IADS CW2

S1976502

Part A

Memory Recycling

Q1

```
Mark():

    for all stack entries e:

        if e != null:

        if  e.color == white:

                    Q.enqueue(e)

                    e.color = black

        while !(Q.isEmpty)

            v = Q.dequeue()

            if v_right != null:

            if v_right.color = white:

                    Q.enqueue(v_right)

                    v_right.color = black

            if v_left != null:

            if v_left.color = white;

                    Q.enqueue(v_left)

                    v_left.color = black.
```

Q2

```
Sweep():

    for all heap h:

        if occupied(h)

        if obj(h).color = white

                    reclaim(h)
```

Q3

The asymptotic upper bound for the runtime of algorithm is $O(m+n)$. Due to the Sweep() function, despite every time the function has to iterate through the stack, size of m, only once in the "for all stack entries e", has to iterate through the entire heap, size of n, as well since the vertices are not reviewed. We must also visit the queued stack vertices. This is $O(2m+2n)$, which evaluates to $O(m+n)$.

Q4

The tight asymptotic lower bound is $\Omega(m+n)$. The Mark() function runs in the entire stack (m) at least once by using "for all stack entries e" with no break, and the if statement means that each iterations adds to the runtime.

The sweep() function runs in entire heap (n) at least once by using "for all heap h" without break, and the if statement means that each iteration adds to the runtime.

It is $\Omega(m+n)$ since there is an existing similarity of the upper and lower bound.

Q5

Each loop runs independently. Let $f(m,n) = x + y + z$

where

X = the worst case of "for all stack entries e" in Mark() function.

Y = the worst case of while loop of Mark() function

Z = the worst case of " for all heap h" in Sweep() function.

$X = 4m$ since the loop has 4 lines

$Y = 9(m+n)$ Since we have 8 possible lines and an initial line.

$Z = 3n$ Since every heap is occupied so the if parts are passed always.

we have to calculate: // $g(m,n) = m+n$

$f(m,n) = 4m + 9(m+n) + 3n = 4m + 9m + 9n + 3n = 13m + 12n$

* Let $C = 13$, $M = 0$ and $N = 0$

* $f(m,n) \leq g(m,n)$

* $13m + 12n \leq 13(m+n) \rightarrow 13m + 12n \leq 13m + 13n$

* for every $m \geq 0$, $13m \leq 13m$

* for every $n \geq 0$, $12n \leq 13n$

* Thus, $10m + 9n \leq 10m + 10n = 10(m+n)$

$$10m + 9n \leq C(m+n)$$

Q6

Part a

For execution, the program P runs P. (O(P)).

The memory will be filled up during a high number of execution phases. The mark-sweep recycling phase will be executed at that time. The recycling phase would normally be O(n+m) but P works with a set of 10 stacks, n is set to the integer 10. O(n+10) is the mark-sweep which is O(n). The phase of O(n) is carried out and reduces the occupied memory to n/2. The memory must be filled up again before O(n) operation is executed again. In the worst case scenario, the occupied memory is going to be sized exactly n/2. The total running time =

"2p/n x O(n) = 2p x O(1) = O(2p) = O(p)"

Part B

The overall runtime for memory recycling has time complexity O(p). for each of the p measures, the expensed time is O(p)/p. It would be g(p) = p as the limiting function in O(p). The division gives us 1 so it is O(1).

Q7

we know that $f(m,n) = 13m + 12n$ and in the worst, $f(m,n) \leq Cg(m,n)$.

In this case, $m = 10$ because of the stack in P.

The recycling phase time complexity in $f(m,n) =$

$13 \times 10 + 12n = 130 + 12n = f(n)$. In question 6, the cost of $f(n)$ is $2p/n$ times. So we have to calculate

$2p/n \times (130 + 12n) = 260p/n + 24p$

Let C be 30, N = 260, P = 0

$h(n,p) = O(g(n,p))$ iff $h(n,p) \leq Cg(n,p)$ we need $h(n,p) \leq Cg(n,p)$

$260p/n + 24p \leq 24p \Rightarrow 260p/n + 24p \leq 30p$

$260/n \leq 1$ for every $n \geq 260$

$p(260/n) + 24p \leq p + 24p \leq 30p$

$25p \leq 30p$ for all non-negative values

if we think from $O(1)$

$25 \leq 30$ is always true.

Q 8

The occupied memory in the derivation is now reduced to a size rn, leaving $(1-r)n$ free heap in the worst case. In addition, there must be at least $(1-r)n$ program execution steps to fill the memory and require the recycling phase.

It is $P/(1-r)n$ times is run by the phase.

Total runtime $= P/(1-r)n \times O(n) = P(1-r) \times O(1) = O(P/(1-r)$ where $0 < r \leq 1$.

The amortized time:

$$P/P(1-r) \times O(1) = O(1/(1-r)) \text{ where } 0 < r \leq 1.$$

The recycling phase with time cost, $f(n) = 13 \times 10 + 12n = 130 + 12n$

Total cost $= P/(1-r)n \times f(n)$

$$= P/(1-r)n \times 130 + 12n = 130P/(1-r)n + 12P/(1-r)$$

Total runtime $h(n,r,P) = 1/(1-r) + P/(1-r) = (1+P)/(1-r)$

$$h(n,r,P) = O(P/(1-r))$$

For the single operation cost $s(n,r,P)$

$$s(n,r,P) = h(n,r,P)/P = 130P/(1-r)nP + 12P(1-r)P$$

$= 130/(1-r)n + 12(1-r) = s(n,r)$, which equals to $12(1-r)$.

So $s(n,r) = O(1/(1-r))$ where n is ∞.

Q10

In my opinion, usage of Circular queue would be much better since data is not actually removed from the queue. Head Pointer is incremented by one position when dequeue is executed. It is also reinitialize both head and tail Pointers every time they reach the end of the queue.

PART B

The knapsack problem

Q1

Greedy (S; $s_1, \ldots, s_n$; $v_1, \ldots v_n$)

Initialize array Volume (si values)

Initialize array Value (vi values)

Initialize array ratios of length n, fills 0 for i in cost

Initialize array cost of length n

```
for i in cost
    cost[i] <- value[i]/volume[i]
sort.descending(cost)
maxVolume <- 0
for i <1 to n
    if  max Volume + volume <= S
        ratios[i] <- 1
        maxVolume <- maxVolume + Volume[i]
    else
        ratios[i] <- (S - maxVolume) / Volume[i]
        maxVolume <- S
            break

return ratios
```

Q2

For the runtime of the greedy algorithm, the asymptotic upper bound is $O(n \log n)$ since the algorithm wants to sort through the costs and can use the merge sort algorithm which provides $O(n \log n)$. The asymptotic upper bound on the runtime is not affected because the remaining code runs in sorted order.

Q 3

The optimal solution consists of a number of fractions.

As compared to our r, we need a list and lets call it ℓ.

For the optimal algorithm, only 1 item must be present so our base case is $n=1$.

for $n=k+1$ items. where assuming $n=k$

$$V^{k+1} = V^k + \ell_{k+1} \times V_{k+1} - \ell_x \times V_x$$

$$= V^k + S' \times V_{k+1}/S_{k+1} - S' \times V_x/\ell_x$$

$$= V^k + S' (V_{k+1}/S_{k+1} - V_x/S_x)$$

where

$\ell_{k+1} = S'/S_{k+1}$

S' = fractional objects used to be "Volume of objects"

* $V^k + S'(V_{k+1}/S_{k+1} - V_x/S_x)$ is the maximized value

* For any 2 items x and k+1 $\quad 0 \le S' \le S_{k+1}$, $0 \le S' \le S_x$

The solution will be an empty set when there are no items since the statement is true for $n=0$.

If it is true for $n=k$, it is also true for $n=k+1$.

If it is true for $n=0$, it is also true for all n.

Q4

for "highest value per unit volume":

$S = 10$; Volume $= \{6, 9\}$; Values $= \{8, 9\}$ for volume s and value v

Only one item can be chosen. 6 will be the first item chosen since it has unit value of $1,\bar{3}$ while the second item is 1

Thus, the total value will be 8.

for the plain value:

$S = 5$; Volume $= \{4, 2, 3\}$; value $= \{5, 4, 4\}$

This algorithm will select the second item but would not be able to select any more.

* In this case, the "highest value per unit volume" algorithm will yield the optimal solution.

Q5

We need a matrix $(n+1) \times (S+1)$ for value.

Dynamic (S; $s_1, \ldots s_n$; $v_1, \ldots v_n$)

Initialize array Volume (s_i)

Initialize array Value (v_i)

```
for i <- 0 to n
  for j <- 0 to S
    if (i=0) or (j=0)
      matrix[i][i] <- 0
    else if volume(i) <= j
      matrix[i][j] <- max((value[i] + matrix[i-1][j-volume[i]]),
                          (matrix[i-1][j]))
    else
      matrix[i][i] <- matrix[i-1][i]

maxValue <- matrix[n][S]
```

Run time of the algorithm = $O(nS)$

* $O(nS)$ is not polynomial time since S is exponential in Log S and it could dominate the size of input
* If S is small, $O(nS)$ can be useful. (pretty good)