COT 6405
ANALYSIS OF ALGORITHMS

Dynamic Programming

Computer & Electrical Engineering and Computer Science Department
Florida Atlantic University

Outline

- DP – introduction
- Weighted Interval Scheduling (KT – chapter 6.1)
- Principles of DP (KT – chapter 6.2)
- Change-making problem (BB – chapter 8.2)
- 0-1 knapsack problem (BB – chapter 8.4)
- Sequence alignment problem (KT – chapter 6.6)

KT book - *Algorithm Design* by J. Kleinberg and Eva Tardos
BB book - *Fundamentals of Algorithms* by Gilles Brassard and Paul Bratley

Dynamic Programming (DP)

- is a technique, not a specific algorithm (like divide-and-conquer)
- applied to optimization problems (maximization or minimization)
- applicable when subproblems are not independent, that is subproblems share subsubproblems. Then DP solves each subproblem only once, stores the result in a table, and reuses it later

Weighted Interval Scheduling

Problem definition:

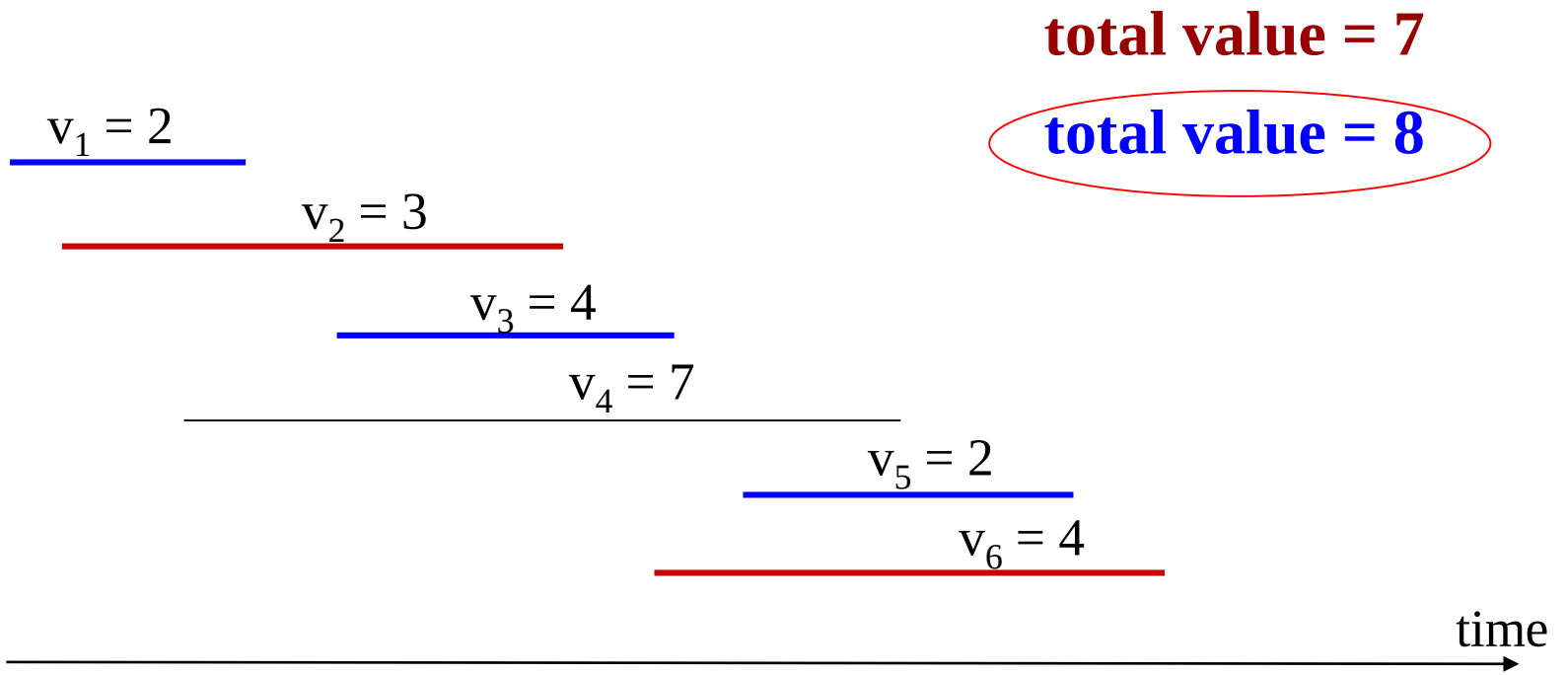
We are given **1 resource** and a number of **n requests** labeled 1,2,…,n. Each request i has a starting time s_i, a finishing time f_i, and a *value* (or *weight*) $v_i > 0$.

Find a *compatible* subset S of requests (intervals) of *maximum total value* $\Sigma_{i \in S} v_i$.

- two requests i and j are compatible if the requests do not overlap
  - request i is earlier than j, $f_i \leq s_j$
  - request j is earlier than i, $f_j \leq s_i$
- a subset of requests is compatible if all pairs i, j ($i \neq j$) are compatible

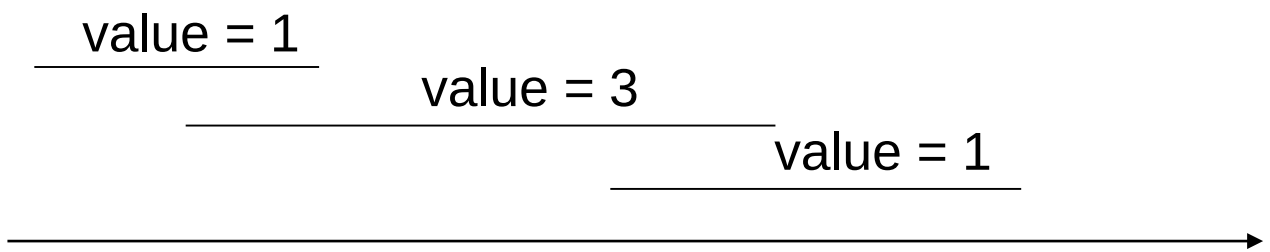
Weighted Interval Scheduling

Example



Weighted Interval Scheduling

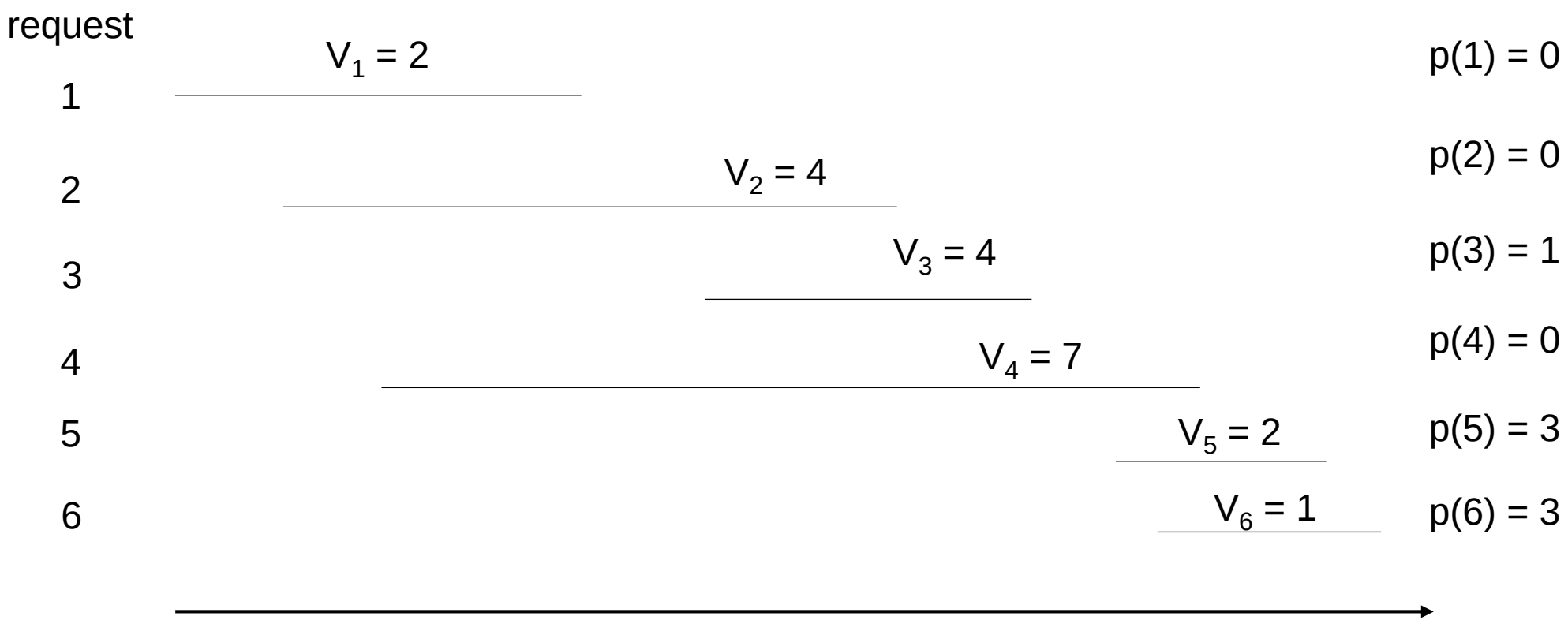
- can be solved using *greedy* when all requests have the same value (equal to 1)
  - greedy choice: choose the compatible request with the earliest finishing time
- when the requests have different values, greedy does not work



- for different request values, DP produces an optimal solution

A recursive algorithm

- suppose that the requests are sorted in nondecreasing finishing time $f_1 \leq f_2 \leq \ldots \leq f_n$. We say that request i *comes before* j if i < j.
- p(j) – the largest index i < j such that requests i and j are disjoint
  - rightmost interval i that ends before j begins



Weighted Interval Scheduling

- optimal solution O: either n belongs to O or it doesn't
  - if $n \in O$, then in addition O contains an *optimal* solution to the problem with requests {1,…,p(n)}
  - if $n \notin O$, then O is an *optimal* solution to the problem with requests {1,2,…,n-1}
- let O_j – optimal solution for the problem w/ requests {1,2,…,j}
  - OPT(j) – value of this solution
- objective: find O_n and OPT(n)

Writing a recursion

$OPT(j) = \max (v_j + OPT(p(j)), OPT(j-1))$

- request j belongs to an optimal solution on the set {1,2,…,j} iff $v_j + OPT(p(j)) \geq OPT(j-1)$
- characteristic of DP: write a recurrence equation that expresses the optimal solution (or its value) in terms of optimal solutions to smaller subproblems

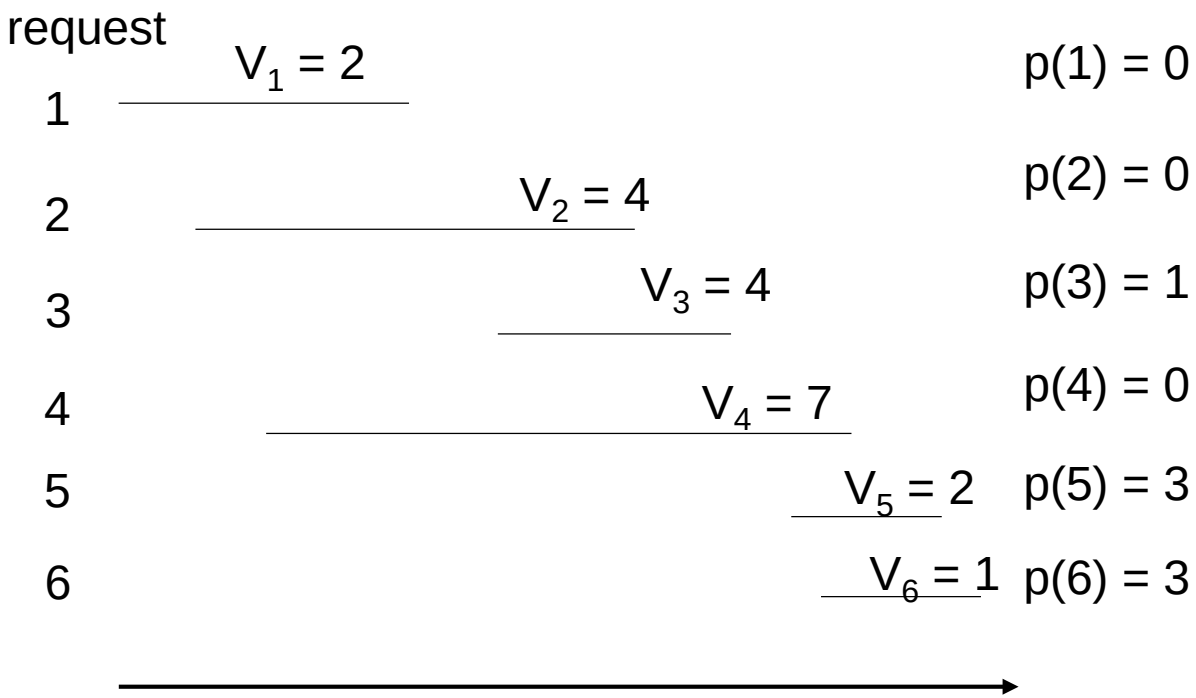
Recursive algorithm

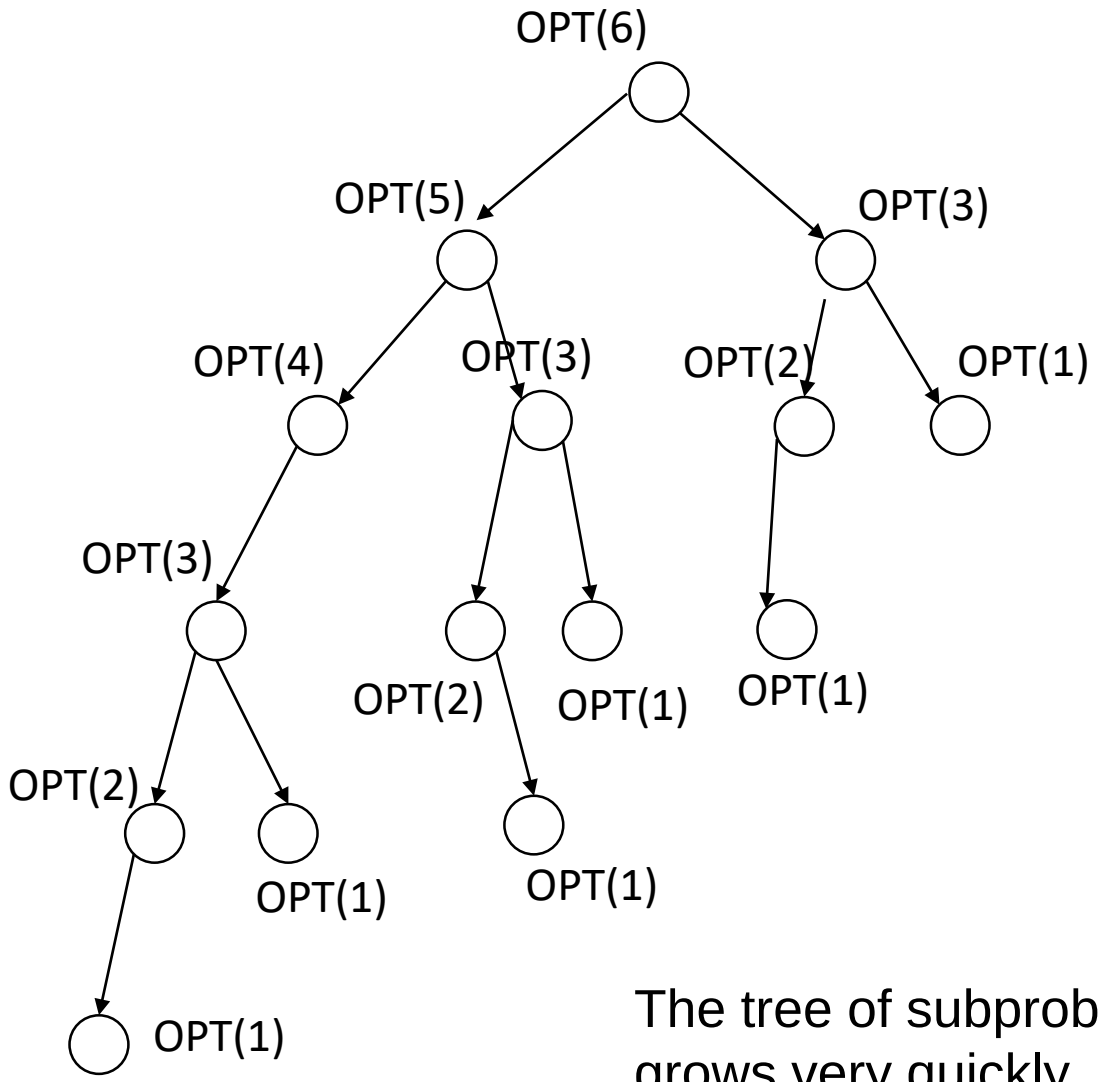
Compute-Opt(j)

```
if j == 0 then
  return 0
else
  return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))
endif
```

- drawback: RT is exponential in n

Example





The tree of subproblems grows very quickly.

Memoizing the Recursion

- Key observations:
  - Compute-Opt(n) is only solving n+1 different subproblems: Compute-Opt(0), Compute-Opt(1),…,Compute-Opt(n)
  - exponential time is due to the redundancy in the number of times the same call is made

- **Memoization technique**: *solve each subproblem only once and store its value in a table. All future calls use the precomputed value.*

Memoizing the Recursion

- array M[0..n]
  - M[j] initially empty, then store Compute-Opt(j) value
- to determine OPT(n), call M-Compute-Opt(n)

M-Compute-Opt(j)

```
if j == 0 then
    return 0
elseif M[j] is not empty then
    return M[j]
else
    M[j] = max(v_j + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
     return M[j]
endif
```

Analyzing the Memoized Version

- Assuming that the input intervals are sorted by their finish time and that the p() values are already computed, the RT to compute M-Compute-Opt(n) is O(n).
  - Proof: since M has n+1 entries, there are at most O(n) calls to M-Compute-Opt, and hence the RT is O(n)

A second solution: iterative approach

Iterative-Compute-Opt

M[0] = 0

for j = 1,2,…,n

    M[j] = max(v_j + M[p(j)], M[j-1])

endfor

RT = O(n)

- Example

Computing a solution in addition to its value

- Find the subset of requests of maximum value

```
Find-Solution(j)
if j == 0 then
    return Ø
else
    if v_j + M[p(j)] ≥ M[j-1] then
        output Find-Solution(p(j)) ∪ { j }
    else
        output the result of Find-Solution(j-1)
    endif
endif
```

- initial call: Find-Solution(n)
- RT = O(n), calls recursively on smaller instances and takes constant time per call

Principles of Dynamic Programming

- write solution to a problem recursively, based on solutions to subproblems
- memoization or iteration over subproblems?
  - iteratively building up subproblems is simpler
- properties of DP:
  - there is only a polynomial number of subproblems
  - the solution to the original problem can be easily computed from the solutions to the subproblems
  - there is a natural ordering of subproblems from smallest to largest

Change-making problem

- Unfortunately, even though greedy algorithm is very efficient, it works only in a limited number of instances
- Dynamic programming works for all systems of coins
- Suppose that the currency has available coins of n different denominations
  - a coin of denomination i, $1 \leq i \leq n$ has value $d_i > 0$ units
- Suppose that we have an unlimited supply of coins of each denomination
- Goal: give the customer coins worth N units, using as few coins as possible

DP approach

- table c[1..n, 0..N] – one row for each denomination and one column for each amount from 0 units to N units

- c[i,j] – minimum number of coins required to pay an amount j, with $0 \leq j \leq N$, using only coins of denominations 1 to i, $1 \leq i \leq n$

- the solution to the original problem is given by c[n, N]

Filling up the table

- c[i,0] = 0 for all i
- then the table can be filled row by row, left to right, or column by column, top to bottom
- to compute c[i,j], two choices:
  - do not use any coins of denomination i, then c[i,j] = c[i-1,j]
  - use at least one coin of denomination i, then c[i,j] = 1 + c[i,j - d_i]
- therefore:

  $c[i,j] = \min(c[i-1,j], 1+c[i,j-d_i])$

- if i = 1 and j < d_1, then set c[i,j] = ∞

DP algorithm

- Algorithm “coins” computes the minimum number if coins needed to make change for N units.
- The array d[1..n] specifies the coinage. In the example, there are coins for 1, 4, and 6 units.

```
Algorithm coins(N)
array d[1..n] = [1,4, 6]
array c[1..n, 0..N]
for i = 1 to n
        c[i,0] = 0
for i = 1 to n
        for j = 1 to n
                if i == 1 and j < d[i] then c[i,j] = ∞
                elseif i == 1 then c[i, j] = 1 + c[1, j – d[1]]
                elseif j < d[i] then c[i,j] = c[i-1,j]
                else c[i,j] = min(c[i-1,j],1+c[i,j-d[i]])
return c[n,N]
```

$RT = \Theta(nN)$

DP algorithm comments

- if an unlimited supply of coins of value 1 is available, then we can always find a solution to our problem
- if no coin with value 1, then there may be values of N for which no solution is possible
  - algorithm returns ∞

Finding how many coins of each denominations are used

PrintCoins(c,i,j)

if c[i,j] = ∞

  then return no change possible

if j == 0 then return

if c[i,j] == c[i-1,j]

  then PrintCoins(c,i-1,j)

elseif c[i,j] == 1 + c[i,j-d_i]

  then PrintCoins(c,i,j-d_i)

  print d_i

- Initial call: PrintCoins(c,n,N)
- RT = O(N + n)
- Example

0-1 knapsack problem

- Given:
  - n objects and a knapsack
  - for i = 1,..,n, object i has a positive weight w_i and a positive value v_i
  - objects may not be broken into smaller pieces, either take the whole object or leave it behind
  - the knapsack can carry a weight $\leq W$
- Objective: fill the knapsack s.t. to maximize the value of the included objects, while respecting the capacity constraints

DP approach

- table V[1..n,0..W] – one row for each available object and column for each weight from 0 to W
- V[i,j] – maximum value of the objects we can transport if the weight limit is j, $0 \leq j \leq W$, and if we only include objects numbered from 1 to i, $1 \leq i \leq n$
- The solution to the original problem is given by V[n,W]

Filling up the table

- V[i,0] = 0, for all i
- then the table can be filled row by row, left to right, or column by column, top to bottom
- to compute V[i,j], two choices:
  - not adding object i to the knapsack, V[i,j] = V[i-1, j]
  - adding object i to the knapsack, V[i,j] = v_i + V[i-1, j-w_i]
- therefore:

  V[i,j] = max(V[i-1,j], v_i + V[i-1,j-w_i])
- for the out-of-bounds-entries, we define:

  V[0,j] = 0 when $j \geq 0$

  V[i,j] = $-\infty$ for all i when $j < 0$

Sequence Alignment – first example

- dictionaries on the Web and spell checkers
  - if you type "*ocurrance*", it may ask: "Perhaps you mean *occurrence*?"
  - the dictionary will search its entries for the word most "similar" to the one typed
- how should we define *similarity* between two words or strings?

Modeling “similarity”

- model similarity between two strings by the number of *gaps* and *mismatches* that occur when lining up the two sequences

o	-	c	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

- one gap
- one mismatch

o	-	c	u	r	r	-	a	n	c	e
o	c	c	u	r	r	e	-	n	c	e

- three gaps
- no mismatch

- which one is better?

Computational biology – second example

- An organism's *genome*:
  - divided into linear DNA molecules, chromosomes, which serve as an one-dimensional chemical storage device
  - string over the alphabet {A,C,G,T} that determines the properties of the organism
    - adenine, cytosine, guanine, thymine
  - the string encodes instructions for building protein molecules
    - using a chemical mechanism to read portions of the chromosomes, a cell can construct proteins that control its metabolism

Computational biology – second example, cont.

- let X and Y be two strains of bacteria, closely related evolutionary
- assume that a certain substring in the DNA of X codes for a certain toxin
- if this substring is found in Y, we may hypothesize that this portion codes for a similar kind of toxin

Sequence Alignment Problem

- let X = $x_1x_2..x_m$ and Y = $y_1y_2..y_n$
- {1,2,…,m} and {1,2,…,n} represent different positions in strings X and Y
- *matching*: set of ordered pairs such that each item occurs in at most one pair
- a matching M of these two sets is an *alignment* if there are no *crossing* pairs: if (i,j), (i',j') $\in$ M and i < i' then j < j'
  - an alignment provides a way to line up the two strings

stop-
-tops

  - alignment: {(2,1),(3,2),(4,3)}

Sequence Alignment Problem

Let M be an alignment of X and Y

- *gap penalty:* each gap incurs a cost $\delta > 0$
- *mismatch cost*: for each pair of letters p,q in the alphabet, there is a mismatch cost α_{pq} for lining up p with q
  - usually $\alpha_{pp} = 0$
- the *cost* of M is the sum of gap and mismatch costs

Objective: find an ***optimal alignment*** of X and Y, that means an alignment of minimum cost.

- values δ and $\{\alpha_{pq}\}$ are given parameters
- the lower the cost, the more similar X and Y are
- going back to the first example *ocurrance* and *occurrence*, the first alignment is better if and only if $\delta + \alpha_{ae} < 3\delta$

Designing the DP algorithm

- in the optimal alignment M:
  - either $(m,n) \in M$ or $(m,n) \notin M$
- in any alignment M:
  - if $(m,n) \notin M$, then either the m^{th} position of X or the n^{th} position of Y are not matched in M

    proof: assume by contradiction $(i,n),(m,j) \in M$ for some i,j.

    Then $i < m$ and $j < n \Rightarrow$ crossing pairs $\Rightarrow$ contradiction

Property: in any alignment M, one of the following is true:
(*i*) $(m,n) \in M$, or
(*ii*) the m^{th} position of X is not matched, or
(*iii*) the n^{th} position of Y is not matched

Designing the DP algorithm

- let OPT(i,j) – minimum cost of an alignment between $x_1x_2\ldots x_i$ and $y_1y_2\ldots y_j$
- recursively define OPT(m,n):

case (*i*):

- pay $\alpha_{x_m y_n}$, then optimally align $x_1x_2\ldots x_{m-1}$ and $y_1y_2\ldots y_{n-1}$

  $OPT(m,n) = \alpha_{x_m y_n} + OPT(m-1,n-1)$

case (*ii*):

- pay gap cost δ, then optimally align $x_1x_2\ldots x_{m-1}$ and $y_1y_2\ldots y_n$

  $OPT(m,n) = \delta + OPT(m-1,n)$

case (*iii*):

- pay gap cost δ, then optimally align $x_1x_2\ldots x_m$ and $y_1y_2\ldots y_{n-1}$

  $OPT(m,n) = \delta + OPT(m,n-1)$

Designing the DP algorithm

Property: The minimum alignment costs satisfy the following recurrence, for $i \geq 1$, $j \geq 1$:

$OPT(i,j) = \min\{\alpha_{x_i y_j} + OPT(i-1,j-1), \delta + OPT(i-1,j), \delta + OPT(i,j-1)\}$

Moreover, (i,j) is in an optimal alignment iff the minimum is achieved by the first of these values.

Alignment (X,Y)

```
array A[0..m,0..n]
initialize array A[i,0] = iδ for each i
initialize array A[0,j] = jδ for each j
for j = 1 to n
  for i = 1to m
     A[i,j] = min {αxiyj + A[i-1,j-1], δ + A[i-1,j], δ + A[i,j-1]}
return A[m,n]
```

RT = Θ (mn)