ANLY 550: Assignment #3
Brody Vogel
Worked with: Jinghao Yan

1

Cut Property: Let $X \subseteq T$, where T is a MST of G. Let $S \subseteq V$. Assume that no edge in X "crosses" S (connects a node in S to a node in X). Let e be a minimum-weight, crossing edge. Then $X \cup \{e\} \subseteq T'$ for some MST, T', of G.

Consider a connected, undirected graph, *G,* with unique edge weights. This means every "cut" of *G* has a unique lightest-weight crossing edge, *e*.
For each *e, e* must be in any MST of *G*. If this were not true, there would be some path taking the place of *e* in a MST of *G.* In that path, there would be an edge $f \neq e$ crossing the cut from which *e* is the lightest-weight edge. But if we replaced *f* with *e*, the MST would have a lighter weight, thus proving each *e* must be in every MST of *G.*
Now, this means any MST of *G* can be built from any source node *s* in *G* by continually adding the lightest-weight crossing edge *e* from growing cuts of *G,* until we arrive at a MST = <*e1, e2, e3, ..., eN-1*>. We know that, because the weights of *G* are unique, each *e* is unique, and so there is only one MST of any connected, undirected graph with unique edge weights.

2

a

```
pseudocode [multiplications emboldened]
Fast3(x, y):
if x, y < 100:
        return x * y
a, b, c = x[1:n/3], x[n/3 + 1: 2n/3], x[2n/3 + 1: n]
d, e,, f = y[1:n/3], y[n/3 + 1: 2n/3], y[2n/3 + 1: n]
temp1 = (a + b + c) * (d + e + f)
temp2 = (a + b) * (d + e)
temp3 = (b +c) * (e + f)
temp4 = a * d
temp5 = b * e
temp6 = c * f
first_term = temp2 – temp4 – temp5
third_term = temp3 – temp5 – temp6
second_term = temp1 – temp4 – first_term – third_term – temp6
return [ $10^{\frac{4n}{3}}(temp4) + 10^{n}(first\_term) + 10^{\frac{2n}{3}}(second\_term) + 10^{\frac{n}{3}}(third\_term) + temp6$ ]
```

The algorithm is the same as Karatsuba's, extended to three parts (n/3) instead of two. We first split each n-digit number, *x* and *y*, into thirds*, [a,b,c] and [d,e,f]*. We observe that the original numbers can then be expressed as: x = $10^{\frac{2n}{3}}(a) + 10^{\frac{n}{3}}(b) + c, and\ y = 10^{\frac{2n}{3}}(d) + 10^{\frac{n}{3}}(e) + f.$
So $x * y = 10^{\frac{4n}{3}}(ad) + 10^{n}(ae + bd) + 10^{\frac{2n}{3}}(af + be + dc) + 10^{\frac{n}{3}}(bf + ce) + cf.$
Computing this would require nine elementary multiplications, but we can reduce this to six by observing:

) ae + bd **= (a+b)*(d+e) – ad – be**
) bf + ce **= (b+c)*(e+f)** *–be* **– cf**
and
) af + be + cd = ***(a + b + c)*(d + e + f)*** *– ad –cf - (a+b)*(d+e) – ad – be - (b+c)*(e+f) –be*

In this case, the only multiplications we must perform are emboldened at their first appearance in the relations; there are only six, which we see we can then use to build the terms we need to plug into the formula for $x * y$, as in the pseudocode.

b
Similar to Karatsuba's, the asymptotic runtime of this algorithm is: $T(n) = 6T(n/3) + O(n) = O(n^{\log_3 6}) = O(n^{1.63})$, by the Master Theorem. It performs 6 multiplications of n/3 digit numbers; $O(n)$ handles the additions and assignments.
Karatsuba's would be faster—$O(n^{1.59})$—and so preferable.

c
In this case, the asymptotic runtime would be: $T(n) = 5T(n/3) + O(n) = O(n^{\log_3 5}) = O(n^{1.46})$, by the Master Theorem.
It would then be faster then—and therefore preferable to—Karatsuba's.

3
pseudocode

```
Max-Sub(array A of length n):
total_max = 0
cur_max = 0
cur_start = 1
total_start = 1
cur_stop = 1
total_stop = 1
for i = 1 to n:
    if A[i] + cur_max > 0:
        cur_max = A[i] + cur_max
        cur_stop = i
    else:
        cur_max = 0
        cur_start = i + 1 or NULL if i == n
        cur_stop = i + 1 or NULL if i == n
    if cur_max > total_max:
        total_max = cur_max
        total_start = cur_start
        total_stop = cur_stop
return [ (total_start, total_stop) , total_max ]
```

In essence, the algorithm starts at the beginning of the array and greedily adds the next number, while keeping track of: the largest sum of a subarray it has thus far come across and its beginning and end indices; and the sum of the subarray it is currently on and its beginning and end indices.

As long as the current sum of the subarray the algorithm is working on (*cur_sum*) stays above 0, the algorithm keeps looking to add more numbers (because it could theoretically compensate for any negative numbers in the subarray with larger positive numbers). If *cur_sum* hits or dips below 0, the algorithm knows it's better to drop that subarray and start working on a new one, starting at the next number in the array. *total_sum* and its indices, though, only update if *cur_sum* reaches a new all-time high. Thus at index 4 of [1 , 3, -2, 1, 4], *cur_sum* would be 3, while *total_sum* would be 4. In this way, when the algorithm returns [(*total_start, total_stop) , total_max*], we're ensured to get the largest sum of a subarray in the original array and the beginning and end indices thereof.

Each of the six variables can be updated in the *for i =1 to n:* loop at most n times, so the runtime of this algorithm is *O(6n) = O(n).*

Proof of Correctness
We can prove the correctness of this algorithm with induction.
Base Cases

When n = 0, it's clear the problem is not defined.
When n = 1, it's clear the algorithm returns A[1], which is correct.
When n = 2, the algorithm compares A[1] + A[2] to *total_max* (set to A[1] after the first iteration), updates *total_max* and its indices if necessary, and then returns them; this is correct.

Inductive Hypothesis:

Assume for all lengths of A up to and including n the algorithm correctly outputs the indices and sum of the largest subarray of A.

Case n+1

If the length of A is increased from n to n+1, the algorithm will run once more through the loop. If adding the new final element makes *cur_max* larger than *total_max*, the algorithm will update *total_max* and its indices accordingly, and then return them; this is correct.

So, by induction, the algorithm has been proved correct.

4
If you knew where to put the first or last partition, you could recursively run the algorithm on everything to the right (left) of that partition. This would be the intuition behind a dynamic solution to the problem. I'm not sure, however, where to go from there. Here's my best shot:

pseudocode

```
imbalance(Array A of length n, k partitions):
temp = []
min = []
partition_places = []
place the first k+1 entries in A in temp
place a partition between each element in temp, partition_places = [1, 2, 3,...k]
for j = k+2 to n:
        add the jth element of A to the end of temp
        for m = k down to 1:
```

```
        slide the mth partition to the right by one element if doing so will decrease the
        imbalance
        if a partition moves:
            min.append(new imbalance)
            update partition_places

return min[-1] and indexes of partitions in temp
```

Each time the array *temp* is updated, the algorithm initially places the last partition, then the second to last, etc. It uses the knowledge of the minimal imbalance of *temp* sized n to find the answer to the minimal imbalance of *temp* sized n+1.

The runtime would be *O(n)* for each addition to *temp,* and *O(nk)* for each tested change to the partition positions. *O(nk)* total.

Theoretically, the algorithm wouldn't change much if we wanted the sum rather than the largest deviation from the average. The only thing that would change is the definition of "decrease the imbalance" in the nested loop above. The sub-problems would stay the same; that is, they'd still involve finding the minimal imbalance on the smallest subarray and then building the solution to the entire problem.

Proof of Correctness
∅

5

```
pseudocode(M, text = $\ell_1, \ell_2, \dots, \ell_n$):
penalty = n x n matrix
cost = [], cost[0] = 0
place = [], place[0] = 0
for i=1 to n:
    for j=i to n:
        if $M - j + i - \sum_{k=i}^{j} \ell_k < 0$:
            penalty[i,j] = inf
        elif j == n AND $M - j + i - \sum_{k=i}^{j} \ell_k >= 0$:
            penalty[i,j] = 0
        else:
            penalty[i,j] = $(M - j + i - \sum_{k=i}^{j} \ell_k)^3$
for x=1 to n:
    cost[x] = inf
    for y=1 to x:
        if cost[y-1] + penalty[y,x] < cost[x]:
            cost[x] = cost[y-1] + penalty[y,x]
            place[x] = y
```

```
total = cost[-1]
print_output = []
a = n
b = 0
while a >= b:
        print_output.append(text[place[a-1]:a])
        a = place[a-1]
        b = place[a-1]
reverse(print_output)
return(total, print_output)
```

In terms of dynamic programming, the sub-problems are finding the minimal penalty that can be achieved for *each line*. Once we know the minimal penalty that can be incurred by the first line, for example, we can use this knowledge to find the minimal penalty that can be incurred by the first *two* lines, on and on through the text.

Basically, the algorithm does this in two parts. The first loop uses the definition from the problem to fill in a *penalty* table that looks, for example, something like this (assuming $M = 8$):

	THE	**FAT**	**BROWN**	**CAT**	**SAT**	**ON**	**THE**	**BLUE**	**DOG**
THE	125	1	inf	inf	inf	inf	inf	inf	inf
FAT		125	inf	inf	inf	inf	inf	inf	inf
BROWN			27	inf	inf	inf	inf	inf	inf
CAT				125	1	inf	inf	inf	inf
SAT					125	1	inf	inf	inf
ON						216	8	inf	inf
THE							125	0	inf
BLUE								64	0
DOG									0

In words, it goes through and computes all the possible penalties for a line that starts with the word on the x-axis and ends with the word on the y-axis.

The second loop then goes through the *penalty* table and finds the minimal penalty that can be achieved if a line starts with the word on the x-axis. In the table, it would start at "The", then walk to "The Fat" and see that it can decrease the penalty incurred by the first line from 125 to 1; it would then start a new line with "Brown", see that it can't decrease the penalty from line 2 below 27 and move to the next line; so on and so forth until it reaches the end. After initiating an array called *cost* as *cost[0] = 0,* it then uses this array to keep a running total of the cumulative costs associated with these results; from the table above, *cost = [0, 125, 1, 28, 153, 29, 161, 37, 101, 37].* This loop also keeps track of the index of the word on the x-axis that achieves that minimum penalty, in an array called *place;* from the table above, *place = [1, 1, 3, 4, 4, 5, 6, 8, 8].* In this way, the algorithm can step backwards through the *text* and *place* arrays to reconstruct the sentence, line-by-line.

Finally, the algorithm uses *cost* and *place* to return the total penalty and print the text.

The runtime is *O*(n^2) for the first two loops because both loops fill in every entry in an n x n matrix, and *O(n)* for the output loop. This gives a total runtime of *O*($2n^2 + n$) = *O*(n^2).

Proof of Correctness
Assume the algorithm *did not* return the minimal penalty possible for a threshold, *M,* and *text* = $\ell_1, \ell_2, \dots, \ell_n$. This means one of two things:

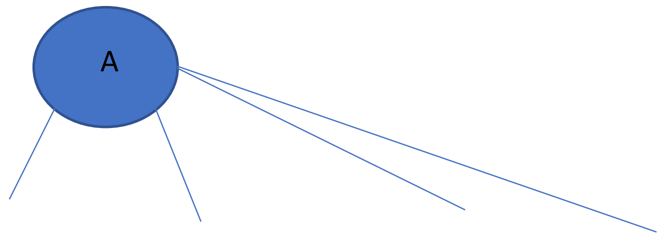
- *penalty* was filled incorrectly
- The second loop did not correctly search *penalty* for the minimal penalty for each line.

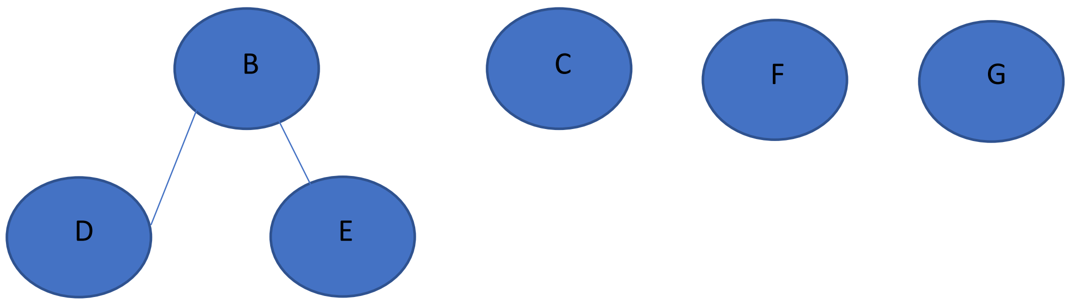
The first bullet is false because the algorithm fills in *penalty* according to the description of the problem. So this means the algorithm did not find the minimal penalty for some line in the text. But we know from the description above that *every possible* combination of lines will be considered by the algorithm, and so the second bullet is also impossible. Thus, by contradiction, the algorithm has been proved correct.

6
pseudocode:

```
Max-Independent-Set(Graph G = (V, E), source s):

if G is directed:
        order = run DFS(G) to get topological ordering
else:
        order = run BFS(G) and rearrange the resulting dist[] array by reversed order of
                distance from s

no_children(node n) = 1 + # of nodes in largest independent set starting at the level of n's
                        grandchildren

children(node n) = # of nodes in largest independent set starting at the level of n's children

largest = []
for x in order:
        largest[x] = max(no_children(x), children(x))

return largest[-1]
```

To use a simple example, on the following graph:





The algorithm would compute:

largest[E] = max(no_children(E), children(E)) = no_children(E) = 1
largest[D] = max(no_children(D), children(D)) = no_children(D) = 1
largest[G] = max(no_children(G), children(G)) = no_children(G) = 1
largest[C] = max(no_children(F), children(F)) = no_children(F) = 1
largest[C] = max(no_children(C), children(C)) = no_children(C) = 1
largest[B] = max(no_children(B), children(B)) = children(B) = 2
largest[A] = max(no_children(A), children(A)) = max(1 +largest[B], [largest[B] + largest[C] + largest[F] + largest[G]])
= (2+1+1+1) = 5

largest[-1] = 5, starting at node A

The sub-problems are finding the size of the largest independent set of nodes rooted at each node. If we then start from the bottom of the graph, we can use the answers to these sub-problems to build the largest independent set rooted at the source, which we know is the largest independent set in the graph.

To accomplish this, the algorithm first sorts the edges (either topologically or in descending order of distance from *s*) so as to only consider each edge at most twice (for example, B→C above is considered once with *children(B)* and once with *no_children(A)*). It then finds the largest independent set rooted at every node in the graph, which is synonymous with finding the largest independent set of nodes starting *at the level of* or *the level below* each node. The algorithm continuously adds the size of these largest sets to the array *largest*. This way, it can use previous entries in *largest* to save time in computing values for nodes higher up in the tree, as in the example above. It then calls *largest[-1]* to find the number of nodes in the largest independent set of *G* rooted at the source, which we know will be the largest independent set of *G*.

For runtime, either DFS or BFS will take *O(m + n).* As mentioned above, the algorithm then considers each node exactly once and each edge at most twice, so we get *O(3m + n).* Because this is a tree graph, though, we know *n > m,* so the final runtime is linear, *O(n)*.

Proof of Correctness

Assume the algorithm *does not* return the size of the largest independent set in some graph *G*. This means there is an independent set, necessarily rooted at the source *s,* that was not found by the algorithm. Since the algorithm compares the number of nodes starting at a level at an odd depth in *G* to that of nodes starting at an even depth (#(*A, D, E) vs #(B, C, F, G) above),* every combination of nodes is tested for the largest independent set except those that contain nodes with an edge between them. Thus, this largest independent set in *G* that was not found by the algorithm must contain at least two nodes that are connected by an edge. But this contradicts the problem. Thus, the algorithm has been proved correct by contradiction.