#1 Assignment - CMPT 405

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

September 21, 2018

#1 - Let C be the array containing all the possible coins {*1, 5, 10, 25, 100, 200* }. Let V be the total change value.

Algorithm:
Input: *C, V*

```
d ← sort C such that $d_1 \geq d_2 \geq ... \geq d_n$
$res \leftarrow \emptyset$; $i \leftarrow 1$
while $V > 0$ do
    if $V \geq d_i$ then
        $n_{coins} \leftarrow \left\lfloor \frac{V}{d_i} \right\rfloor$
        $V \leftarrow V - (n_{coins} * d_i)$
        $res \leftarrow res \cup \{(d_i, n_{coins})\}$
    end if
    $i \leftarrow i + 1$
end while
return $res$
```

Intuition: For all $d_i, d_j \in C$, $1 \leq i < j \leq n$, $d_i \geq 2 * d_j$, meaning that if I the algorithm chooses any d_j over any d_i, it will have to pick at least 2 times more coins for some value V that satisfies both d_i and d_j.

Proof. The only way to give more coins than the smallest possible number of coins for any change would be in a case where the algorithm chooses d_j over a d_i (see *Intuition*). Now, imagine that the algorithm chooses d_j over d_i, then we have two cases:
- Case 1. $d_i > V$:

If so, we are done because there are no possible ways of choosing d_i for value V.
- Case 2. $d_i \leq V$:

If that was the case (as a mean of contradiction), we would have an optimal set OPT such that $OPT_{i-1} \cup d_j \subseteq OPT$, which is not the case, once the iteration i will happen before the iteration j, causing the algorithm to choose d_i over d_j (and never the opposite) for any value V that satisfies both d_i and d_j. □

#2 a)

Greedy approach to the fractional knapsack:
- n objects and a knapsack
- item i weighs $w_i > 0$ and has utility $u_i > 0$
- fill knapsack so as to **maximize** total utility/weight, not exceeding total capacity W

Algorithm approach:
- sort items in decreasing order of their utility-to-weight ratio u_i/w_i
- repeatedly add item with max ratio u_i/w_i. If not possible to add the whole object, add a fraction $\alpha \in (0, 1)$ of it, if possible.

Proof. Let $K_{opt} \subseteq \{i_1, i_2, i_3, ..., i_n\}$ be the optimal set of items in a knapsack and let K_j be the chosen items after an iteration j, $0 \leq j \leq n$. Let K_j be considered "promising" if $K_j \subseteq K_{opt}$.

Base case: K_0: K_0 is promising since the total number of chosen objects, in this case *none*, does not exceed total capacity W. Thus, there exists some optimal K_{opt} such that $K_0 \subseteq K_{opt} \subseteq K_0 \cup \{i_1, i_2, ..., i_n\}$.

Induction step: Assume K_{j-1} is promising for stage $j-1$, meaning that $K_{j-1} \subseteq K_{opt} \subseteq K_{j-1} \cup \{i_j, i_{j+1}, ..., i_n\}$. We want to show K_j. On a stage j we have two cases:

Case 1. i_j is rejected. Then $K_{j-1} \cup \{i_j\}$ or $K_{j-1} \cup \{i_j * \alpha\}$ (any fraction $\alpha \in (0, 1)$ of i_j) exceed the capacity W; thus, $K_{j-1} = K_j$. Since $K_{j-1} \subseteq K_{opt}$ and K_{opt} does not exceed the total capacity W, $i_j \notin K_{opt}$. So $K_j \subseteq K_{opt} \subseteq K_j \cup \{i_{j+1}, i_{j+2}, ..., i_n\}$.

Case 2. i_j or $i_j * \alpha$ is added to K_{j-1}. Let item i_{chosen} be i_j or $i_j * \alpha$ (whichever was added to K_{j-1}). Then $K_{j-1} \cup \{i_{chosen}\}$ does not exceed the total capacity W and we have $K_{j-1} \cup \{i_{chosen}\} = K_j$.

Case 2.1. $i_{chosen} \in K_{opt}$. Then we have $K_j \subseteq K_{opt} \subseteq K_j \cup \{i_{j+1}, i_{j+2}, ..., i_n\}$.

Case 2.2. $i_{chosen} \notin K_{opt}$. We show that there is another maximum set of utility-to-weight items K'_{opt} that witnesses the fact that K_j is promising. For example, consider an item i_{chosen} added to K_{opt}. This will exceed the capacity W and the knapsack will contain at least one item of $\{i_{j+1}, i_{j+2}, ..., i_n\}$.

Proof of claim (Case 2.2): K_{opt} contains all elements of K_{j-1} and can be obtained from K_{j-1} by adding some items from the set $\{i_j, i_{j+1}, ..., i_n\}$. Adding i_j does not exceed capacity W, so the excess in $K_{opt} \cup \{i_{chosen}\}$ must contain some elements other items in $\{i_{j+1}, i_{j+2}, ..., i_n\}$.

□

#2 b)

item	utility	weight
1	2	1
2	1000	1000

$W = 1000$

$K_{opt} = \{i_2\}$(utility $= 1000$), $K_{greedy} = \{i_1\}$(utility $= 2$)

#3
The idea here is to use the greedy approach to check ahead and count the number of tiles a_{xy}, $1 \leq i < x \leq k$, $1 \leq y \leq n_x$ adjacent to a tile a_{ij}. Basically, if a tile a_{ij} touches the bounds of a tile a_{xy}, we say they are adjacent. The MAX number of adjacent tiles is then then minimum required number of colors for our solution. For example, if we have the tiles $a_{11} = 0.6$, $a_{12} = 0.4$ and $a_{21} = 0.35$, $a_{22} = 0.35$, $a_{23} = 0.3$, then a_{11} is adjacent to a_{21} and a_{22} and a_{12} is adjacent to a_{22} and a_{23}. The minimum number of colours required would then be 4 in this case.

Algorithm:

```
Input: wall, k
  for $i$ from 1 to $k$ do
    for $j$ from 1 to $n_i$ do
      $wall_i \leftarrow$ sort tiles such that $a_{ij} \geq a_{ij+1} \geq ... \geq a_{in_i}$
    end for
  end for
  $C \leftarrow 0$
  for $i$ from 1 to $k-1$ do
    $c \leftarrow 0$
    for $j$ from 1 to $n_i$ do
      $x \leftarrow i+1$
      for $y$ from 1 to $n_x$ do
        if $a_{ij}$ adjacent to $a_{xy}$ then
          $c \leftarrow c+1$
        else
          break  //not adjacent to $a_{xy+1}$ through $a_{xn_x}$ as well
        end if
      end for
    end for
    $C \leftarrow MAX(C, c)$
  end for
  return $C$
```

Counterexample:

Suppose:
col 1 : $a_{11} = 0.6$, $a_{12} = 0.4$
col 2 : $a_{21} = 0.4$, $a_{22} = 0.4$, $a_{23} = 0.2$

Our Greedy algorithm selects a_{11} as being adjacent to a_{21} and a_{22}. After that, it selects a_{12} as being adjacent to a_{22} and a_{23}, returning 4 as the minimal number of colours. The algorithm could not rearrange a_{22} and a_{23} so that a_{11} would be adjacent to a_{21} and a_{23}; and a_{12} would be adjacent to a_{22}, returning 3, the optimal number of colors for this problem. Thus, the Greedy approach given is not optimal. See below:

$W_{greedy} = \{(a_{11}, a_{21}), (a_{11}, a_{22}), (a_{12}, a_{22}), (a_{12}, a_{23})\}$. **min:** 4 colors
$W_{opt} = \{(a_{11}, a_{21}), (a_{11}, a_{23}), (a_{12}, a_{22})\}$. **min:** 3 colors

#4
Definition: For $0 \leq i \leq m$, $0 \leq j \leq n$, define $M[i, j]$ as the optimal number of paths in the Cartesian plane from $(0, 0)$ to (i, j) that uses the combined number of steps of type U(up, $M[i-1, j]$), R(right, $M[i, j-1]$) and D(diagonal, $M[i-1, j-1]$). The optimal number of paths from $(0, 0)$ to (m, n) is then $M[m, n]$.

Recurrence:

$$M[i,j] = \begin{cases} 1 & \text{if } i = 0 \text{ or } j = 0 \\ M[i-1,j] + M[i,j-1] + M[i-1,j-1] & \text{otherwise} \end{cases}$$

Algorithm:
Input: m, n

```
  Make matrix $M$ with dimensions $m \times n$
  for $i$ from 0 to $m$ do
      $M[i, 0] \leftarrow 1$
  end for
  for $j$ from 0 to $n$ do
      $M[0, j] \leftarrow 1$
  end for
  for $i$ from 1 to $m$ do
      for $j$ from 1 to $n$ do
          $M[i, j] \leftarrow M[i-1, j] + M[i, j-1] + M[i-1, j-1]$
      end for
  end for
  return $M[m, n]$
```

Running time: All operations on M (inside the loops) are constant and therefore the running time of the algorithm is $O(mn)$

Expression:
For d steps of type D(diagonal), there must have $m-d$ steps of type U and $n-d$ steps of type R, in order to reach (m, n). It can then be represented by:

$$M(m,n) = \sum_{d=0}^{min(m,n)} \binom{m+n-d}{m} \binom{m}{d}$$

#5

The idea of the problem is to use the Greedy approach on the **set cover** problem to get maximal rectangles, meaning that if I have a position $p_{v_i} = 0$, where v is a leaf of T and i is an index of u's binary string, and I also have $p_{u_i} = 0$, for a leaf u sibling of v, then I can maximize a rectangle for a w parent of v and u, such that $p_{w_i} = 0$ (the same holds for sequences of 0's). We start the algorithm by setting up the sets for the set cover problem such that the longest number of consecutive zeroes and ones stay make a group within a string (e.g S_v for $v = 1011000$ would be $S_v = \{1, 0, 11, 000\}$. We then traverse T such that we get maximal rectangles, see the algorithm below:

Algorithm:

```
Input: $T$
  for each $v \in T, children(v) = \emptyset$ do
      $S_v \leftarrow$ group 0's and 1's $\in v$ as a list, such that no 0 in position $i$ is followed
  by another (group of) 0 in position $i+1$ and no 1 in position $i$ is followed by
  another (group of) 1 in position $i+1$
  end for
  for each $w \in T_{postorder}, children(w) = \{u, v\}$ do
      $S_w \leftarrow$ matching-indexes 0's in both $S_v$ and $S_u$
      $S_v \leftarrow S_v - S_w$
      $S_u \leftarrow S_u - S_w$
  end for
  $Res \leftarrow \emptyset$
  for each $v \in T$ do
      if $S_v \neq \emptyset$ then
          $rectangle \leftarrow \forall(v, (i, j)) \in S_v$ where $p_{v_i j} = \{0\}^{|v|}$
          $Res \leftarrow Res \cup rectangle$
      end if
  end for
  return $Res$
```

Running time:

Let b be the number of bits in a binary string on the leaves of T. The algorithm then takes $O(nlogn)$ to traverse T and $O(b)$ for each set-related operation inside the loops. Therefore, the total running time of the algorithm is $O(bnlogn)$.