Walsh, Bayard Algorithms Final

Bayard Walsh

March 2023

1 Frog and Rabbit Save the World

Problem Statement: From the description $G = (V, E, s, t), L$, please describe how to build a related directed graph $G' = (V', E', s', t')$ such that: Rabbit and Frog can save the world in $G = (V, E, s, t), L$ IF, AND ONLY IF, there is a directed path from s' to t' in G' . Prove the correctness property above for your reduction, by proving both directions in the "if and only if".

Reduction:
Starting with $G = (V, E, s, t), L$, we will performing the following steps to obtain $G' = (V', E', s', t')$.

First, we will introduce the subroutine, $p(u, v)$ which is defined as follows:
Given two nodes u and v, return a set of edges E_p, where E_p is the list of **all** possible outgoing edges **which are "reachable"** from u, given our other path is currently at v, with respect to the graph G. We are defining a "reachable" edge as $L(e) = NULL$ or $L(e) = v$. We will add every possible edge on G that is either "reachable" from u or is "reachable" from some vertex u' that we can reach starting at u along a path where every edge is "reachable" consider our other path is currently at v. Think DFS but only adding edges that are "reachable". Additionally, we will label edges from our $p(u, v)$ function with respect to the current location of u and v. For example, say we can add edge (u, u') to our $p(u, v)$ edge set, because it is "reachable" from u with respect to v. Then we will label the edge as $((u, v), (u', v))$ and add that edge to E_p, because we can reach u and u' from one path given the other path stays at v. The important part is that we maintain that the edge is reachable with respect to our other path being at v, and this data is stored in every node. Note, we will only return **a set of edges** to be added to G', and as we will add every potential (u, v) pair as a node to G' before we add any edges to G', therefore we know that any edge we return can be added to the graph.

With our $p(u, v)$ subroutine defined, we will now move to reduction implementation.

1. Create a matrix S such that if $G = (V, E)$, we have $S = |V \cdot V|$. We

will use this matrix to store our reachable edge sets from a given (u, v) node. Next, compute all possible (u, v) pairs made from V. Note that a given (u, v) pair is **one** node, and will represent the current state of the two paths on G'. For $G' = (V', E')$, add all pairs as nodes to V'. Therefore we will have V^2 total nodes $\in V'$. Note that all nodes are unconnected at this point.

2. For every possible (u, v) pair from V, compute $p(u, v)$, and store each given edge set in $S[u][v]$. There will be V^2 of these problems computed.

3. For $G' = (V', E')$, then $E' \cup S[s][s]$, or add the edge set of reachable edges from (s, s) to G'. Then create list of tuples T, and we have $T = T \cup [s, s]$, which will be a list where we keep track of the reachable edges we have added.

4. For every node $n = (u, v)$ with a path to (s, s) and $n \notin T$, we will append the reachable edges set $S[u][v]$ and $S[v][u]$ to G', and then $T = T \cup [v, u] \cup [u, v]$. We will also add an edge to connect (u, v) to (v, u), **because while they are different nodes in our implementation, in relation to G they represent the same data.** Therefore if any edge is connected to one of them from (s, s) (which is the only way to be added) it is now be connected to both.

Some Notes on this step: This computation will cause G' to grow its amount of edges and therefore create more available nodes based on nodes added from previous edges and so on, which is the intention of the algorithm, as we want paths from (s, s) to all (u, v) nodes that represent a reachable state for our two paths with respect to G. We also may have duplicate edges from an incoming edge set which is already in G', in which case we will only maintain one edge in G' (extra edges are redundant and unnecessary). However note that first, **our amount of nodes are not increasing**, as we never add more nodes to G' after initiating the V^2 amount in the first step, and second our matrix T will bound the amount of edge checking computations to worst case V^2 iterations. This is because if we add a new edge to a node where we have already checked its reachable tree, we don't want to repeat work. Also note that we append the edge set from (v, u) and (u, v) in a given step, as if we achieve (v, u), we also have (u, v), because we can have either path explore more edges or stay at the node, meaning both positions for the path are reachable, even if they have different edge sets.

5. $s' = (s, s)$ and $t' = (u, t)$ **AND** (t, u), where u is any node in V. *Note here that if we have t for either u or v for **any** (u, v), we know that Rabbit and Frog can save the world, because that node represents a state where either Rabbit or Frog are at t*

After we have completed these steps for $G = (V, E, s, t), L$, we have $G' = (V', E', s', t')$.

I will now prove feasibility for our reduction, or show that given $G = (V, E, s, t), L$

we can create $G' = (V', E', s', t')$ in polynomial run time, based on analyzing each step made:

1. Initializes a matrix S of size $|V \cdot V|$, then compute V^2 pairs and add all vertices to $G' = (V', E')$. Both of these operations can be done in $O(V^2)$ run time.

2. We will consider computation for a single sub problem, then apply to every sub problem. Finding all reachable edges from a given node u if the other path is at v for G is worst case E run time, as we will check every possible edge for viability. We will do this work $O(V^2)$ times, so we have a bound of $O(E \cdot V^2)$ run time.

3. Here we add a single edge set to G', and create a list of tuples T. Initializing T and adding $[s, s]$ should be $O(1)$, and adding a single edge set to G' is worst case $O(E)$ run time.

4. Here, we can potentially add V^2 edges sets to G', and each edge set can potentially have E edges. Therefore, we have worst case $O(E \cdot V^2)$ run time. *Note that once a node has had its edge set added once, we record that in T (which is $O(1)$ run time). This recording offers a bound for our computation.*

5. $O(1)$ for assigning constant nodes.

As each incremental step is polynomial run time, we know that we have an overall polynomial reduction, which proves feasibility.

Before the formal proof, I will prove the following:

Lemma 1: For G', every node (u, v) with a reachable path from (s, s) represents a "reachable state" on G, that is there is some sequence of edges where one path is at u and the other path is at v, given both start at s

Base case: $(s, s) = (u, v)$
As both paths start s, we know (s, s) is a reachable state for both paths, so (u, v) on G' represents a reachable state for G. This proves our base case.

Now we will assume that any node (u', v') with a path k distance from (s, s) represents a reachable state for Rabbit and Frog on G.

Induction, distance from (s, s) to (u, v) is $k + 1$, assuming (u, v) is connected to (s, s)
We know that there is a path from (s, s) that can reach (u, v), by assumption. Therefore, say this path is $k+1$ distance. Next, we know that any (u', v') on G' with a path k distance from (s, s) represents a reachable state for the two paths

on G, by inductive hypothesis. Therefore, let (u', v') be the last node before (u, v) on the path from (s, s) to (u, v). Therefore we must show that given our paths are at (u', v'), they can move to the state (u, v). However, as (u', v') has an edge to (u, v), and we know the only way an edge could be added is to the graph connected to (s, s) on G' is if (u, v) is a "reachable" state if our two paths are at (u', v'). Therefore, we know that (u, v) is a reachable state for our two paths, which proves our Lemma.

Lemma 2: For G, if our two paths start at s and can do some number of steps where one path reaches v and the other path reaches u, then there will be some node (u, v) or (v, u), connected to (s, s) on G'
As we define (u, v) on G' as a state of our two paths on G, we will do induction on the number of "steps" from the start to reach the given (u, v) state. A step is either path crossing a single edge.

Base case: 1 "step" to reach (u, v) from both paths at s
Then we know that either path is one step from s. This means that either u or v is "reachable" from s, given the other path stays at s. Therefore it will be added to G', as it is a "reachable" path starting at (s, s) so we have proved base case.

Now we will assume that for any amount of steps k to reach a reachable state for one path, u' and the other path, v' on G, there will be some node (u', v') on G' such that (u', v') is connected to (s, s).

Induction, $k + 1$ "steps" to reach (u, v) from both paths at s
By inductive hypothesis, we know that we will have any node (u', v') after k steps. We also know that we can do some number of steps where one path reaches v and the other path reaches u, by assumption, and we will let this number of steps be $k + 1$. Therefore, have (u', v'), the node on G' connected to (s, s) by inductive hypothesis, be a node that represents the state of our two paths one step before they reach u and v. As we know that we can have one path reach u and the other path reach v by assumption, then there must be some valid step from the states u' and v' to u and v. Therefore, from the node (u', v') there must be some outgoing "reachable" edge connected to (u, v), because we check every possible edge from (u', v') in G', and our paths can reach u and v in G. As (s, s) is connected to (u', v') by assumption, we know that there is now some path from (s, s) to (u, v). This proves our induction.

Now to prove correctness, I will show that:
$G = (V, E, s, t), L <=> G' = (V', E', s', t')$ for the given problem statement.

If $G = (V, E, s, t), L$, then $G' = (V', E', s', t')$
Here, we know there must be some valid state where at least one path is at t, because "they save the world". Therefore, we will apply **Lemma 2**, with regard to the state where one path is at t. Therefore, this proves that there must be

some connected path (u, t) or (t, u) from (s, s), proving there is a path on G' given we have the save the world problem.

If $G' = (V', E', s', t')$, then $G = (V, E, s, t), L$
Here, we know there must be some reachable path from (s, s) to either (u, t) or (t, u) in G', as given. Therefore, we will apply **Lemma 1**, with regard to this (u, t) or (t, u). Therefore, this proves that there must be a reachable state where one path is at t, which this proves our save the world problem, given there is a path on G'.

Now we have $G = (V, E, s, t), L <=> G' = (V', E', s', t')$ which completes the reduction relationship stated in the problem statement.

2 Balanced Intervals

A)
Input: arrays $X[1, \cdots, n]$ and $Y[1, \cdots, n]$, where each entry $X[i]$, $Y[j]$ from either array is an integer in the range $[-3n, 3n]$.

Output: a pair $i, j \in 1, ..., n$ of (not necessarily distinct) values, such that $X[i] = Y[j]$. If no such pair does not exist, output "DNE".

```
Algorithm 2 A):( $X[1, ..., n]$ and $Y[1, ..., n]$)
  $A[-3n \cdots 3n]$ // create array for ORDERED tuples with the $-3n \cdots 3n$ total indices to access

  from $i = -3n$ to $i = 3n$ do: // $i = 3n$ inclusive
      $A[i] = \{i, \infty\}$ // make tuple at each i index in A. i first means i value not found yet

  from $i = 1$ to $i = n$ do:
      $A[x[i]] = \{\infty, i\}$ // $x[i]$ value possible at i, update tuple. $\infty$ first means i found, i stored second

  from $j = 1$ to $j = n$ do:
      if $(A[y[j]].1 = \infty)$ do: // some $y[j] = x[i]$
          return $(A[y[j]].2$ , j)

  return DNE // no matching index possible
```

Algorithm Notes:
Not a formal proof, but a few notes on clarifying the algorithm:

1. Overall run time is 3 loops: initiating A, adding X, and then checking for Y matches. Those loops have the following iterations $O(6n)$, $O(n)$, and $O(n)$. It follows that we have reached our $O(n)$ runtime.

2. In terms of syntax $A[i].1$ and $A[i].2$ mean checking the first and second terms in the ORDERED tuple, respectively.

3. The basic idea of the algorithm as that we will create a tuple for $\{i, \infty\}$ for every possible index ($-3n$ to $3n$). Then every time we can find a given value i in the array x, we replace that value in A, changing the first part of the tuple to ∞, which is a essentially a flag for i occurs in x. The second part of the tuple is the i index, because we want to return where the pair occurs. We store all possible x values in A, as range is given. Note we don't need to do anything for duplicates entries in x, because any index is fine, so we can replace a previously found x with a newer one - it doesn't change the problem. Next we query trough y. If any $y[j]$, or y value at j index access an $A[]$ which $= \infty$, we know that we have a match, because the value at j for y equals some value in x, or $x[i] = i = y[j]$, meaning we have a pair, so we return the i index on the second half of the tuple, with the j from the y loop. We return the first pair because any pair will do. We enter every x and check every y, which implies that if no pair is returned, no pair is possible so we **return DNE**

B)
Input: array $A[1, ..., n]$ of integers $A[i]$, each in the restricted range $\{-3, -2, -1, 0, 1, 2, 3\}$.

Output: a pair $[i, j]$ with $1 \leq i \leq j \leq n$, such that $\sum_{k=i}^{j} A[k] = 0$

```
procedure: Find-Interval ($K[1, ..., n]$)

    $n = \text{size}(K)$

    if $n$ is 1 then:
        if ($A[k_1] = 0$) then: // base case, $k_1 = 0$ (only index in $K$), then set $j = k_1$ and $i = k_1$
            return ($\{k_1, k_1\}$)

        else:
            return DNE

    $m = [n/2]$ // partition $K$ in half

    $B =$ Find-Interval ($K[1, ..., m]$)
    if $B \neq$ DNE do:
        return $B$

    $C =$ Find-Interval ($K[m + 1, ..., n]$)
```

```
if $C \neq$ **DNE do:**
    **return** $C$

$D=$ **Algorithm** $2$ **A):(**$K[1,...,m], K[m+1,...,n]$**, where BOUND=**$-3m, 3m$**)**
**return** $D$
```

```
**Algorithm:** $A[1,...,n]$

    $S[1 \cdots n]$ // *initialize sum array*
    $S[1] = A[1]$ // *set base for sum loop*

    from $i = 2$ to $i = n$ **do:** //*sum loop*
        $S[i] = S[i-1] + A[i]$

    from $i = 1$ to $i = n$ **do:** //*check edge case where* $0$ *interval starts at front*
        if $S[i] = 0$ **do:**
            **return** $\{1, i\}$

    $O =$ **Find-Interval** $(S[1,...,n])$
    **return** $O$
```

Proof of Correctness:
Before our formal proof, we will prove some simple Lemma:

Lemma 1: If for two points i and j (where $1 \leq i \leq j \leq n$) we have $S[i] = S[j]$, then we know that $\sum_{k=i}^{j} A[k] = 0$
We compute the array S as an ongoing sum while we process A, so for a given i, $S[i]$ is the sum of every value in A from $1 \cdots i$, or $S[i] = \sum_{k=1}^{i} A[k]$. Therefore we have for $S[i] = \sum_{k=1}^{i} A[k]$, $S[j] = \sum_{k=1}^{j} A[k]$, and $S[i] = S[j]$. Therefore, consider the following steps:
$S[i] = S[j]$
$S[i] = \sum_{k=1}^{i} A[k] = S[j] = \sum_{k=1}^{j} A[k]$
$\sum_{k=1}^{i} A[k] = \sum_{k=1}^{j} A[k]$
$\sum_{k=1}^{i} A[k] - \sum_{k=1}^{i} A[k] = \sum_{k=1}^{j} A[k] - \sum_{k=1}^{i} A[k]$
$0 = \sum_{k=1}^{j} A[k] - \sum_{k=1}^{i} A[k]$
$0 = \sum_{k=i}^{j} A[k]$
This completes our lemma.

Lemma 2: For any two points i in $K[1,...,m]$ and j in $K[m+1,...,n]$, where both arrays are sub arrays of S, and where $1 \leq i \leq m < m+1 \leq j \leq n$, and given that $n = \frac{m}{2}$, if we have $S[i] = S[j]$, then we know that $-3m \leq i = j \leq 3m$
Here we will show that any potential $S[i] = S[j]$ pair among two arrays will obey the bounds that we have designed for **Algorithm** 2 **A)**.

As each integer in $A[i]$ is restricted to $\{-3, -2, -1, 0, 1, 2, 3\}$, given we start our sum at 0, and we know that at **most** over the array $K[1, ..., m]$, the overall sum of S can increase by $3m$ (every number is 3) or decrease by $-3m$ (every number is -3). Therefore, if we run **Algorithm** 2 **A)** on $K[1, ..., m]$, because the the range $[-3m, 3m]$, we know that every possible number in $K[1, ..., m]$ will be stored in the array we implement in that solution (see above). While this is not inherently true for every possible number in $K[m+1, ..., n]$, because a given index could be greater than other less than the $[-3m, 3m]$ bound (say strictly increasing by 3 from all of $K[1, ..., n]$, none of these numbers can be equal to some number in $K[1, ..., m]$, so our bound holds for any possible $S[i] = S[j]$ pairs. This proves our Lemma
*Side Note: the computation in **Algorithm** 2 **A)** could cause some out of bounds referencing to the array built in $2A$ as some $K[m+1, ..., n]$ indices could be beyond the array bounds as mentioned, but we will ignore these invalid references because none of those references will result in a $S[i] = S[j]$ pair as proved above.*

We are using a divide and conquer approach, so we will follow that proof strategy of base case, inductive hypothesis, and inductive step.

Base Case, size($K[]$)=1 and $A[k_1] = 0$
If size($K[]$)= 1, then this is the smallest size of an interval, and if $A[k_1] = 0$, then we know that we can return $\{k_1, k_1\}$ as a valid interval which sums to 0 across its indexes (which is just itself when size is 1). Furthermore, no other value of $A[k_1]$ can have a sum to 0 across an interval of size 1, and we return **DNE** in this case. As our algorithm follows this approach we have proved our base case for any $K[]$ where size($K[]$)= 1.

Inductive Hypothesis
Now we will assume that for any interval $K[1, ..., n]$ where $1 \leq$ size($K[]$) $< H$ which obeys our integer range restrictions stated in the problem, we will correctly return a valid interval that sums to 0 or **DNE** if none is possible.

Inductive Step
Consider an interval of size($K[]$) $= H$ which obeys our integer range restrictions stated in the problem, and I will prove that we will correctly return a valid interval that sums to 0 or DNE if none is possible. As we have proved base case, I will assume $1 < H$. Therefore, we will partition $K[]$ in half, and then run recursive **Find-Interval** calls on each side of the partition. As we assume any $K[]$ of size($K[]$) $< H$ is correct, if either of these calls return a valid interval we return it in the algorithm, and we are done, so we will assume that both cases return **DNE**, or there are no valid sub intervals within either side that equal 0. Therefore we must prove that
$D=$ **Algorithm** 2 **A):(**$K[1, ..., m], K[m+1, ..., n]$**, where BOUND=**$-3m, 3m$**)** is correct. Note that every array we are working on is a continuous sub array of $S[1, ..., n]$. And as proved in **Lemma** 1, any two points in $S[1, ..., n]$ returned by Algorithm 2 A) represent a valid interval for our solution. Secondly, as proved

in **Lemma** 2, every $S[i] = S[j]$ pair is within the bound of the sub problem. As we know equivalent pairs in $S[]$ correlate to 0 sum indexes, and we can find all these equivalent pairs through Algorithm 2 A), and by assuming correctness for Algorithm 2 A) as given by problem statement, this completes our proof, so our algorithm will return the correct interval if there is one for any interval of size($K[]$) $= H$. This proves correctness of our algorithm.

Edge Case Comment:
Note that there is a scenario where the approach taken above fails; when there is one occurrence where the overall sum S is 0, but there is no matching other 0 value (take example $A = \{-1, 2, -1\}$ which offers $S = \{-1, 1, 0\}$, and the whole interval should be returned as a solution, but evades divde and conquer approach). As this case is not caught in our recursion, we will include a simple for loop before the first recursive call to see if any S sums to 0, and then returning the interval up to that point. As S is a running sum for the whole interval, if at any i it equals 0 we know that we can return $1, i$ as a valid interval. This proves correctness for edge case.

Run time:
First we initialize a sum $S[1 \cdots n]$ and then add to it through our "sum loop". This will take $O(n)$ time. Next we use our edge case loop to check for the interval starting at 1 edge case mentioned above. This will take $O(n)$ time again. Therefore, before our **Find-Interval (**$S[1, ..., n]$**)** call, we have $O(n) + O(n)$ runtime, which simplifies to $O(n)$. Now we consider **Find-Interval (**$S[1, ..., n]$**)** runtime. Before our partition we check for base case with a few $O(1)$ operations. Then we form a half way partition. Afterwards we have two **Find-Interval** calls, each with half of the size of our initial interval caused by partition. We check for a solution, which is $O(1)$, for each recursive call. Then we call **Algorithm 2 A):(**$K[1, ..., m], K[m+1, ..., n]$**, where BOUND=**$-3m, 3m$**)**. Note that we have an $O(m)$ run time for this call (by assumption, and as proved in **Lemma 2** we can use only the size of one interval to check all pairs, so we can use m instead of n), which reduces to $O(\frac{n}{2})$. Therefore, we have $O(\frac{n}{2}) + 2 \cdot T(\frac{n}{2})$ in terms of run time for our **Find-Interval**, where T represents recursive calls to **Find-Interval**. By the Master Theorem, our recursion reduces to $O(n \cdot \log n)$. Therefore from our initial algorithm call we have $O(n \cdot \log n) + O(n) + O(n)$, reducing to $O(n \cdot \log n)$, which completes our run time proof.

3 Triple-Free Sub sequences.

Input: a sequence $a = (a_1, ..., a_n)$ of positive integers (not necessarily sorted or distinct).

Output: the maximum possible sum $\sum_{i \in T} a_i$ across all i of any triple-free sub sequence.

By the problem restrictions, we note that across a, we cannot have any set of consecutive indices $\{i, i+1, i+2\}$. From this restriction, we will create the following procedure for a sub problem for a given i with respect to a: $P[i,j]$, where $i+2=j$
$P[i,j]$ will take the largest sum among the triple with respect to a. As we know the sums are all positive, we only need to return $\max\{a_i+a_{i+1}, a_i+a_j, a_{i+1}+a_j\}$, because the sum of positive numbers can only be a larger or equal positive number, so we have a simple comparison between these 3 options, which are computed in a few constant operations; therefore we will determine $P[i,j]$ to have a run time of $O(1)$ for a given i. Also, for computing the algorithm, we will have $P[i,j]$ store three pieces of data: the overall sum for the interval M, and the starting and ending position S and E, stored as $P[i,j].M$,$P[i,j].S$, $P[i,j].E$. Note that S will be n and E will be $n+2$, regardless of which two numbers we pick, because we cannot pick the third number as it would violate the constraints, so we will treat it as if it was chosen.

We will create another procedure, which is $c(j)$. As we will create intervals from the procedure above, $c(j)$ will be used to determine which jobs are overlapping. Therefore, let $c(j)$ return the largest index $i < j$ such that interval i is compatible with interval j. This calculation can be easily done through comparing the value of given $P[i,j].S$, $P[i,j].E$ for various intervals, and we will use $c(j)$ to make sure we have no conflicting intervals through a simple for loop. Upper bound run time is $O(n)$ as $c(n)$ can consider every interval, and note that we have n total intervals as we each distinct interval starts with a given a, and the array is n long.

With this in mind, we will create our algorithm

Algorithm:
```
    $S = [n]$ // *creation of array storing intervals starting at n*

    from $i = 1$ to $i = n-2$ **do:** // *end at $n-2$, will reach $n$ for $M[n-2, n]$*
        $S[i] = P[i, i+2]$ // *store procedure in $S[i]$, $P$ computes with respect to sequence $a$*

    **SORT, RELABEL** $S[i]$ by largest $S[i].E$ // *earliest end time first*

    $Q = \emptyset$
    $M[0] = 0$
    from $i = 1$ to $i = n-2$ **do:** // *DP calls*
        $M[i] = \max\{S[i].M + M[c(i)], M[i-1]\}$

        if $M[i] \neq M[i-1]$ **do:** // *maintain set of unavailable indexes*
            $Q = Q \cup (S[i].S) \cup (S[i].S) + 1 \cup (S[i].E)$

    **max**= $M[i]$
```

```
from $i = 1$ to $i = n$ do: // checking edge cases
    if $i \notin Q$ do: // available index, add to sum (doesn't violate intervals)
        max=max+$a_i$

return max
```

Proof of Correctness: This will follow a traditional DP proof as recommended
Family of optimization problems
We compute the max for a given **continuous** 3 integer index on a, through procedure $P[i, j]$. As we compute this sub problem from $i = 1$ to $i = n - 2$, this covers all sub problems in the overall problem, and as every index starts at some i and extends two indexes beyond that, we cover the scope of problems.

Obtaining overall output from family of optimization problems
Our only constraint on the maximization problem is that we can't have any set of three consecutive indices. By computing every intervals possible with this constraint in mind as sub problems and using $c(i)$ to check for when we can add another (non-overlapping) interval, we will build towards the optimal solution.

Base Case
Here $OPT[0] = M[0] = 0$. As we iterate from $i = 1$ to $i = n - 2$, we will build our max interval referring to the previous max. As $OPT[0] = M[0]$ represents the state before we have considered our first index, $OPT[0] = M[0] = 0$ is optimal.

Recurrence Relationship
We will consider a given $OPT[i]$ to be the optimal solution, or max sum, given it has considered i intervals (ordered from earliest ending time), where $i \leq n-2$. Note that at this point we have sorted $S[]$, and in this case $i \neq$ an index in a_i, but rather the $i-th$ interval, but since we have $n-2$ overall intervals we use $n-2$ for the loop. Consider if the i interval is $\notin OPT[i]$. Then $OPT[i] = OPT[i-1]$, as we haven't added anything. So consider if the i interval is $\in OPT[i]$. Then $OPT[i] = OPT[i] + OPT[c(i)]$, as we have added $OPT[i]$, and we know there are no incompatible intervals with i in $OPT[i]$, in regard to creating a continuous triple, meaning that all intervals, $c(i) + 1 \cdots i - 1$ will not be in $OPT[i]$. So we have $OPT[i] = OPT[i] + OPT[c(i)]$ in this case. These observations with the base case given the following recurrence.

$$OPT[i] = \left\{ \begin{array}{ll} 0, & i = 0 \\ \max\{OPT[i] + OPT[c(i)], OPT[i-1]\}, & i > 0 \end{array} \right\} \quad (1)$$

Because we can assume $M[j]$ is optimal for all $j < i$, and as we have $M[i] = \max\{S[i].M + M[c(i)], M[i-1]\}$, and we consider our base case $M[0] = 0$, then we have incorporated this recurrence into our algorithm which proves our algorithm, so $OPT(i) = M[i]$.

While we have maximized with respect to **intervals**, note that even if we have the max sum from intervals to maximize our function, there could be gaps in between our solution, as it might not be possible to form a max interval sum which has every "usable" number in it, and as we want to maximize overall sum, we will do a final loop afterwards to fill in those gaps, which is the purpose of the set Q. Q maintains numbers which are currently in (or would be in violation of) an interval after we have computed $M[i]$, based on when $M[i]$ updates. Checking for Q will be worst case $O(n)$ runtime, as that is the longest the set can be and we will use pessimistic implementation of set lookup for runtime.

Runtime:
First we implement $S[i]$, which means running $P[i, i+2]$ $n-2$ times. As discussed above, this is $O(n-2)$ total work as each $P[i, i+2]$ does some simple computations in $O(1)$ time. Next, we sort and relabel $S[i]$. Assuming $O(n \cdots n$ log $n)$ sorting time, this step takes that long. Next, more constant operations. Next, we loop through our intervals and make a number of calls to $M[]$. However, this work can be constant with the exception of $c(i)$ lookup time, as we save every call made, so we only need to make each call once, meaning we don't have to worry about optimal recursion calls. Therefore we have $O(n)$ work for each of $O(n)$ sub problems, which is $O(n^2)$. From here we will check for "gaps" with our Q ;pp[, which will do potentially $O(n)$ work for $O(n)$ iterations, so we have $O(n^2)$ again. In conclusion, we get $O(n-2) + O(n \cdots n$ log $n) + O(n^2) + O(n^2)$, which simplifies to $O(n^2)$, and is polynomial.

4 Path interception

Input: a directed, unweighted graph $G = (V, E)$, with designated multiple "start" vertices $s_1, \cdots, s_k$ and "terminal" vertices $t_1, \cdots, t_{k'}$. You may assume all these $k + k'$ vertices are distinct.

Output: $W \subseteq E$ of edges where all directed paths from any start vertex s_i to t_j pass through at least one edge of W

Algorithm:

```
create capacity function $c(e)$.

for every $e = (u, v) \in E$, label the edge with capacity as follows:
    **if** $u = s$ **and** $v = t$: // *check for $s - t$ edge*
        $c(e) = 1$
    **else:**
        $c(e) = \infty$
```

```
for $G = (V, E)$, $V \cup s_m \cup t_m$ // *add master edges*

for every $s \in s_1, \cdots, s_k$ do:
    connect $s$ to $s_m$ with some edge $e$, where $c(e) = \infty$

for every $t \in t_1, \cdots, t_{k'}$ do:
    connect $t$ to $t_m$ with some edge $e$, where $c(e) = \infty$
```

Run Edmonds and Karp algorithm to find residual graph G_f, with respect to G, using s_m as a source node and e_m as a sink node

Compute minimum $s_m - t_m$ cut using G_f. Partition these sets as S and T, and add all edges that cross the partition to min cut-set W
return W

Proof of correctness:
Lemma 1: After our changes to G every $s_i - t_j$ path can be extended to an $s_m - t_m$ path, and every $s_i - t_j$ path is a valid path for capacity in the max flow problem with source s_m and sink t_m
We know that every s node directly connects to s_m, and we also know that every t node directly connects to t_m, as we have designed. Therefore, any $s_i - t_j$ path can be extended to be a $s_m - s_i - t_j - t_m$ path. Therefore, any $s_i - t_j$ path will be a valid path for capacity in our max flow problem, where source is s_m and sink is t_m, which proves our lemma.

Lemma 2: For any $s_m - t_m$ path, an $s - t$ edge will always be in the min cutset for the max flow problem, or our overall min cut will always only have $s - t$ edge
We know that every $s_m - t_m$ path must have at least one edge which crosses from some s node to some t node, say e, because we have only connected s_m to s nodes and t_m to t nodes, so there must be some edge between these two sets on a $s_m - t_m$ path. Furthermore, we know that $c(e) = 1$, whereas every non $s - t$ edge has capacity equal to ∞. Therefore we know that for every $s_m - t_m$ path, we can always pick an $s - t$ edge as the min value for our cutset, and even in the worst case where our cutset is every $s - t$ edge, it will always be smaller than picking one non-$s - t$ edge, because $E < \infty$. Therefore, when we run the max flow problem, where the source is s_m and sink is t_m, on every $s_m - t_m$ path we can pick an $s - t$ edge which must be in the min cutset. This proves our Lemma.

Lemma 3: Every path $s_i - t_j$ passes through W at some point
First we use **Lemma** 1 to extend our path to some $s_m - t_m$. Then we consider how we have computed W, which is the set of edges of a minimum-cut for the $s_m - t_m$ max flow. By **Lemma** 2 we know every $s_m - t_m$ path must have an $s - t$ edge in the min cutset for the max flow problem. We also know that extending a path to $s_m - t_m$ never creates $s - t$ edges (only $s - s$ and $t - t$ edges added). Therefore, we have proved that every $s_i - t_j$ path is also a $s_m - t_m$ path with

an $s-t$ edge in the min cutset, and as we take the min cutset as W, every edge must cross W, completing our proof. Furthermore, note that if we remove every edge in W, from G we will have a max flow of 0, because W is a min cut. As s_m is connected to every s with ∞ capacity and t_m is connected to every t with ∞ capacity, then if we have a max flow of 0, then there is no path from any s_i to any t_j for the graph $G' = (V, E - W)$. Note also that every edge which is in W was originally in the graph of G, as we only added $s-s$ and $t-t$ edges in our implementation, so even in our modifications the graph holds for the original problem.

Lemma 4: W is minimal, or there is no W' which satisfies the requirements where $|W'| < |W|$
Assume that there is a W' which is smaller. Therefore, there must be some other way of selecting edge where $|W'| < |W|$. Then at most, $|W'| = |W| - 1$, or there must be at least one less edge. However, we have used the reduction to the max flow problem to find the minimum cut set between s nodes and t nodes. This is because:
1. We only feature edges between s and t nodes as the edges between the two are bottlenecks for the min cut, and a cutset of every $s-t$ edge is smaller than a cutset with any non $s-t$ edge, so therefore our cutset will always feature $s-t$ edges.
2. We will consider every s and t node as each node is has ∞ capacity to and from their respective master node, so every potential $s-t$ path must be in the min cut.
Furthermore, as we have given every $s-t$ edge the value 1, and we know that the Edmonds and Karp algorithm is always bound by min cut set, then the minimum value of a cut also corresponds to minimum edges. Therefore, if we remove a single edge from W, the cutset would no longer be valid by definition, and therefore there would be some $s-t$ path which doesn't cross W. Therefore we cannot find a smaller W', proving the minimization of W.

Run Time:
Capacity function is $O(1)$. Next, we add capacity to every edge, which is $O(E)$. We add two nodes to the graph which is $O(1)$. Then we add edges between each s and t node to either s_m or t_m. As each s, t node are distinct, we have worst case $O(V)$ iterations. Next we run Edmonds and Karp algorithm to find residual graph G_f. This takes $O(|V||E|^2)$ run time. Next we compute the minimum $s-t$ cut using G_f, where we will do a search to find all vertices that are reachable from s_m through G_f and put them in S. All unreachable vertices in G_f are in T. This will be our minimum cut, W, and all edges that cross $S-T$ will be added to W. Given G_f from the previous step, this step will be worst case $O(E+V)$ run time. Now we have $O(|V||E|^2) + O(|E| + |V|) + O(|E|) + O(|V|)$, simplifying to $O(|V||E|^2)$ as an upper bound.