#2 Assignment - CMPT 405

Luiz Fernando Peres de Oliveira - 301288301 - lperesde@sfu.ca

October 5, 2018

#1

First, we draw the table with cost c of multiplying two matrices in the dimensions $\{1 \times 1, 1 \times d, d \times 1, d \times d\}$

m_1	m_2	m_{res}	cost
1×1	1×1	1×1	1
1×1	$1 \times d$	$1 \times d$	d
$1 \times d$	$d \times 1$	1×1	d
$1 \times d$	$d \times d$	$1 \times d$	d^2
$d \times 1$	1×1	$d \times 1$	d
$d \times 1$	$1 \times d$	$d \times d$	d^2
$d \times d$	$d \times 1$	$d \times 1$	d^2
$d \times d$	$d \times d$	$d \times d$	d^3

Intuition:

The algorithm should always choose the lowest possible cost $c \in \{1, d, d^2, d^3\}$ because according to the table above, for any two given fixed matrices m_1, m_2 in a formula φ, where φ is the formula multiplying n matrices with dimensions $\{1 \times 1, 1 \times d, d \times 1, d \times d\}$, if it chooses the lowest cost operation, it will also generate a matrix m_{res} that, by multiplying another (possible) matrix m_3, will have cost $c' \geq c$. If $c' = c$, we keep the same cost, which is good, since we picked the lowest cost c at first. If $c' > c$, then, by the table above, $c' = cd$, which is also the lowest possible cost. As a mean of contradiction, if the algorithm chooses a matrix for which $c' > cd$, then the matrix would have non-compatible dimensions. Therefore, it is important that formula φ be a valid formula. Because we know that φ is always valid, we know that we can group (or parenthesize) the matrices by their dimensions and you will see that it can be done in linear time. The approach of the algorithm is the that with a stack S, every time it comes across a matrix $m_i = (1 \times x), x \in \{1, d\}, 1 \leq i \leq n$, it pushes i onto S, as the idea of the algorithm is to prioritize low costs. At the end, it subdivides φ in ascending order subformulas $\varphi' = \varphi'_1, ...\varphi'_m$ (recall that we push onto S the indexes of $(1 \times x)$ matrices). For each $\varphi'_j \in \varphi'$, we left-associate the matrices inside them. See the example and algorithm below:

$\varphi = (d\times 1)(1\times 1)(1\times d)(d\times 1)(1\times 1)$

$S = \boxed{1 \mid 2 \mid 4}$

$\varphi' = \boxed{(d\times 1) \mid (1\times 1) \mid (1\times d)(d\times 1) \mid (1\times 1)}$

$res = (d\times 1)\Big((1\times 1)\Big(\Big((1\times d)(d\times 1)\Big)(1\times 1)\Big)\Big)$

Algorithm:

Input: φ, n

```
Make stack $S$
for $i$ from 1 to $n$ do
    $(row, col) \leftarrow$ matrix$_i$
    if row $= 1$ then
        push $i$ onto $S$
    end if
end for
$\varphi' = \varphi$
while $S \neq \emptyset$ do
    $t \leftarrow$ pop $S$
    $v \leftarrow$ top $S$ // check top of S without popping
    diff $\leftarrow t - v$
    if diff $> 0$ then
        $\varphi' \leftarrow$ break $\varphi'$ and add parenthesis at pos $v$
        diff $\leftarrow$ diff $- 1$
        while diff $> 0$ do
            $\varphi' \leftarrow$ break $\varphi'$ and add parenthesis by left-associating extra matrices
            diff $\leftarrow$ diff $- 1$
        end while
    end if
end while
return $\varphi'$
```

#2

Definition: Let A be an array with size $n+1$. Initialize $A[0] = 1$, $A[1] = 1$. Define $A[i]$ as the number of possible orders in which you can multiply after an iteration i. At the end, the number total of possible orders will be $A[n]$.

Recurrence:

$$A[i] = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2 \\ \sum_{j=1}^{i} A[j-1] * A[i-j] & n \geq 2 \end{cases}$$

Algorithm:

Input: n

```
Make array $A$ of size $n+1$
$A[0] \leftarrow 1$
$A[1] \leftarrow 1$
for $i$ from 2 to $n$ do
    for $j$ from 1 to $i$ do
        $A[i] \leftarrow A[j-1] * A[i-j]$
    end for
end for
return $A[n]$
```

Closed form:

The closed form for the catalan number is (as per [2]):

$C_n = \frac{1}{n+1}\binom{2n}{n}$

#3

Definition: Let A be an array with size $n+1$ and s be a sequence of integers. Initialize $A[0] = -\infty$ and for $1 \leq i \leq n$, define $A[i]$ as the largest contiguous subsequence sum in s after an iteration i. At the end, the largest possible sum will then be the highest element in A.

Recurrence:

$$A[i] = \begin{cases} -\infty & \text{if } i = 0 \\ \max\{A[i-1]+s[i], s[i]\} & \text{otherwise} \end{cases}$$

Algorithm:

Input: s, n

```
Make array $A$ of size $n+1$
$A[0] \leftarrow -\infty$
insert *none* in the index 0 of $s$ // *make $|s| = |A|$ for the loop*
$best_i \leftarrow 0$
for $i$ from 1 to $n$ do
    $A[i] \leftarrow \max\{A[i-1]+s[i], s[i]\}$
    if $A[i] > A[i-1]$ then
        $best_i \leftarrow i$
    end if
end for
$s' \leftarrow \emptyset$ // *find best subsequence index set $s'$*
while $A[best_i] = A[best_i - 1] + s[best_i]$ do
    $s' \leftarrow s' \cup \{best_i\}$
    $best_i \leftarrow best_i - 1$
end while
$s' \leftarrow s' \cup best_i$ // *add the lowest index of subsequence*
return $s'$
```

Demonstration:

$s =$

none	-2	11	-4	13	-5	-2

$A =$

$-\infty$	-2	11	7	**20**	15	13

$s' = \{2, 3, 4\}$

At the end, as s' demonstrates, we will have the range 2..4 comprising the largest possible sum of a contiguous subsequence in s.

Running time: The running time of the loops are $O(n)$. All operations on A (inside the loops) are constant ($O(1)$) and therefore the total running time of the algorithm is $O(n)$.

#4
The idea of the problem is to take any node on the tree T, consider it as a root and then do a **postorder traversal** applying a recurrence similar do the one on question #3. Consider all vertices are labelled $v_1, ..., v_i, ..., v_n$, $1 \leq i \leq n$. Let A be an array that keeps the highest sum of a subtree with root on vertex v. As we are doing a postorder traversal, when we compute $A[v_p]$, v_p being a parent vertex, we will have the sum of the children already computed. We only include $A[v_c]$ on the sum of the children of v_p if $A[v_c] > 0$, and, if this holds true, v_c is also denominated v_c+, for all children v_c of v_p. The recurrence and algorithm would then be:

$$A[v_i] = \Big\{ \max \{ \textstyle\sum_{v_c^+ \in children(v_i)} A[v_c^+] + w(v_i), w(v_i) \}$$

Algorithm:
Input: T

```
  Make array $A$ of size $n$
  $best_i \leftarrow 0$ // $v_{best_i}$ will be the root of $H$
  for each $v_i \in T_{postorder}$ do
      $A[v_i] \leftarrow \max \{ \sum_{c^+ \in children(v_i)} A[v_c^+] + w(v_i), w(v_i) \}$
      if $A[v_i] > A[best_i]$ then
          $best_i \leftarrow i$
      end if
  end for
  $H \leftarrow$ Recursively do ($v_i$, all $v_c^+ \in$ children($v_i$))} starting on root $A[v_{best_i}]$
  return $H, A[v_{best_i}]$
```

#5
Intuition:
The intuition here is that if there exists any opportunities to start with an initial

capital of 1 unit of i and exchange it until we get more than 1, then there is a cycle among some edges in the graph G such that we end up with more units than we started. For some edge e_{ij}, we will then have a path that looks like: $e_{ij} \rightarrow e_{jk} \rightarrow e_{kl} \rightarrow e_{li}$ such that cost $c = e_{ij} * e_{jk} * e_{kl} * e_{li} > 1$. This path is a cycle because it starts and finishes in i and that is the reason why we can exchange it back to i. If there is no cycle, then there is no possible ways to get back to currency i. A good approach for this algorithm is to pre-process our edges e_{ij} (as per [1]) and then run Bellman-Ford algorithm to find whether or not there are any negative cycles. The pre-processing starts by using logarithms (it holds true for the problem because the logarithm function is a monotonic increasing function) to convert the multiplication into a summation and then setting the new values so we can run Bellman-Ford:

$$\prod_{1 \leq i < j \leq k} e_{ij} \qquad \text{to} \qquad \sum_{1 \leq i < j \leq k} ln(e_{ij}) \tag{1}$$

Because we want to find negative cycles, we will take equation (1) and invert the sign, because the log is negative for the range $(0, 1)$ and we want the opposite: we want to find negative cycles if the total amount is greater than 1. So we have :

$$\sum_{1 \leq i < j \leq k} -ln(e_{ij}) \tag{2}$$

After the pre-processing, it is possible to run Bellman-Ford to find if there is a negative cycle. If so, we know that there are opportunities for currency i to be exchanged and get a value greater than the initial one. We repeat that for all currencies. See below:

Algorithm:

```
Input: $G$
  $s \leftarrow \emptyset$
  for each node $i \in G$ do
     nodes $\leftarrow$ all relaxed nodes on $n_{th}$ iterarion of Bellman-Ford (source on $i$)
     cycle$^-$ $\leftarrow$ false
     while nodes $\neq \emptyset$ and not cycle$^-$ do
        $j \leftarrow$ get any element from nodes
        nodes $\leftarrow$ nodes$-\{j\}$
        $T' \leftarrow$ run BFS with source $j$
        if $i$ is reachable from $j$ (using $T'$) then
           cycle$^-$ $\leftarrow$ true
        end if
     end while
     if cycle$^-$ then
        $s \leftarrow s \cup i$
     end if
  end for
  return $s$
```

References

[1] - *Coursera: Algorithms on Graphs (https://www.coursera.org/lecture/algorithms-on-graphs/infinite-arbitrage-MrQ2H)*
[2] - *Catalan number - https://en.wikipedia.org/wiki/Catalan_number*