# CS 124, Problem Set 4

Jessica Wang

March 4, 2016

## Problem 1

### Algorithm

This algorithm will store 5 integer values which are intialized as follows: $maxSum = 0$, $startIndex = 0$, $endIndex = 0$, $currentSum = 0$ and $currentSumStart = 0$. $maxSum$ will hold the current maximum consecutive sum found within the array with $startIndex$ and $endIndex$ representing those consecutive numbers. $currentSum$ represents the current consecutive sum being started with $currentSumStart$ representing the starting index of that current consecutive sum. The function will iterate through the array $A$ and at each index $i$ it will calculate $currentSum + i$ can work out acording to the following three cases:
**Case 1** $currentSum + A[i] > currentSum$**:** This means adding the value $A[i]$ will increase the current consecutive sum, thus if the new $currentSum + A[i] > maxSum$ update $maxSum = currentSum + i$, $startIndex = currentSumStart$ and $endIndex = i$. Lastly no matter what, update $currentSum+ = A[i]$ before moving to the next index.
**Case 2** $0 < currentSum + A[i] < currentSum$**:** This means adding the value $A[i]$ will decrease the current consecutive sum, so only update $currentSum+ = A[i]$ before moving to the next index.
**Case 3** $0 > currentSum + A[i]$**:** Since adding $A[i]$ will make the consecutive sum negative, it is better to start the next consecutive sum at $A[i + 1]$ instead of adding the previous negative value. Therefore update $currentSumStart = i + 1$ and $currentSum = 0$ before moving to the next index.

### Recursive Equation

Let $o(i)$ represent the largest connsective sum of the subsets $A[s, i]$ for all $0 \leq s \leq i$. $o(i)$ represents the largest connsective sum possible out of all the subsets that end with $A[i]$. $o(i)$ can be found with the recursion below

$$o(i) = max(o(i + 1) + A[i], 0)$$

The maximum consecutive sum can be found by

$$\textit{Maximum consecutive sum of n numbers } = max(o(1), o(2), ...., o(n))$$

The base case for this recusion is the $o(1)$. If $A[1]$ is positive or 0, $o(1) = A[1]$. If $A[1]$ is negative then $o(1) = 0$. Therefore by induction our algorithm is correct.

### Time and Size Bounds

This algorithm only stores 5 constants so (assuming the space for the array is given seperately) the algorithm takes $O(1)$ space. The algorithm iterates through $A$, which has size $n$, once so the run time is $O(n)$.

# Problem 2

## Constructed special functions

Define the following functions, $optimal(j,k)$ is the minimal imbalance from of the subarray $A[j,n]$ with $k$ partitions. Let $weightDifference(a,b)$ be $|w(A[a,b]) - \sum_{l=1}^{n} A[l]/k+1|$, thus the weight difference between the subarray $A[a,b]$ to the optimal weight distribution. Let $ideal = \sum_{l=1}^{n} A[l]/k+1$.

## Algorithm

Construct a lookup $n$x$n$ matrix $weights$ and iterate through the upper triangle of this matrix such that $weights[i][j] = weightDifference(i,j)$. Construct another $n$x$k+1$ matrix $optimalImbalance$ (with columns indexing at one and rows indexing at 0) which will eventualy, in the upperleft corner, hold $optimalImbalance[i][j] = optimal(i,j)$ with $optimalImbalance[1][k] = optimal(1,k)$ the minimum imbalance of the entire array. Make an additional $n$ by $k+1$ matrix $divisorIndex$ (with columns indexing at one and rows indexing at 0) that mirrors $optimalImbalance$ and allows us to record where the partitions are placed.

$optimalImbalance$ will fill by iterating through $j$ from 0 to $k$ and for each $j$ iterate throught $i$ from 1 to $n-j$. Each $optimalImbalance[i][j]$ will be calculated by finding the $min(optimalImbalance[i+1][j-1], weights[i], [i+1], max(optimalImbalance[i+2][j-1], weights[i][i+2]), \ldots.., max(optimalImbalance[n-j+1][j-1], weights[i][n-j+1]))$. This finds the optimal placement of $j$ divisors in the subarray $A[i,n]$ by first determining the imbalance of putting the first divisor at index $i$ (making partition $A[i,i]$ and dividing the remaining $A[i+1,n]$ optimally by calculating $max(optimalImbalance[i+2][j-1], weights[i][i+2])$ which is equal to calculating$max(optimal(i+2, j-1), weightDifference(i, i+2))$. It will then determine the imbalance of putting the first divisor at index $i+1$ and iterate through all possible placements of the first divisor (up to $n-j+1$). The minimum of all these specific partition's imbalances will be the optimal mimimum imbalance for placing of $j$ divisors in the subarray $A[i,n]$. Therefore there must be some index $i+d$ where $i \leq d \leq n-j+1$ where $i+d$ is the optimal index to place the first partition. $i+d$ can be easily aquired since it will be the minimum $max(optimalImbalance[i+d][j-1], weights[i][i+d])$. Store $divisorIndex[i][j] = i+d$.

The minimum imbalance of placing $k$ divisors into $A$ can be found in $optimalImbalances[1][k] = optimal(i,k)$. To find the indicies of each divisor $d_1, d_2, ..., d_k$ use the $divisorIndex$. Iterate through $j$ from $k$ to 1, starting at $divisorIndex[1][k]$ the value stored there, say $d_1$, will indicate where to put the first divisor. To find the next divisor to to $divisorIndex[d_1][k-1]$, which will give you value $d_2$ to put the second divisor. Looking at $divisorIndex[d_2][k-2]$ will give you $d_3$, so on and so forth, allowing you to find all $d_1, d_2, ..., d_k$.

## Recursive Equation

Calculating the values of $optimalImbalances[i][j]$ is the same as calculating $optime(i,j)$. This can be done by the following recursive equation:

$$optimal(i,j) = min(max(weightDifference(i,p), optimal(p+1, j-1)))$$

$$\forall p \text{ such that } i \leq p \leq n-j+1$$

The base case for this algorithm would be $optimal(i,0)$ for all $1 \leq i \leq n$. $optimal(i,0)$ represents the optimal imbalance of the sequence $i...n$ with 0 dividors, thus as its own subset. Thus $optimal(i,0) =$

$weightDifference(i, n)$ which is easily calculated at the beginning. Therefore by induction this algorithm is correct.

### Generalization

Changing the definition of imbalance would change our algorithm to calculate $optimal(i, j)$ by the following recursion equation.

$$optimal(i, j) = min(weightDifference(i, p) + optimal(p + 1, j - 1))$$

$$\forall p \text{ such that } i \leq p \leq n - j + 1$$

The base case would again be $optimal(i, 0)$ for all $1 \leq i \leq n$ and as shown above these are easy to calculate. Therefore by induction this algorithm is correct.

### Time and Size Bounds

The $n$x$n$ lookup matrix $weightDifference$ is the biggest matrix needed to be stored so the space needed is $O(n^2)$. The algorithm iterates through element of $optimalImbalances$, which has size $n$x$k$+1, and at each element iterates through a possible $n$ different $x$ values for $max(optimalImbalance[i+x][j-1], weights[i][i+x])$. Therefore the run time is $O((k+1)n * n) = O((k+1)n^2)$.

## Problem 3

### Constructed special functions

Define the function $linePenlty(i, j) = (M - j + i + \sum_{k=i}^{j} \ell_k)^3$ if the words $i$ through $ij$ fit on a single line and $None$ otherwise. Thus $linePenalty(i, j)$ represents the penalty a line of words $i$ through $j$.

### Algorithm

Construct lookup matrix $penaltyMatrix$ of sixe $n$x$n$ by iterating through the upper triangle and intializing $penaltyMatrix[i][j] = linePenalty(i, j)$. Create an array $optimalPenalty$ of size $n$ (index at 1). This will eveually be evaluated so $optimalPenalty[i]$ will store the penalty number for creating the optimal paragraph of word $i$ through $n$. Create an additional array $lineBreaks$ of size $n$ (index at 1) that mirrors $optimalPenalty$ that will eventually be evaulated such that $lineBreaks[i]$ stores the number of words that are located on the first line of the optimal arrangement indicated by $optimalPenalty[i]$.

Evaluate $optimalPenalty$ by iterating from index $n$ to 1 (backwards through the array). This will reflect starting with the last word of the paragraph and adding the word in front of it one by one. To find the optimal arrangement of adding a word to the beginning of the text, first consider the overall penalty of placing the additional word on its own line and optimally placing the remaining words starting on the next line ($penaltyMatrix[i][i]+optimalPenalty[i+1]$). Next consider the overall penalty of placing the first two words of the new text on the first line and optimally placing the remaining words starting on the next line ($penaltyMatrix[i][i+1]+optimalPenalty[i+2]$). We can continue this until the first line cannot fit any additional words. Picking the minimum of these overal penalties will allow us to find the new optimal placement of the new text. In this way we find $optimalPenalty[i]$ by looping through $penaltyMatrix[i][i+j]+optimalPenalty[i+j+1]$ with

$j$ from 0 until $penaltyMatrix[i][i+j] == None$. Save this value $j$ such that $penaltyMatrix[i][i+j] + optimalPenalty[i+j+1]$ is the minimum (the optimal placement of words, with $j+1$ words on the first line) by updating $lineBreaks[i] = j+1$. By the time we iterate from $n$ to 1, the value at $optimalPenlty[1]$ will be the overall penalty for the entire text.

There is a special case, which is if the optimal arrangement from words $i$ to $n$ only takes one line. This is dealt with by considering first $penaltyMatrix[i][n]! = None$ (happens when words $i$ to $j$ fit on a single line) when arriving at index $i$ of $optimalPenalty$. If $penaltyMatrix[i][n]! = None$ then $optimalPenalty[i] = 0$ and $lineBreaks[i] = n$ because the most optimal format would be the words fitting into the last line.

With $optimalPenlty$ and $lineBreaks$ evaluated, to determine how to print the paragraph start by accessing the line break index $lb_1$ at $lineBreaks[1]$. The next line will begin at word $lb_1 + 1$ so the next line break index $lb_2$ will be stored at $lineBreaks[lb_1 + 1]$. Continue this format until $lb_i = n$ which indicates the $i^{th}$ line is the last line of the paragraph.

## Algorithm Code

The python code written to run the algorithm is attached in a seperate zip file. problem3.py is the python code itself. buffy.txt is the input, as taken from the course website. output40.txt is the optimal paragraph for $M = 40$ and output72.txt is the optimal paragraph for $M = 72$

## Results

The code was written in python and attached to the submission. The results were as follows:

**M=40, Found in output40.txt**

Buffy the Vampire Slayer fans are
sure to get their fix with the DVD
release of the show's first season.
The three-disc collection includes all
12 episodes as well as many extras.
There is a collection of interviews
by the show's creator Joss Whedon in
which he explains his inspiration for
the show as well as comments on the
various cast members. Much of the same
material is covered in more depth with
Whedon's commentary track for the show's
first two episodes that make up the
Buffy the Vampire Slayer pilot. The
most interesting points of Whedon's
commentary come from his explanation
of the learning curve he encountered
shifting from blockbuster films like Toy
Story to a much lower-budget television
series. The first disc also includes
a short interview with David Boreanaz

who plays the role of Angel. Other
features include the script for the
pilot episodes, a trailer, a large photo
gallery of publicity shots and in-depth
biographies of Whedon and several of the
show's stars, including Sarah Michelle
Gellar, Alyson Hannigan and Nicholas
Brendon.

**M=72, Found in output72.txt**

Buffy the Vampire Slayer fans are sure to get their fix with the
DVD release of the show's first season. The three-disc collection
includes all 12 episodes as well as many extras. There is a collection
of interviews by the show's creator Joss Whedon in which he explains
his inspiration for the show as well as comments on the various cast
members. Much of the same material is covered in more depth with
Whedon's commentary track for the show's first two episodes that make
up the Buffy the Vampire Slayer pilot. The most interesting points of
Whedon's commentary come from his explanation of the learning curve
he encountered shifting from blockbuster films like Toy Story to a
much lower-budget television series. The first disc also includes a
short interview with David Boreanaz who plays the role of Angel. Other
features include the script for the pilot episodes, a trailer, a large
photo gallery of publicity shots and in-depth biographies of Whedon and
several of the show's stars, including Sarah Michelle Gellar, Alyson
Hannigan and Nicholas Brendon.

## Recursive Equation

*optimalPenalty* can be found using the following recursive equation where $optimalPenalty(i)$ is the value stored at $optimalPenalty[i]$.

$$optimalPenalty(i) = min(linePenalty(i, i + p) + optimalPenalty(i + b + 1))$$

$$\forall p \textit{ from } 0 \textit{ until linePenalty(i,i+p)==None}$$

The base case for this algorithm would be all the cases where the worlds $i$ through $n$ fit on a single line. As explain in the algorithm for all these cases $optimalPenalty(i) = 0$. Thus by induction this algorithm is correct.

## Time and Size Bounds

$penaltyMatrix$ is size $n$x$n$ so the algorithm takes $O(n^2)$ time. It takes $n^2$ to fill $penaltyMatrix$. The algorithm iterates through *optimalPenalty*, which has size $n$, and at each index iterates through up to $n$ values of $x$ where $penaltyMatrix[i][i + x] + optimalPenalty[i + x + 1]$. Thus calculating *optimalPenalty* also takes $n^2$ time. The algorithm overall has run time $O(n^2)$.

## Problem 4

### Algorithm

Let $s_1s_2...s_n$ represent the data string and $dictionary = [w_1, w_2, ...., w_m]$ represent the dictionary strings in an array. Let $l_{w_i}$ represent the length of dictionary string $w_i$. Construct array $optimalWords$ of size $n$ that indexes from 1 where $optimalWords[i]$ relates to $s_i$. $optimalWords$ will be evaluated such that $optimalWords[i]$ will hold the minimum number of words needed to create the substring $s_1s_2....s_i$.

Initalize each index of $optimalWords$ to the maximum integer. Iterate through $optimalWords$ starting at $i = 1$ and continuing to $n$. At each index $optimalWords[i]$ iterate through the array $dictionary$ starting at $j = 1$ to $m$. For each dictionary word iterate through the letters of $w_j$ and compare them to the letters $s_is_{i+1}s_{i+2}...s_{i+l_{w_j}}$. If the two sets of letters match that indicates that the word $w_j$ starts at $s_i$. Thus the substring $s_1s_2....s_{i+l_{w_j}}$ could be constructed using the minimal configuration of words to make $s_1s_2...s_{i-1}$ and adding $w_j$ starting at the $s_i$ spot. Thus it would take $optimalWords[i-1]+1$ words to create $s_1s_2....s_{i+l_{w_j}}$. If $optimalWords[i-1]+1$ is less than what is currently stored at $optimalWords[i+l_{w_j}]$ this indicates that you found a new minimum configuration of words needed to create $s_1s_2....s_{i+l_{w_j}}$ so update $optimalWords[i+l_{w_j}] = optimalWords[i-1]+1$.

Once $optimalWords$ has been completely evaluated the minimum number of dictionary words needed to create the string $s_1s_2...s_n$ will be $optimalWords[n]$. If $optimalWords[n]$ is still the maximum integer then there exists no encoding of dictionary words that create the string $s_1s_2...s_n$.

### Recursion

The calculation of $optimalWords$ can be defined by the recursion below where $optimalWords(i) = optimalWords[i]$

$$optimalWords(i) = min(optimalWords(i-j-1)+1)$$

$$\forall j \ where \ 0 \leq j \leq k \ and \ s_{i-j}s_{i-j+1}...s_i \ is \ a \ dictionary \ word$$

$optimalWords[i]$ will be initialized to the maximum integer which represents that $optimalWords(i)$ is $null$. Thus the base case is the first few $optimalWords[i]$ which are not $null$. Therefore the base case are all the $optimalWords[i]$ where $s_1s_2....s_i$ is a word in the dictionary. This is found in our algorithm and recorded in our algorithm when evaluating $optimalWords[1]$ showing how these base cases exist and how to easily calculate them. Therefore by induction this algorithm is correct.

### Time and Size Bounds

This algorithm stores two arrays, one of size $n$ ($optimalWords$) and one of size $m$ ($dictionary$). Thus the size bound of this algorithm is $O(max(n, m))$. The algorithm iterates through all $n$ letters and for each letter iterates through all $m$ dictionary words and for each dictionary word iterates up to $k$ letters. Thus the run time is $O(nmk)$.