Dynamic Programming

Algorithm : Design & Analysis
[16]

In the last class…

- Shortest Path and Transitive Closure
- Washall's Algorithm for Transitive Closure
- All-Pair Shortest Paths
- Matrix for Transitive Closure
- Multiplying Bit Matrices - Kronrod's Algorithm

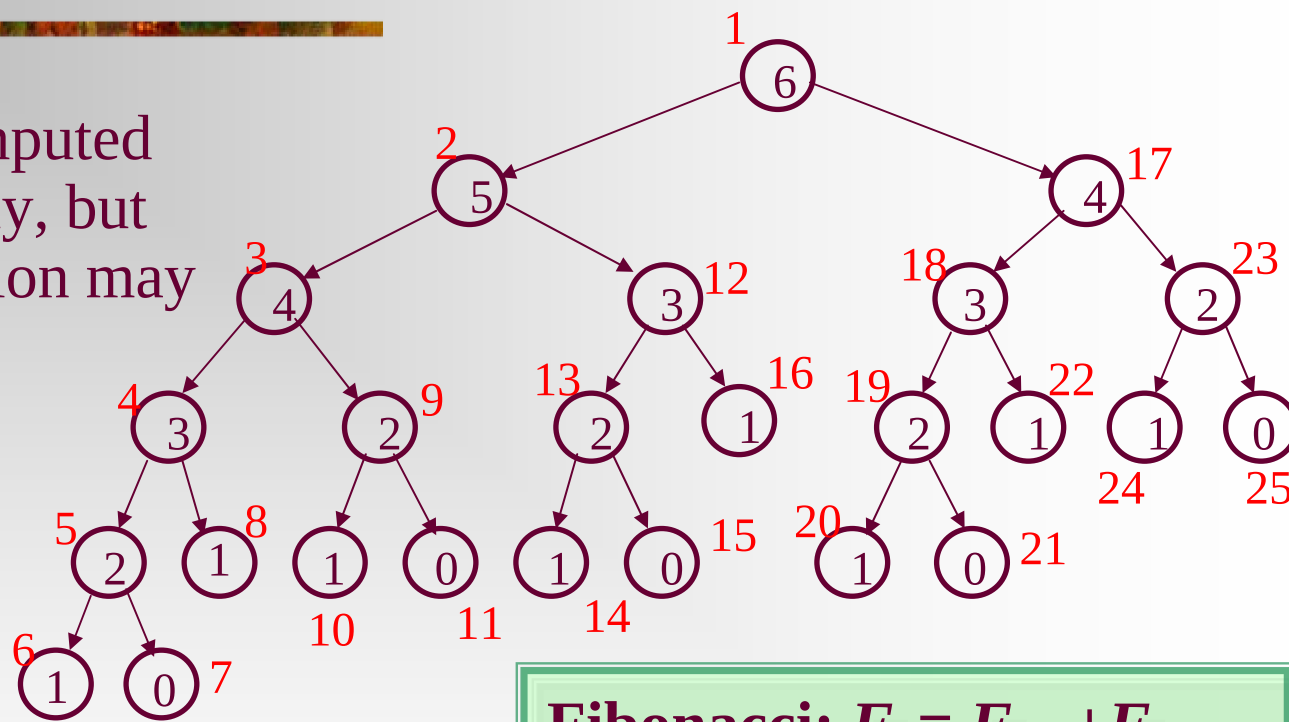
Dynamic Programming

- Recursion and Subproblem Graph
- Basic Idea of Dynamic Programming
- Least Cost of Matrix Multiplication
- Extracting Optimal Multiplication Order

Natural Recursion may be Expensive

The F_n can be computed in linear time easily, but the cost for recursion may be exponential.

The number of activation frames are $2F_{n+1}$-1



Fibonacci: $F_n = F_{n-1} + F_{n-2}$

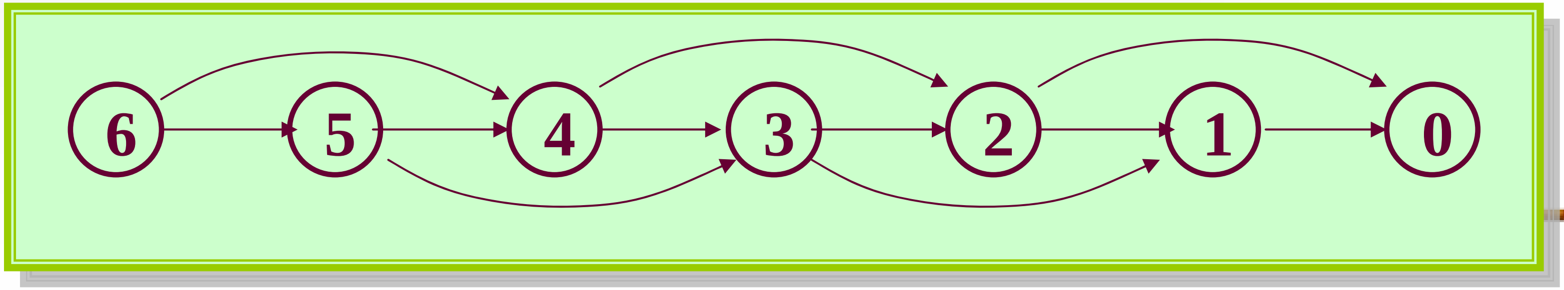
1, 1, 2, 3, 5, 8, 13, 21, 35, ...

For your reference

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

Subproblem Graph

- For any known recursive algorithm A for a specific problem, a subproblem graph is defined as:
  - vertex: the instance of the problem
  - directed edge: the subproblem graph contains a directed edge I→J if and only if when A invoked on I, it makes a recursive call directly on instance J.
- Portion A(*P*) of the subproblem graph for Fibonacci function: here is fib(6)



Properties of Subproblem Graph

- If A always terminates, the subproblem graph for A is a DAG.
- For each path in the tree of activation frames of a particular call of A, A(*P*), there is a corresponding path in the subproblem graph of A connecting vertex *P* and a base-case vertex.
- A top-level recursive computation traverse the entire subproblem graph in some memoryless style.
- The subproblem graph can be viewed as a dependency graph of subtasks to be solved.

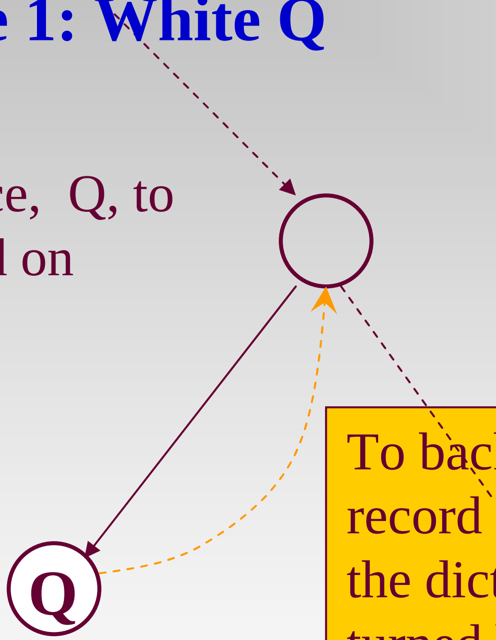
Basic Idea for Dynamic Programming

- Computing each subproblem **only once**
  - Find a reverse topological order for the subproblem graph
    - In most cases, the order can be determined by particular knowledge of the problem.
    - General method based on DFS is available
  - Scheduling the subproblems according to the reverse topological order
  - Record the subproblem solutions for later use

Dynamic Programming Version $\mathcal{DP}$(A) of a Recursive Algorithm A

Case 1: White Q

a instance, Q, to be called on

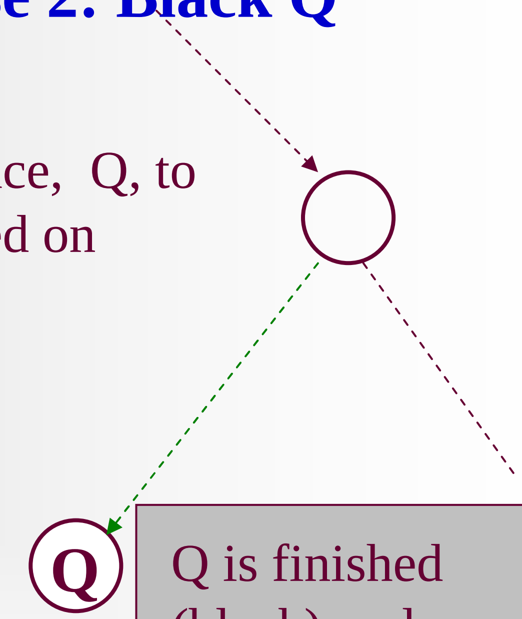


To backtracking, record the result into the dictionary (Q, turned black)

Q is undiscovered (white), go ahead with the recursive call

Case 2: Black Q

a instance, Q, to be called on



Q is finished (black), only “checking” the edge, retrieve the result from the dictionary

Note: for DAG, no gray vertex will be met

DP(fib): an Example

```
fibDPwrap(n)
  Dict soln=create(n);
return fibDP(soln,n)
```

This is the wrapper, which will contain processing existing in original recursive algorithm wrapper.

```
fibDP(soln,k)
    int fib, f1, f2;
    if (k<2) fib=k;
    else
        if (member(soln, k-1)==false)
            f1=fibDP(soln, k-1);
        else
            f1= retrieve(soln, k-1);
        if (member(soln, k-2)==false)
            f2=fibDP(soln, k-2);
        else
            f2= retrieve(soln, k-2);
        fib=f1+f2;
    store(soln, k, fib);
return fib
```

Matrix Multiplication Order Problem

- The task:

  Find the product: $A_1 \times A_2 \times \ldots \times A_{n-1} \times A_n$

  A_i is 2-dimentional array of different legal size
- The issues:
  - Matrix multiplication is associative
  - Different computing order results in great difference in the number of operations
- The problem:
  - Which is the best computing order

Cost of Matrix Multiplication

An example: $A_1 \times A_2 \times A_3 \times A_4$

$30\times1 \quad 1\times40 \quad 40\times10 \quad 10\times25$

$((A_1\times A_2)\times A_3)\times A_4$: 20700 multiplications

$A_1\times(A_2\times(A_3\times A_4))$: 11750

$(A_1\times A_2)\times(A_3\times A_4)$: 41200

$A_1\times((A_2\times A_3)\times A_4)$: 1400

Let $C = A_{p\times q}\times$ …

$$c_{i,j} = \sum_{k=1}^{q} a_{ik} \ldots$$

There are *q* multiplication

C has $p\times r$ elements as $c_{i,j}$

So, *pqr* multiplications altogether

Looking for a Greedy Solution

- Greedy algorithms are usually simple.
- Strategy 1: "**cheapest multiplication first**"
  - Success: $A_{30\times1}\times((A_{1\times40}\times A_{40\times10})\times A_{10\times25}$
  - Fail: $(A_{4\times1}\times A_{1\times100})\times A_{100\times5}$
- Strategy 2: "**largest dimension first**"
  - Correct for the second example above
  - $A_{1\times10}\times A_{10\times10}\times A_{10\times2}$: two results

Problem and Sub-problem: Intuition

- Matrices: $A_1, A_2, \ldots, A_n$
- Dimension: dim: $d_0, d_1, d_2, \ldots, d_{n-1}, d_n$, for A_i is $d_{i-1} \times d_i$
- Sub-problem: seq: $s_0, s_1, s_2, \ldots, s_{k-1}, s_{len}$, which means the multiplication of k matrices, with the dimensions: $d_{s0} \times d_{s1}, d_{s1} \times d_{s2}, \ldots, d_{s[len]-1} \times d_{s[len]}$.
  - Note: the original problem is: seq=$(0,1,2,\ldots,n)$

Cost of the Optimum Order by Recursion

Recursion on index sequence: (seq): 0, 1, 2, …, *n* (len=n)
with the kth matrix is A_k ($k \neq 0$) of the size $d_{k-1} \times d_k$,
and the kth(k<n) multiplication is $A_k \times A_{k+1}$.

```
mmTry1(dim, len, seq)
    if (len<3) bestCost=0
    else
        bestCost=∞;
        for (i=1; i≤len-1; i++)
            c=cost of multiplication at position seq[i];
            newSeq=seq with ith element deleted;
            b=mmTry1(Dim, len-1, newSeq);
            bestCost=min(bestCost, b+c);
    return bestCost
```

$T(n)=(n-1)T(n-1)+n$, in $\Theta((n-1)!)$

Constructing the Subproblem Graph

- The key issue is: how can a subproblem be denoted using a **concise identifier**?
- For mmTry1, the difficult originates from the varied intervals in each newSeq.
- If we look at the **last** (contrast to the first) multiplication, the **two** (not one) resulted subproblems are both contiguous subsequences, which can be uniquely determined by the pair:

  <head-index, tail-index>

Best Order by Recursion: Improved

Only one matrix

```
mmTry2(dim, low, high)
    if (high-low= =1) bestCost=0
    else
        bestCost=∞;
        for (k=low+1; k≤high-1; k++)
            a=mmTry2(dim, low, k);
            b=mmTry2(dim, k, high);
            c=cost of multiplication at position k;
            bestCost=min(bestCost, a+b+c);
    return bestCost
```

with dimensions: dim[low], dim[k], and dim[high]

Still in $\Omega(2^n)$!

Best Order by Dynamic Programming

- DFS can traverse the subproblem graph in time $O(n^3)$
  - At most $n^2/2$ vertices, as <i,j>, $0 \le i < j \le n$.
  - At most $2n$ edges leaving a vertex

```
mmTry2DP(dim, low, high, cost)
    ……
    for (k=low+1; k≤high-1; k++)
        if (member(low,k)==false) a=mmTry2(dim, low, k);
            else a=retrieve(cost, low, k);
        if (member(k,high)==false) b=mmTry2(dim, k, high);
            else b=retrieve(cost, k, high);
        ……
    store(cost, low, high, bestCost);
    return bestCost
```

Corresponding to the recursive procedure of DFS

Simplification Using Ordering Feature

- For any subproblems: (low1, high1) depending on (low2, high2) if and only if low2≤low1, and high2≤high1
- Computing subproblems according the dependency order

- matrixOrder(*n*, cost, last)
- **for** (low=*n*-1; low≥1; low--)
- **for** (high=low+1; high≤*n*; high++)

Compute solution of subproblem (low, high) and store it in cost[low][high] and last[low][high]

- **return** cost[0][*n*]

Matrix Multiplication Order: Algorithm

- **Input**: array **dim** $=(d_0, d_1, \ldots, d_n)$, the dimension of the matrices.
- **Output**: array **multOrder**, of which the *i*th entry is the index of the *i*th multiplication in an optimum sequence.

```
float matrixOrder(int[] dim, int n, int[] multOrder)
  <initialization of last,cost,bestcost,bestlast…>
   for (low=n-1; low≥1; low--)
     for (high=low+1; high≤n; high++)
       if (high-low==1) <base case>
       else bestcost=∞;
       for (k=low+1; k≤high-1; k++)
         a=cost[low][k];
         b=cost[k][high]
         c=multCost(dim[low], dim[k], dim[high]);
         if (a+b+c<bestCost)
           bestCost=a+b+c; bestLast=k;
       cost[low][high]=bestCost;
       last[low][high]=bestLast;
  extrctOrderWrap(n, last, multOrder)
  return cost[0][n]
```

Using the stored results

An Example

- Input: d_0=30, d_1=1, d_2=40, d_3=10, d_4=25

***cost* as finished**

$$\begin{bmatrix} - & 0 & 1200 & 700 & \boxed{1400} \\ - & - & 0 & 400 & 650 \\ - & - & - & 0 & 10000 \\ - & - & - & - & 0 \\ - & - & - & - & - \end{bmatrix}$$

First entry filled

Last entry filled

Note: *cost*[i][j] is the least cost of $A_{i+1} \times A_{i+2} \times \ldots A_j$.

For each selected *k*, retrieving:

- least cost of $A_{i+1} \times \ldots \times A_k$.
- least cost of $A_{k+1} \times \ldots \times A_j$.

and computing:

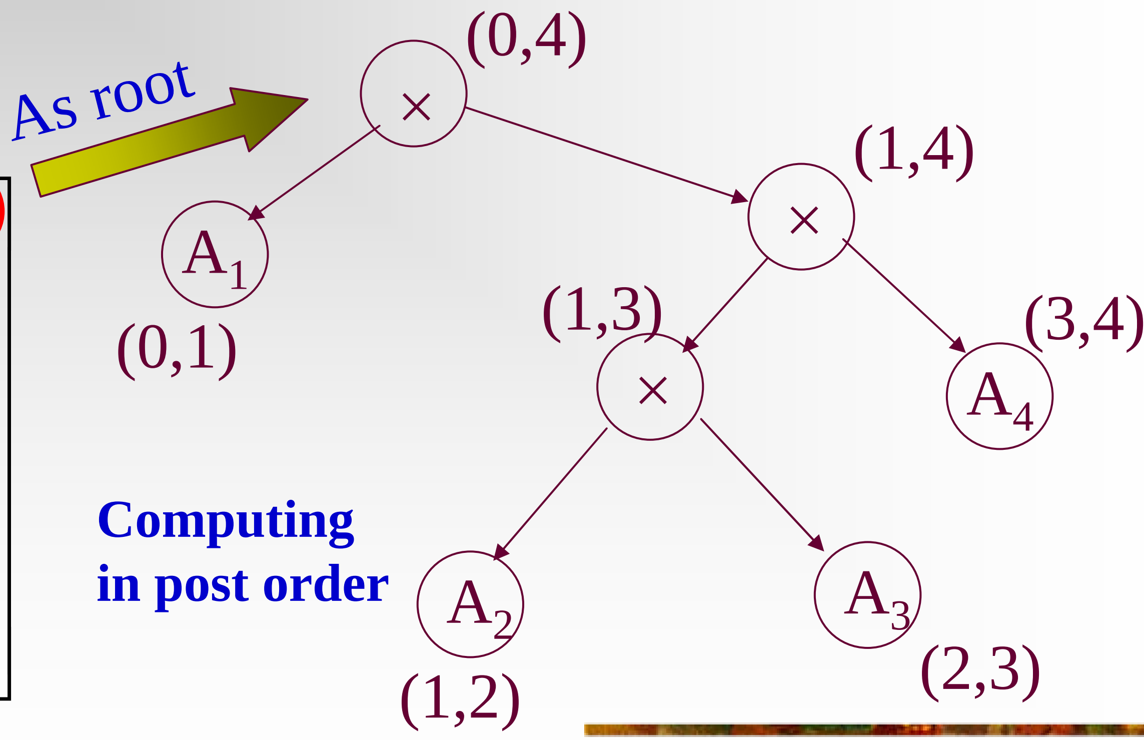
- cost of the last multiplication

Array *last* and the Arithmetic-Expression Tree

- Example input: $d_0=30$, $d_1=1$, $d_2=40$, $d_3=10$, $d_4=25$

***last* as finished**

$$\begin{bmatrix} - & -1 & 1 & 1 & 1 \\ - & - & -1 & 2 & 3 \\ - & - & - & -1 & 3 \\ - & - & - & - & -1 \\ - & - & - & - & - \end{bmatrix}$$



Extracting the Optimal Order

- The core procedure is extractOrder, which fills the multiOrder array for subproblem (low,high), using informations in *last* array.

```
extractOrder(low, high, last, multOrder)
    int k;
    if (high-low>1)
        k=last[low][high];
        extractOrder(low, k, last, multOrder);
        extractOrder(k, high, last, multOrder);
        multOrder[multOrderNext]=k;
        multOrderNext++;
```

Just a post-order traversal

initialized in the wrapper

Calling Map

Output, passed to extractOrder

```
float matrixOrder (int [ ] dim, int n, int [ ] multOrder )
    int [ ] last; float [ ] cost; int low, high, ......
    for (low=n-1; low≥1; low--)
        for (high=low+1; high≤n; high++)
            ......
            for (k=low+1; k≤high-1; k++)
                <Computing all possible multCost by calling multCost>
        <Filling the entries in cost and last (one entry for each)>
    extractOrderWrap(n, last, multOrder)
    return cost[0][n];
```

```
extractOrder(low, high, last, multOrder)
<Whenever high>low, call recursively on
(low,k) and (k,high) where k=last[low][high]>
```

Analysis of matrixOrder

- Main body: 3 layer of loops
  - Time: the innermost processing costs constant, which is executed $\Theta(n^3)$ times.
  - Space: extra space for *cost* and *last*, both in $\Theta(n^2)$
- Order extracting
  - There are $2n$-1 nodes in the arithmetic-expression tree. For each node, extractOrder is called once. Since non-recursive cost for extractOrder is constant, so, the complexity of extractOrder is in $\Theta(n)$

Home Assignment

- 10.1
- 10.4
- 10.6
- 10.7