Dynamic Programming

Weighted Interval Scheduling, Matrix Chain Multiplication, Longest Common Subsequence, Sequence Alignment,

General Comments

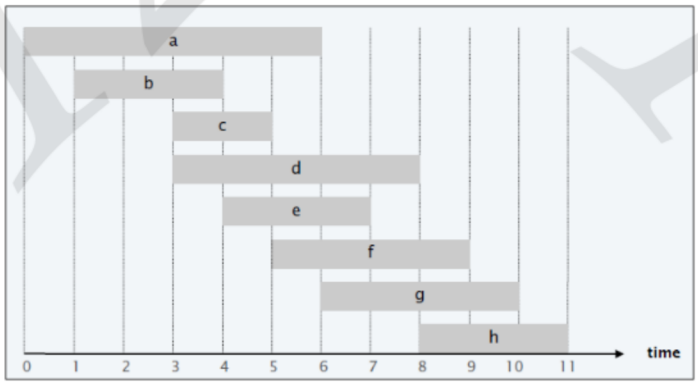
- Three Algorithmic Paradigms:
- **Greedy**. Build up a solution incrementally, myopically optimizing some local criterion.
- **Divide-and-conquer**. Break up a problem into independent subproblems, solve each subproblem, and combine solution to subproblems to form solution to original problem.
- **Dynamic programming**. Break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems. ["Programming" in this context is not writing computer codes, just a tabular way to remember past partial results]
- **Notes:**
  - D&C can be used for problems with overlapping subproblems, but, almost always that would be more expensive (depending on amount of overlap) – an extreme example is computing n^{th} Fibonacci number.
  - D&C is a top down approach while DP is bottom up.
  - Recursion is elegant but almost always more expensive to implement (too many function calls and associated stack processing and other issues).
  - D&C, in many cases, is intuitive, but a problem must have certain characteristics for DP to be a viable solution technique. It is generally easier to rewrite DP solutions iteratively than D&C solutions.

Dynamic Programming Applications

- **Areas.**
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations research.
  - Computer science: theory, graphics, AI, compilers, systems, ….
  - ...
- **Some famous dynamic programming algorithms.**
  - Unix diff for comparing two files.
  - Viterbi for hidden Markov models.
  - De Boor for evaluating spline curves.
  - Smith-Waterman for genetic sequence alignment.
  - Bellman-Ford for shortest path routing in networks.
  - Cocke-Kasami-Younger for parsing context-free grammars.
  - ...

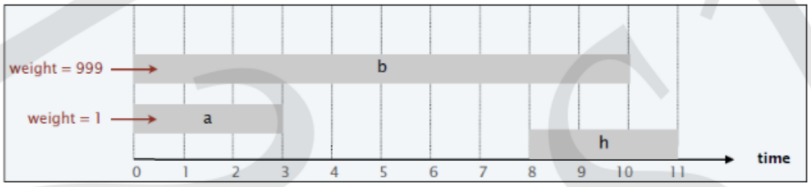
Weighted Interval Scheduling

- **Input:**
  - Job j starts at s_j, finishes at f_j, and has weight or value v_j.
  - Two jobs compatible if they don't overlap.
- **Goal:** find maximum weight subset of mutually compatible jobs.
- **Note:** This is a generalization of simple interval scheduling where each job (activity) has equal value.



Earliest-finish-time first algorithm

- **Earliest finish-time first.**
  - Consider jobs in ascending order of finish time [i.e., we need a presorting, O(n log n)]
  - Add job to subset if it is compatible with previously chosen jobs.
- **Recall.** Greedy algorithm is correct if all weights are 1.
- **Observation**. Greedy algorithm fails spectacularly for weighted version.

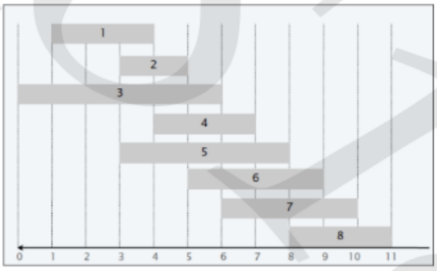


Weighted Interval Scheduling

- **Presort:** Label jobs by their finishing time $f_1 \leq f_2 \ldots \leq f_n$.
- **Define** p(j) = largest index i < j such that job i is compatible with j. Example: p(1) = 0, p(2) = 0, p(3) = 0, p(4) = 1, p(5) = 0, p(6) = 2, p(7) = 3, p(8) = 5. How to compute the p(·) array and what is complexity?

Observations: Assuming an optimal solution Opt, we have 2 possibilities:

1. last interval n ∈ Opt: no interval strictly between p(n) and n can belong to Opt. Also, Opt must include an optimal solution to the problem consisting of requests {1, . . . , p(n)}.
2. last interval n ∉ Opt: Opt is simply equal to the optimal solution to the problem consisting of requests {1, . . . , n − 1}.



Thus, the optimal solution on intervals {1, 2, . . . , n} involves looking at the optimal solutions of smaller problems of the form {1, 2, . . . , j}.

For any value of j between 1 and n, let Opt_j denote the optimal solution to the problem consisting of requests {1, . . . , j}, and let OPT(j) denote the value of this solution. (We define OPT(0) = 0, based on the convention that this is the optimum over an empty set of intervals.) The optimal solution we're seeking is precisely Opt_n, with value OPT(n).

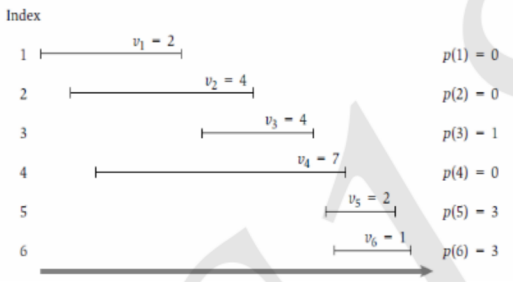
Weighted Interval Scheduling

- **Presort:** Label jobs by their finishing time $f_1 \leq f_2 \ldots \leq f_n$.

Dynamic Programming: Binary Choice

Notation: OPT(j) = optimal solution value for jobs 1, 2, …, j. Observe: There are 2 possible cases:

- **Case 1:** OPT selects job j collects profit v_j, can't use incompatible jobs {p(j) + 1, p(j) + 2, …, j – 1}, Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j).
- **Case 2:** OPT does not select job j Must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j – 1.
- Both cases follow from optimal substructure definition, proof via exchange argument. Easy recursive formulation



Another Example 2

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{v_j + OPT(p(j)), OPT(j-1)\} & \text{otherwise} \end{cases}$$

Weighted interval scheduling: brute force

- Easy to write the code.

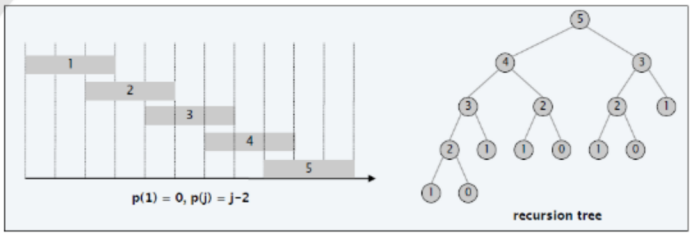
Input: n, s[1..n], f[1..n], v[1..n]

Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n]. [O(n log n)]

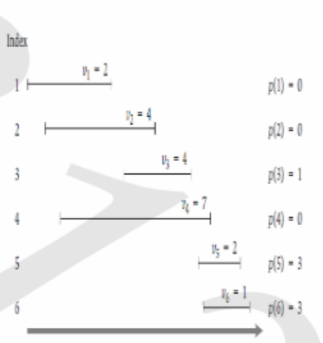
Compute p[1], p[2], …, p[n].

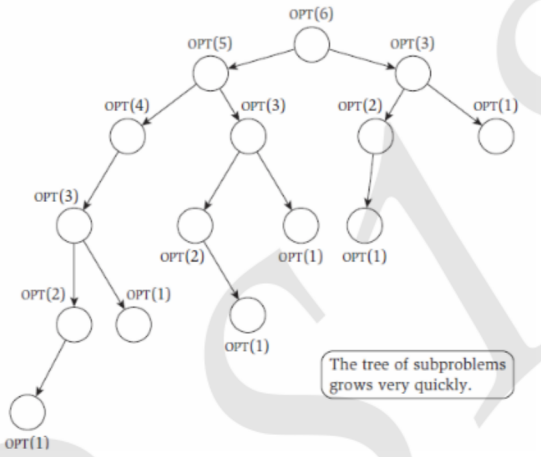
```
Compute-Opt(j)
if j = 0
    return 0.
else
    return max(v[j] + Compute-Opt(p[j]), Compute-Opt(j–1)).
```

- This is not acceptable, because of overlapping subproblems exponential number of recursive calls exponential execution time (like simple minded formulation of Fibonacci numbers)



Example 2





Improved Recursion via Memoization

- **Memoization**: Store results of each subproblem, almost always in a 1D array or 2D array; lookup as needed. [To determine OPT(n), we invoke M-Compute-Opt(n)].

```
Input: n, s[1..n], f[1..n], v[1..n]
Sort jobs by finish time so that f[1] ≤ f[2] ≤ … ≤ f[n].  [O(n log n)]
Compute p[1], p[2], …, p[n].
for j = 1 to n {set M[j] = −1;} M[0] = 0; // M is a global array
M-Compute-Opt(j)
    if M[j] is −1
        M[j] ← max(v[j] + M-Compute-Opt(p[j]), M-Compute-Opt(j − 1)).
    return M[j].
```

Claim. Memoized version of algorithm takes O(n log n) time.

- Sort by finish time: O(n log n).
- Computing p(·) : O(n log n) via sorting by start time.
- M-COMPUTE-OPT(j): each invocation takes O(1) time and either
  - (i) returns an existing value M[j]
  - (ii) fills in one new entry M[j] and makes two recursive calls
- Progress measure Φ = # nonempty entries of M[].
  - initially Φ = 0, throughout Φ ≤ n.
  - (ii) increases Φ by 1 ⇒ at most 2n recursive calls.
- Overall running time of M-COMPUTE-OPT(n) is O(n).
- **Remark**. O(n) if jobs are presorted by start and finish times.

Weighted interval scheduling: finding a solution

- DP algorithm computes optimal value. How to find the actual solution itself? It's easy. Once the optimal value is computed by DP, using memoization, (i.e., the array M[] has been computed, make a second pass on the array M[].

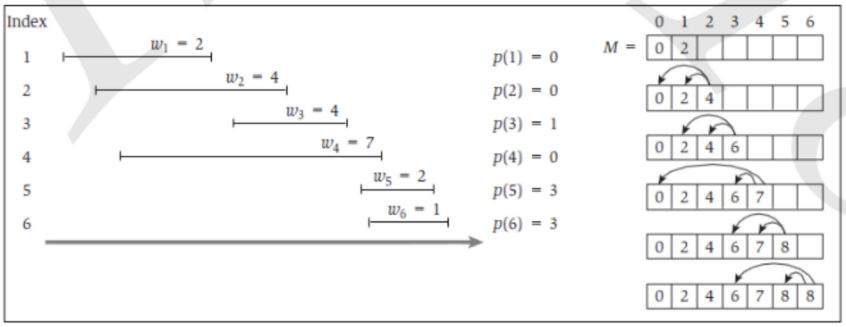
```
Find-Solution(j)
if j = 0
    return ∅.
else if (v[j] + M[p[j]] > M[j–1])
    return { j } ∪ Find-Solution(p[j]).
else
    return Find-Solution(j–1).
```

- Analysis. # of recursive calls ≤ n ⇒ O(n). Hand execute the algorithm to get a feeling how the algorithm works; we will expand this idea in the next few example problems.

Weighted interval scheduling: bottom-up DP

- In this case just unwind the recursion.

```
BOTTOM-UP (n, s1, …, sn , f1, …, fn , v1, …, vn)
Sort jobs by finish time so that f1 ≤ f2 ≤ … ≤ fn.
Compute p(1), p(2), …, p(n).
M [0] ← 0.
FOR  j = 1 TO n
M [ j ] ← max { vj + M [ p(j) ], M [ j – 1] }.
```



*Matrix Chain Multiplication**

- We are given a sequence (chain) $A_1, A_2, \ldots, A_n$ of n matrices to be multiplied, and we wish to compute the product $A_1 \times A_2 \times \ldots, An$ so that the number of element products is minimized.
- Facts: Assume matrices are compatible (conformable) [otherwise you cannot multiply]
  - Matrix multiplication is associative – i.e., $A \times B \times C = A \times (B \times C) = (A \times B) \times C$ – the more the number of matrices in the chain, more is the number of choices.
  - The number of products to multiply two matrices of orders $(m \times n)$ and $(n \times p)$ is O(mnp).
  - Consider 3 matrices – A $(m \times n)$, B $(n \times p)$, C $(p \times q)$, say m =p =10, q = n = 2
    - #products in (AB)C = mnp +mpq = mp(n +q) = 400
    - #products in A(BC) =npq +mnq = nq(p + m) = 80
    - If the number of matrices in the chain is n, the possible number of ways of parenthesizing is known to be exponential, $\Omega(4^n/n^{3/2}) = \Omega(2^n)$
    - Thus, computing the product according to some parenthesization may be 100, 1000 times faster than another.
  - **Note** that in the matrix-chain multiplication problem, we are not actually multiplying matrices. *Our goal is only to determine an order for multiplying matrices that has the lowest cost.*

*Adopted from CLRS Chapter 15

Counting the number of parenthesizations

- Before solving the matrix-chain multiplication problem by dynamic programming, let us convince ourselves that exhaustively checking all possible parenthesizations does not yield an efficient algorithm.
- Denote the number of alternative parenthesizations of a sequence of n matrices by P(n).
- When n = 1, we have just one matrix and therefore only one way to fully parenthesize the matrix product.
- When $n \geq 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between the k^{th} and $(k + 1)^{st}$ matrices for any k = 1,2, …, n-1. Thus, we obtain the recurrence

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$ with a solution $$Catalan_n = \frac{1}{n+1}\binom{2n}{n}$$

- Catalan numbers grow as $\Omega(4^n/n^{3/2}) = \Omega(2^n)$. Thus, exhaustive search is a very poor strategy.

Optimal Substructure of the problem

- We use $A_{i\ldots j}$ where $i < j$ to denote the matrix that is the result of the product $A_iA_{i+1}\ldots A_j$.
- Suppose that to optimally parenthesize $A_iA_{i+1}\ldots A_j$ we split the product between A_k and A_{k+1} for some integer k in the range $i \le k \le j$, i.e., for some value of k, we first compute the matrices $A_{i\ldots k}$ and $A_{k+1\ldots j}$ and then multiply them together to produce the final product $A_{i\ldots j}$. The cost of parenthesizing this way is the cost of computing the matrix $A_{i\ldots k}$, plus the cost of computing $A_{k+1\ldots j}$, plus the cost of multiplying them together.
- The optimal substructure of this problem is as follows. Suppose that to optimally parenthesize $A_iA_{i+1}\ldots A_j$, we split the product between A_k and A_{k+1}. Then the way we parenthesize the "prefix" subchain $A_iA_{i+1}\ldots A_k$ within this optimal parenthesization of $A_iA_{i+1}\ldots A_j$ must be an optimal parenthesization of $A_iA_{i+1}\ldots A_k$. Why?
- We can build an optimal solution to an instance of the matrix-chain multiplication problem by splitting the problem into two subproblems (optimally parenthesizing $A_iA_{i+1}\ldots A_k$ and $A_{k+1}\ldots A_j$), finding optimal solutions to subproblem instances, and then combining these optimal subproblem solutions.
- We must ensure that when we search for the correct place to split the product, we have considered all possible places, so that we are sure of having examined the optimal one.
- Assume we maintain an array p such that each matrix A_i is of order $p_{i-1} \times p_i$.

A Recursive Solution

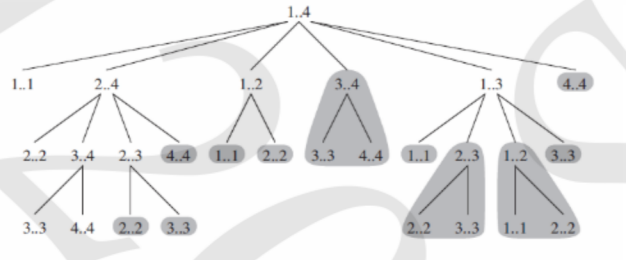
- We choose as our sub problems the problems of determining the minimum cost of parenthesizing $A_iA_{i+1}\ldots A_j$ for $1 \le i \le j \le n$. Let m[i, j] be the minimum number of scalar multiplications needed to compute the matrix $A_{i\ldots j}$ for the full problem, the optimum way to compute $A_{i\ldots n}$ would thus be m[1, n].
- We observe m[i, j] = 0 if i = j since there is nothing to do with a single matrix. Thus, m[i, i] = 0 for all i.
- To compute m[i, j], we use recursion: Let us assume that to optimally parenthesize, we split the product $A_iA_{i+1}\ldots A_j$ between A_k and A_{k+1}, where $i \le k < j$. Then, m[i, j] equals the minimum cost for computing the sub-products $A_{i\ldots k}$ and $A_{k+1\ldots j}$, plus the cost of multiplying these two matrices together. We get m[i, j] = m[i, k] + m[k=1, j] $= p_{i-1}p_kp_j$.

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j}\{m[i,k] \quad m[k \quad 1,j] \quad p_{i \ 1}p_kp_j\} & \text{if } i < j \end{cases}$$

- It is now easy towrite the pseudocode. We have two drawbacks:
  - The recursive solution is exponential due to overlapping sub problems.
  - The m[i, j] values give the costs of optimal solutions to sub problems, but they do not provide all the information we need to construct an optimal solution
- We will see how dynamic programming solves both the problems.

Recursive Matrix Chain

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i == j
2      return 0
3  m[i, j] = ∞
4  for k = i to j − 1
5      q = RECURSIVE-MATRIX-CHAIN(p, i, k)
           + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
           + p_{i−1} p_k p_j
6      if q < m[i, j]
7          m[i, j] = q
8  return m[i, j]
```



Dynamic Programming Approach

- We compute the optimal cost by using a tabular, bottom-up approach (dynamic programming approach) [one can also use memorization to implement a top down approach]. The idea is, as we have seen before, to store the result of any sub problem, as soon as it is computed, in order to be able to use it later if need be.
- As before, we use an array p[0 … n] of length n+1 such that the matrix A_i has the dimension $p_{i-1} \times p_i$.
- We use an auxiliary table m[1... n][1..n] to store the m[i, j] costs and another auxiliary table s[1... n – 1][2... n] that records which index of k achieved the optimal cost in computing m[i, j]. We use the table s to construct an optimal solution.
- We have seen cost m[i, j] of computing a matrix-chain product of j – i + 1matrices depends only on the costs of computing matrix-chain products of fewer than j – i + 1 matrices.
- Thus, the algorithm should fill in the table min a manner that corresponds to solving the parenthesization problem on matrix chains of increasing length.
- See the pseudo-code of the DP algorithm that determines the optimal number of scalar multiplications needed to compute a matrix-chain product, it does not directly show how to multiply the matrices.
- The table s[1... n – 1][2... n] will give us info to do so.

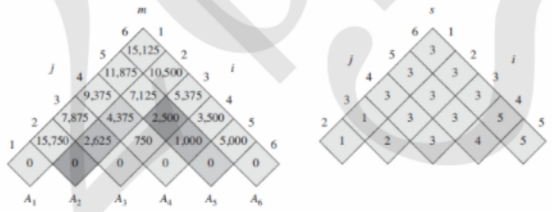
Pseudo Code

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

- It is easy, by looking at the nested loop structure, to claim that the running time of the algorithm is $O(n^3)$; each loop index takes on at most n – 1 values. Also the algorithm uses $\Theta(n^2)$ space to store the m and s tables.

An Example

- n = 6 and p[.] = [30, 35, 15, 5, 10, 20, 25]



The tables are rotated so that the main diagonal runs horizontally. The m table uses only the main diagonal and upper triangle, and the s table uses only the upper triangle. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 10 when computing

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases} = 7125\,.$$

The algorithm first computes $m[i, i] = 0$ for $i = 1, 2, \ldots, n$ (the minimum costs for chains of length 1) in lines 3–4. It then uses recurrence (15.7) to compute $m[i, i + 1]$ for $i = 1, 2, \ldots, n - 1$ (the minimum costs for chains of length $l = 2$) during the first execution of the **for** loop in lines 5–13. The second time through the loop, it computes $m[i, i+2]$ for $i = 1, 2, \ldots, n-2$ (the minimum costs for chains of length $l = 3$), and so forth. At each step, the $m[i, j]$ cost computed in lines 10–13 depends only on table entries $m[i, k]$ and $m[k + 1, j]$ already computed.

Constructing an optimal solution

- We need to explicitly construct the sequence of products for the optimum solution. The table s[1... n – 1][2... n] in the previous algorithm gives us info to do so. Each entry s[i,j] records a value k such that an optimal parenthesization of A_iA_{i+1} …A_j splits the product between A_k and A_{k+1} .
- Thus, we know that the final matrix multiplication in computing $A_{1...n}$ optimally is $A_{1..s[1,n]}A_{s[1,n]+1...n}$. We can determine the earlier matrix multiplications recursively, since s[1,s[1,n]] determines the last matrix multiplication when computing $A_{1..s[1,n]}$ and s[s[1,n]+1,n]] determines the last matrix multiplication when computing $A_{s[1,n]+1...n}$.
- Here is a brief sketch of the pseudocode; it should be invoked as Print_OS(s,1,n). It's straightforward to write an iterative version and to include the storing the solution as a character string.

```
Print_OS(s, i, j)
if i = j then print "A"i
 else print "("
      Print_OS (s, i, s[i, j])
      Print_OS (s, s[i,j]+1, j)
       print ")".
```

The Longest Common Subsequence Problem

- A common text processing problem is to test the similarity [similarity can be defined in many different ways] between two text strings. In molecular biology, genetics applications, the two strings correspond to two strands of DNA, which could, for example, come from two individuals, who we will consider genetically related if they have a long subsequence common to their respective DNA sequences. Other applications include software engineering (version control), web crawlers, File compression (diff in Linux).
- String matching, in general, is a fundamental problem in text editing. [Think of "grep" when we perform search]
- Consider any finite alphabet Σ [Example: {a, c, t, g} in genomics]. A string Y over Σ is a finite sequence of symbols from Σ; |Y| denotes length of Y, Y is represented by an array Y[0… m-1] where m =|Y|, XY denotes concatenation 2 strings X and Y [X.Y].
- Y[i:j], $0 \le i \le j$, denotes a **substring** of Y. Y[0… i], $i \le m$ is a **prefix** of Y; Y[j:m], $j \le m-1$, is a **suffix** of Y. Example: if Y = "abcdf", then, abc, a, bcd, cdf are all substrings of Y while bdf, acd, cf are not substrings of Y.
- A string P [called pattern] occurs in T [called text] at position i [or, P matches T at position i], iff P(j) = T(i+j–1), for $1 \le j \le |X|$
- Examples: (1) Let Σ = {a, b, c}, and let T = aabcabccaa. Then, P = abc is a substring of T that occurs at positions 1 and 4. The prefix T(0:4) is the substring aabca, and the suffix T(4:9) is the substring abccaa. Note: If the pattern P is a substring of the text T, then we just need to identify the start position of P in T

Longest Common Subsequence (LCS)

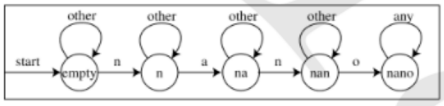
- An important way to measure the "similarity" between two stings, over a finite alphabet is the length of the longest common subsequence (LCS) between the two strings. Given a string X of size n, a subsequence of X is any string that is of the form $X[i_1]X[i_2] \cdots X[i_k]$, $i_j < i_{j+1}$ for $j = 1, \ldots, k-1$; it is a sequence of characters that are not necessarily contiguous but are nevertheless taken in order from X. For example, the string AAAG is a subsequence of the string CGATAATTGAGA. Note that the concept of subsequence of a string is different from the one of substring of a string.
- The specific text similarity problem we address here is the longest common subsequence (LCS) problem. In this problem, we are given two character strings, X of size n and Y of size m, over some alphabet and are asked to find a longest string S that is a subsequence of both X and Y . If S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA and S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA, then the longest strand S_3 is GTCGTCGGAAGCCGGCCGAA. If X = (ABCBDAB) and Y = (BDCABA), the sequence (BCA) is a common subsequence of both X and Y, but not an LCS of X and Y, since it has length 3 and the sequence (BCBA), which is also common to both X and Y, has length 4. The sequence (BCBA) is an LCS of X and Y, as is the sequence (BDAB) [*since X and Y have no common subsequence of length 5 or greater*].
- *Applications*: DNA sequence analysis [alphabet = (A, G, C, T)*, length of strings is in hundreds of millions], web crawlers, version control in software engineering, etc.

*adenine, guanine, cytosine, and thymine

A simpler Problem

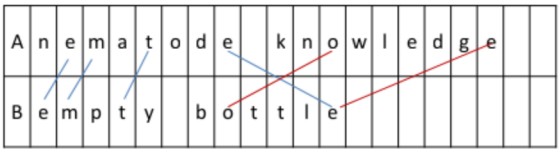
- If X = (abracadabra) and Y = (aadaa), Y is a subsequence of X. Given two strings X and Y, the longest common subsequence of X and Y is a longest sequence Z which is both a subsequence of X and Y. If X = (abracadabra) and Y = (yabbadabbadoo), the longest common subsequence is (abadaba). It is not always unique – LCS of (abc) and (bac) is either (ac) or (bc).
- A much simpler problem first: Consider two character arrays: P[m], T[n]; we want to test if P[m] is a subsequence or not, assuming m < n. [in applications, m ≪ n]

```
int main(void){
char P[45] = "nematode knowledge";
char T[10] = "nano";
while (*T != '\0')
 if (*P == (*T)++ && ++*P == '\0')
{printf ("TRUE\n");  return(0);}
 printf ("FALSE\n"); return (0); }
```



Some Simple Observations

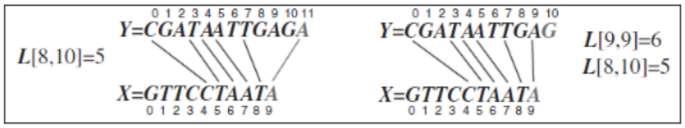
- Consider two arbitrary strings, one top of the other



- Observe several important facts:
  - If we draw the lines such that no line crosses any others, we get a common subsequence (not necessarily the longest though!); we can get many; which one is the longest?
  - If the two strings A and B start with the same letter, it's always safe to choose that starting letter as the first character of the subsequence. Why?
  - If the two first characters differ, then it is not possible for both of them to be part of a common subsequence – one or the other (or maybe both) will have to be removed.
  - Once we've decided what to do with the first characters of the strings, the remaining subproblem is again a longest common subsequence problem, on two shorter strings. Therefore we can solve it recursively.
  - It is much easier to compute the length of the LCS than the LCS itself by recursion; let's do that first.

LCS Substructure Property

- We define a subproblem, therefore, as that of computing the length of the longest common subsequence of X[0..i] and Y [0..j], denoted L[i, j]. We want to rewrite L[i, j] in terms of optimal subproblem solutions. There are two cases:
  - **Case 1**: X[i] = Y[j]. Let c = X[i] = Y[j]. We claim that a LCS of X[0..i] and Y [0..j] ends with c. To prove this claim, assume it is not true. Then, there exists some longest common subsequence X[i1]X[i2] . . . X[ik] = Y[j1]Y[j2] . . . Y[jk]. If X[ik] = c or Y [jk] = c, then we get the same sequence by setting ik = i and jk = j. Alternately, if X[jk] ≠ c, then we can get an even longer common subsequence by adding c to the end →←. Thus, a longest common subsequence of X[0..i] and Y [0..j] ends with c = X[i] = Y [j]. Therefore, we set L[i, j] = L[i − 1, j − 1] + 1 if X[i] = Y [j].
  - **Case 2:** X[i] ≠ Y[j]. In this case, we cannot have a common subsequence that includes both X[i] and Y[j]. That is, a common subsequence can end with X[i], Y[j], or neither, but not both. Therefore, we set L[i, j] = max{L[i − 1, j], L[i, j − 1]} if X[i] ≠ Y[j].
- **Note**: In order to make the Equations make sense in the boundary cases when i = 0 or j = 0, we define L[i,−1] = 0 for i = −1, 0, 1, . . . , n−1 and L[−1, j] = 0 for j = −1, 0, 1, . . . , m − 1.



Length of LCS by Recursion

```
//In the LCS problem, subproblems consist of a pair of suffixes of the two input strings.
int max (int a, int b){
if (a > b) return a;
else return b; }
int lcs_length (char *A, char *B){
if (*A == '\0' || *B == '\0') return 0;
else if (*A == *B) return (1 + lcs_length(A+1, B+1));
else return max(lcs_length(A+1, B) , lcs_length(A, B+1)); }
```

Pointer Arithmetic

- We assume A and B contain valid strings. Can you rewrite to get LCS of any two subarrays of a given array? **Try**
- It is similar to Fibonacci series; computing the same function again and again and over again.

```
int main(){
char A[20] = "naematode knowledge";
char B[20] = "nanoee";
printf ("Length of LCS of A and B is %d\n", lcs_length (A, B));
return 0 ;
```

- Since each character of A is either in or not in a subsequence, there are potentially 2^n different subsequences of A, each of which requires O(m) time to determine whether it is a subsequence of B. Thus, the brute-force approach yields an exponential algorithm that runs in $O(2^n m)$ time, which is very inefficient.

Length of LCS using iteration

```
1.  int lcs_length (char *A, char *B){
2.  // Need a 2-D int array L,
3.  int i, j; int *L;
4.  int m = strlen(A), n = strlen(B);
5.  L = (int *) malloc ((m)*(n)*sizeof(int));
6.    for (i=m;i>= 0; i--)
7.       for (j=n; j>= 0;j--)
8.       {
9.         if (A[i] == '\0' || B[j]=='\0') L[i*m+j] = 0;
10.        else if (A[i]==B[j]) L[i*m+j]= 1+L[(i+1)*m+j+1];
11.        else L[i*m+j]= max(L[(i+1)*m+j],L[i*m+j+1]);
12.      }
13. return L[0]; }
```

- Think of how 2-D arrays are stored in computer's memory.
- It's easier, less error prone, to think of them as an array [read the matrix in a row major way] and adjust the index manipulations.

1. The steps 1 – 5 allocates a 2-D array L of size #rows and #columns, which are respectively the lengths of the two strings – doesn't matter which one is which. Everything else follows directly from the recursive formulation.
2. We think of the problem as a way of computing the entries in the array L where L[i,j] is the length of the length of LCS restricted to those substrings. We'd get the same results if we filled them in any order. We use something simpler, like a nested loop, that visits the array systematically. The only thing we have to worry about is that when we fill in a cell L[i,j], we need to already know the values it depends on, namely in this case L[i+1,j], L[i,j+1], and L[i+1,j+1]. So, we traverse the array backwards, from the last row working up to the first and from the last column working up to the first. This is iterative (because it uses nested loops instead of recursion) or bottom up (because the order we fill in the array is from smaller simpler subproblems to bigger more complicated ones).

LCS Algorithm (A different look)

- Using the relations, we can write a recursive routine to compute the length of the LCS; that will take exponential time. We will do a bottom up DP algorithm noting that there are only $\Theta(mn)$ distinct subproblems.
- We initialize an (n + 1) × (m + 1) array, L, for the boundary cases when i = 0 or j = 0. Namely, we initialize L[i,−1] = 0 for i = −1, 0, 1, . . . , n − 1 and L[−1, j] = 0 for j = −1, 0, 1, . . . , m − 1. (This is a slight abuse of notation, since in reality, we would have to index the rows and columns of L starting with 0.) Then, we iteratively build up values in L until we have L[n − 1,m − 1], the length of a longest common subsequence of X and Y.
- The algorithm is dominated by two nested for-loops, with the outer one iterating n times and the inner one iterating m times. Since the if-statement and assignment inside the loop each requires O(1) primitive operations, this algorithm runs in O(nm) time.

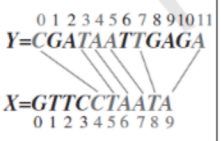
Algorithm LCS(X, Y):
Input: Strings X and Y with n and m elements, respectively
Output: For $i = 0, \ldots, n-1$, $j = 0, \ldots, m-1$, the length $L[i, j]$ of a longest common subsequence of $X[0..i]$ and $Y[0..j]$

```
for i ← −1 to n − 1 do
    L[i, −1] ← 0
for j ← 0 to m − 1 do
    L[−1, j] ← 0
for i ← 0 to n − 1 do
    for j ← 0 to m − 1 do
        if X[i] = Y[j] then
            L[i, j] ← L[i − 1, j − 1] + 1
        else
            L[i, j] ← max{L[i − 1, j] , L[i, j − 1]}
return array L
```

Extracting the LCS

- Given the table of L[i, j] values, constructing a longest common subsequence is simple. One method is to start from L[n−1,m−1] and work back through the table, reconstructing a longest common subsequence from back to front. At any position L[i, j], we determine whether X[i] = Y [j]. If this is true, then we take X[i] as the next character of the subsequence (noting that X[i] is **before** the previous character we found, if any), moving next to L[i−1, j−1]. If X[i] ≠ Y [j], then we move to the larger of L[i, j − 1] and L[i − 1, j]. We stop when we reach a boundary entry (with i = −1 or j = −1). This method constructs a longest common subsequence in O(n + m) additional time. Write the pseudocode and implement.

L	-1	0	1	2	3	4	5	6	7	8	9	10	11
-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	2	2	2	2	2	2	2	2	2
2	0	0	1	1	2	2	2	3	3	3	3	3	3
3	0	1	1	1	2	2	2	3	3	3	3	3	3
4	0	1	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	2	2	2	3	4	4	4	4	4
6	0	1	1	2	2	3	3	3	4	4	5	5	5
7	0	1	1	2	2	3	4	4	4	4	5	5	6
8	0	1	1	2	3	3	4	5	5	5	5	5	6
9	0	1	1	2	3	4	4	5	5	5	6	6	6



Retrieve the LCS itself

```
int i = 0, j=0; char S[20]; int sp=0;
while (i<m && j<n)
{
 if (A[i] == B[j])
 {S[sp] = A[i]; sp++; i++; j++;}
 else if (L[(i+1)*m+j] >= L[i*m+j+1]) i++;
 else j++;
}
S[sp] = '\0'; //pad with a null to make it a string
```

Note:

- i and j are running indices on the character arrays A and B
- S is a character array to store the retrieved LCS and sp is a an index to S, initialized at 0.
- One can print the LCS by using something like printf ("%s \n", S)

A Look at the array L

We ran the program for A = "nematode knowledge" and B ="empty bottle". The LCS is "emt ole"

0 – 1 Knapsack Problem

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ and has value $v_i > 0$.
- Knapsack has capacity of W.
- Goal: fill knapsack so as to maximize total value.
- Examples: {1, 2, 5} has value 35; {3,4} has value 40; {3, 5} has value 46, but exceeds weight limit.
- Possible Ways to choose a greedy algorithm:
  - **Greedy by value**. Repeatedly add item with maximum v_i.
  - **Greedy by weight**. Repeatedly add item with minimum w_i.
  - **Greedy by ratio**. Repeatedly add item with maximum ratio v_i / w_i.

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

knapsack instance (weight limit W = 11)

- **Observation.** None of greedy algorithms is optimal [Show by counterexamples]
- **Note**: This problem is called a "0-1" problem, because each item must be entirely accepted or rejected.

DP Solution

- **Attempt 1**: OPT(i) max profit subset contains item 1 …, i.
  - Case 1: OPT does not select item i ⇒ OPT selects best of { 1, 2, …, i – 1 } [by optimum substructure property]
  - Case 2: OPT selects item i. Selecting item i does not immediately imply that we will have to reject other items; also, without knowing what other items were selected before i, we don't even know if we have enough room for i.
  - NEED Something more.
- **Attempt 2**: OPT(i, w) = max profit subset of items 1, …, i with weight limit w.
  - Case 1: OPT does not select item i ⇒ OPT selects best of { 1, 2, …, i – 1 } using weight limit w.
  - Case 2: OPT selects item i ⇒ New weight limit = w – wi. and OPT selects best of { 1, 2, …, i – 1 } using this new weight limit.
  - Note: both cases can be shown true by optimal substructure property (using exchange argument). Thus,

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w), v_i + OPT(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$

Bottom-up DP Pseudocode

KNAPSACK $(n, W, w_1, \ldots, w_n, v_1, \ldots, v_n)$

FOR $w = 0$ TO W

  $M[0, w] \leftarrow 0$.

FOR $i = 1$ TO n

  FOR $w = 0$ TO W

  IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w]$.

  ELSE $M[i, w] \leftarrow \max \{ M[i-1, w],\ v_i + M[i-1, w-w_i] \}$.

RETURN $M[n, W]$.

Recovering the actual choices of items is done by backward tracing M[.][.], as we have done before.

Example

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{ OPT(i-1,w),\ v_i + OPT(i-1,w-w_i) \} & \text{otherwise} \end{cases}$$

weight limit w

subset of items 1, ..., i	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

0 – 1 Knapsack problem: running time

- **Theorem.** There exists an algorithm to solve the knapsack problem with n items and maximum weight W in $\Theta(nW)$ time and $\Theta(nW)$ space.

Weights are integers between 1 and W

- **Proof:**
  - Takes O(1) time per table entry.
  - There are $\Theta(nW)$ table entries.
  - After computing optimal values, can trace back to find solution: take item i in OPT(i, w) iff M [i, w] > M [i – 1, w].

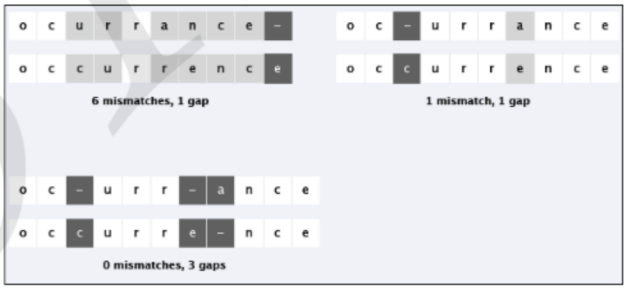
Remarks:

- Not polynomial in input size! The running time of our algorithm depends on a parameter W that strictly speaking is not proportional to the size of the input (the n items, together with their weights and benefits, plus the number W).
  - Assuming that W is encoded in some standard way (such as a binary number), then it takes only $O(\log_2 W)$ bits to encode W.
  - Moreover, if W is very large (say $W = 2^n$), then this dynamic programming algorithm would actually be asymptotically slower than the brute-force method.
  - It's **Pseudo-polynomial** algorithm, since its running time depends polynomially on the magnitude of a number given in the input, not its encoding size.
- Decision version of knapsack problem is NP-COMPLETE. [CHAPTER 8]
- There exists a poly-time algorithm that produces a feasible solution that has value within 1% of optimum. [SECTION 11.8]

Sequence Alignment

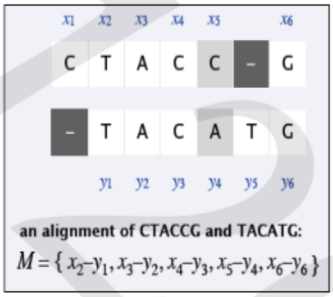
- How similar are two given strings (sequences)? How do we measure? There are several ways to formalize the notion of distance between strings. One common, and simple, formalization called edit distance, focuses on transforming (or editing) one string into the other by a series of edit operations on individual characters.
  - The permitted edit operations are insertion (I) of a character into the first string, the deletion (D) of a character from the first string, or the substitution (R) (or replacement) of a character in the first string with a character in the second string. Using M to denote the non-operation "match", the sequence RIMDMDMMI, known as edit transcript, can be used to edit the string "vintner" into "writers".

R	I	M	D	M	D	M	M	I
v		i	n	t	n	e	r	
w	r	i		t		e	r	s



Sequence Alignment

- There are many other ways of "lining up" two strings. Consider 2 strings: $X = x_1x_2\ldots x_m$, and $Y = y_1y_2\ldots y_n$. A **matching M** is a set of ordered pairs with the property that each item occurs in at most one pair. We say that a matching M of these two sets is an alignment if there are no "crossing" pairs: if for any 2 pairs $(x_{i1}, y_{j1}), (x_{i2}, y_{j2}) \in M$ and $i1 < i2$, then $j1 < j2$.
- Intuitively, an alignment gives a way of lining up the two strings, by telling us which pairs of positions will be lined up with one another.



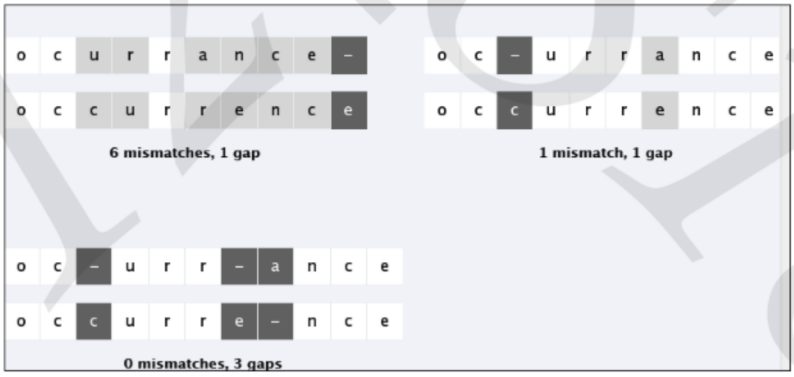
The cost of a matching M is defined as sum of its gap and mismatch costs:

1. **Gap penalty** is a parameter $\delta > 0$ [For each position of X or Y that is not matched in M – it is a gap.
2. For each pair of letters p, q in our alphabet, there is a **mismatch penalty** of α_{pq} for lining up p with q. One generally assumes that $\alpha_{pp} = 0$ for each letter p – there is no mismatch cost to line up a letter with another copy of itself.

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i : x_i \text{ unmatched}} \delta + \sum_{j : y_j \text{ unmatched}} \delta}_{\text{gap}}$$

Sequence Alignment

- As an example, look at different possible matchings of two strings ["occurrence" and "ocurrance"]with different costs.



- Applications: Unix diff, speech recognition, computational biology, error checking, ...

Sequence Alignment: problem structure

- **Definition**. OPT(i, j) = min cost of aligning prefix strings $x_1\ x_2\ ...\ x_i$ and $y_1\ y_2\ ...\ y_j$.
- **Case 1**. OPT matches $x_i - y_j$.
  Pay mismatch for $x_i - y_j$ + min cost of aligning $x_1\ x_2\ ...\ x_{i-1}$ and $y_1\ y_2\ ...\ y_{j-1}$.
- **Case 2a**. OPT leaves x_i unmatched.
  Pay gap for x_i + min cost of aligning $x_1\ x_2\ ...\ x_{i-1}$ and $y_1\ y_2\ ...\ y_j$.
- **Case 2b**. OPT leaves y_j unmatched.
  Pay gap for y_j + min cost of aligning $x_1\ x_2\ ...\ x_i$ and $y_1\ y_2\ ...\ y_{j-1}$.
- **Note:** All three cases derive from the optimal substructure property (prove via exchange argument)

$$OPT(i,j) = \begin{cases} j\delta & \text{if } i = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \\ i\delta & \text{if } j = 0 \end{cases}$$

- Writing the pseudocode is as easy and straightforward as before.

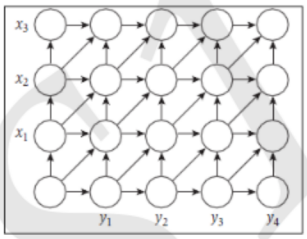
Sequence Alignment Pseudocode

```
SEQUENCE-ALIGNMENT (m, n, x1, ..., xm, y1, ..., yn, δ, α)
-----------------------------------------------------------
FOR i = 0 TO m
   M[i, 0] ← iδ.
FOR j = 0 TO n
   M[0, j] ← jδ.

FOR i = 1 TO m
   FOR j = 1 TO n
      M[i, j] ← min { α[xi, yj] + M[i−1, j−1],
                      δ + M[i−1, j],
                      δ + M[i, j−1]).

RETURN M[m, n].
```

Sequence Alignment Analysis

- **Theorem**. The dynamic programming algorithm computes the edit distance (and optimal alignment) of two strings of length m and n in $\Theta(mn)$ time and $\Theta(mn)$ space.
- **Correctness**: The correctness of the algorithm follows directly from the recurrence relation. The running time is O(mn), since the array M has O(mn) entries, and at worst we spend constant time on each. Also note that as in previous dynamic programming algorithms, we can trace back through the array M, using the second part of the recurrence to construct the alignment itself.

Interesting Observation: Build a two-dimensional $m \times n$ grid graph G_{XY}, with the rows labeled by symbols in the string X, the columns labeled by symbols in Y, and directed edges. We number the rows from 0 to m and the columns from 0 to n; we denote the node in the i^{th} row and the j^{th} column by the label (i, j). We put costs on the edges of G_{XY}: the cost of each horizontal and vertical edge is δ, and the cost of the diagonal edge from $(i-1, j-1)$ to (i, j) is $\alpha_{xi,yj}$. Convince yourself that the rule is exactly the same as the recurrence we have seen before.

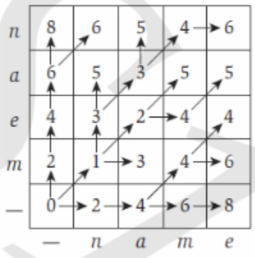


Claim: Let f (i, j) denote the minimum cost of a path from (0, 0) to (i, j) in G_{XY}. Then for all i, j, we have f (i, j) = OPT(i, j).

Sequence Alignment Analysis

Example: Consider an example: assume gap penalty $\delta = 2$; $\alpha_{XY} = 1$ if both X and Y are different vowels or both are different consonants and 3 if X, Y form a vowel, consonant pair.

- For each cell in the table (representing the corresponding node), the arrow indicates the last step of the shortest path leading to that node, i.e., the way that the minimum is achieved using the recurrence relation.
- Also, by following arrows backward from node (4, 4), we can trace back to construct the alignment.



- **Claim**: Let f (i, j) denote the minimum cost of a path from (0, 0) to (i, j) in G_{XY}. Then for all i, j, we have f (i, j) = OPT(i, j).
- **Proof:** Use strong induction on i + j. Base case: When i + j = 0, we have i = j = 0, andf (0, 0) = OPT (0, 0) = 0. Now consider arbitrary values of i and j, and suppose the statement is true for all pairs (k_1, k_2) with $k_1 + k_2 < i + j$. The last edge on the shortest path to (i, j) is either from $(i-1, j-1)$, $(i-1, j)$, or $(i, j-1)$. Thus we have f (i, j) = $\min[\alpha_{xiyj} + f(i-1, j-1), \delta + f(i-1, j), \delta + f(i, j-1)] = \min[\alpha_{xiyj} + OPT(i-1, j-1), \delta + OPT(i-1, j), \delta + OPT(i, j-1)] =$ OPT(i, j) where we pass from the first line to the second using the induction hypothesis, and we pass from the second to the third using our previous recurrence relation.