TP2 Remise: Analyse Asymptotique

Jackson Li 1785104; Daniel Dermont 2043595

November 2020

Our sample size is 3 times the number of blocks, as each block generated has three orientations. For simplicity, we simply refer to this sample size as n.

1 Sorting

```
def sortBySADecreasing(blocks):
    inc=np.argsort(blocks[:,4])#indexes of rows sorted by last column in increasing order
    dec = inc[::-1]#now in decreasing order
    sorted_blocks=blocks[dec]#reorganise list according to "dec" indexes
    return sorted_blocks

def sortBySAIncreasing(blocks):
    inc=np.argsort(blocks[:,4])#indexes of rows sorted according to last column in increasing orc
    sorted_blocks=blocks[inc]#reorganise list according to "inc" indexes
    return sorted_blocks
```

The functions **sortBySADecreasing** and **sortBySAIncreasing** are utility function used by the algorithms to sort the blocks according to their surface area. We will start by analysing the time complexity of the sortBySADecreasing. The first line of sortBySADecreasing indicates the use of a numpy method called argsort which uses quicksort to return a list of indices sorted in increasing order according to the surface area. Quicksort in numpy has a time complexity of $\Theta(nlogn)$, given it uses a variant called introsort. The function also reverses a python list, which has $\Theta(n)$ since it has to travel through all indices. The last line the function is also $\Theta(n)$ since it is populating a new list of size n. The overall time complexity of the **sortBySADecreasing** function is therefore $\Theta(nlogn)$ as the sorting dominates the growth rate of the other operations. Time complexity wise, the function **sortBySAIncreasing** only differs from the previous function by having no reverse operation, and therefore also has a overall time complexity of $\Theta(nlogn)$.

$$sort \in \Theta(nlogn) \tag{1}$$

2 Greedy Algorithm

```
def greedy(blocks):#Select block with largest top surface area to place on tower
    sorted_block = sortBySADecreasing(blocks)
    n = sorted_block.shape[0]
    tower = [[sorted_block[0, 0], sorted_block[0, 1], sorted_block[0, 2]]]
    for index in range(1, n):
        if canPlaceBlock(sorted_block[index], tower[-1]):#check if next block can be placed
            tower = tower + [[sorted_block[index, 0], sorted_block[index, 1],
                    sorted_block[index, 2]]]
    return sum(row[0] for row in tower), tower

def canPlaceBlock(blockK, blockI):#try to place k on i
    return blockK[1] < blockI[1] and blockK[2] < blockI[2]
```

The **greedy** algorithm starts off by using the sortByIncreasing function which has time complexity $\Theta(nlogn)$. It then obtains the length of the sorted blocks list using numpy's shape function, which is $\Theta(n)$ as it travels through each element to count the size. The third line refers to a simple assignment which has $\Theta(1)$. The algorithm then uses a loop that has will iterate exactly n-1 times. Inside the loop are simple operations with $\Theta(1)$. The loop as a whole therefore has time complexity of $\Theta(n)$. Finally, at the return line, the sum of all block heights in the tower is calculated with time complexity of $\Theta(n)$. Overall, the **greedy** algorithm has time complexity of $\Theta(nlogn)$ as the rate of growth of the sorting function overshadows those of all other operations.

$$greedy \in \Theta(nlogn) \tag{2}$$

3 Dynamic Programming

```
def dyn_prog(blocks):
    n = blocks.shape[0]#n in this case is actually 3N, where N is the number of unique blocks
    sorted_blocks = sortBySAIncreasing(blocks)#sorted in increasing surface area
    h_tower = np.copy(sorted_blocks[:, 0]) #tableau de calculs for dynamic programming
    pointers = [None] * n #refers to index of the block on top of the sub-tower
    isTerminated = [False] * n #terminated blocks have reached their optimal subsolution
    while False in isTerminated:# at most n iterations
        for k in range(n): #k is the index of the current block to be placed on top
            if not isTerminated[k]:
                canStillBePlaced = False
                for i in range(k+1,n): #i is the index of the top block of subtower
```

```
                    if not isTerminated[i] and canPlaceBlock(sorted_blocks[k], sorted_blocks[i]):
                        if worthPlacingBlock(sorted_blocks[k, 0], h_tower[i], h_tower[k]):
                            h_tower[k] = sorted_blocks[k, 0] + h_tower[i]
                            pointers[k] = i
                        canStillBePlaced = True
                if not canStillBePlaced:
                    isTerminated[k] = True
    index_max = np.argmax(h_tower)
    return h_tower[index_max], getBlocksInTower(sorted_blocks[:, 0:3], pointers, index_max)

def canPlaceBlock(blockK, blockI):
    return blockK[1] < blockI[1] and blockK[2] < blockI[2]

def worthPlacingBlock(block_height, subtower_height, current_tower_height):
    return block_height + subtower_height > current_tower_height
```

Below are two helper functions that the main dynamic programming algorithm uses. Both are of $\Theta(1)$

```
def getBlocksInTower(block_Dims, pointers, top_block_index):
    current_Block_Dims = block_Dims[top_block_index]
    if pointers[top_block_index] == None:
        return [[current_Block_Dims[0], current_Block_Dims[1], current_Block_Dims[2]]]
    return getBlocksInTower(block_Dims, pointers, pointers[top_block_index]) +
                [[current_Block_Dims[0], current_Block_Dims[1], current_Block_Dims[2]]]
```

The **dynamic programming** algorithm starts off by obtaining the number of elements in the "blocks" array, which has a time complexity of $\Theta(n)$. It sorts the blocks array with $\Theta(nlogn)$ by calling sortBySAIncreasing. A column of the "blocks" array is then copied to initialize the towers which has time complexity of $\Theta(n)$ as it has to travel through each element of the column. A couple of vectors of size n are initialized with $\Theta(n)$. An outermost loop is introduced. This outermost loop will iterate at most n-1 times and at least once. It will loop at most n-1 times because a minimum of one block is placed on top of a sub-tower every iteration and after n-1 iterations there are no more blocks to place. If no block is placed during an iteration, it means that all elements in isTerminated are True and the outermost loop will exit. The outermost loop will iterate at least once because it needs to check if any block can be placed. In a case where none of the blocks can be placed, say in the case where all blocks have the same width, the outermost loop will exit after one iteration. Inside the outermost loop are two nested inner loops. The combination of the two inner loops will iterate at most $n^2/2 - n/2$ times. This number of iteration happens for the very first iteration of the outermost loop since at that point the isTerminated vector just got initialised. All operations inside the two loops have time complexities of $\Theta(1)$. Therefore, in worst case scenario, the outermost loop will iterate n-1 times and the combined inner loops $n^2/2 - n/2$ times. In the best case scenario, the outermost loop iterates only once and the inner loops $n^2/2 - n/2$ times. As such the combined time complexity of all the loops combined is $O(n^3)$ and $\Omega(n^2)$. Once outside the loops, Numpy's argmax is called on a vector of n length which results in a time complexity of $O(n)$. Finally, **getBlocksInTower** is called. The function getBlocksInTower has $O(n)$, since the tallest possible tower contains n elements, and $\Omega(1)$, for a tower of one block. Therefore, the overall time complexity of the **dynamic programming** algorithm is $O(n^3)$ and $\Omega(n^2)$ due to the loops.

$$dynamic\_programming \in O(n^3) \tag{3}$$

$$dynamic\_programming \in \Omega(n^2) \tag{4}$$

4 Tabu Search

We first note that no ordering is required in our candidate list to perform the Tabu search.

```
def mustReplace(c,s):
        fitsAbove = (c[1] < s[1]) and (c[2] < s[2]) #candidate fits above if the 2
                            #surface dimensions are strictly less than the solution block
        fitsBelow = (c[1] > s[1]) and (c[2] > s[2]) #candidate fits below if the 2
                            #surface dimension are strictly greater than the solution block
        # replace if the candidate does not fit above or below the sol'n block,
        #  or the sol'n block shares the candidate ID
        replace =  not (fitsAbove or fitsBelow) # or (c[3] == s[3]) [to compare IDs]

        return replace, fitsAbove, fitsBelow
```

The **mustReplace** function is simply a comparison and therefore has a time complexity of order-1.

```
def tabu_score(c,S):
        # c is a member of Candidates (C) and S is the proposed solution
        # score is the tabu score of this candidate's inclusion in the stack
        # score = -(height of blocks replaced) + (height of c)
        score = c[0] # height is the first entry of each candidate
        # Different paradigm if S is 1D (S only consists of one block)
        if(S.ndim > 1):
                for i in range(S.shape[0]):
                        replace, _, _ = mustReplace(c,S[i])
                        if(replace):
                                score -= S[i,0]
        else:
                replace, _, _ = mustReplace(c,S)
                if(replace):
                        score -= S[0]
        return score
```

The **tabu_score** function has its best case time complexity order-1 when the input S is 1D or empty. If all n blocks of our sample fit into a potential solution, we have a worst case time complexity of order-n.

```
def scoring(C,S):

        hi_score = -1000000
        hiC_index = 0
        for i in range(C.shape[0]):
                score = tabu_score(C[i],S)
                if (score > hi_score):
                        hi_score = score
                        hiC_index = i

        return hi_score, hiC_index
```

The **scoring** function loops through all values in the Candidate stack C and finds the highest **tabu_score** when compared to the current solution S. If C were empty on a given iteration, we would have a best case time complexity of order-1. If the candidate stack C included the entire sample size, we would have our worst case time complexity of order-n, since we loop an order-1 operation n times, as S would necessarily be empty.

```
def updateS(c,S):
# Add best Candidate to solution
        tabBlocks = np.array([[]])
        if(S.ndim == 1): # If S is 1D
                replace, fitsAbove, fitsBelow = mustReplace(c,S)
                if(replace):
                        newS = c
                        tabBlocks = S
                elif(fitsBelow):
                        newS = np.vstack((c,S))
                elif(fitsAbove):
                        newS = np.vstack((S,c))
        else:
                replaceIndices = [] # indices of parts of the solution to replace
                tabBlocks = np.zeros(6,int) # blocks bound for T
                for i in range(S.shape[0]):
                        replace, _, _ = mustReplace(c,S[i])
                        if(replace):
                                if (len(replaceIndices) == 0):
                                        replaceIndices = [i]
                                else:
                                        replaceIndices += [i]
                                tab = np.append(S[i],np.random.randint(8,12))
                                tabBlocks = np.vstack((tabBlocks,tab))

                # Delete bad blocks from S to form newS
                newS = np.delete(S,replaceIndices,0)
                # handle tabBlocks
                if(tabBlocks.ndim == 1): # if no blocks are replaced
                        tabBlocks = np.array([])
                else: # if not, discard the initialization value
                        tabBlocks = np.delete(tabBlocks, 0, 0)

                # now fit c to newS
                if(newS.ndim == 1): # If newS is 1D
                        _, fitsAbove, fitsBelow = mustReplace(c,newS)
                        if(fitsBelow):
                                newS = np.vstack((c,S))
                        elif(fitsAbove):
                                newS = np.vstack((S,c))
                else:
                        # if c fits below the base of the tower, stack on it
                        _, _, fitsBelow = mustReplace(c,newS[0])
                        if(fitsBelow):
                                newS = np.vstack((c,newS))
                        else:
                                inserted = False
                                for j in range(newS.shape[0]):
                                        _, _, fitsBelow = mustReplace(c,newS[j])
                                        if(fitsBelow and (not inserted)):
                                                newS = np.insert(newS, (j), c, 0)
                                                inserted = True
                                # finally, if c block fits on top of tower
                                _, fitsAbove, _ = mustReplace(c,newS[j])
                                if(fitsAbove):
                                        newS = np.vstack((newS, c))

        return newS, tabBlocks
```

The **updateS** function has its best case scenario time complexity if the stack S is empty or is 1D. In this case, only simple assignment or stacking operations are performed once, making this an order 1 operation. In the worst-case scenraio, the

potential solution S includes all n samples, and the input c is necessarily empty. Even if c is empty, the **updateS** function starts with a loop through all entries of S with the contents of the loop being an order-1 operation. This loop in the worst case is therefore an order n operation. The variable newS would be the same stack as S, equally composed of n samples. Since newS is necessarily an $nx2$ matrix, the **updateS** function would again loop through all n samples, performing an order 1 operation, making the second loop also an order-n operation in the worst case. We can therefore say that the worst case time complexity is a order-$2n$ operation, which is an order-n operation.

```
def appendTabBlocks(T,tabBlocks):
        if(T.size == 0):
                appendedT = tabBlocks
        elif(tabBlocks.size == 0):
                appendedT = T
        else:
                appendedT = np.vstack((T,tabBlocks))
        return appendedT
```

In all cases, assuming the stacking operation is order-1, the **appendTabBlocks** function is order-1.

```
def updateT(T,tabBlocks, C):
        # the following function adds the new tabu blocks to T
        currentT = appendTabBlocks(T,tabBlocks)

        # there are three possible cases
        # if the currentT is empty, do nothing
        if(currentT.size == 0): # if there are no tabu blocks to add, pass
                newT = currentT
                newC = C
        # if currentT is 1D vector
        elif(currentT.ndim == 1):
                #decrement the tabu counter
                currentT[5] -= 1
                #if we hit zero on tabu counter
                # stack the tabu block back onto the candidates
                if(currentT[5] == 0):
                        newCandidate = np.delete(currentT,5)
                        newC = np.vstack((newCandidate, C))
                        newT = np.array([])
                # if we don't hit zero on tabu counter, pass
                else:
                        newT = currentT
                        newC = C
        # if currentT is 2D
        elif(currentT.ndim > 1):
                # we instantiate newT with this dummy value so that we can use np.vstack(())
                newT = np.zeros(6,int)
                # we want to stack any spent tabu blocks back under the Candidates
                newC = C
                for i in range(currentT.shape[0]):      # iterate over Tabu list
                        currentT[i,5] -= 1              # decrement tabu counter value
                        if(currentT[i,5] == 0):
                        # if counter zero stack the current tabu block on top of C
                                newCandidate = np.delete(currentT[i],5,0)
                                newC = np.vstack((newCandidate,C))
                        else:
                        # else, stack current block in newT
                                newT = np.vstack((newT,currentT[i]))
                # Two cases:
                # if newT is empty (apart from dummy value)
                if(newT.ndim == 1):
                        newT = np.array([])
                # else, delete dummy variable
                else:
                        newT = np.delete(newT,0,0)
        return newT, newC
```

The **updateT** function has its best case scenario when the Tabu list T is empty, which results in order-1 time complexity for **updateT**. If the Taboo list is full of n samples, the for loop that decrements the Tabu list becomes an order-n operation, making the **updateT** function an order-n operation.

```
def tabu_search(blocks):
        # Initial Candidates and Solution sets formed
        C = blocks
        i_0 = np.argmax(C[:,0])  # index of largest h

        prevS = np.zeros(5,int)
        prevS = C[i_0]
        bestS = prevS
        bestH = prevS[0]
        C = np.delete(C, i_0, 0)
        T = np.array([])
        convCount = 0
```

```
    ## MAIN LOOP ##
        while(convCount < 100):

                hi_score, hiC_index = scoring(C,prevS)

                newS, tabBlocks = updateS(C[hiC_index],prevS)

                C = np.delete(C,hiC_index,0)

                T, C = updateT(T, tabBlocks, C)


                # compare height of resulting tower to height of best Solution
                # if thisH <= bestH, we increment convCount

                newH = np.sum(newS, axis=0)[0]
                if(newH > bestH):
                        bestH = newH
                        bestS = newS
                        convCount = 0
                else:
                        convCount += 1
                prevS = newS
    ## END LOOP ##
        return bestH, bestS[:, 0:3]
```

The function **tabu_search** is the overall implementation of the Taboo search algorithm. The initial candidate stack C is composed of all entries of the sample, and therefore the **np.argmax** function is an order-n operation in any case. The main loop of the function executes so long as the solution has not converged for a number of iterations (here 100) we will denote k. The maximum tabu list counter value will be denoted l. In our best case scenario, the initial solution S is our best solution, converging after k iterations. The worst case time complexity is harder to explain
We first consider the case where the potential solution S at iteration i of the main loop is populated by $(n-1)$ blocks, and there is a single candidate c in the stack C, and an empty tabu list T. If this candidate does not form the best solution, we have attained the best solution and our solution converges in k iterations. If this solution maximizes the current solution by stacking in S or replacing another block or set of blocks in S, we will have attained the maximal solution. Since the candidate c was in the stack C, it was immediately available, and convergence takes $k+1$ iterations. If we now consider the case where c was in T, i would take at most $k+l$ iterations to converge. Now we consider the case that S at iteration i is populated by $(n-2)$ blocks, and there are two candidates among C and T, where $c1$ is a more optimal solution than $c2$. In the best case scenario here, the tower S is already maximal and converge is reached in k iterations. In the worst case, $c1$ was placed in T on iteration $i-1$ for l iterations, and $c2$ is in the candidate list C. It can be seen that the amount of iterations of the main loop until $c1$ is available in C and and has stacked to form the optimal solution is proportional to l. Using this pattern, we can say that for a sample size n, the worst case amount of iterations for the main loop of **tabu_search** is proportional to the sample size n. The contents of the loop include individual functions whose time complexity is described above, along with a sum of the height column of a potential solution newS whose time complexity is order-1 at best and order-n at worst. The contents of the loop will, in all cases, perform at least one elementary operation on every block included in the sample for every iteration of the loop, since $\Omega = S \cup T \cup C$, where Ω here denotes the sample space (not to be confused with big-Ω):

$$\Theta_{loopcontents} = \Theta(n) \tag{5}$$

Since in the worst case, the main loop iterates $n+k$ times, we have:

$$O_{loop} = ((const)n)O_{loopcontents} = O(n^2) \tag{6}$$

and in the best case, the loop iterates k times, we have the lower bound on time complexity for the loop:

$$\Omega_{loop} = (k)\Omega_{loopcontents} = \Omega(k) = \Omega(1) \tag{7}$$

in terms of sample size n.
Finally, since in all cases, we must do an initial search over the entire sample size in all cases.

$$O_{tabu\_search} = O(n) + O(n^2) = O(n^2) \tag{8}$$

$$\Omega_{tabu\_search} = \Omega(n) + \Omega(1) = \Omega(n) \tag{9}$$

$$tabu\_search \in O(n^2) \tag{10}$$

for cases that terminate

$$tabu\_search \in \Omega(n) \tag{11}$$