In this case, after the iteration has been completed, The number of blacks not in queue has increased by 1 since 1 black cell has been dequeued after interation. Hence the inequality can be formulate as such:

$$Q - i \leq m + i$$

where i = number of while loop iteration

when i = 0, the invariant holds.
assume when i = k, the invariant holds,
when i = k + 1:

$$Q-(k+1) \leq m+(k+1) \rightarrow Q \leq m + 2(k+1)$$

which holds the invariant given, by induction, the invariant would be maintained for any positive integer i.

- case 2: either s.left or s.right is valid and referenced by the current queue entry

In this case, after the iteration has been completed:

$$Q+i-i \leq m + i \rightarrow Q \leq m + i$$

which holds the invariant since $Q \leq m$ before entering the while loop.

when i = 0, the invariant holds.
assume when i = k, the invariant holds,
when i = k + 1:

$$Q \leq m + (k+1)$$

which holds the invariant given, by induction, the invariant would be maintained for any positive integer i.

- case 3: both s.left and s.right are valid and referenced by the current queue entry

In this case, after the iteration has been completed:

$$Q+2i-i \leq m + i \rightarrow Q + i \leq m + i \rightarrow Q \leq m$$

which holds the invariant.

when i = 0, the invariant holds.
assume when i = k, the invariant holds,
when i = k + 1:

$$Q + (k+1) \leq m + (k+1)$$
$$Q \leq m$$

which holds the invariant given, by induction, the invariant would be maintained for any positive integer i.

Since in all 3 cases the invariant holds, the invariant would be maintained

at all times.

A9

$$\text{queue length} \le m + \text{number of blacks not in queue}$$

Hence:

$$(m+n)/2 \le m + \text{number of blacks not in queue}$$

if we exceed the maximum queue length by 1:

Assuming this inequality still holds:

$$(m+n)/2 + 1 \le m + \text{number of blacks not in queue}$$

Since the number of blacks not in queue at most could be:

$$\text{number of blacks not in queue} = (m+n)/2 - m$$

In this case,

$$(m+n)/2 + 1 \le m + (m+n)/2 - m$$
$$(m+n)/2 + 1 \le (m+n)/2$$

Which is not true.

By contradiction,the statement of queue length would never exceeds (m+n)/2 is valid.

A10

PART B:

B1

```
Greedy(S;s_1, . . . , s_n;v_1, . . . ,v_n)
    Initialise Array array with size n
    Initialise ZeroArray r_array with size n
    for i=1 to n do
        array.append(v_i / s_i, i)
    MergeSort(array) using ratio as key
    counter = n - 1
    while S > 0 do
        index = array[counter][1]
        if s_index > S do
            r_array[counter] = S / s_index
        else do
            r_array[counter] = 1
        counter = counter - 1
        S = S - r_array[counter]
    return r_array
```

B2

- Initialise an array with size n takes 1 line execution.

- Initialise ZeroArray r_array with size n could be O(1) or O(n) depending on whether the array is initialized as a static or automatic storage duration variable. In this case, it is assumed it takes 1 line execution.

- for loop for calculating ratios take n line executions.

- MergeSort takes O(nlogn) time. As the exact implementation could lead to varying line executions, it is assumed it takes nlogn line executions to complete the sorting procedure.

- initialise the counter takes 1 line execution.

- while loop takes 5 line executions in each interation, but the exact number of iterations depends on the size of S(the total volume). In this case, the worst case is used: every thing can fit in the knapsack. This would take 5n+1 operations, the 5 iterations plus one last check to exit the loop.

- returning takes 1 line execution.

In total, it would take (1+1+n+nlogn+1+5n+1+1) or (6n+nlogn+5) line executions to complete.

B3

The algorithm does always finds the optimal solution in the fractional setting. It is true as the algorithm always prioritises the item with the best value to volume
ratio, as well as the divisability of each item which allows the left over spaces to be
entirely filled, which maximises the total value of items in the knapsack.

B4

good A is in total valued 2 units, and takes 1 unit of space.
good B is in total valued 3 units, and takes 3 unit of space.
There a 3 unit of free space in the knapsack.
A greedy strategy would take good A as it has the higher value to volume ratio, the goods in the sack would have valued 2 units in the end, although taking good B would a better solution.

B5

```
Dynamic(S; s1, . . . , sn; v1, . . . , vn)
    Initialise 2DArray V with n rows and S columns
    Initialise ZeroArray array with length n
    for vol = 0 to S do
        V[0, vol] = 0
    for i = 1 to n do
        for vol = 0 to S do
            if s_i ≤ vol do
                V[i, vol] = max(V[i-1, vol], v_i + V[i-1, vol-s_i])
            else do
                V[i, vol] = V[i-1, vol]
    vol_left = S
    for n to 1 do
        if V[n, vol_left] != V[n-1, vol_left] do
            array[n-1] = 1
            vol_left = vol_left - s_n
    return array
```

- initialise V takes 1 line execution.
- initial array takes 1 line execution.
- first row setting for loop takes S line executions.
- generating the rest of V takes 2S(n-1) line executions, 2 line executions in each column for n-1 rows.

- setting vol_left takes 1 line execution.
- line executions for solution solving for loop can vary depending on the items and the given total volume. In the worst case, everything fits in the sack, which takes 3(n-1) line executions to complete.
- returning takes 1 line execution.

In total, it takes (1+1+S+2S(n-1)+1+3(n-1)+1) or (2S+3)n-S+1 line executions to complete.