Dynamic Programming

Algorithms Design Paradigms

- Divide and Conquer
  - Quick Sort; Binary Search; ....
- Greedy
  - Where a divide phase doesn't work !
  - Choose the best possible recourse leading to the optimality
- Dynamic Algorithms
  - Where a divide phase doesn't work!
  - Use backtracking when a selection doesn't turn to be optimal.

Algorithms Design Paradigms (contd)

- Greed
  - Build up a solution incrementally, myopically optimizing some local criterion.
- Divide-and-conquer
  - Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.
- Dynamic programming
  - Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.
  - a powerful technique that can be used
    - to solve many problems in time $O(n^2)$ or $O(n^3)$ for which a naive approach would take exponential time.

Introduction to Dynamic Programming

- "Those who do not remember the past are condemned to repeat it"
  - has evolved into a major paradigm of algorithm design in CS.
- How did the term originate?
  - In 1957, Richard Bellman coined it as a method of solution
  - to describe solution to a type of an optimum control problem.

Etymology

- Dynamic
  - is something that depends on the current state.
- Programming imply a series of choices
  - statically programmed radio programs vis-à-vis phone-in programs.
- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
  - Bellman sought an impressive name to avoid confrontation.
  - "it's impossible to use dynamic in a pejorative sense"
  - "something not even a Congressman could object to"
- Top-down design vrs Bottom-up design
  - Dynamic Progg may take either of the Bottom-up OR the Top-Down approach – though more logical is Bottom-up

Applications

- Areas.
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations research.
  - Computer science: theory, graphics, AI, systems, ….

- Some famous dynamic programming algorithms.
  - Viterbi for hidden Markov models.
  - Unix diff for comparing two files.
  - Smith-Waterman for sequence alignment.
  - Bellman-Ford for shortest path routing in networks.
  - Cocke-Kasami-Younger for parsing context free grammars.

Introduction to Dynamic Programming(contd)

- Dynamic Progg is similar to divide and conquer approach
  - i.e. typically provides many ways
    - to divide the original problem into subproblems.
  - it is not evident which division leads to solution
    - then, how can this approach work ?

Introduction to Dynamic Programming(contd)

- On the other hand it is
  - opposite of the greedy approach
    - implicitly explores all the possible solutions
  - is useful in those applications
    - which require solution to all the subproblems that may be used in framing the eventual optimal solution
    - e.g…………..
  - careful decomposition of a problem into subproblems
    - avoids any repetition
    - builds solutions to larger & larger subproblems
  - is dangerously close to the brute-force approach
    - but, then is it advisable ?

Introduction to Dynamic Programming (contd)

- "Top-down design is natural, powerful
  - but what about the efficiency concerns?"
- Recursion if not controlled properly…………
- Often compilers do not do justices to such top-down designed algorithms
  - need to provide more information to the compilers
- How?
  - Using a table of choices instead of recursion.
  - e.g. a language like Haskell

Introduction to Dynamic Programming (contd)

- Underlying idea
  - Avoid duplication of efforts
  - Use a table where solutions of subinstances are stored
  - implicitly explore the space of all possible solutions
    - but without ever examining them all explicitly

An illustration – Fibonacci series

- Calculate Fibonacci numbers

```
int fib-recurs(int n) {
  if (n < 2) return n;
  else
    return fib(n-1) + fib(n-2);
}
```

  - Simple, elegant!
  - But .... Let's analyze it's performance!

Analysis - Fibonacci series

- Analysis
  - If time to calculate Fib(n) is t_n
    then $t_n = t_{n-1} + t_{n-2}$
  - Now $t_1 = t_2 = c$
  - So $t_n = c'\text{Fib}(n-2) = O(2^n)$

```
int fib( int n ) {
  if ( n < 2 ) return n;
  else
    return fib(n-1) + fib(n-2);
}
```

Analysis (contd)

- Prove that time to calculate F(n) is $O(2^n)$
- Actually $\mathbf{f_{n+1} = O(1.618^n)}$

  this is definitely not an efficient algorithm!!

Simulation

- To appreciate the root-cause of inefficiency
  - draw the call-graph for fib-recurs(6)
- What is problematic in the call-graph ?
- What is the minimum running time required ?
- So the issue is
  - Is it not possible to compute F_n with $\theta(n)$ simple statements (which involve no further calls) and remembering n smaller values?
- How about the iterative solution to the problem?

Fibonacci series – Iterative approach

```
int fib( int n ) {
  int f1, f2, f;
    if ( n < 2 ) return n;
  else {
    f1 = f2 = 1;
    for( k = 2; k < n; k++ ) {
      f = f1 + f2;
      f2 = f1;
      f1 = f;
      }
  return f;
}
```

Fibonacci series – Iterative approach

```
int fib( int n ) {
   int f1, f2, f;
     if ( n < 2 ) return n;
  else {
    f1 = f2 = 1;
    for( k = 2; k < n; k++ ) {
      f = f1 + f2;
      f2 = f1;
      f1 = f;
      }
  return f;
}
```

Note the `f1`, `f2` here

We start by solving the smallest problems

Then use those solutions to solve bigger and bigger problems

Fibonacci series – Iterative approach

- Exercise:
  - draw the call structure for the above.
  - Is it a tree of a DAG ?
- Which approach ‘ve we followed here?
  - Bottom up or top down?

Dynamic Programming – Basic Paradigm

- Dynamic Approach
  - Solve the small problems
  - Store the solutions
  - Use those solutions to solve larger problems
- But, what does dynamic programming approach require additionally ?
  - space
- Compare the time complexities of the Iterative Fibonacci with Recursive Fibonacci.
  - $O(\,n\,)$ *vrs* $O(\,n^n\,)$ for the recursive case!

Basic Paradigm (contd)

- It is clear that
  - the growth of the redundant calculations is explosive in recursive algorithms.
- Why can't a compiler handle a recursive program the same way ? i.e.
  - by keeping a stored list of precomputed values and NOT making a recursive call each time.
- But, then this efficiency is a "borrowed" one and
  - NOT inherently built into the algorithmic design approach.

Binomial Coefficients

Problem: Binomial Coefficients

- A binomial coefficient, denoted *C(n, k),* is the number of combinations of *k* elements from an *n*-element set *(0 ≤ k ≤ n).*
- That is……….

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \text{ for } 0 < k < n$$

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < \mathrm{k} < \mathrm{n} \\ 1 & k = 0 \text{ or } k = n \end{cases}$$

$$\binom{n}{0} = 1 \qquad \binom{n}{n} = 1$$

Problem: Binomial Coefficients

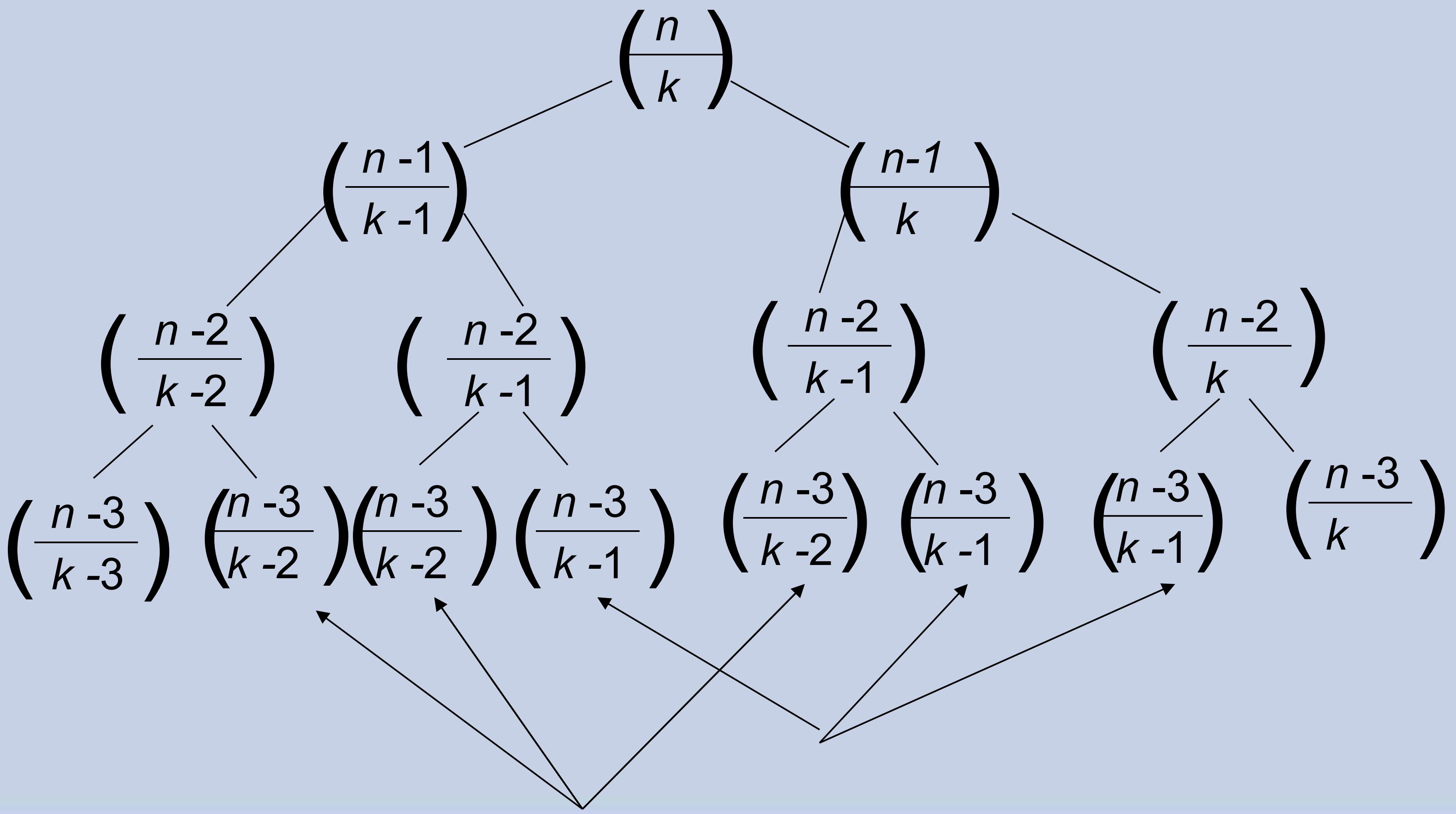
- We can calculate it directly by the function below

```
Algorithm C(n,k)
if k = 0 or k = n
    then return 1
else return C(n-1, k-1)+ C(n-1, k)
```

- The recurrence relation (a problem → 2 overlapping subproblems) would be given by……
  - $C(n, k) = C(n-1, k-1) + C(n-1, k)$, for $n > k > 0$, and
  - $C(n, 0) = C(n, n) = 1$

Analysis

- If is apparent that many values of **C(i j) i< n, j<k** are calculated over and over again………



Dynamic Solution

- Use a matrix B of n+1 rows, k+1 columns where

$$B[n, k] = \binom{n}{k}$$

- *Establish* a recursive property. Rewrite in terms of matrix B:

$$B[i, j] \begin{cases} = B[i-1, j-1] + B[i-1, j] & , 0 < j < i \\ = 1 & , j = 0 \text{ or } j = i \end{cases}$$

- Solve all "smaller instances of the problem" in a *bottom-up* fashion by computing the rows in *B* in sequence starting with the first row.

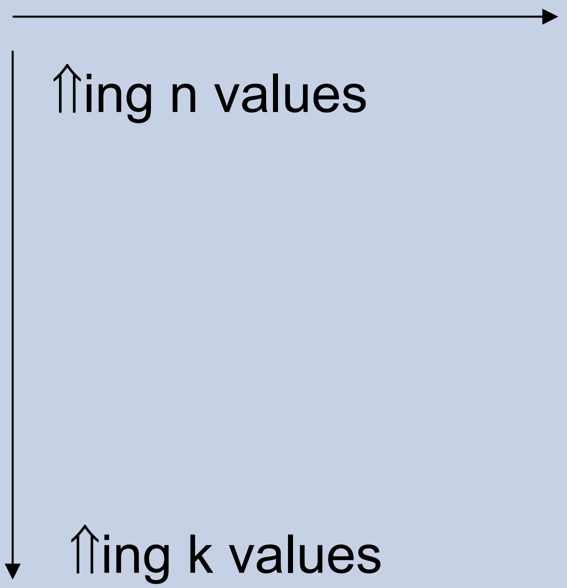
Compute B[4,2]=$\binom{4}{2}$

- Row 0: **B[0,0] =1**
- Row 1: **B[1,0] = 1**
  B[1,1] = 1
- Row 2: **B[2,0] = 1**
  B[2,1] = B[1,0] + B[1,1] = 2
  B[2,2] = 1
- Row 3: **B[3,0] = 1**
  B[3,1] = B[2,0] + B[2,1] = 3
  B[3,2] = B[2,1] + B[2,2] = 3
- Row 4: **B[4,0] = 1**
  B[4,1] = B[3, 0] + B[3, 1] = 4
  B[4,2] = B[3, 1] + B[3, 2] = 6

Alternate Approach

- Instead, a table of intermediate values – as

```
                 1                      k=0
              1     1
           1     2     1
        1     3     3     1
     1     4     6     4     1
  1     5    10    10     5     1     k=5
n=0                            n=5
```



- Each entry O(1) time and there are O(n^2) entries
- Therefore, O(n^2) time to calculate B.C.

Dynamic Program

```
bin($n$,$k$ )
1. **for** $i$ = 0 **to** $n$ // every row
2.    **for** $j$ = 0 **to** minimum( $i$, $k$ )
3.       **if** $j$ = 0 or $j$ = $i$   // column 0 or diagonal
4.          **then** B[ $i$ , $j$ ] = 1
5.          **else** B[ $i$ , $j$ ] =B[$i$ -1, $j$ -1] + B[$i$ -1, $j$ ]
6. **return** B[ $n$, $k$ ]
```

Dynamic Program

- What is the run time?
- How much space does it take?
- If we only need the last value, can we save space?
- All values in column 0 are 1
- All values in the first k+1 diagonal cells are 1
- j != i and 0 < j <=min{i,k} ensures we only compute B[i, j] for j < i and only first k+1 columns.

Number of iterations

$$\sum_{i=0}^{n} \sum_{j=0}^{\min(i,k)} 1 = \sum_{i=0}^{k} \sum_{j=0}^{i} 1 + \sum_{i=k+1}^{n} \sum_{j=0}^{k} 1 =$$

$$\sum_{i=0}^{k} (i+1) + \sum_{i=k+1}^{n} (k+1) =$$

$$\frac{(k+2)(k+1)}{2} + (n-k)(k+1) =$$

$$\frac{(2n-k+2)(k+1)}{2}$$

Overview and Comments

Problems/Subproblems call structure

- Consider a recursive algorithm A known for a problem P. Then,
  - the relationship that exists between the problem instance of P and its subproblems can be characterized by a DAG, whose
    - vertices
      - all input instances of the problem
    - directed edges i.e. $I \rightarrow J$
      - all pairs such that when algorithm is invoked on instance I, it makes a recursive call to instance J.
- E.g. again draw the call-graph of fib-iters(6)

Problems/Subproblems call structure (contd)

- Basically a graph traversal procedure like DFS, but doesn't color the vertex.
  - Therefore, a memory less graph traversal
  - in a recursive call, it traverses every path in the DAG

- Summarizing……….
  - if using a recursive strategy to solve a problem with input instance P,
    - it is to be solved for all the vertices that are reachable from P, in the subproblem graph.
  - What if there are multiple paths to a subproblem in DAG?
    - the procedure will take exponential no of paths in DAG.

Summarizing.....(contd)

- What if we aren't able to determine the topological order (dependency graph)
  - from the knowledge of the subproblem?
    - invoke the DFS and note the reverse topological order at the finish/postorder time.
- A recursive algorithm explores every edge in the sense of making the recursive call while
  - the DFS skeleton only explores edges to undiscovered vertices and
  - it checks the other edges.

DP version of a recursive algorithm

- def: A dynamic programming version of a given recursive algorithm A, denoted as DP(A)
  - is a procedure that, given a top-level problem P to solve,
    - performs a DFS on the subproblem graph for A(P) and
    - as the intermediate solutions are found for the subproblems, they are recorded in the ADT dictionary viz. **soln**…..(i.e**. Dict soln**)
- a process called memo…ization.

Memoization

- Recall that
  - the operations on ADT **Dict** are **create, member, insert** and **store**.
- We shall devise an algorithm for this process viz algorithm memo-ization(A, DP(A)).
  - What is expected out of memo-ization(A, DP(A))?
- Memoization can be be applied
  - only to functions which are referentially transparent i.e.
    - those that do not have side effects

Algorithm Memoization

- Before any recursive call on subproblem **Q**, check the dictionary **soln** to see if the solution for **Q** has been stored.
  - If no solution has been stored, go ahead with the recursive call,
    - thereby treating **Q** as a white vertex and treating **P**→**Q** as a tree edge.
  - IF a solution has been stored for **Q**,
    - retrieve the stored solution and do not make the recursive call, thereby treating Q as a black vertex.
  - Just before returning the solution for **P**,
    - store it in the dictionary **soln**……effectively coloring the vertex **P** black.
- /* The subproblem graph must be acyclic, because vertices are not colored gray */

Fibonacci – Dynamic Programming version

```
Algorithm fibDP(soln, k)
1.  if (k<2)
2.    fib = k;
3.  else
4.    if (member(soln, k-1) == false)
5.       $f_1$= fibDP(soln, k-1);
6.    else $f_1$= retrieve(soln, k-1);
7.    if (member(soln, k-2) == false)
8.       $f_2$= fibDP(soln, k-2);
9.    else $f_2$= retrieve(soln, k-2);
10.   fib = $f_1$ + $f_2$
11. store(soln, k, fib)
12. return fib
```

Fibonacci – Dynamic Programming version

Algorithm DP(fib(n)):

```
1.    Dict soln = create(n);
3.          return fibDP(soln,n);
4.    fibDP(soln, k)
```

Coin Changing Problem

The Coin changing problem

- to devise an algorithm for paying a given amount to a customer using the smallest possible number of coins.
- Two variations
  - Given a fixed denomination, unlimited supply of coins
  - Given different series of denominations, but limited supply.
- Two approaches
  - Greedy approach works……but….not always…
  - Dynamic programming approach is required

Algorithm MakeChange(SetofCoins n)

```
// Algorithm MakeChange(SetofCoins n) /* Make change for n
   units using the least possible number of coins.
1. const C = {100, 25, 10, 5}
2. S ← ∅ {S – the set to hold the solution}
3. s ← 0 {s – the sum of items in S}
4. while s ≠ n do
5.      x ← the largest item in C such that s + x ≤ n
6.     if there is no such item then
7.                     return "no solution found"
8.      S ← S U { a coin value x}
9.      s ← s + x
10. return S.
```

- Which design approach does this algorithm use. Why ?
- Limitations ?

The Dynamic Programming approach

- Setup a table c[1…n, 0..N]
  - columns: denominations available viz. $1 \leq i \leq n$
  - rows: amount to be paid is j viz. $0 \leq j \leq N$
- let **c[n, N]** - the minimum number of coins of denominations n required for the amount **N**
- **c[i,0]** – has to be zero for every **i**
- To pay an amount j using coins of denomination **1** to **i**
  - either do not use any coins of denomination **i** in which case, **c[i,j] = c[i-1, j]**
  - or, $c[i, j] = 1 + c[i, j-d_i]$
- But, the objective is to minimize the number of coins used and so chose
  - $c[i, j] = \min \{1 + c[i, j-d_i], c[i-1, j]\}$
  - e.g.

Illustration

- e.g. consider paying an amount of 8 here.
- which coins would be selected ?

	Amount	0	1	2	3	4	5	6	7	8
0	$d_0=0$	0	0	0	0	0	0	0	0	0
1	$d_1=1$	0	1	2	3	4	5	6	7	8
2	$d_2=4$	0	1	2	3	1	2	3	4	2
3	$d_3=6$	0	1	2	3	1	2	1	2	2

– Note that the table is filled using

```
c[i, j] = min {1+ c[i, j-di], c[i-1, j]}
```

Dynamic Programming Algorithm

```
Algorithm coins(N)
// making change for N units using coinages from d[1..n]
1. d[1..n] = [1,4,6]
2. for i=1 to n
3.   for j=1 to N
4.     if i=1 && j < d[i] then c[i,j]=∞
5.     elseif i=1 then c[i,j]= 1 + c[1,j-d[1]]
6.     elseif j < d[i] then c[i,j]=c[i-1, j]
7.     else min{c[i-1,j], 1 + c[1,j-d[1]]}
8. return c[n,N]
```

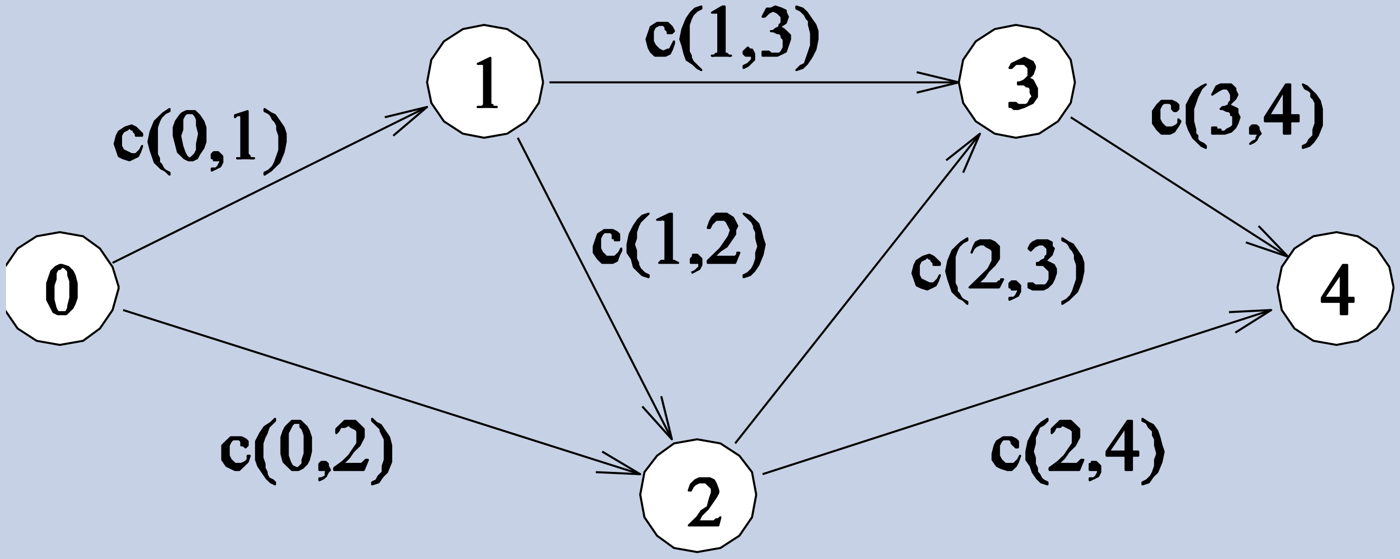
Shortest Paths in DAGs

DP: Shortest Paths in DAGs

- Consider the problem of finding a shortest path between a pair of vertices in an acyclic graph.
- An edge connecting node *i* to node *j* has cost *c(i,j)*
- The graph
  - contains *n* nodes numbered *0, 1,…, n-1*, and
  - has an edge from node *i* to node *j* only if *i < j*.
  - Node 0 is source and node *n-1* is the destination.

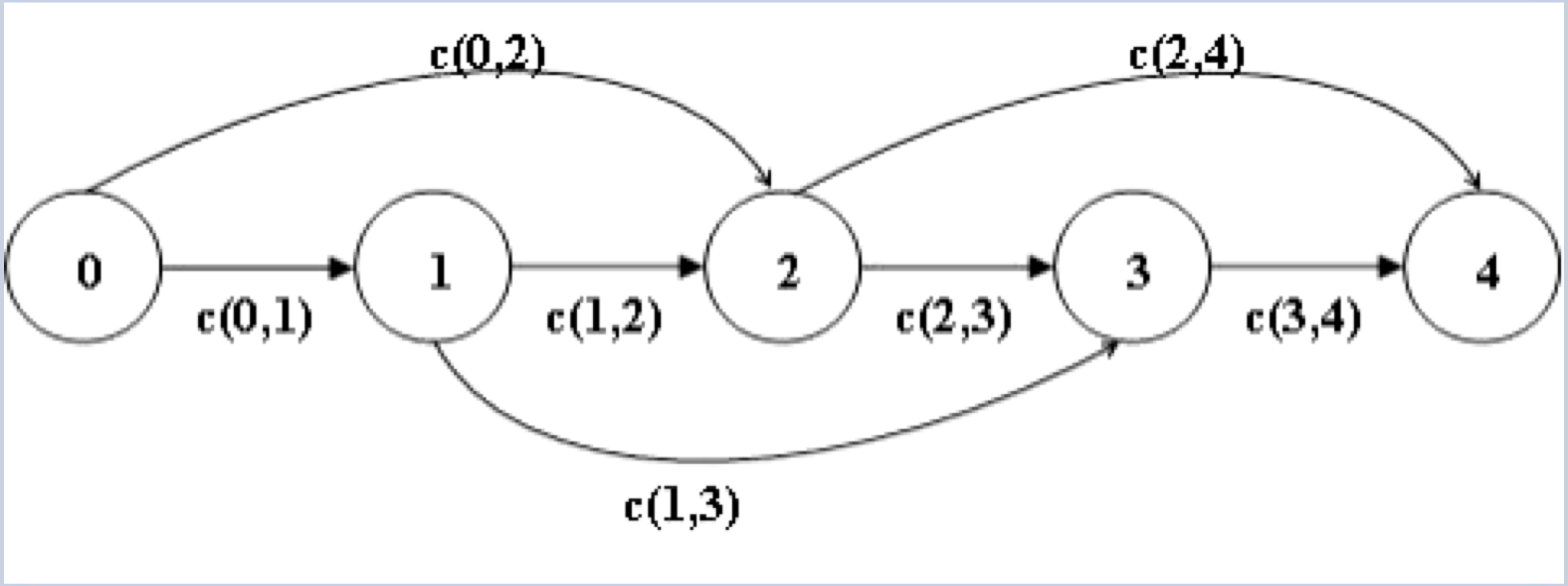
DP: Shortest Paths in DAGs

- The nodes of a DAG can be linearized
  - e.g. for the graph show below, what would be the linearized DAG



DP: Shortest Paths in DAGs

- The nodes of a DAG can be linearized
  - e.g. for the graph show above, the linearized DAG would be as follows



DP: Shortest Paths in DAGs

- The advantages are :
  - It is easy to compute the shortest distances from a source node to any other node in the graph e.g.
  - `dist(4) = min{dist(3) + c(3,4), dist(2) + c(2,4)}`
  - such relation can be written for every node in the DAG
    - by the time we reach a particular node, we already have the “DP table” information we need to compute the shortest distance to that node.
  - hence, we are able to compute all distances in a single pass

DP: Shortest Paths in DAGs

- Let *f(x)* be the cost of the shortest path from node 0 to node *x*.

$$f(x) = \begin{cases} 0 & x = 0 \\ \min_{0 \le j < x} \{ f(j) + c(j, x) \} & 1 \le x \le n - 1 \end{cases}$$

DP: Shortest Paths in DAGs

Algorithhm SPDAG

```
•   initialize all dist(.) values to infinity
•   dist(s)=0
1.  for each v Є V\{s}, in linearized order:
2.      dist(v) = min_{u,vЄE}{dist(u) + l(u,v)}
```

0/1 Knapsack Problem

The 0/1 Knapsack problem

- Knapsack problem.
  - Given n objects and a "knapsack."
  - Item i weighs w_i > 0 kilograms and has value v_i > 0.
  - Knapsack has capacity of W kilograms.
  - Goal: fill knapsack so as to maximize total value.

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

- Ex: Let W = 11, then
  - { 3, 4 } has value 40
- Greedy
  - repeatedly add item with maximum ratio v_i / w_i.
  - Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

The 0/1 Knapsack problem

- The greedy approach does not work
- consider an instance of $\mathbf{i}$ items, $1 \le i \le n$,
  - weights $w_1, w_2, w_3 \ldots w_n$,
  - values $v_1, v_2, v_3 \ldots . v_n$,
  - knapsack capacity j with $1 \le j \le W$,
  - $V[i,j]$ be the value of an optimal solution to this instance
    - i.e. the value of the most valuable subset of first i items that fit into the knapsack of capacity j.

Dynamic Programming: False Start

- Define
  - OPT = v(i) = max profit subset of items 1, …, i.
    - Case 1: OPT does not select item i.
      - OPT selects best of { 1, 2, …, i-1 }
    - Case 2: OPT selects item i.
      - accepting item i does not immediately imply that we will have to reject other items
      - without knowing what other items were selected before i, we don't even know if we have enough room for i
  - Conclusion. Need more sub-problems!

DP: Adding a New Variable

- Define
  - OPT = v(i, j) = max profit subset of items 1, …, i with weight limit j.
    - Case 1: OPT does not select item i.
      - OPT selects best of { 1, 2, …, i-1 } using weight limit j
    - Case 2: OPT selects item i.
      - new weight limit = j – w_i
      - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

$$OPT = v(i,\ j) = \begin{cases} 0 & \text{if } \mathrm{i} = 0 \\ v(i-1,\ j) & \text{if } \mathrm{w_i} > j \\ \max\left\{ v(i-1,\ j),\quad v(i)\ +\ v(i-1,\ j-w_i) \right\} & \text{otherwise} \end{cases}$$

DP: Adding a New Variable : Further Exp

- Two cases to consider
  - if the item **`i`** is not included in the above subset, the optimal solution is **`V[i-1,j]`**
  - if the item **`i`** is included in the above subsets, the optimal subset is made of
    - this item and
    - an optimal subset of first **`i-1`** items that fit into the knapsack of capacity $j-w_i$.
  - Hence, the optimal solution is $v_i + V[i-1,j-w_i]$
- Therefore, the optimal solution among all feasible subsets of the first i items is the maximum of the above two values i.e.

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

The 0/1 Knapsack problem : DP Table

- The table shows the values of V[i,j] where
  - i is the number of items that fit into the knapsack of capacity j

	0	$j-w_i$	j	W
0	0	0	0	0
i-1	0	$V[i-1, j-W_i]$	V[i-1, j]	
i	0		V[i, j]	
n	0			goal

The 0/1 Knapsack problem : DP Illustration

- Consider W = 5, n=5,v_i and w_i as shown below

item	Weight	value
1	2	12
2	1	10
3	3	20
4	2	15

The 0/1 Knapsack problem : DP Illustration

- The recurrences

$$V[i,j] = \max\{V[i\text{-}1,j],\ vi + V[i-1, j\text{-}wi]\},\quad if\ \ j - w_i >= 0$$
$$= V[i\text{-}1,j]\ \ if\ \ j - w_i < 0$$

	i and j	0	1	2	3	4	5
	0	0	0	0	0	0	0
$W_1=2$, $v_1=12$	1	0	0	12	12	12	12
$W_2=1$, $v_2=10$	2	0	10	12	22	22	22
$w_3=3$, $v_3=20$	3	0	10	12	22	30	32
$w_4=2$, $v_4=15$	4	0	10	15	25	30	37

The DP approach with memoization

	i & j	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1=2$, $v_1=12$	1	0	0	12	12	12	12
$w_2=1$, $v_2=10$	2	0	–	12	22	–	22
$w_3=3$, $v_3=20$	3	0	–	–	22	–	32
$w_4=2$, $v_4=15$	4	0	–	–	–	–	37

DP approach with memoization algorithm

```
ALGORITHM   MKnapsack(i, j)
//Implementn of the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first
// items being considered and a nonnegative integer j indicating the
   knapsack's capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays Weights [1...n], Values [1...n]
//and table V[0…n, 0…W] whose entries are initialized with -1's except for
//row 0 and column 0 initialized with 0's

1. if  V[i, j] < 0
2.       if j < Weights[i]
3.              values ← MKnapsack(i-1, j)
4.      else
5.              values ← max(MKnapsack(i-1, j),
6.                     Values[i] +  MKnapsack(i-1, j-Weights[i])
7.      V[i, j]← Value
8. return V[i, j]
```

Tutorial Exercise1

- Solve the following instance of the 0/1 knapsack problem using dynamic programming

Weight	1	2	3	2
Profit	20	15	25	12

- The capacity of the knapsack is W = 5
- Use the following recurrence

$$OPT = v(i,\ j) = \begin{cases} 0 & \text{if } \mathrm{i} = 0 \\ v(i-1,\ j) & \text{if } \mathrm{w_i} > j \\ \max\left\{ v(i-1,\ j),\quad v(i)\ +\ v(i-1,\ j-w_i) \right\} & \text{otherwise} \end{cases}$$

Tutorial Exercise1…Correct solution ??

- Verify whether the following formulation is correct or not ?

$$OPT = v(i,\ j) = \begin{cases} 0 & \text{if } \mathrm{i} = 0 \\ v(i-1,\ j) & \text{if } \mathrm{w_i} > j \\ \max\left\{v(i-1,\ j),\quad v(i)\ +\ v(i-1,\ j-w_i)\right\} & \text{otherwise} \end{cases}$$

	i and j	0	1	2	3	4	5
	0	0	0	0	0	0	0
$W_1=1$, $v_1=20$	1	0	20	20	20	20	20
$W_2=2$, $v_2=15$	2	0	20	20	35	35	35
$w_3=3$, $v_3=25$	3	0	20	20	35	45	45
$w_4=2$, $v_4=12$	4	0	20	20	35	45	55

Tutorial Exercise2

- Suppose the semester is coming to an end and you have to work on the assigned projects of n courses in a total of H hours. Each course is to be graded on a scale of 1 to g with a higher grade being the better one. You diligently examined the nature of each project problem, amount of work needed, the psychology of the instructor and how much work your partners are doing in the project, to come up with the following estimate functions.

  For all $1 \le i \le n$, you have a function

  $f_i : \{0, \ldots\ldots\ldots, H\} \rightarrow \{1, \ldots\ldots\ldots\ldots, g\}$

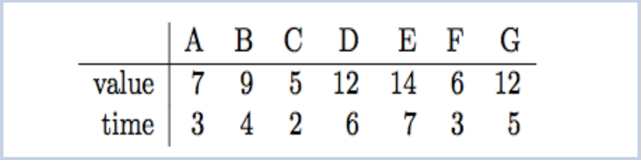
  such that $f_i(k)$ gives the grade you are likely to get if you spend k hours working on the project for course i.

  Your goal, of course, is to maximize the total of all the grades that you get.

  Design an algorithm to do so. Assume that all the quantities n, g, H are positive natural numbers and the functions f_i are non-decreasing i.e. you do not get a worse grade if you put in more time.

Tutorial Problem3

- Imagine you have a homework assignment with different parts labeled A through G.
- Each part has a “value” (in points) and a “size” (time in hours to complete). For example, say the values and times for our assignment are:



	A	B	C	D	E	F	G
value	7	9	5	12	14	6	12
time	3	4	2	6	7	3	5

- Say you have a total of 15 hours: which parts should you do?
- Two cases :
  - Partial credit allowed
  - Partial credit not allowed
- what is the best total value possible?

Tutorial Exercise4

- Problem Definition
- An example:
  - Let n =4 i.e. four weeks
  - The values of l_i and h_i are as per the following table.

	Week 1	Week 2	Week 3	Week 4
l_i	10	1	10	10
h_i	5	50	5	1

  - What would be the plan for maximum value?
- Analyze whether the algorithm next, solves the problem correctly. Give a counterexample with
  - the correct optimal answer and the one returned by the algorithm for the counterexample

Tutorial Exercise4….

```
Algorithm JobPlanSched($l_i$,$h_i$, n)
// assume that $h_i$=$l_i$=0, when i>n
1. for i = 1 to n
2.    if $h_{i+1}$ > $l_i$+$l_{i+1}$ then
3.        print "choose no job in week i"
4.        print "choose a high-stress job in week
   i+1"
5.        continue with iteration i+2
6.    else
7.        print "choose a low-stress job in week i"
8.        continue with iteration i+1
9.    endif
10.end
```

	Week 1	Week 2	Week 3
l_i	2	2	2
h_i	1	5	3

Tutorial Exercise5

- Suppose the job of a firm is to manage the construction of billboards on the Jammu-KAtra Gauravpath that runs South-North for M kms. The possible sites for the billboards are given by numbers $x_1, x_2, x_3, x_4, x_5, \dots x_n$ each in the interval $[0 \dots M]$. x_i's are indicating the position of the billboards along the Gauravpath in kms, measured from its northern end. If it is decided to place a billboard at location x_i, then the firm receives a revenue of $r_i > 0$.

  Regulations of the JK UT require that no two of the billboards be within less than or equal to 5 kms of each other. Thus, as part of the optimization problem, the firm has to decide where to place the billboards at a subset of the sites $x_1, x_2, x_3, x_4, x_5, \dots x_n$ so as to maximize the total revenue, subject to this constraint.

  Give an algorithm that takes in an instance of this problem as input and returns the maximum total revenue that can be obtained from any valid subset of sites. Give its running time also.

Tutorial Exercise5…

- An example :

  Suppose **M** = 20, n = 4

  $[x_1,x_2,x_3,x_4] = [6,7,12,14]$

  $[r_1,r_2,r_3,r_4] = [5,6,5,1]$

  Then, what would be the optimal solution ?