COT 6405
ANLYSIS OF ALGORITHMS

Dynamic Programming

Computer & Electrical Engineering and Computer Science Dept.
Florida Atlantic University

Spring 2017

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 & 6.7)

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* $\Sigma_{i \in S} v_i$.

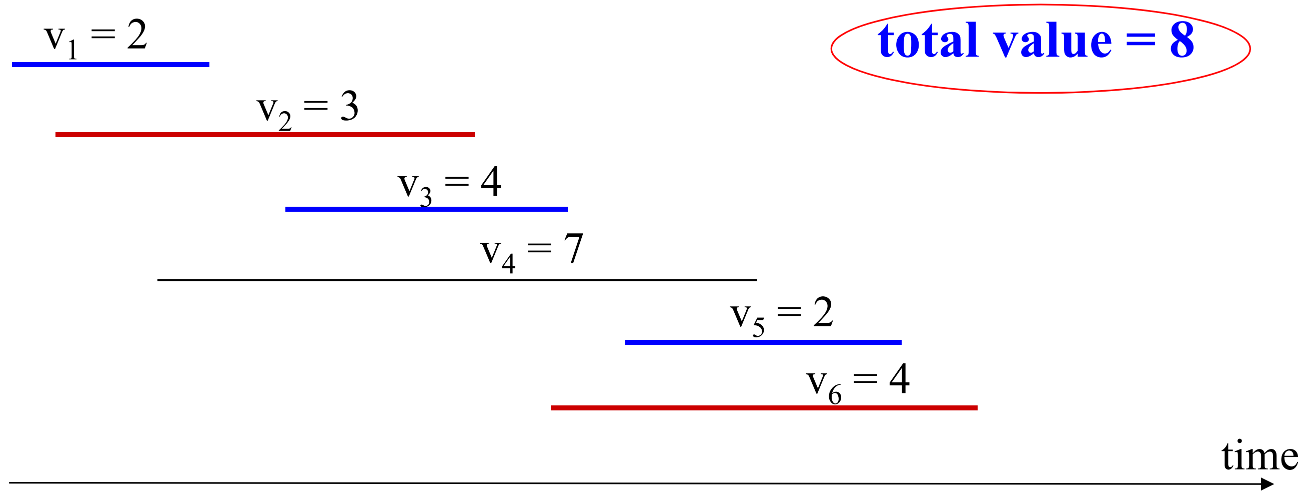
- two requests i and j are compatible if the requests do not overlap
  - request i is earlier than j, $f_i \leq s_j$
  - request j is earlier than i, $f_j \leq s_i$
- a subset of requests is compatible if all pairs i, j ($i \neq j$) are compatible

Weighted Interval Scheduling

Example

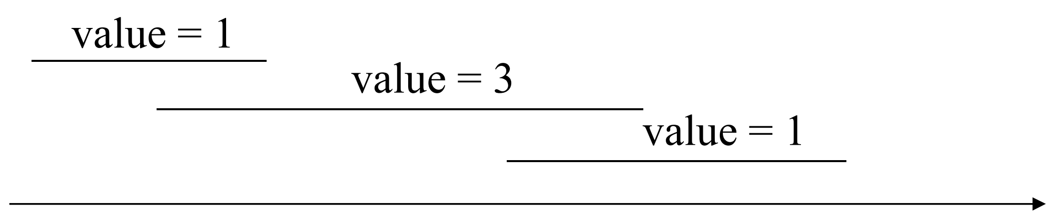
total value = 7

total value = 8



Weighted Interval Scheduling

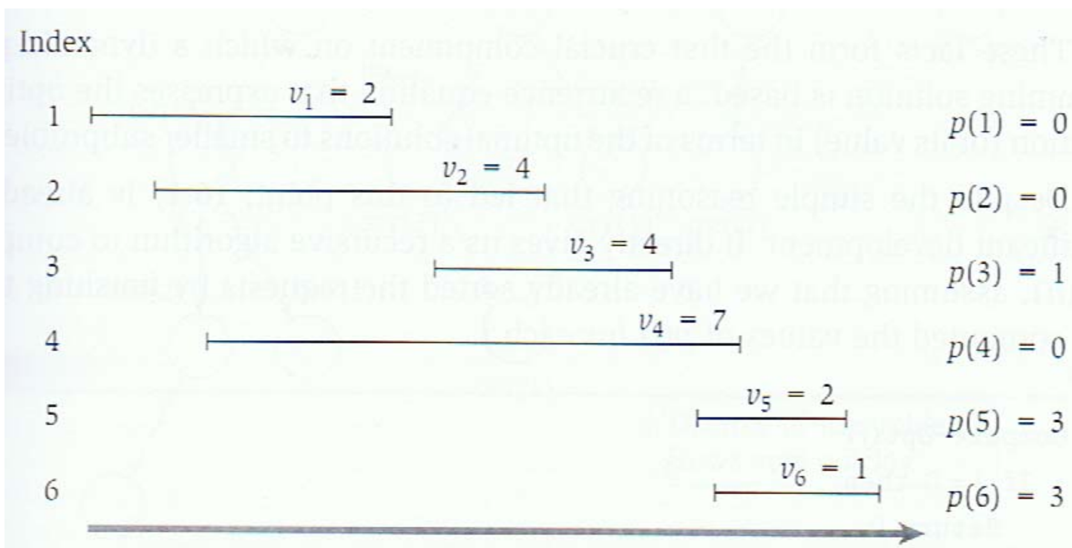
- can be solved using *greedy* when all requests have the same value (equal to 1)
  - greedy choice: choose the compatible request with the earliest finishing time
- when the requests have different values, greedy does not work



- 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
- define p(j) of an interval j – the largest index i < j s.t. 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 $\{1,\ldots,p(n)\}$
  - if $n \notin O$, then O is an *optimal* solution to the problem with requests $\{1,2,\ldots,n-1\}$
- let O_j – optimal solution for the problem w/ requests $\{1,2,\ldots,j\}$
  - OPT(j) – value of this solution
- objective: find O_n and OPT(n)

Writing a recursion

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

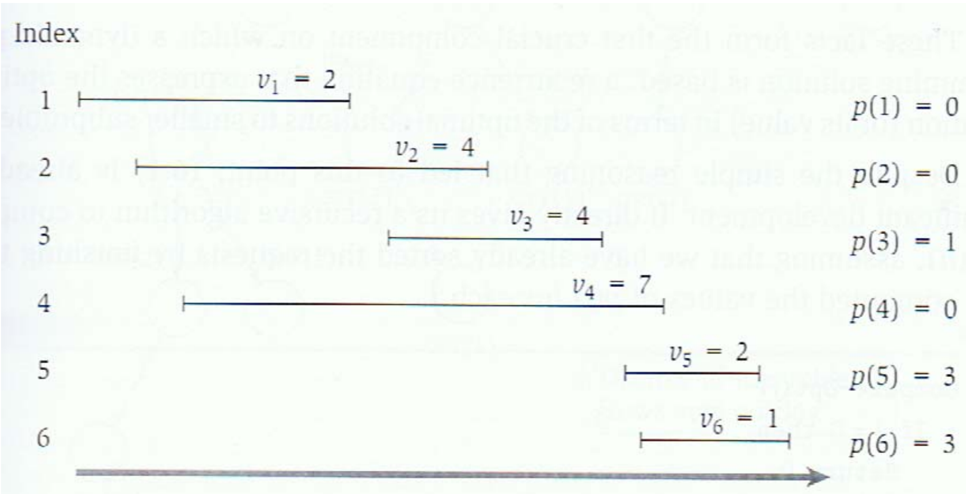
- request j belongs to an optimal solution on the set {1,2,…,j} iff $v_j + OPT(p(j)) \geq OPT(j-1)$
- characteristic of DP: write a recurrence equation that expresses the optimal solution (or its value) in terms of optimal solutions to smaller subproblems

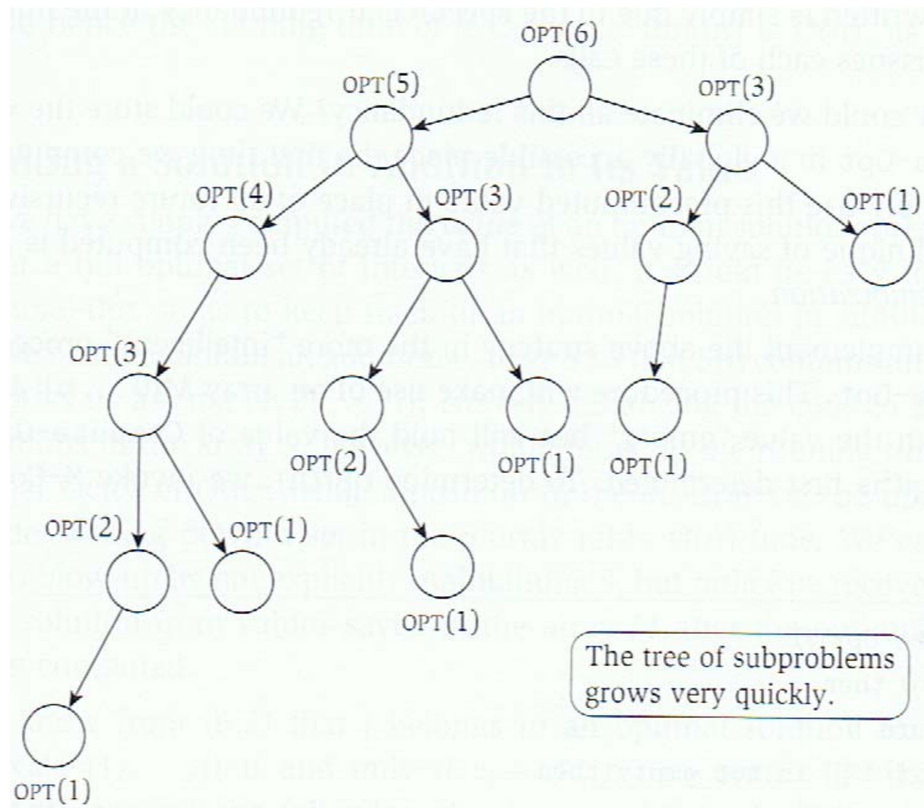
Recursive algorithm

```
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


Index
1
$v_1 = 2$
$p(1) = 0$
2
$v_2 = 4$
$p(2) = 0$
3
$v_3 = 4$
$p(3) = 1$
4
$v_4 = 7$
$p(4) = 0$
5
$v_5 = 2$
$p(5) = 3$
6
$v_6 = 1$
$p(6) = 3$


OPT(6)
OPT(5)
OPT(3)
OPT(4)
OPT(3)
OPT(2)
OPT(1)
OPT(3)
OPT(2)
OPT(1)
OPT(1)
OPT(2)
OPT(1)
OPT(1)
OPT(1)
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
else if 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 $c[i,j] = \infty$

DP algorithm

```
function coins(N)
   {Gives the minimum number of coins needed to make
    change for N units. Array d[1..n] specifies the coinage:
    in the example there are coins for 1, 4 and 6 units.}
   array d[1..n]= [1,4,6]
   array c[1..n,0..N]
   for i ← 1 to n do c[i,0] ← 0
   for i ← 1 to n do
       for j ← 1 to N do
           c[i,j] ← if i = 1 and j < d[i] then +∞
                    else if i = 1 then 1 + c[1, j − d[1]]
                    else if j < d[i] then c[i − 1, j]
                    else 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)
else if 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, $0 \le j \le W$, and if we only include objects numbered from 1 to i, $1 \le i \le n$
- 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-currance
occurrence
```

- one gap
- one mismatch

```
o-curr-ance
occurre-nce
```

- three gaps
- no mismatch

- which one is better?

Application – computational biology

- 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

Application – computational biology

- let X and Y be two strains of bacteria, closely related evolutionary
- assume 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 s.t. 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

Suppose M is a given alignment between 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***, 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 $i \geq 1$, $j \geq 1$:

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

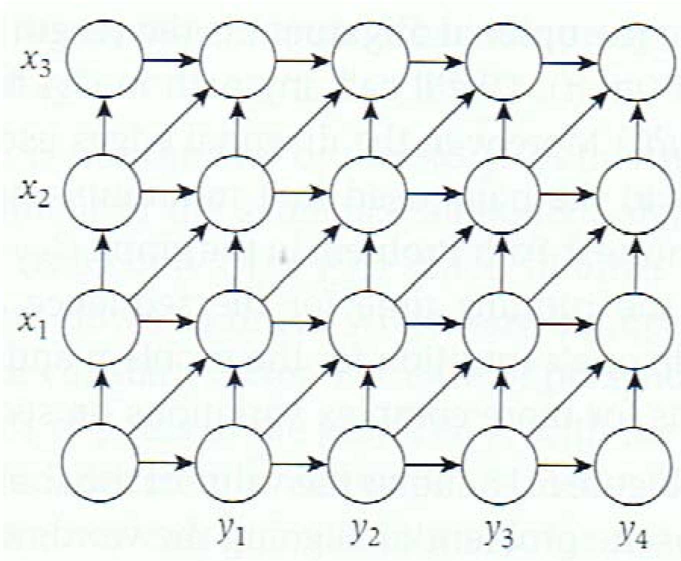
Moreover, (i,j) is in an optimal alignment iff the minimum is achieved by the first of these values.

<u>Alignment (X,Y)</u>

```
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 {α_{xiyj} + A[i-1,j-1], δ + A[i-1,j], δ + A[i,j-1]}
return A[m,n]
```

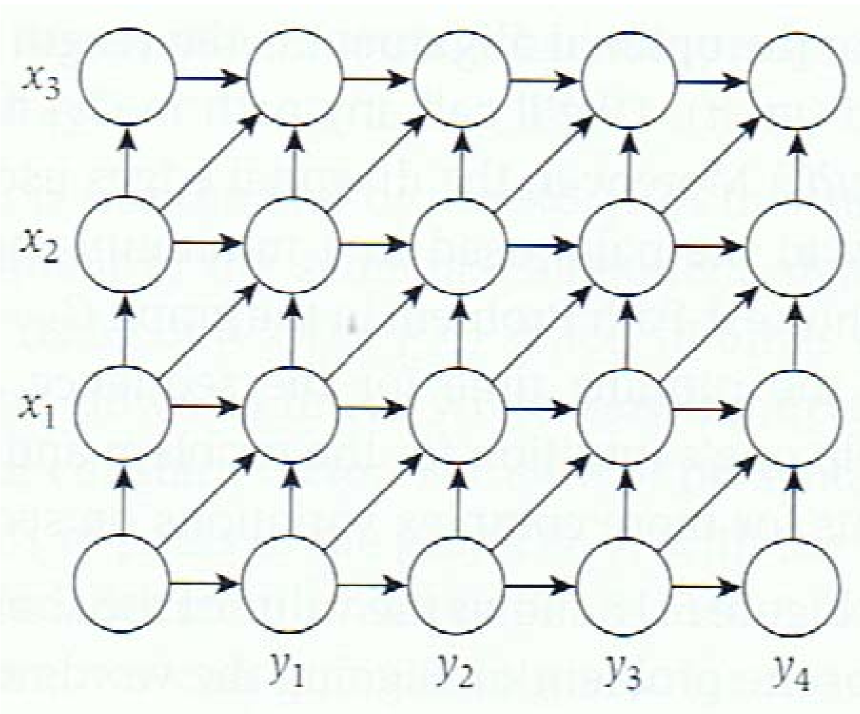
RT = Θ (mn)

Graph – based picture of sequence alignment



- Cost of the edges:
  - horizontal & vertical edges have cost δ
  - diagonal edge (i-1,j-1) to (i,j) has cost $\alpha_{x_i y_j}$
- Value of an optimal alignment is the minimum-cost of a path from (0,0) to (m,n)

Graph – based picture of sequence alignment



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 in Linear Space via Divide and Conquer

- Finding the optimal alignment is equivalent to constructing the graph G_{XY} with mn nodes laid out in a grid and looking for the cheapest path between opposite corners
- RT for this approach is O(mn) and space O(mn)
- This is too large for biological applications where strings are very long
  - if the two strings have ~ 100,000 symbols each, then RT ~ 10 billion primitive operations and space is ~10 billion array
- Objective: enhancement of the sequence alignment algorithm that has RT = O(mn) and space O(m+n)
  - uses divide-and-conquer

Designing the Algorithm

- First, we'll show that if we care only about the *value* of an optimal alignment, then it's easy to have linear space
- Key observation: to fill out the array A, we only need information on the current column of A and the previous column of A
- Instead of using array A of size $(m+1)\times(n+1)$, use array B of size $(m+1)\times 2$
- As the algorithm iterates through values of j, entries B[i,0] will hold the *previous* column's value A[i,j-1] and entries of the form B[i,1] will hold the *current* column's values A[i,j]

Designing the Algorithm

```
Space-Efficient-Alignment (X,Y)
array B[0..m,0..1]
initialize B[i,0] = iδ for each i (just as in column 0 of A)
for j = 1,…,n
    B[0,j] = jδ (since this corresponds to entry A[0,j])
    for i = 1,…,m
        B[i,1] = min[ α_{x_i y_j} + B[i-1,0], δ + B[i-1,1], δ + B[i,0]} ]
    endfor
    // move col 1 of B to col 0 to make room for the next iteration
    update B[i,0] = B[i,1] for each i
endfor
```

- RT = O(mn)
- space =O(m)

Space-efficient-alignment

- when the alg. terminates, B[i,1] holds the value of OPT(i,n) for i = 0,…m
- issue: how to find the assignment itself?
  - we haven't left enough information to find the alignment
  - B has only the last two columns, so we cannot trace back the optimal alignment (shortest path)
- different approach if we want to recover the optimal alignment

A backward formulation of the DP

- $f(i,j)$ – length of the shortest path (0,0) to (i,j) in the graph G_{XY}

$$f(i,j) = OPT(i,j)$$

- define $g(i,j)$ – length of the shortest path from (i,j) to (m,n) in G_{XY}

- build g using DP in reverse: start with $g(m,n) = 0$, and the answer we want is $g(0,0)$

- for $i < m$ and $j < n$ we have:

$$g(i,j) = \min [\alpha_{x_{i+1}\, y_{j+1}} + g(i+1, j+1), \delta + g(i, j+1), \delta + g(i+1, j)\}$$

- g is built using DP backward from (m,n)

- we can also design the space-efficient version,

  Backward-Space-Efficient-Alignment(X,Y)

in space O(m) and RT = O(mn)

Combining the Forward and Backward Formulations

Important properties:

- The length of the shortest corner-to-corner path in G_{XY} that passes through (i,j) is $f(i,j) + g(i,j)$
- Let k be any number in {0,…,n} and let q be an index minimizes the quantity $f(q,k) + g(q,k)$. Then there is a corner-to-corner path of minimum length that passes through the node (q,k).

Designing the algorithm

- divide G_{XY} along the center column and compute f(i,n/2) and g(i,n/2) for each i, using the two space-efficient algorithms
- find the minimum f(i,n/2) + g(i,n/2) for same value i
- then there is a shortest corner-to-corner path that passes through (i,n/2)
- recursively find the shortest-path in G_{XY} between (0,0) and (i,n/2) and in the portion between (i,n/2) and (m,n)
- MAIN IDEA:
  - Apply these recursive calls sequentially and reuse the working space from one call to the next
- then the space usage is O(m+n)

Designing the algorithm

- maintain a globally accessible list P with nodes on the shortest corner-to-corner path as they are discovered
  - initially, P is empty
  - P has at most m+n entries, since any path has at most m+n edges
- notation:

  $X[i:j]$, for $1 \leq i \leq j \leq m$, is the substring $x_i x_{i+1} \ldots x_j$

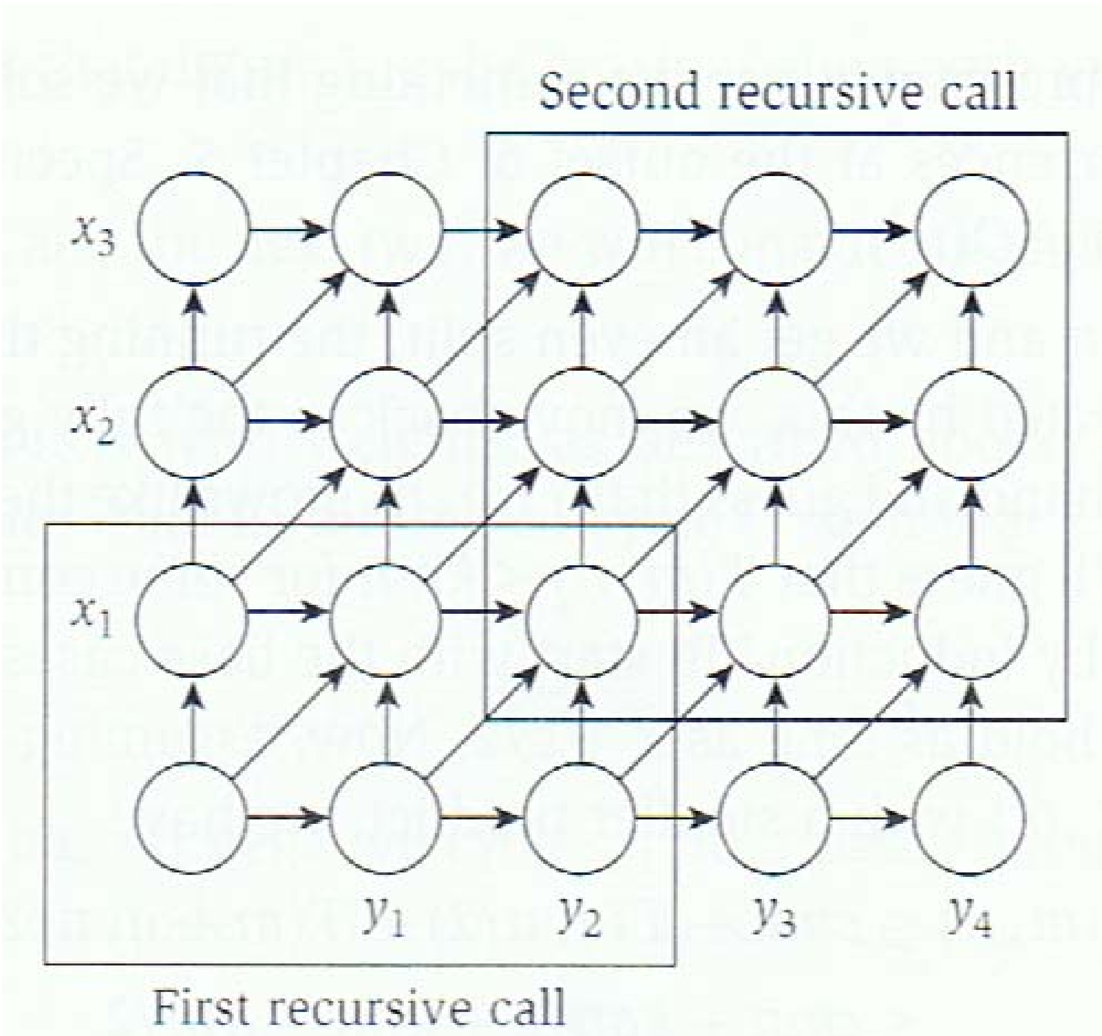
  similar for $Y[i:j]$
- assume for simplicity n is a power of 2

Designing the algorithm

```
Divide-and-Conquer-Alignment(X,Y)
m is the number of symbols in X
n is  the number of symbols in Y
if m ≤ 2 or n ≤ 2 then
  compute optimal alignment using Alignment(X,Y)
call Space-Efficient-Alignment(X,Y[1:n/2])
call Backward-Space-Efficient-Alignment(X,Y[n/2+1:n])
let q be the index minimizing f(q,n/2) + g(q,n/2)
add (q,n/2) to the global list P
Divide-and-Conquer-Alignment(X[1:q],Y[1:n/2])
Divide-and-Conquer-Alignment(X[q+1:m],Y[n/2+1:n])
return P
```

- ✓ Space used is $O(m+n)$

Example



Second recursive call
x_3
x_2
x_1
y_1
y_2
y_3
y_4
First recursive call

RT analysis

The RT of Divide-and-Conquer-Alignment on strings of length m and n is O(mn).

Proof:

$T(m,n)$ – running time

$T(m,n) \leq cmn + T(q,n/2) + T(m-q,n/2)$

$T(m,2) \leq cm$

$T(2,n) \leq cn$

Particular case: m = n and q is in the middle

$T(n) \leq cn^2 + 2T(n/2)$

case 3 of the Master Thm $\Rightarrow T(n) = \Theta(n^2)$

RT analysis

General case:

$$T(m,n) \leq cmn + T(q,n/2) + T(m-q,n/2)$$

Show by induction that $T(m,n) = O(mn)$, that means

$$T(m,n) \leq kmn \text{ for some constant } k$$

Base case: $m \leq 2$ or $n \leq 2$ is true

Inductive step:

$$\begin{aligned} T(m,n) &\leq cmn + T(q,n/2) + T(m-q,n/2) \\ &\leq cmn + kqn/2 + k(m-q)n/2 \\ &= cmn + kqn/2 + kmn/2 - kqn/2 \\ &= (c + k/2)mn \end{aligned}$$

Inductive step works if $c+k/2 = k$, that means $c = k/2$ or $k = 2c$