Dynamic Programming

Julius February 23&28, 2024

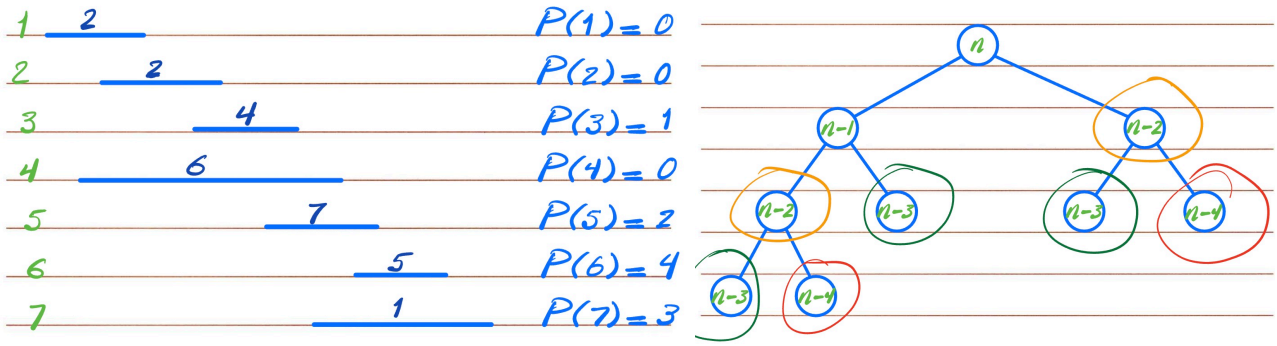
1 General Approach

- Characterize the structure of an optimal solution
- Recursively define the value of an optimal solution
- Compute the value of an optimal solution in a bottom-up fashion
- Construct an optimal solution from computed information

2 Weighted Interval Scheduling Problem

2.1 State Definition and State Transition

Input: Given a set of n requests, where i^{th} interval has a start time $s(i)$, a finish time $f(i)$, and a weight $w(i)$.
Output: A subset of compatible request intervals $S \subset \{1, 2, \ldots, n\}$, so at to maximize $\sum_{i \in S} w(i)$.
Observation: Either job i is part of an optimal solution or not.
If it is, value of optimal solution = w(i) + value of the optimal solution for the subproblem that consists only of **compatible requests with job i**.
If it isn't, value of optimal solution = value of the optimal solution without job i.
To find the compatible subproblems quickly: sort the requests in order of non-decreasing finish times and reindex the requests so that $f(1) \leq f(2) \ldots \leq f(n)$ Then define $P(j)$ to be the largest index $i < j$ such that request i is compatible with request j.



Denote the optimal solution to the problem consisting of first $\{1, \ldots, j\}$ requests O_j, and $OPT(j)$ denotes the value of O_j. we can now rewrite the observation as:
If $j \in O_j \Rightarrow OPT(j) = w(j) + OPT(P(j))$
If $j \notin O_j \Rightarrow OPT(j) = OPT(j-1)$

Solution:

```
Compute_Opt(j)
    if j = 0: Return 0
    else: Return max(w(j) + Compute_Opt(P(j)), Compute_Opt(j-1))
```

Worst case occurs when the size of the subproblem go down very slowly, e.g. $T(n) = T(n-1) + T(n-2)$ similar to the Fibonacci series T(n) grows exponentially. If we check the above worst case recursion tree, we can observe that we are solving the same exact subproblems multiple times.

2.2 Memorization Recursion

Memorization: Store the value of each subproblem in a globally accessible place the first time we compute it, then simply use this precomputed value in all future recursive calls.

Memorization Solution:

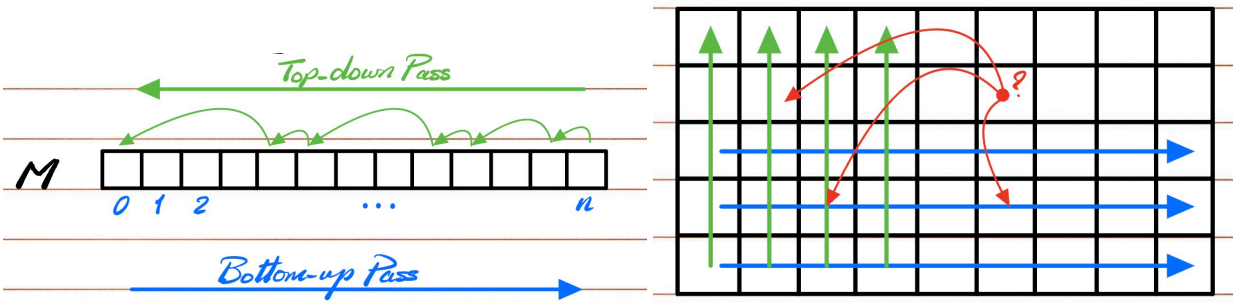
```
M_Compute_Opt(j)
    if j = 0: Return 0
    else if M[j] is not empty: Return M[j]
    else: Define M[j] = max(w(j) + M_Compute_Opt(P(j)), M_Compute_Opt(j-1))
        Return M[j]
```

Complexity Analysis: Initial Sorting Requests take $\Theta(nlgn)$; Build the $P(\cdot)$ array by Binary Search takes $\Theta(nlgn)$; M_Compute_Opt takes $\Theta(n)$; Total time complexity is $\Theta(nlgn)$.
However, so far the **bottom-up pass from 1 to n** compute only the value of the optimal solution, we can't find the actual optimal solution because at a higher up node we might realize this whole branch is not optimal. Thus we need the **Top-down pass from n to 1** to compute an optimal solution: j belongs to O_j if and only if $w(j) + M\_Compute\_Opt(P(j)) \geq M\_Compute\_Opt(j-1)$.

Find the optimal solution:

```
Find_Solution(j)
    if j > 0:
        if w(j) + M[P(j)] ≥ M[j − 1]:
            Output j together with the results of Find_Solution(P(j))
        else:
            Output the results of Find_Solution(j-1)
```

2.3 Iterative Solution



Iterative Solution:

```
Iterative_Compute_Opt(n)
    M[0] = 0
    for j = 1 to n:
        M[j] = max(w(j) + M[P(j)], M[j-1])
    return M[n]
```

Usually the values of the optimal solutions to unique subproblems are stored in 1, 2, 3, ... dimensional arrays. The order in which these values are computed depends on the recurrence formula.

3 K Intervals Scheduling Problem

Input: an array a[n] containing positive integers $a_1, a_2, \ldots, a_n$ and positive integers m and k such that $mk \leq n$.
Output: choose **k mutually disjoint subarrays** of length m from array a[n] to maximize the total sum of elements in these subarrays. Formally, let the start and end indices of the k intervals of length m be $[L_1, R_1], [L_2, R_2], \ldots, [L_k, R_k]$, such that $1 \leq L_1 < R_1 < L_2 < R_2 < \ldots < L_k < R_k \leq n$, then the objective is to maximize the value of $\sum_{i=1}^{k} \sum_{j=L_i}^{R_i} a_j$.
However, we can't only define the OPT(i) as the maximum sum of the first i elements of a[n] using k intervals of length m, because we need to keep track of the number of intervals constructed as well.
Define OPT(i, j): the maximum sum when choosing i mutually disjoint subarrays for the first j elements of a[n].
Recurrence Formula:

$$OPT(i,j) = \max\{OPT(i,j-1), OPT(i-1,j-m) + \sum_{k=j-m+1}^{j} a_k\}$$

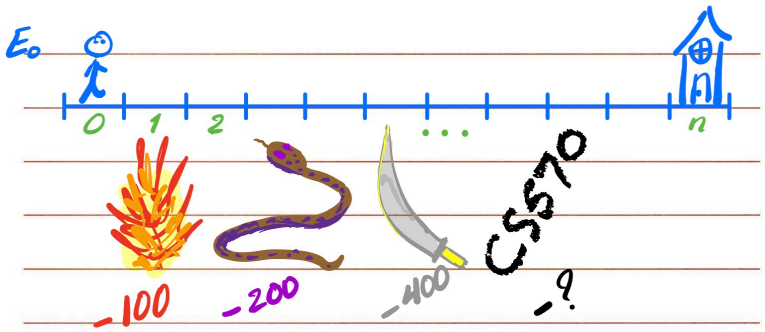
Initialization: $\forall i = 0, 0 \leq j \leq n, OPT(i,j) = 0$ and $\forall 1 \leq i \leq m, 0 \leq j < i * m, OPT(i,j) = 0$ Initialize the prefix sum array of size (n + 1) with prefix_sum[0] = 0;

```
Bottom Up Pass:
for i = 1 to n: prefix_sum[i] = prefix_sum[i - 1] + a[i];
for i = 1 to k:
    for j = i*m to n:
        OPT(i, j) = max{OPT(i, j-1), OPT(i-1, j-m) + prefix_sum[j] - prefix_sum[j - m]}
return OPT(k, n)
```

Time Complexity: The initialization of the prefix sum array takes $\Theta(n)$, the bottom-up pass takes $\Theta(kn)$, so the total time complexity is $\Theta(n + kn) = \Theta(kn)$.

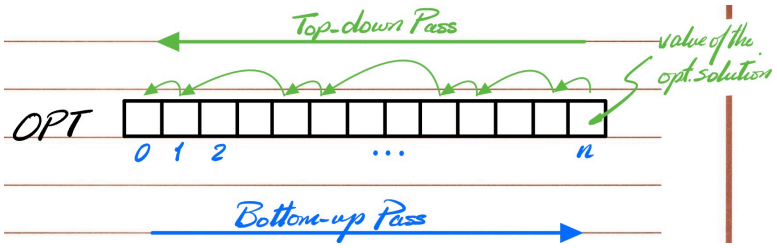
4 Jump Video Game Problem

Start with energy E_0, the player need to reach home with the maximum level of remaining energy. The player will lose e_i units of energy when landing in stage i. Choices at each stage: 1. Spend 50 units of energy to walk to the next stage; 2. Spend 150 units of energy to jump over one stage; 3. Spend 350 units of energy to jump over two stages.



Denote $OPT(i)$: optimal level of energy when the palyer reaches stage i.
Recurrence Formula: $OPT(i) = \max(OPT(i-1) - 50 - e_i, OPT(i-2) - 150 - e_i, OPT(i-3) - 350 - e_i)$

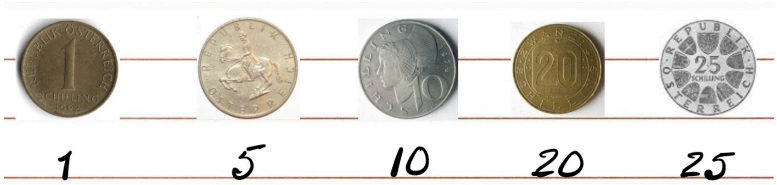
```
Bottom Up Pass:
OPT(0) = E0, OPT(1) = E0 − 50 − e1, OPT(2) = max(E0 − 150 − e2, OPT(1) − 50 − e2)
for i = 3 to n:
    OPT(i) = max(OPT(i − 1) − 50 − ei, OPT(i − 2) − 150 − ei, OPT(i − 3) − 350 − ei)
return OPT(n)
```




```
Top Down Pass:
j = n
while(j > 0):
    if OPT(j) = OPT(j − 3) − 350 − ej:
        Print "from stage" (j-3) "jump over 2 stages", j = j - 3
    else if OPT(j) = OPT(j − 2) − 150 − ej:
        Print "from stage" (j-2) "jump over 1 stage", j = j - 2
    else:
        Print "from stage" (j-1) "walk to stage", j = j - 1
```

5 Coin Problem

Given a set of coin denominations $d_1, d_2, \ldots, d_k$, and a target amount N, find the minimum number of coins needed to make change for N.



Let $OPT(i)$ be the minimum number of coins needed to make change for i. We have the following Recurrence Formula: $OPT(i) = \min(OPT(i-d_1), OPT(i-d_2), \ldots, OPT(i-d_k)) + 1$

Bottom Up Pass:

Initialize $OPT(0) = 0, OPT(1) = 1, \ldots, OPT(24) = 5$
for i = 25 to N:
$OPT(i) = \min(OPT(i-1), OPT(i-5), OPT(i-10), OPT(i-20), OPT(i-25)) + 1$
return $OPT(N)$

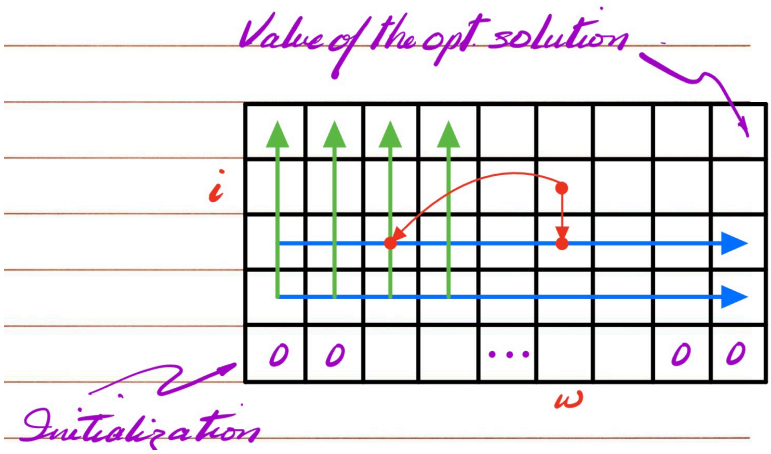
The run time complexity is $\Theta(N) = \Theta(2^{log_2 N})$, so this is a pseudo-polynomial time algorithm, because its running time is a polynomial in the numeric value of the input.

6 0/1 Knapsack Problem && Subset Sum Problem

Given a single resource available for W units of weight, and a set of requests $\{1, \ldots, n\}$ each of them takes w_i time to process with a value of v_i and can be scheduled at any time between 0 to W. The Objective is to schedule jobs such that we maximize the accumulated value.
Define OPT(i): the value of the optimal solution for requests $\{1, \ldots, i\}$ is not enough. Because if $n \notin O_n$: $OPT(n) = OPT(n-1)$. If $n \in O_n$: $OPT(n) = v_n + OPT(n-1)$. The two OPT(n-1) is not the same because the available time remaining is not the same.
Define OPT(i, w): the value of optimal solution using a subset of the items $\{1, \ldots, i\}$ with max allowed time w. If $n \notin O_n$: $OPT(n, w) = OPT(n-1, w)$. If $n \in O_n$: $OPT(n, w) = v_n + OPT(n-1, W-w_n)$

Recurrence Formula:

If $w_i > w$: $OPT(i, w) = OPT(i-1, w)$
else: $OPT(i, w) = \max(OPT(i-1, w), v_i + OPT(i-1, W-w_i))$



Iteration Solution:

Initialize $\forall i \in [1,n], w \in [0,W] : OPT(0,w) = OPT(i,0) = 0$
for i = 1 to n:
  for w = 0 to W:
    if $w_i > w$: $OPT(i,w) = OPT(i-1,w)$
    else: $OPT(i,w) = \max(OPT(i-1,w), v_i + OPT(i-1, W-w_i))$
return $OPT(n,W)$

Runtime Complexity: $\Theta(nW) = \Theta(n2^{\log_2 W})$, where W is not the size of input, rather it's the numerical value of an input term. Therefore, our complexity is exponential with reference to the input size, so it's not efficient.

7 Efficient Algorithm

An algorithm is considered **efficient** if its running time is polynomial in the size of the input.
Polynomial Time: its running time is a polynomial in the length of the input.
Pseudo-polynomial Time: its running time is a polynomial in the numeric value of the input, but exponential in the number of bits required to write the input.

8 Complete Knapsack Problem for Coins

Compute the total number of ways to make a change for a given amount m. Assume that we have an unlimited supply of coins and all denominations are sorted in ascending order: $1 = d_1 < d_2 < \ldots < d_n$
Define COUNT(n, m): the total number of ways to make change for amount m using coin denominations 1 to n
Recurrence Formula: $COUNT(n,m) = COUNT(n-1,m) + COUNT(n, m-d_n)$
Initialization: $\forall i \in (0,n],\ COUNT(i,0) = 1$ since there is a way to pay the amount 0 by paying 0 for all coin types. $\forall j \in (0,m],\ COUNT(0,j) = 0$ since there is no way to pay a non-zero amount without any coins.
Bottom Up Pass:
for i = 1 to n:
  for j = 1 to m:
    if $d_i > j : COUNT(i,j) = COUNT(i-1,j)$
    else: $COUNT(i,j) = COUNT(i-1,j) + COUNT(i, j-d_i)$
return COUNT(n, m)
This will take $\Theta(nm)$ which is pseudo-polynomial since m is the numerical value of an input term.

9 Calender Food Problem

Graduate students get a lot of free food at various events. Suppose you have a schedule of the next n days marked with those days when you get a free dinner, and those days on which you must acquire dinner on your own. On any given day you can buy dinner at the cafeteria for $3. Alternatively, you can purchase one week's groceries for $10, which will provide dinner for each day that week (that day and the six that follow). However, because you don't have a fridge, the groceries will go bad after seven days (including the day of purchase) and any leftovers must be discarded. Due to your very busy schedule, these are your only two options for dinner each night. Your goal is to eat dinner every night while minimizing the money you spend on food.
Define OPT(i): the minimum amount of money you need to spend to eat dinner on the first i days.
Recurrence Formula: If there is free dinner on day i, $OPT(i) = OPT(i-1)$, otherwise $OPT(i) = \min(OPT(i-1)+3, OPT(i-7)+10)$
Initialization: $OPT(0) = 0, \ldots, OPT(6) = 10$
Bottom Up Pass:
for i = 7 to n:
  if there is free dinner on day i: $OPT(i) = OPT(i-1)$
  else: $OPT(i) = \min(OPT(i-1)+3, OPT(i-7)+10)$
return $OPT(n)$
Time Complexity: This will take $\Theta(n)$ which is polynomial if we assume that the input consists of an array of size n called Free, where Free(i) is true when there is free food on day i, and false otherwise.
Top Down Pass:
j = n;
while(j > 0):
  if $OPT(j) = OPT(j-1)$:

Print "Day" j "eat free food", j = j - 1;
else if $OPT(i-1)+3 < OPT(i-7)+10$:
Print "Day" j "cafeteria", j = j - 1;
else:
Print "Day" j-6 "Go groceries to buy food" and "Eat at home on Day" j-6 to j, j = j - 7

10 Manhattan Walk Problem

You are in Downtown of a city and all the streets are one-way streets. You can only go east (right) on the east-west (left-right) streets, and you can only go south (down) on the north-south (up-down) streets.
a. In Figure A below, how many unique ways are there to go from the intersection marked S (coordinate (0,0)) to the intersection marked E (coordinate (n,m))?
b. Repeat this process with Figure B, be wary of dead ends.

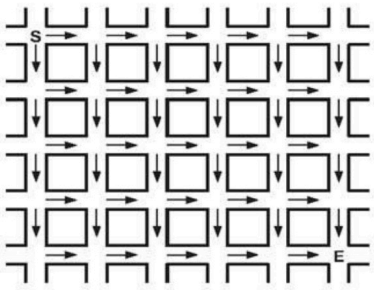


Figure A.

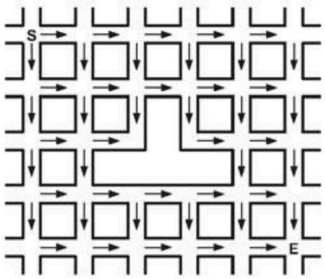


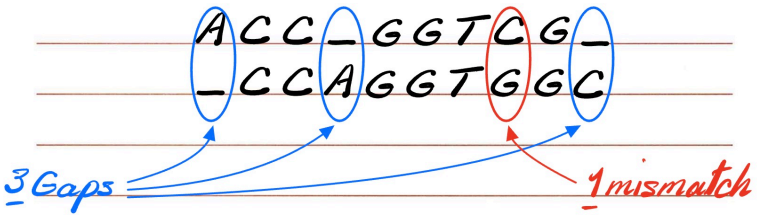
Figure B.

Part a: Let's say E is at coordinates (0,0) and s is at coordinates (n,m). We will define COUNT(i, j) as the total number of ways to go from coordinates (i, j) to E (0,0).
Recurrence Formula: $COUNT(i,j) = COUNT(i-1,j) + COUNT(i,j-1)$
Initialization: $\forall i \in [1,n], COUNT(i,0) = 1$ since there is only one way to go (horizontally) from (i,0) to (0,0). $\forall j \in [1,m], COUNT(0,j) = 1$ since there is only one way to go (vertically) from (0,j) to (0,0)
Bottom up pass:
for i = 1 to n:
for j = 1 to m:
$COUNT(i,j) = COUNT(i-1,j) + COUNT(i,j-1)$
return COUNT(n, m)

Part b: we need to apply special recurrence formula to the intersections that are affected namely (2, 2) and (3, 2). Bottom up pass:
for i = 1 to n:
for j = 1 to m:
if (i, j) is not a dead end:
$COUNT(i,j) = COUNT(i-1,j) + COUNT(i,j-1)$
else if (i, j) == (2, 2): $COUNT(i,j) = COUNT(i-1,j)$
else if (i, j) == (3, 2): $COUNT(i,j) = 0$
return COUNT(n, m)
Runtime Complexity: If we assume that the input only consists of the coordinates n and m, this will take $\Theta(nm)$ which is pseudo-polynomial. If the input consisted of the 2D array of all intersections, then the run time is polynomial in the size of the input.

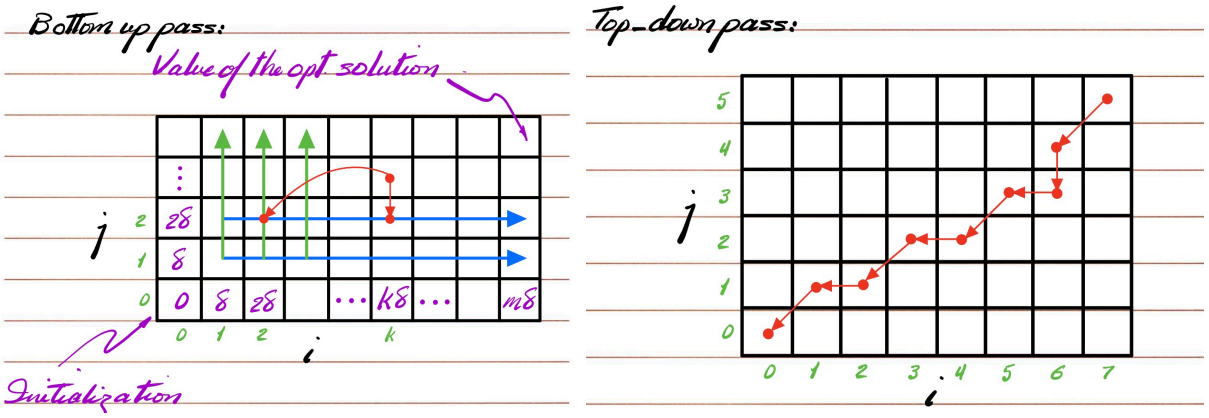
11 Sequence Alignment Problem

Given two DNA sequences S_1, S_2 consisting of base molecules A C G T only, compute their similarity score.
For example, $S_1 = ACCGGTCG, S_2 = CCAFFTGGC$, one possible alignment will be:
we need to come up with a concise definition of the similarity between two sequences.
Matching: a set of ordered pairs with property that each item occurs at most once.
However, a perfect matching does NOT indicate the two strings are similar, e.g. $S_1 = SEASIDE, S_2 = DISEASE$.
Alignment: a matching with no crossing pairs, i.e. in an alignment M if pairs $(i,j),(i',j') \in M, i < i'$ then $j < j'$
Cost of Alignment: incur a gap penalty of δ for each gap and α_{pq} for mismatching of letters p and q.
Similarity: the similarity of two sequences is the minimum cost of all possible alignments.



11.1 State Definition and State Transition

Define OPT(i, j): the minimum cost of an alignment between the first i letters of S_1 with the first j letters of S_2. Then in an optimal alignment M, at least one of the following holds:
If $(x_i, y_j) \in M$, then $OPT(i,j) = OPT(i-1, j-1) + \alpha_{x_i y_j}$
If $(x_i, -) \in M$, then $OPT(i,j) = OPT(i-1, j) + \delta$
If $(-, y_j) \in M$, then $OPT(i,j) = OPT(i, j-1) + \delta$
Recurrence Formula: $OPT(i,j) = \min(OPT(i-1, j-1) + \alpha_{x_i y_j}, OPT(i-1, j) + \delta, OPT(i, j-1) + \delta)$



Buttom Up Pass:

Initialize $OPT(0,j) = j\delta, OPT(i,0) = i\delta$
for i = 1 to m:
  for j = 1 to n:
    $OPT(i,j) = \min(OPT(i-1, j-1) + \alpha_{x_i y_j}, OPT(i-1, j) + \delta, OPT(i, j-1) + \delta)$
return $OPT(m,n)$

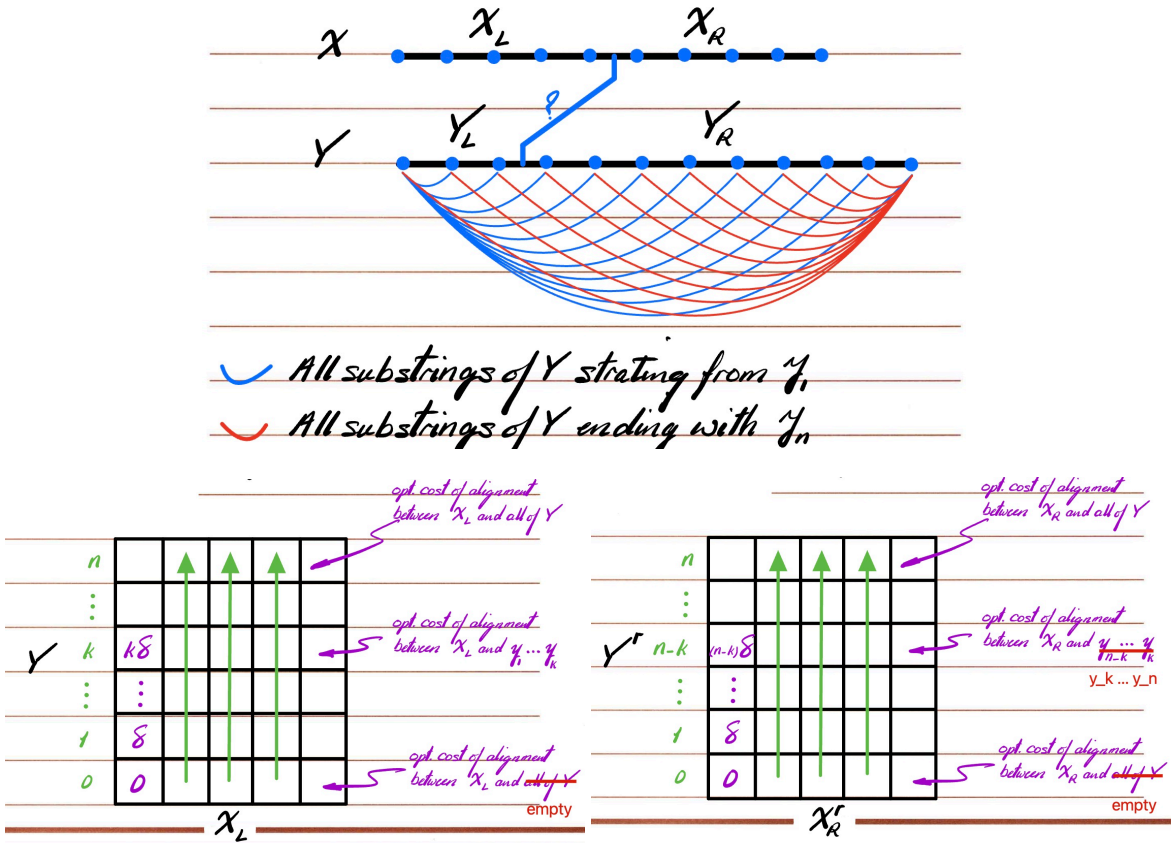
Runtime Complexity: If we assume that the input only consists of the two sequences S_1, S_2, this will take $\Theta(mn)$ which is polynomial time in the size of the input.
Memory Complexity: The space complexity is $\Theta(mn)$ because we need to store the values of all subproblems for getting the optimal Alignment by top down pass.

Top Down Pass:

i = m, j = n;
while(i > 0 and j > 0):
  if $OPT(i,j) = OPT(i-1, j-1) + \alpha_{x_i y_j}$:
    Print "Match" x_i "with" y_j, i = i - 1, j = j - 1;
  else if $OPT(i,j) = OPT(i-1, j) + \delta$:
    Print "Gap" x_i, i = i - 1;
  else:
    Print "Gap" y_j, j = j - 1;

11.2 Memory Efficient Solution by Divide and Conquer

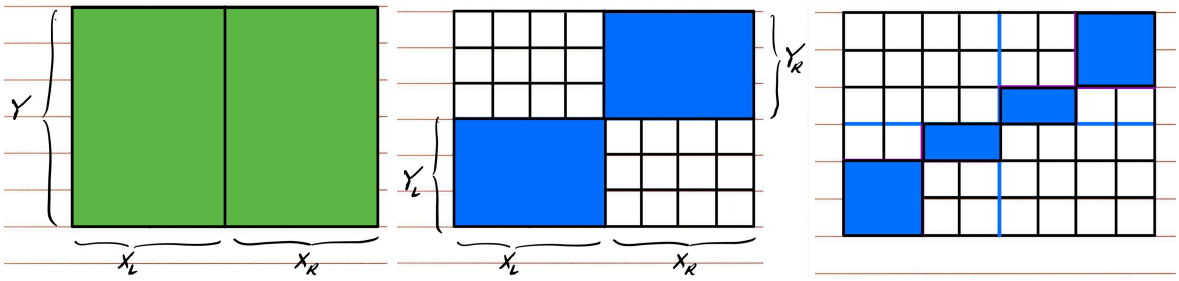
We can just split X into two halves, and find the best alignment of X_L, X_R with Y. Intuitively, we need to find the optimal split point in Y using Dynamic Programming with only **two columns memoery requirement**.
For Left part, we need to compute the optimal cost of the alignments between X_L and each of the substrings of Y starting from y_1. For Right part, we need to compute the optimal cost of the alignments between X_R and each of the substrings of Y starting from y_n. (Symmetric Dynamic Problem)



Then compute the total cost of the alignment between X_L and X_R with Y by trying all possible split points in Y.

$$\min_{0 \le k \le n} \{OPT(X_L, \{y_1, \dots, y_k\}) + OPT(X_R, \{y_{k+1}, \dots, y_n\})\}$$

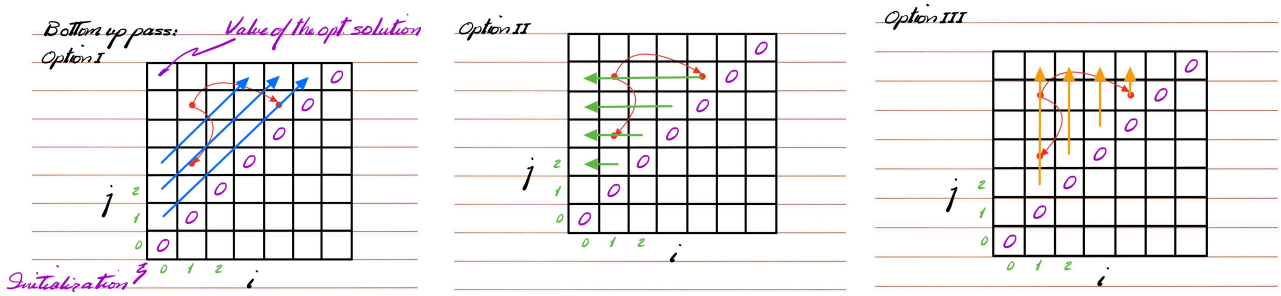
The k that minimizes the above expression is the **optimal split point** for this divide step.



Runtime Complexity: number of operations at the root level takes $\Theta(Cmn)$, at the second level takes $\Theta(Cmn/2)$, at the third level takes $\Theta(Cmn/4)$...
Total number of operations is $\Theta(Cmn + Cmn/2 + Cmn/4 + ...) = \Theta(2Cmn) = \Theta(mn)$.

12 Matrix Chain Multiplication Problem

Given a sequence of n dense matrices $A_1 \dots A_i \dots A_n$ along with their dimensions $r_i \times c_i$, find the order of matrix multiplications that minimizes the total number of element multiplications.
Recall that without using Strassen's algorithm, the number of multiplications needed to compute the product of two matrices $A_{m \times n}$ and $A_{n \times k}$ is mnk because we need n multiplications for each element in result $m \times k$ matrix.
Consider the sequence of matrices $A_1, A_2, \dots, A_n$, then the last multiplication in this sequence must be $(A_1 \cdots A_k) \times (A_{k+1} \cdots A_n)$ for some $1 \le k \le n$.
Define OPT(i, j): the minimum number of multiplications needed to compute the product of matrices $A_i \cdots A_j$.
Recurrence Formula: $OPT(i, j) = \min_{i \le k < j} \{OPT(i, k) + OPT(k + 1, j) + r_i c_k c_j\}$
Initialization: $\forall i \in [1, n], OPT(i, i) = 0$ and $\forall i \in [1, n), OPT(i, i + 1) = r_i c_i c_{i+1}$
Bottom Up Pass:



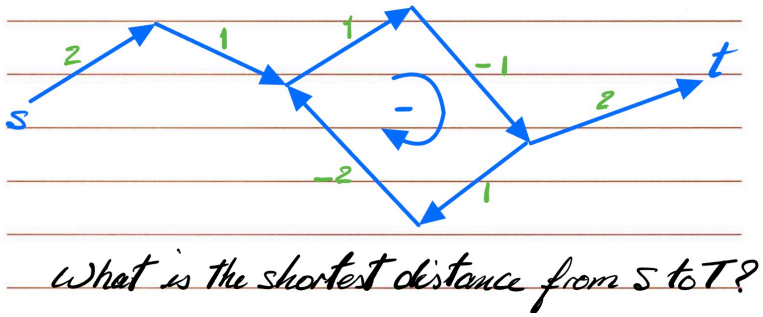
Buttom Up Pass for Option II and III:

for i = 1 to n: OPT(i, i) = 0
for j = 2 to n:
  for i = j-1 to 1 by -1:
    $OPT(i,j) = \min_{i \le k < j} \{OPT(i,k) + OPT(k+1,j) + r_i c_k c_j\}$
return $OPT(1,n)$
for i = n-1 to 1 by -1:
  for j = i+1 to n:
    $OPT(i,j) = \min_{i \le k < j} \{OPT(i,k) + OPT(k+1,j) + r_i c_k c_j\}$
return $OPT(1,n)$

Time Complexity: If we assume that the input only consists of the sequence of n matrices with their size, this will take $\Theta(n^3)$ which is polynomial time in the size of the input.

13 Shortest Path Problem for Graphs wtih Negative Weight Edges

There are no shortest path algorithms for graphs with **negative weight cycles**, because the shortest path is not well-defined. If G has no negative cycles then there is a simple (no self cycle) shortest path from s to t and has at most (n-1) edges.

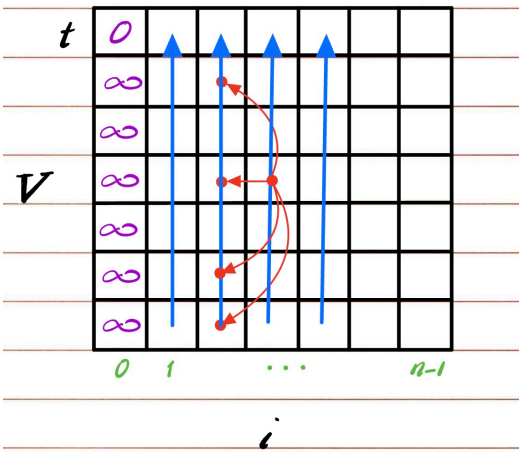


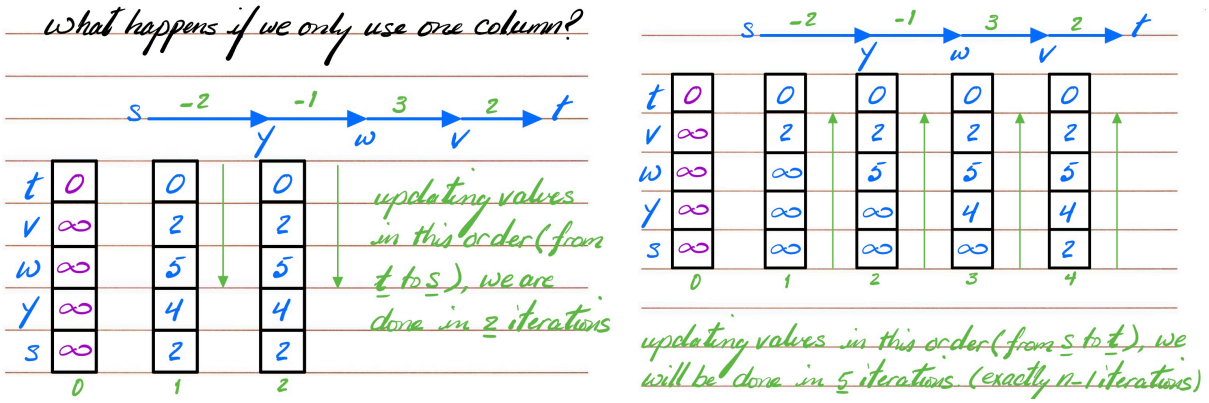
Define OPT(i, v): the shortest path from v to t with at most i edges. Thus we need to find $OPT(n-1, s)$.
Recurrence Formula: $OPT(i,v) = \min(OPT(i-1,v), \min_{w \in Adj(v)} \{OPT(i-1,w) + C_{vw}\})$

Bellman-Ford Algorithm (G, s, t):

n = the number of vertices in G
Initialize $OPT(0,v) = \infty$ for all $v \in V - \{t\}$, $OPT(0,t) = 0$
for i = 1 to n-1: (Iterate n-1 times to get path of at most n-1 edges)
  for all $v \in V$ in any order:
    $OPT(i,v) = \min(OPT(i-1,v), \min_{w \in Adj(v)} \{OPT(i-1,w) + C_{vw}\})$
return $OPT(n-1,s)$

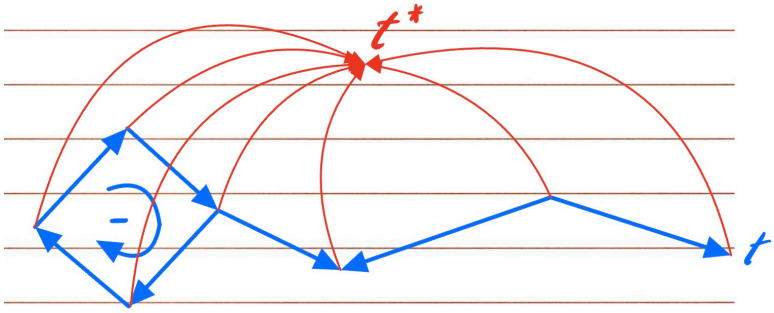
Time Complexity: the time complexity is $\Theta(nm)$ where n is the number of vertices and m is the number of edges, since the total number of comparisons in the inner loop can't exceed m.
So to get faster propogation of information, we need to update the nodes in increasing order of their distance to destination t, i.e. in order of BFS Tree level of root t.





Detect Negative Weight Cycles

If there is **a path from the negative cycle to t**, distance from nodes on the cycle to t will be $-\infty$. Thus we can detect negative weight cycles by running the Bellman-Ford algorithm. If there is a change in the distance of any node after n-1 iterations, then there is a negative weight cycle.
However, if there is **no such path from the negative cycle to t**, the initial distance of ∞ will never go down after n-1 iterations. To ensure all negative weight cycles are detected, we need to **add a virtual destination node** t^* and add edges from all nodes to t^*.



Bellman-Ford VS Dijkstra's Algorithm

They takes $\Theta(mn), \Theta(mlgn)$ respectively, where n is the number of vertices and m is the number of edges.
But this doesn't mean that Dijkstra's algorithm is always faster than Bellman-Ford algorithm for positive weight graphs. Bellman-Ford algorithm is more suitable for distributed parallel processing. There are three cost components in distributed parallel algorithm: **communication cost (I/O Cost), computation cost (CPU Cost), synchronization cost (Message Passing Cost)**. The message passing cost is much slower than I/O cost.
In each iteration of Bellman-Ford algorithm, there are 2m messages at every iteration because for each edge we need the information passing from and to both ends. This is preaty expensive. Instead, we only need the **node with recently updated distance** to send out messages. Start from t, and send out message to all its neighbors, then the neighbors propogate this information out to their neighbors, and so on.

14 Expanded 0/1 Knapsack Problem

For the upcoming semester, we have S students who want to volunteer their time, and B buses to help get them between campus and the location of their volunteering. **There are F projects under consideration: project i requires s_i student and b_i buses to accomplish, and will generate $g_i > 0$ units of goodwill for the university.** Our goal is to maximize the goodwill generated for the university subject to these constraints. Note that each project must be undertaken entirely or not done at all.
Define OPT(i, s, b): the maximum goodwill generated by the first i projects with at most s students and b buses.
Recurrence Formula: $OPT(i, s, b) = \max(OPT(i-1, s, b), g_i + OPT(i-1, s-s_i, b-b_i))$
Initialization: $\forall i \in [1, F], j \in [1, S], k \in [1, B]\ OPT(0, j, k) = OPT(i, 0, k) = OPT(i, j, 0) = 0$

```
Bottom Up Pass:
for i = 1 to F:
    for j = 1 to S:
        for k = 1 to B:
            If s_i > j or b_i > k: OPT(i, j, k) = OPT(i − 1, j, k)
            Else: OPT(i, j, k) = max(OPT(i − 1, j, k), g_i + OPT(i − 1, j − s_i, k − b_i))
return OPT(F, S, B)
```

Time Complexity: If we assume that the input only consists of the sequence of F projects with their requirements, this will take $\Theta(FSB)$ which is pseudo-polynomial because S and B represent numerical values of the input terms rather than the size of the input.

```
Top Down Pass Θ(F):
i = F, j = S, k = B;
while(i > 0 and j > 0 and k > 0):
    if OPT(i, j, k) = OPT(i − 1, j, k):
        Print "Project" i "not taken", i = i - 1;
    else:
        Print "Project" i "taken", i = i - 1, j = j - s_i, k = k - b_i;
```

15 Relationship Tree Problem

The corporation has a hierarchical ranking structure. Suppose you are organizing a company party, and **you will not invite any employee whose immediate superior is invited. Each employee j has a positive value** v_j, representing how enjoyable their presence would be at the party. Our goal is to determine which employees to invite, subject to these constraints, to maximize the total value of invitees.
Define OPT(i): the maximum value of the invitees of the subtree rooted at invitee i.
Recurrence Formula: say c_i are all the children of i, g_i are all the grandchildren of i

$$OPT(i) = \max(\sum_{c \in c_i} OPT(c), \sum_{g \in g_i} OPT(g) + v_i)$$

Initialization: for all Leaf nodes of tree: $OPT(i) = v_i$
Memory Efficiency: For each node, we need to store both $OPT(i)$ and $\sum_{c \in c_i} OPT(c)$ (0 if no children).

```
Bottom Up Pass Θ(n):
For each node in descending order of BTS Tree level:
    OPT(i) = max(∑_{c∈c_i} OPT(c), ∑_{g∈g_i} OPT(g) + v_i)
return OPT(root)
```

```
Top Down Pass Θ(n):
def Find_Invitees(i):
    if OPT(i) > ∑_{c∈c_i} OPT(c):
        Output i
        for all g in g_i: Find_Invitees(g)
    else:
        for all c in c_i: Find_Invitees(c)
Find_Invitees(root)
```

16 Pile Boxes Problem

You are given a set of n types of rectangular 3-D boxes, where the i^{th} box has height h_i, width w_i and length l_i (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only **stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly (both length and width) larger than those of the 2-D base of the higher box.** Of course, it's allowable to use multiple instances of the same type of box, by rotating a box so that any side functions as its base.
First remove the complexity related to the rotation of the boxes by creating 3*n box types as follows: Given a box with sides measuring X < Y < Z, we will create 3 box types (Length, Width, Height): (X, Y, Z), (X, Z, Y), (Y, Z, X), remember only Height matters, Width and Length are symmetric.
Then sort and reindex the 3*n boxes based on decreasing base area (Width * Depth).
Define OPT(i): the maximum height of the stack of boxes with the i^{th} box at the top.
Recurrence Formula:

$$OPT(i) = h_i + \max_{0<j<i \ \&\& \ l_j>l_i \ \&\& \ w_j>w_i} \{OPT(j)\}$$

If no compatible j < i, such that $l_j > l_i \ \&\& \ w_j > w_i$, then the second maximum term is zero.

Bottom Up Pass:

for i = 1 to 3n:
$OPT(i) = h_i + \max_{0<j<i \ \&\& \ l_j>l_i \ \&\& \ w_j>w_i} \{OPT(j)\}$
$Base(i) = j$ corresponding to the id of the box we are putting i on, or 0 if i is the base box.
return $\max_{1 \le i \le 3n} \{OPT(i)\}$

Time Complexity: If we assume that the input only consists of the sequence of n boxes with their dimensions, this will take $\Theta(3n * 3n) = \Theta(n^2)$ which is polynomial time in the size of the input.

Top Down Pass:

Suppose i is the id of the box with the maximum OPT value.
Output i
While(Base(i) != 0):
    i = Base(i)
    Output i

Time Complexity: This will take $\Theta(n)$ because we have **stored the base of each box** in the bottom up pass. Otherwise, it will take $\Theta(n^2)$ to recompute the base of each box in the top down pass.