CSE 421: Introduction to Algorithms

Dynamic Programming

Winter 2005
Paul Beame

Dynamic Programming

- Dynamic Programming
  - Give a solution of a problem using smaller sub-problems where all the possible sub-problems are determined in advance
  - Useful when the same sub-problems show up again and again in the solution

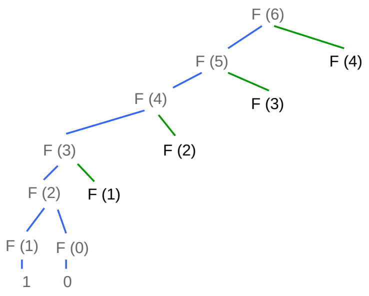
---

A simple case:
Computing Fibonacci Numbers

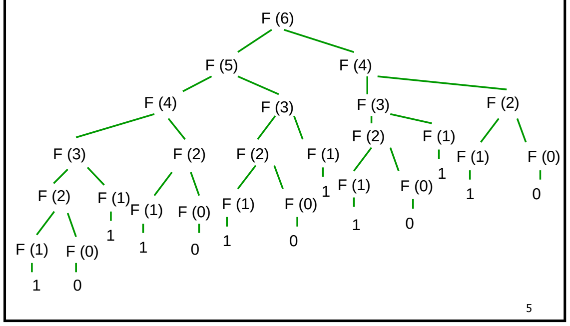
- Recall $F_n=F_{n-1}+F_{n-2}$ and $F_0=0, F_1=1$
- Recursive algorithm:
  - Fibo(**n**)

```
if n=0 then return(0)
else if n=1 then return(1)
else return(Fibo(n-1)+Fibo(n-2))
```

Call tree - start



Full call tree



Memoization (Caching)

- Remember all values from previous recursive calls
- Before recursive call, test to see if value has already been computed
- Dynamic Programming
  - Convert memoized algorithm from a recursive one to an iterative one

Fibonacci
Dynamic Programming Version

- FiboDP(**n**):

```
FiboDP(n):
    F[0]← 0
    F[1] ←1
    for i=2 to n do
        F[i]←F[i-1]+F[i-2]
    endfor
    return(F[n])
```

Fibonacci: Space-Saving Dynamic Programming

- FiboDP(**n**):

```
FiboDP(n):
    prev← 0
    curr←1
    for i=2 to n do
        temp←curr
        curr←curr+prev
        prev←temp
    endfor
    return(curr)
```

Dynamic Programming

- Useful when
  - same recursive sub-problems occur repeatedly
  - Can anticipate the parameters of these recursive calls
  - The solution to whole problem can be figured out with knowing the internal details of how the sub-problems are solved
    - principle of optimality

      "Optimal solutions to the sub-problems suffice for optimal solution to the whole problem"

Three Steps to
Dynamic Programming

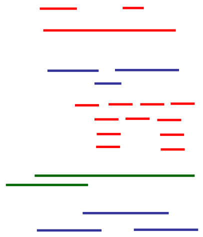
- Formulate the answer as a recurrence relation or recursive algorithm
- Show that the number of different values of parameters in the recursive calls is "small"
  - e.g., bounded by a low-degree polynomial
  - Can use memoization
- Specify an order of evaluation for the recurrence so that you already have the partial results ready when you need them.

Weighted Interval Scheduling

- Same problem as interval scheduling except that each request **i** also has an associated value or weight $\mathbf{w_i}$
  - $\mathbf{w_i}$ might be
    - amount of money we get from renting out the resource for that time period
    - amount of time the resource is being used $\mathbf{w_i=f_i-s_i}$
- Goal: Find compatible subset **S** of requests with maximum total weight

Greedy Algorithms for Weighted Interval Scheduling?

- No criterion seems to work
  - Earliest start time s_i
    - Doesn't work
  - Shortest request time f_i-s_i
    - Doesn't work
  - Fewest conflicts
    - Doesn't work
  - Earliest finish fime f_i
    - Doesn't work
  - Largest weight w_i
    - Doesn't work



Towards Dynamic Programming: Step 1 – A Recursive Algorithm

- Suppose that like ordinary interval scheduling we have first sorted the requests by finish time f_i so $f_1 \le f_2 \le \ldots \le f_n$
- Say request **i** comes **before** request **j** if **i< j**
- For any request **j** let **p**(**j**) be
  - the largest-numbered request before **j** that is compatible with **j**
  - or **0** if no such request exists
- Therefore {**1**,…,**p**(**j**)} is precisely the set of requests before **j** that are compatible with **j**

Towards Dynamic Programming: Step 1 – A Recursive Algorithm

- Two cases depending on whether an optimal solution **O** includes request **n**
  - If it **does** include request **n** then all other requests in **O** must be contained in {**1**,…,**p**(**n**)}
    - Not only that!
      - Any set of requests in {**1**,…,**p(n)**} will be compatible with request **n**
      - So in this case the optimal solution **O** must contain an optimal solution for {**1**,…,**p(n)**}
      - **"Principle of Optimality"**

Towards Dynamic Programming: Step 1 – A Recursive Algorithm

- Two cases depending on whether an optimal solution **O** includes request **n**
  - If it **does not** include request **n** then all requests in **O** must be contained in {**1**,…, **n-1**}
    - Not only that!
      - The optimal solution **O** must contain an optimal solution for {**1**,…, **n-1**}
      - **"Principle of Optimality"**

Towards Dynamic Programming: Step 1 – A Recursive Algorithm

- All subproblems involve requests {**1**,..,**i**} for some **i**
- For **i**=**1**,…,**n** let **OPT**(**i**) be the weight of the optimal solution to the problem {**1**,…,**i**}
- The two cases give

  $\mathbf{OPT}(\mathbf{n})=\max(\mathbf{w_n}+\mathbf{OPT}(\mathbf{p}(\mathbf{n})),\mathbf{OPT}(\mathbf{n-1}))$
- Also
  - $\mathbf{n} \in \mathbf{O}$ iff $\mathbf{w_n}+\mathbf{OPT}(\mathbf{p}(\mathbf{n}))>\mathbf{OPT}(\mathbf{n-1})$

Towards Dynamic Programming: Step 1 – A Recursive Algorithm

- Sort requests and compute array **p**[**i**] for each **i**=**1**,…,**n**

```
ComputeOpt(n)
    if n=0 then return(0)
    else
        u←ComputeOpt(p[n])
        v←ComputeOpt(n-1)
        if w_n+u>v then return(w_n+u)
                    else return(v)
    endif
```

Towards Dynamic Programming: Step 2 – Small # of parameters

- ComputeOpt(**n**) can take exponential time in the worst case
  - $\mathbf{2^n}$ calls if **p**(**i**)=**i**-**1** for every **i**
- There are only **n** possible parameters to ComputeOpt
- Store these answers in an array **OPT**[**n**] and only recompute when necessary
  - Memoization
- Initialize **OPT**[**i**]=**0** for **i**=**1**,…,**n**

Dynamic Programming: Step 2 – Memoization

```
ComputeOpt(n)
    if n=0 then return(0)
    else
        u←MComputeOpt(p[n])
        v←MComputeOpt(n-1)
        if $w_n$+u>v then
            return($w_n$+u)
        else return(v)
    endif
```

```
MComputeOpt(n)
    if OPT[n]=0 then
        v←ComputeOpt(n)
        OPT[n]←v
        return(v)
    else
        return(OPT[n])
    endif
```

Dynamic Programming Step 3: Iterative Solution

- The recursive calls for parameter **n** have parameter values **i** that are < **n**

```
IterativeComputeOpt(n)
    array OPT[0..n]
    OPT[0]←0
    for i=1 to n
        if $w_i$+OPT[p[i]] >OPT[i-1] then
            OPT[i] ←$w_i$+OPT[p[i]]
        else
            OPT[i] ←OPT[i-1]
        endif
    endfor
```

Producing the Solution

```
IterativeComputeOptSolution(n)
    array OPT[0..n], Used[1..n]
    OPT[0]←0
    for i=1 to n
        if $w_i$+OPT[p[i]] >OPT[i-1] then
            OPT[i] ←$w_i$+OPT[p[i]]
            Used[i]←1
        else
            OPT[i] ← OPT[i-1]
            Used[i] ←0
        endif
    endfor
```

```
i←n
S←∅
while i> 0 do
    if Used[i]=1 then
        S←S ∪ {i}
        i←p[i]
    else
        i←i-1
    endif
endwhile
```

Example

	1	2	3	4	5	6	7	8	9
s_i	4	2	6	8	11	15	11	12	18
f_i	7	9	10	13	14	17	18	19	20
w_i	3	7	4	5	3	2	7	7	2
p[i]									
OPT[i]									
Used[i]									

Example

	1	2	3	4	5	6	7	8	9
s_i	4	2	6	8	11	15	11	12	18
f_i	7	9	10	13	14	17	18	19	20
w_i	3	7	4	5	3	2	7	7	2
p[i]	0	0	0	1	3	5	3	3	7
OPT[i]									
Used[i]									

Example

	1	2	3	4	5	6	7	8	9
s_i	4	2	6	8	11	15	11	12	18
f_i	7	9	10	13	14	17	18	19	20
w_i	3	7	4	5	3	2	7	7	2
p[i]	0	0	0	1	3	5	3	3	7
OPT[i]	3	7	7	8	10	12	14	14	16
Used[i]	1	1	0	1	1	1	1	0	1

Example

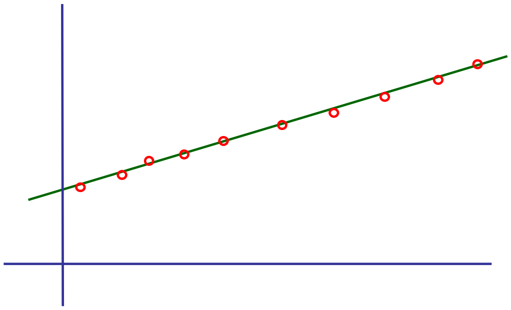
	1	2	3	4	5	6	7	8	9
s_i	4	2	6	8	11	15	11	12	18
f_i	7	9	10	13	14	17	18	19	20
w_i	3	7	4	5	3	2	7	7	2
p[i]	0	0	0	1	3	5	3	3	7
OPT[i]	3	7	7	8	10	12	14	14	16
Used[i]	1	1	0	1	1	1	1	0	1

S={9,7,2}

Segmented Least Squares

- Least Squares
  - Given a set **P** of **n** points in the plane $\mathbf{p_1}=(\mathbf{x_1},\mathbf{y_1}),\ldots,\mathbf{p_n}=(\mathbf{x_n},\mathbf{y_n})$ with $\mathbf{x_1}<\ldots<\mathbf{x_n}$ determine a line **L** given by $\mathbf{y}=\mathbf{ax}+\mathbf{b}$ that optimizes the totaled 'squared error'
    - $\mathrm{Error}(\mathbf{L},\mathbf{P})=\Sigma_i(\mathbf{y_i}-\mathbf{ax_i}-\mathbf{b})^2$
  - A classic problem in statistics
  - Optimal solution is known (see text)
    - Call this line(**P**) and its error error(**P**)

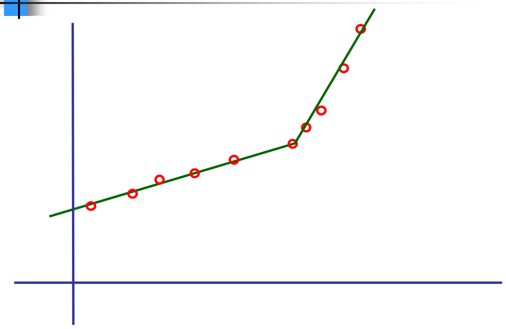
Least Squares



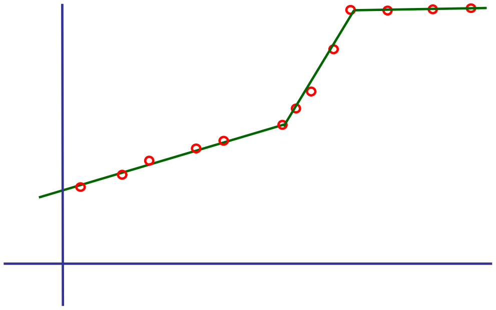
Segmented Least Squares

- What if data seems to follow a piece-wise linear model?

Segmented Least Squares



Segmented Least Squares



Segmented Least Squares

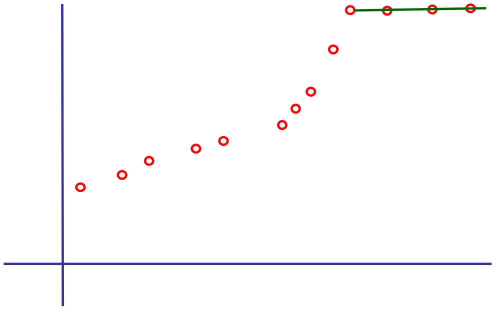
- What if data seems to follow a piece-wise linear model?
- Number of pieces to choose is not obvious
- If we chose **n-1** pieces we could fit with **0** error
  - Not fair
- Add a penalty of **C** times the number of pieces to the error to get a total penalty
- How do we compute a solution with the smallest possible total penalty?

Segmented Least Squares

- Recursive idea
  - If we knew the point p_j where the **last** line segment began then we could solve the problem optimally for points $p_1,\ldots,p_j$ and combine that with the last segment to get a global optimal solution
    - Let OPT(**i**) be the optimal penalty for points $\{p_1,\ldots,p_i\}$
    - Total penalty for this solution would be

$$\text{Error}(\{p_j,\ldots,p_n\}) + C + \text{OPT}(j-1)$$

Segmented Least Squares



Segmented Least Squares

- Recursive idea
  - We don't know which point is p_j
    - But we do know that $1 \le j \le n$
    - The optimal choice will simply be the best among these possibilities
  - Therefore

$$\text{OPT}(n)=\min_{1\le j\le n} \{\text{Error}(\{p_j,\ldots,p_n\}) + C + \text{OPT}(j-1)\}$$

Dynamic Programming Solution

```
SegmentedLeastSquares(n)
  array OPT[0..n], Begin[1..n]
  OPT[0]←0
  for i=1 to n
     OPT[i]←Error{(p_1,…,p_i)}+C
     Begin[i]←1
     for j=2 to i-1
          e←Error{(p_j,…,p_i)}+C+OPT[j-1]
          if e <OPT[i] then
               OPT[i] ←e
               Begin[i]←j
          endif
     endfor
  endfor
  return(OPT[n])
```

```
FindSegments
i←n
S←Ø
while i> 1 do
     compute Line({p_Begin[i],…,p_i})
     output (p_Begin[i],p_i), Line
     i←Begin[i]
endwhile
```

Knapsack (Subset-Sum) Problem

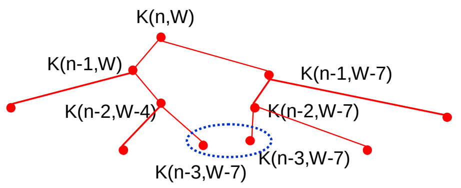
- Given:
  - integer **W** (knapsack size)
  - **n** object sizes $x_1, x_2, \ldots, x_n$
- Find:
  - Subset **S** of $\{1,\ldots, n\}$ such that $\sum_{i\in S} x_i \le W$ but $\sum_{i\in S} x_i$ is as large as possible

Recursive Algorithm

- Let **K**(**n**,**W**) denote the problem to solve for **W** and x_1, x_2, … , x_n
- For **n**>**0**,
  - The optimal solution for **K**(**n**,**W**) is the better of the optimal solution for either
  - **K**(**n-1**,**W**) or x_n+**K**(**n-1**,**W**-x_n)
  - For **n**=**0**
    - **K**(**0**,**W**) has a trivial solution of an empty set **S** with weight **0**

Recursive calls

- Recursive calls on list …,3, 4, 7



Common Sub-problems

- Only sub-problems are **K**(**i**,**w**) for
  - **i** = **0**,**1**,..., **n**
  - **w** = **0**,**1**,..., **W**
- Dynamic programming solution
  - Table entry for each **K**(**i**,**w**)
    - **OPT** - value of optimal soln for first **i** objects and weight **w**
    - **belong** flag - is x_i a part of this solution?
  - Initialize **OPT**[**0**,**w**] for **w**=**0**,...,**W**
  - Compute all **OPT**[**i**,*] from **OPT**[**i-1**,*] for **i**>**0**

Dynamic Knapsack Algorithm

```
for w=0 to W;  OPT[0,w] ← 0;   end for
for i=1 to n do
    for w=0 to W do
        OPT[i,w]←OPT[i-1,w]
        belong[i,w]←0
        if  $w \geq x_i$ then
            val ←$x_i$+OPT[i,w-$x_i$]
            if val>OPT[i,w] then
                OPT[i,w]←val
                belong[i,w]←1
    end for
end for
return(OPT[n,W])
```

Time O(nW)

Sample execution on 2, 3, 4, 7 with K=15

Saving Space

- To compute the value OPT of the solution only need to keep the last two rows of OPT at each step
- What about determining the set S?
  - Follow the **belong** flags **O**(**n**) time
  - What about space?

Three Steps to Dynamic Programming

- Formulate the answer as a recurrence relation or recursive algorithm
- Show that the number of different values of parameters in the recursive algorithm is “small”
  - e.g., bounded by a low-degree polynomial
- Specify an order of evaluation for the recurrence so that you already have the partial results ready when you need them.

Sequence Alignment: Edit Distance

- Given:
  - Two strings of characters **A**=$\mathbf{a_1\ a_2\ \ldots\ a_n}$ and **B**=$\mathbf{b_1\ b_2\ \ldots\ b_m}$
- Find:
  - The minimum number of edit steps needed to transform **A** into **B** where an edit can be:
  - insert a single character
  - delete a single character
  - substitute one character by another

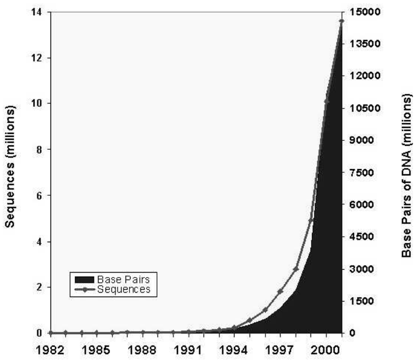
Sequence Alignment vs Edit Distance

- Sequence Alignment
  - Insert corresponds to aligning with a “–” in the first string
    - Cost δ (in our case **1**)
  - Delete corresponds to aligning with a “–” in the second string
    - Cost δ (in our case **1**)
  - Replacement of an **a** by a **b** corresponds to a mismatch
    - Cost α_{ab} (in our case **1** if **a≠b** and **0** if **a=b**)
- In Computational Biology this alignment algorithm is attributed to Smith & Waterman

Applications

- "diff" utility – where do two files differ
- Version control & patch distribution – save/send only changes
- Molecular biology
  - Similar sequences often have similar origin and function
  - Similarity often recognizable despite millions or billions of years of evolutionary divergence

Growth of GenBank



Recursive Solution

- Sub-problems: Edit distance problems for **all prefixes** of **A** and **B** that don’t include all of both **A** and **B**
- Let **D**(**i**,**j**) be the number of edits required to transform $\mathbf{a_1\ a_2\ \ldots\ a_i}$ into $\mathbf{b_1\ b_2\ \ldots\ b_j}$
- Clearly **D**(**0**,**0**)=**0**

Computing D(n,m)

- Imagine how best sequence handles the last characters a_n and b_m
- If best sequence of operations
  - deletes a_n then **D(n,m)=D(n-1,m)+1**
  - inserts b_m then **D(n,m)=D(n,m-1)+1**
  - replaces a_n by b_m then **D(n,m)=D(n-1,m-1)+1**
  - matches a_n and b_m then **D(n,m)=D(n-1,m-1)**

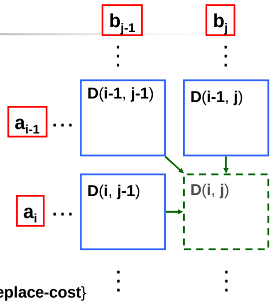
Recursive algorithm D(n,m)

```
if  n=0  then
    return (m)
elseif  m=0  then
    return(n)
else
    if  $a_n$=$b_m$  then
        replace-cost ← 0
    else
        replace-cost ← 1
    endif
    return(min{ D(n-1, m) + 1,
                D(n, m-1) +1,
                D(n-1, m-1) + replace-cost})
```

cost of substitution of a_n by b_m (if used)

Dynamic Programming

```
for j = 0 to m;  D(0,j) ← j; endfor
for i = 1 to n;  D(i,0) ← i; endfor
for i = 1 to n
    for j = 1 to m
        if  $a_i$=$b_j$ then
            replace-cost ← 0
        else
            replace-cost ← 1
        endif
        D(i,j) ←  min { D(i-1, j) + 1,
                        D(i, j-1) + 1,
                        D(i-1, j-1) + replace-cost}
    endfor
endfor
```



Example run with AGACATTG and GAGTTA

			A	G	A	C	A	T	T	G
		0	1	2	3	4	5	6	7	8
	0									
G	1									
A	2									
G	3									
T	4									
T	5									
A	6									

Example run with AGACATTG and GAGTTA

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2								
G	3								
T	4								
T	5								
A	6								

Example run with AGACATTG and GAGTTA

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1					
G	3								
T	4								
T	5								
A	6								

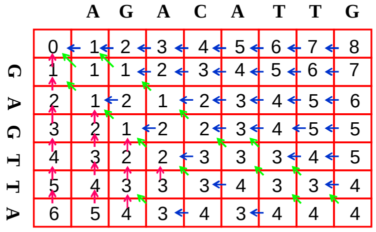
Example run with AGACATTG and GAGTTA

		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1	2	3	4	5	6
G	3	2	1	2	2	3	4	5	5
T	4								
T	5								
A	6								

Example run with AGACATTG and GAGTTA

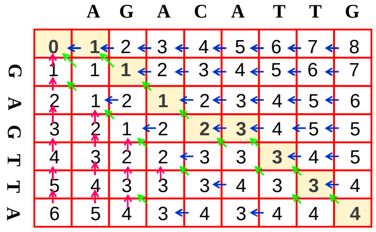
		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1	2	3	4	5	6
G	3	2	1	2	2	3	4	5	5
T	4	3	2	2	3	3	3	4	5
T	5	4	3	3	3	4	3	3	4
A	6	5	4	3	4	3	4	4	4

Example run with AGACATTG and GAGTTA



		A	G	A	C	A	T	T	G
	0	1	2	3	4	5	6	7	8
G	1	1	1	2	3	4	5	6	7
A	2	1	2	1	2	3	4	5	6
G	3	2	1	2	2	3	4	5	5
T	4	3	2	2	3	3	3	4	5
T	5	4	3	3	3	4	3	3	4
A	6	5	4	3	4	3	4	4	4

Example run with AGACATTG and GAGTTA



		A	G	A	C	A	T	T	G
	0	**1**	2	3	4	5	6	7	8
G	1	1	**1**	2	3	4	5	6	7
A	2	1	2	**1**	2	3	4	5	6
G	3	2	1	2	**2**	**3**	4	5	5
T	4	3	2	2	3	3	**3**	4	5
T	5	4	3	3	3	4	3	**3**	4
A	6	5	4	3	4	3	4	4	**4**

Reading off the operations

- Follow the sequence and use each color of arrow to tell you what operation was performed.
- From the operations can derive an optimal alignment

A G A C A T T G
_ G A G _ T T A

Saving Space

- To compute the distance values we only need the last two rows (or columns)
  - **O**(min(**m**,**n**)) space
- To compute the alignment/sequence of operations
  - seem to need to store all **O**(**mn**) pointers/arrow colors
- Nifty divide and conquer variant that allows one to do this in **O**(min(**m**,**n**)) space and retain **O**(**mn**) time
  - In practice the algorithm is usually run on smaller chunks of a large string, e.g. **m** and **n** are lengths of genes so a few thousand characters
    - Researchers want all alignments that are close to optimal
    - Basic algorithm is run since the whole table of pointers (**2** bits each) will fit in RAM
  - Ideas are neat, though

Saving space

- Alignment corresponds to a path through the table from lower right to upper left
  - Must pass through the middle column
- Recursively compute the entries for the middle column from the left
  - If we knew the cost of completing each then we could figure out where the path crossed
  - Problem
    - There are **n** possible strings to start from.
  - Solution
    - Recursively calculate the right half costs for each entry in this column using alignments starting at the other ends of the two input strings!
  - Can reuse the storage on the left when solving the right hand problem

Shortest paths with negative cost edges (Bellman-Ford)

- Dijsktra's algorithm failed with negative-cost edges
  - What can we do in this case?
  - Negative-cost cycles could result in shortest paths with length **-∞**
- Suppose no negative-cost cycles in G
  - Shortest path from **s** to **t** has at most **n-1** edges
    - If not, there would be a repeated vertex which would create a cycle that could be removed since cycle can't have –ve cost

Shortest paths with negative cost edges (Bellman-Ford)

- We want to grow paths from **s** to **t** based on the # of edges in the path
- Let Cost(**s**,**t**,**i**)=cost of minimum-length path from **s** to **t** using up to **i** hops.
  - $\mathrm{Cost}(\mathbf{v},\mathbf{t},\mathbf{0})=\begin{cases}\mathbf{0} \text{ if } \mathbf{v}=\mathbf{t}\\ \infty \text{ otherwise}\end{cases}$
  - $\mathrm{Cost}(\mathbf{v},\mathbf{t},\mathbf{i})=\min\{\mathrm{Cost}(\mathbf{v},\mathbf{t},\mathbf{i-1}), \min_{(v,w)\in E}(\mathbf{c}_{vw}+\mathrm{Cost}(\mathbf{w},\mathbf{t},\mathbf{i-1}))\}$

Bellman-Ford

- Observe that the recursion for Cost(**s**,**t**,**i**) doesn't change **t**
  - Only store an entry for each **v** and **i**
    - Termed OPT(**v**,**i**) in the text
- Also observe that to compute OPT(*,i) we only need OPT(*,i-1)
  - Can store a current and previous copy in O(**n**) space.

Bellman-Ford

```
ShortestPath(G,s,t)
    for all $v \in V$
        $OPT[v] \leftarrow \infty$
    $OPT[t] \leftarrow 0$
    for $i=1$ to $n-1$ do
        for all $v \in V$ do
            $OPT'[v] \leftarrow \min_{(v,w)\in E} (c_{vw}+OPT[w])$
        for all $v \in V$ do
            $OPT[v] \leftarrow \min(OPT'[v], OPT[v])$
    return $OPT[s]$
```

O(**mn**) time

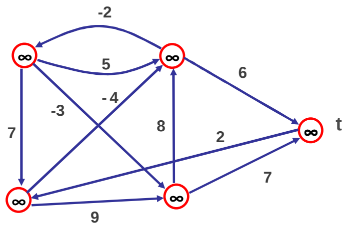
Negative cycles

- Claim: There is a negative-cost cycle that can reach t iff for some vertex **v∈V,** Cost(**v**,**t**,**n**)<Cost(**v**,**t**,**n-1**)
- Proof:
  - We already know that if there aren't any then we only need paths of length up to **n-1**
  - For the other direction
    - The recurrence computes **Cost**(**v**,**t**,**i**) correctly for **any** number of hops **i**
    - The recurrence reaches a fixed point if for every **v∈V,** **Cost**(**v,t,i**)=**Cost**(**v,t,i-1**)
    - A negative-cost cycle means that eventually some **Cost**(**v,t,i**) gets smaller than any given bound
      - Can't have a –ve cost cycle if for every **v∈V,** **Cost**(**v**,**t**,**n**)=**Cost**(**v**,**t**,**n-1**)

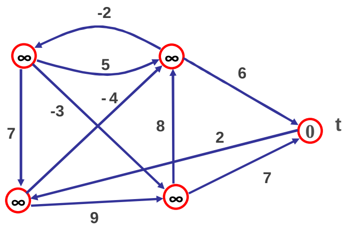
Last details

- Can run algorithm and stop early if the OPT and OPT’ arrays are ever equal
  - Even better, one can update only neighbors **v** of vertices **w** with OPT’[**w**]≠OPT[**w**]
- Can store a successor pointer when we compute OPT
  - Homework assignment
- By running for step **n** we can find some vertex **v** on a negative cycle and use the successor pointers to find the cycle

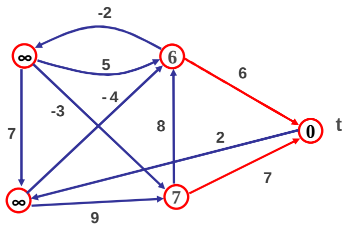
Bellman-Ford



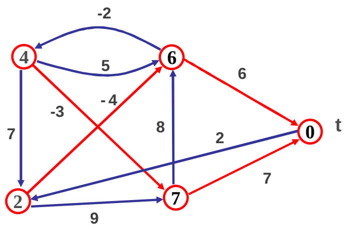
Bellman-Ford



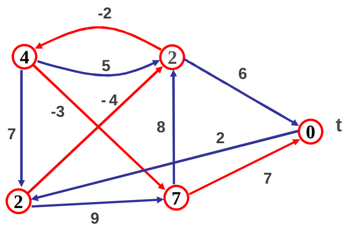
Bellman-Ford



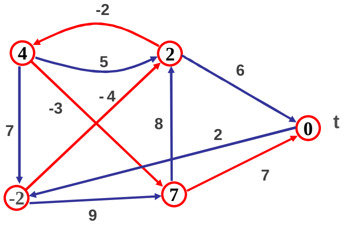
Bellman-Ford



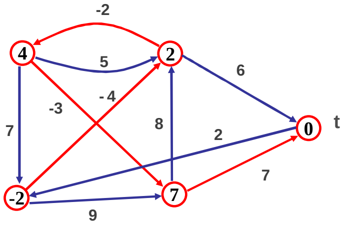
Bellman-Ford



Bellman-Ford



Bellman-Ford



Bellman-Ford with a DAG

Edges only go from lower to higher-numbered vertices
- Update distances in reverse order of topological sort
- Only one pass through vertices required
- O(**n**+**m**) time