COT 6405 Introduction to Theory of Algorithms

Final exam review

About the final exam

- The final will cover everything we have learned so far.
- Closed books, closed computers, and closed notes.

Distribution of Questions

- The content covered by midterms I and II takes 50%
- The content we studied after midterm II takes 50%

Quick summary of previous content

- How to solve the recurrences
  - Substitution method
  - Tree method
  - Master theorem
- Comparison based sorting algorithms
  - Merge sort, quick sort, and Heap sort
- Linear time sorting algorithms
  - Counting sort, Bucket sort, and Radix sort

Quick summary (cont'd)

- Basic heap operations:
  - Build-Max-Heap, Max-Heapify
- Order statistics
  - How to find the k-th largest element : BigFive algorithm

Hash Table

- We use a table of size proportional to |K|: hash tables
  - Define hash functions that map keys to slots of the hash table.
  - However, we lose the direct-addressing ability.

Hash function

- Hash function h: Mapping from Universe U to the slots of a hash table T[0..m–1].
- h : U → {0,1,…, m–1}
  - With arrays, key k maps to slot A[k].
  - With hash tables, key k maps or “hashes” to slot T[h(k)]
    - h(k) is the hash value of key k
- Example of Hash Function
  - h(k) = return (k mod m)
  - where k is the key, and m is the size of the table

Issues with Hashing?

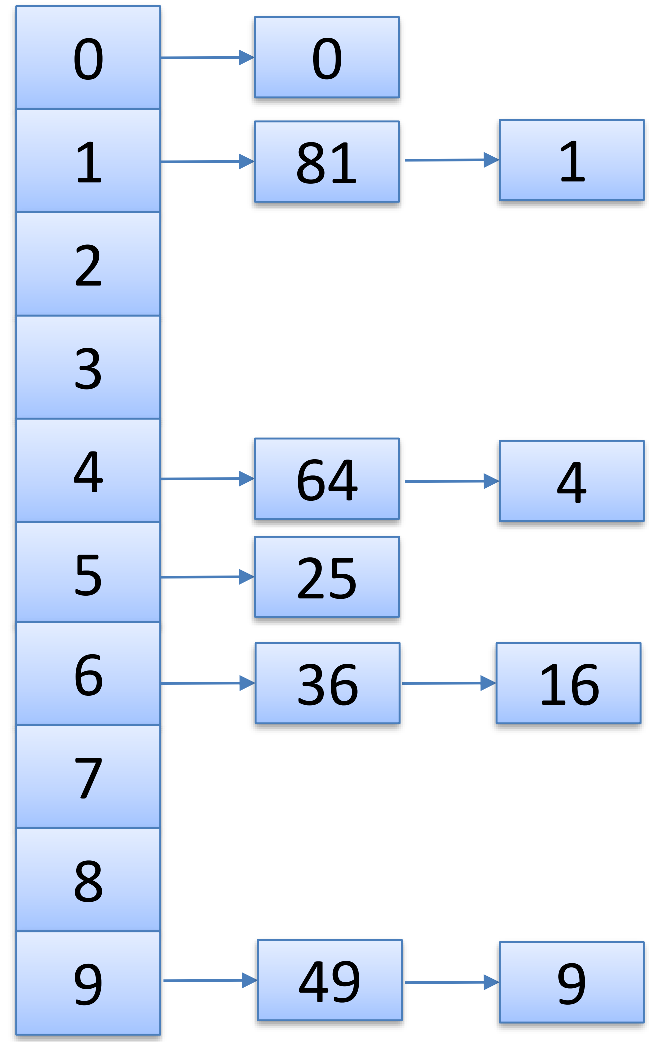
- Multiple keys can hash to the same slot: collisions
  - Design hash functions such that collisions are minimized
  - But avoiding collisions is sometimes impossible
    - Must have collision-resolution techniques

Collision Resolution Scheme 1: Chaining

- The hash table is an array of linked lists
- Insert Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

Notes:

- As before, elements would be associated with the keys
- We're using the hash function h(k) = k mod m
  - m=10



Chaining Algorithms

Chained-Hash-Insert(T, x)

insert x at the head of list T[h(x.key)]

Chained-Hash-Search(T, k)

search for an element with key k

in list T[h(k)]

Chained-Hash-Delete(T, x)

delete x from the list T[h(x.key)]

Analysis of hashing with chaining

- m = hash table size
- n = number of elements in hash table
- load factor α= n/m : average number of keys per slot
- Assume each key is equally likely to be hashed into any slot: using simple uniform hashing (SUH)
- What is the worst-case search time?
  - Unsuccessful Search → we find none
  - Successful Search → we find one

Collision Resolution Scheme 2: Open addressing

- No list and no element stored outside the table
  - If a collision occurs, try alternate cells until empty cell is found.
  - Pro: No pointers!
- Advantage: avoid pointers, potentially yield fewer collisions and faster retrieval
  - Extra memory freed from storing pointers → more hash slots → less collisions!

Common Probing Sequence

- Assume uniform hashing
- Collision Resolution Strategies for open address
  - Linear Probing
  - Quadratic Probing
  - Double Hashing
- We try cells $h(k,0), h(k,1), h(k,2), \ldots, h(k, m-1)$
  - where $h(k,i) = (h'(k) + f(i)) \bmod m$, with $f(0) = 0$
  - Function f is the collision resolution strategy
  - Function h' is the original hash function.

Collision Resolution Comparison:

Expected Number of Probes in Searches

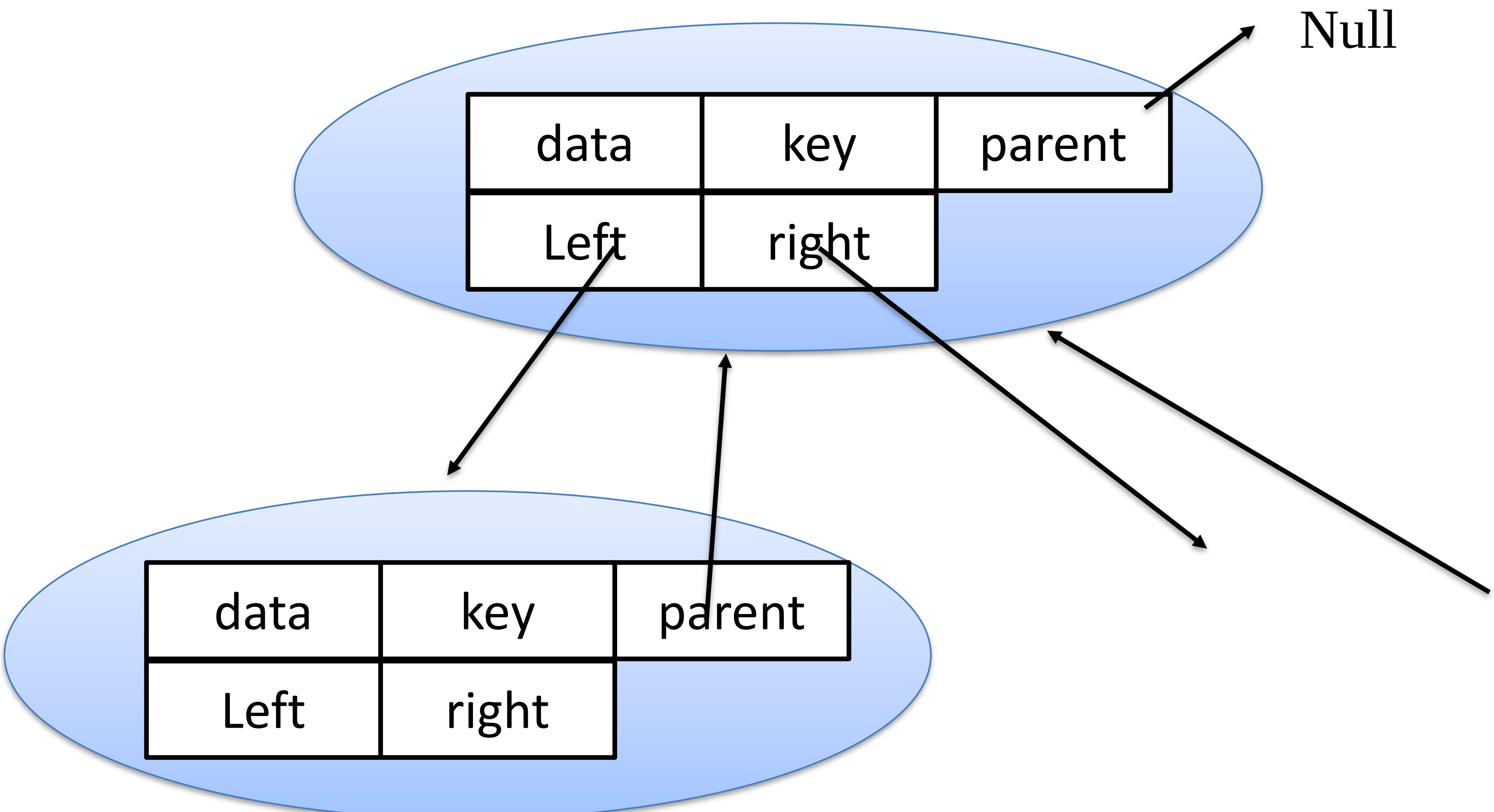
load factor $\alpha = n/m$

	Unsuccessful Search	Successful Search
Chaining	$1+\alpha$ (1 + average number of elements in chain)	$1 + \alpha/2 - \alpha/(2n)$ (1 + average number before element in chain)
Open Addressing (assuming uniform hashing)	$1 / (1 - \alpha)$	$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

Binary Search Trees

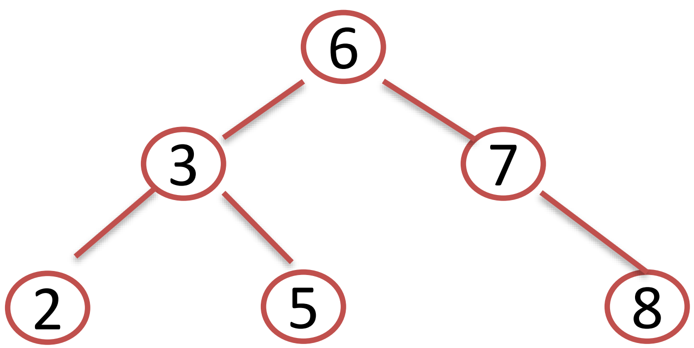
- Binary Search Trees (BSTs) are an important data structure for dynamic sets
- In addition to satellite data, nodes have:
  - **key**: an identifying field inducing a total ordering
  - **left**: pointer to a left child (may be NULL)
  - **right**: pointer to a right child (may be NULL)
  - **p**: pointer to a parent node (NULL for root)

Node implementation

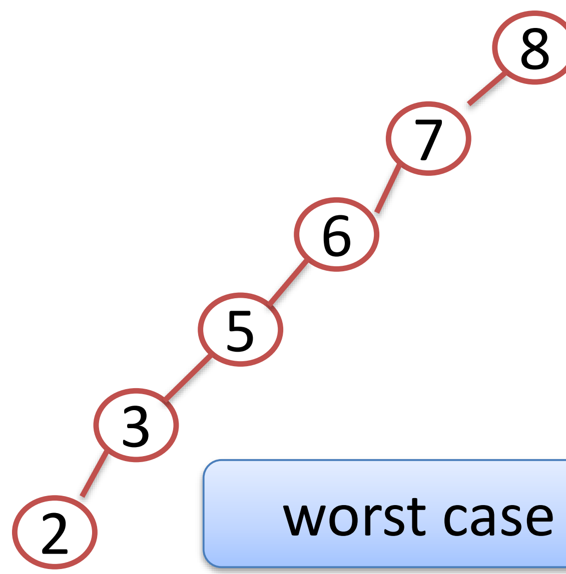


Binary Search Trees

- BST property: Let x be a node in a binary search tree. If y is a node in the left subtree of x, then y.key < x.key. If y is a node in the right subtree of x, then y.key > x.key. Different BSTs can be constructed to represent the same set of data



Average case O(lgn)



worst case O(n)

Walk on BST

- A: prints elements in sorted (increasing) order

```
InOrderTreeWalk(x)
    InOrderTreeWalk(x.left);
    print(x);
    InOrderTreeWalk(x.right);
```

- This is called an inorder tree walk
  - *Preorder tree walk*: print root, then left, then right
  - *Postorder tree walk*: print left, then right, then root

Operations on BSTs: Search

- Given a key and a pointer to a node, returns an element with that key or NULL:

```
TreeSearch(x, k)
     if (x = NULL  or  k = x.key)
         return x;
     if (k < x.key)
         return TreeSearch(x.left, k);
     else
         return TreeSearch(x.right, k);
```

Operations on BSTs: Search

- Here’s another function that does the same

Iterative-Tree-Search **(x, k)**

```
while (x != NULL and k != x.key)
    if (k < x.key)
        x = x.left;
    else
        x = x.right;
return x;
```

BST Operations: Minimum

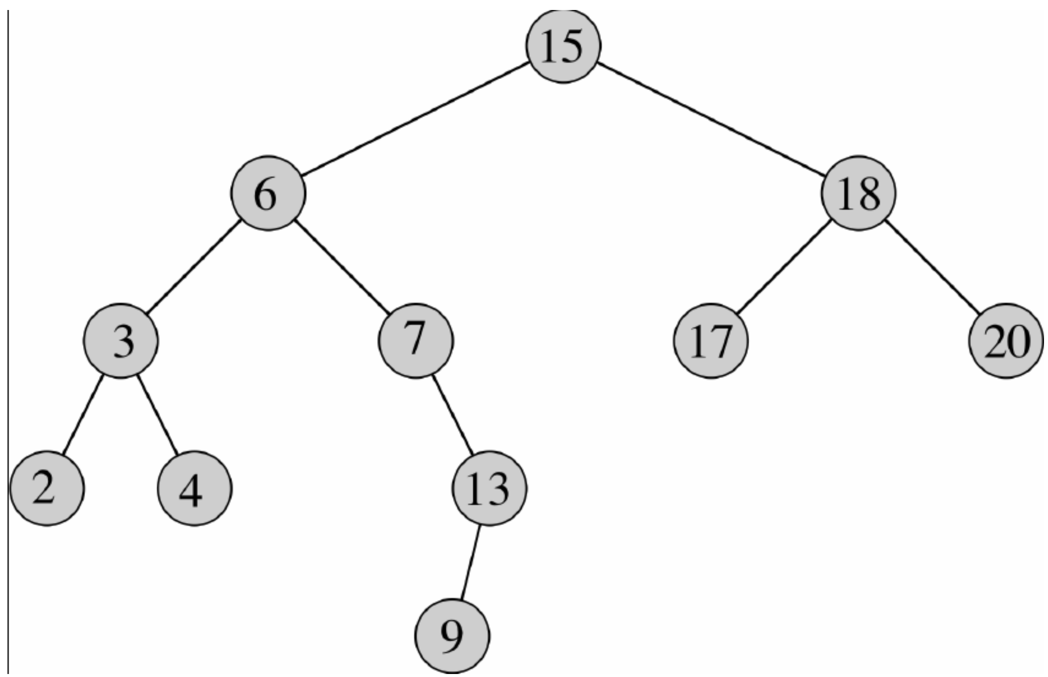
- How can we implement a Minimum() query?

```
TREE_MINIMUM(x)
        while x.lef <> NIL
              x = x.left
         Return x
```

- What is the running time?
- Minimum→ Find the leftmost node in tree
- Maximum → find the rightmost node in the tree

BST Operations: Successor

- Successor of x: the smallest key greater than *key*[*x*].
- What is the successor of node 3? Node 15? Node 13?
- What are the general rules for finding the successor of node x? (hint: two cases)



BST Operations: Successor

- Two cases:
  - x has a right subtree: its successor is minimum node in right subtree
  - x has no right subtree: x must be on the left sub tree of the successor such that x <= successor. So the successor is the first ancestor of x whose left child is an ancestor of x (or x)
    - Intuition: As long as you move to the left up the tree, you’re visiting smaller nodes.

BST Operations: predecessor

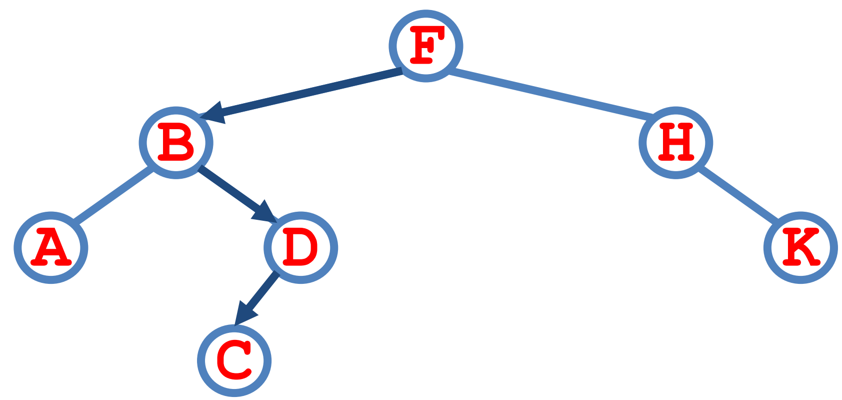
- Two cases:
  - x has a left subtree: its predecessor is maximum node in left subtree
  - x has no left subtree: x must be on the right sub tree of the predecessor such that x >= predecessor. So the predecessor is the first ancestor of x whose right child is an ancestor of x (or x)

Operations of BSTs: Insert

- Adds an element x to the tree
  - → the binary search tree property continues to hold
- The basic algorithm
  - Like the search procedure above
  - Use a "trailing pointer" to keep track of where you came from
    - like inserting into singly linked list

BST Operations: Delete

- Several cases:
  - x has no children:
    - Remove x
    - Set parent's link NULL
  - x has one child:
    - Replace x with its child
    - Set the child's link NULL
  - x has two children:
    - replace x with its successor
    - Perform case 0 or 1 to delete it



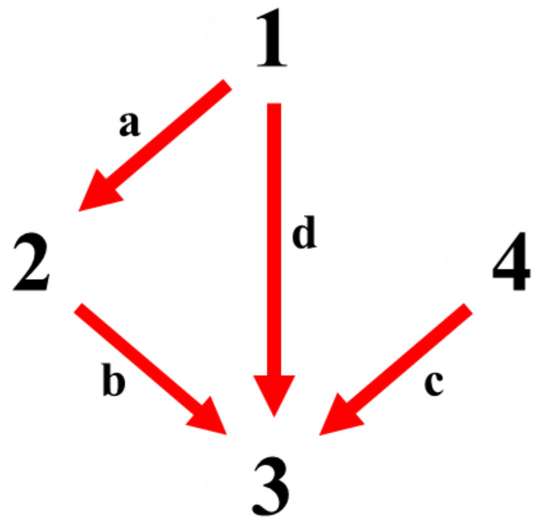
Example: delete K or H or B

Elementary Graph Algorithms

- How to represent a graph?
  - Adjacency lists
  - Adjacency matrix
- How to search a graph?
  - Breadth-first search
  - Depth-first search

Graphs: Adjacency Matrix

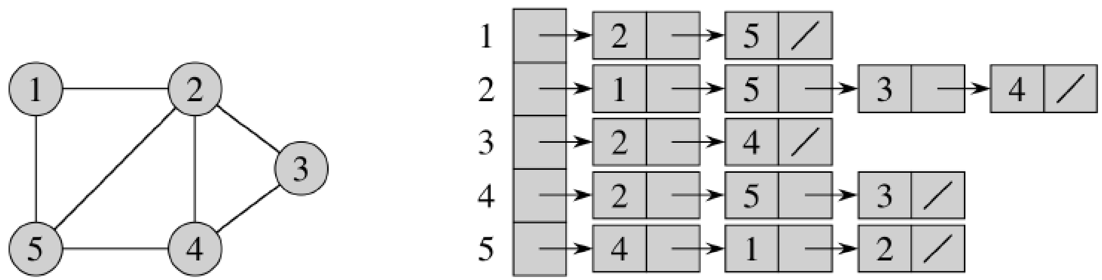
- Example:



A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

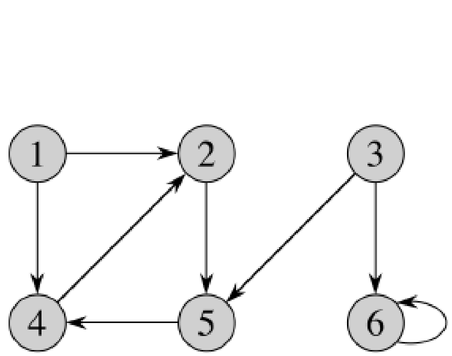
Graphs: Adjacency List

- Undirected



(a) (b)

- Directed Graph



(a) (b)

Graphs: Adjacency List

- How much storage is required?
  - The degree of a vertex *v* = # incident edges
    - Two edges are called incident, if they share a vertex
    - Directed graphs have in-degree, out-degree
  - For directed graphs, # of items in adjacency lists is
    Σ out-degree(*v*) = $|E|$
    takes $\Theta(V + E)$ storage
  - For undirected graphs, # items in adjacency lists is
    Σ degree(v) = 2 $|E|$
    also $\Theta(V + E)$ storage
- So: Adjacency lists take O(V+E) storage

Breadth-First Search (BFS)

- “Explore” a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the breadth of the frontier
- Builds a tree over the graph
  - Pick a source vertex to be the root
  - Find (“discover”) its children, then their children, etc.

Breadth-First Search

```
BFS(G, s) {
    initialize vertices;
    Q = {s};
    while (Q not empty) {
        u = Dequeue(Q);
        for each v ∈ G.adj[u] {
            if (v.color == WHITE)
                v.color = GREY;
                v.d = u.d + 1;
                v.p = u;
                Enqueue(Q, v);
        }
        u.color = BLACK;
    }
}
```

Time analysis

- The total running time of BFS is $O(V + E)$
- Proof:
  - Each vertex is dequeued at most once. Thus, total time devoted to queue operations is $O(V)$.
  - For each vertex, the corresponding adjacency list is scanned at most once. Since the sum of the lengths of all the adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(E)$.
  - Thus, the total running time is O(V+E)

Breadth-First Search: Properties

- BFS calculates the shortest-path distance to the source node
  - Shortest-path distance $\delta(s,v)$ = minimum number of edges from s to v, or ∞ if v not reachable from s
- BFS builds breadth-first tree, in which paths to root represent shortest paths in G
  - Thus, we can use BFS to calculate a shortest path from one vertex to another in O(V+E) time

Depth-First Search

- Depth-first search is another strategy for exploring a graph
  - Explore “deeper” in the graph whenever possible
  - Edges are explored out of the most recently discovered vertex v that still has unexplored edges
    - Timestamp to help us remember who is “new”
  - When all of v’s edges have been explored, backtrack to the vertex from which v was discovered

Depth-First Search: The Code

```
DFS(G)
{
  for each vertex u ∈ G.V
  {
      u.color = WHITE
      u.π  = NIL
  }
  time = 0
  for each vertex u ∈ G.V
  {
    if (u.color == WHITE)
       DFS_Visit(G, u)
  }
}
```

```
DFS_Visit(G, u)
{
   time = time + 1
   u.d = time
   u.color = GREY
   for each v ∈ G.Adj[u]
  {
     if (v.color == WHITE)
        v.π = u
        DFS_Visit(G, v)
  }
   u.color = BLACK
   time = time + 1
   u.f = time
}
```

DFS: running time (cont'd)

- How many times will DFS_Visit() actually be called?
  - The loops on lines 1–3 and lines 5–7 of DFS take time Θ(V), exclusive of the time to execute the calls to DFS-VISIT.
  - DFS-VISIT is called exactly once for each vertex v
  - During an execution of DFS-VISIT*(v)*, the loop on lines 4–7 is executed $|Adj[v]|$ times.
  - $\sum_{v \in V} |Adj[v]| = \Theta(E)$
  - Total running time is $\Theta(V + E)$

DFS: Different Types of edges

- DFS introduces an important distinction among edges in the original graph:
  - Tree edge: encounter new vertex
  - Back edge: from a descendent to an ancestor
  - Forward edge: from an ancestor to a descendent
  - Cross edge: between a tree or subtrees
- Note: tree & back edges are important
  - most algorithms don't distinguish forward & cross

Minimum Spanning Tree

- Problem:
  - given a <u>connected</u>, <u>undirected</u>, <u>weighted</u> graph
    G = (V, E)
  - find a spanning tree using edges that connects all nodes with a minimal total weight w(T)= SUM(w[u,v])
    - w[u,v] is the weight of edge (u,v)
- Objectives: we will learn
  - Generic MST
  - Kruskal's algorithm
  - Prim's algorithm

Growing a minimum spanning tree

- Building up the solution
  - We will build a set *A* of edges
  - Initially, *A* has no edges.
  - As we add edges to *A*, maintain a loop invariant
- Loop invariant: *A* is a subset of some MST
  - Add only edges that maintain the invariant
  - Definition: If *A* is a subset of some MST, an edge *(u, v)* is safe for *A*, if and only if $A \cup \{(u, v)\}$ is also a subset of some MST
  - So we will add only safe edges

Generic MST algorithm

GENERIC-MST(G, w)

$A = \emptyset$

while A is not a spanning tree

    find an edge (u, v) that is safe for A

    $A = A \cup \{(u, v)\}$

return A

How do we find safe edges?

- Let edge set A be a subset of some MST
- $(S, V - S)$ be a cut that respects edge set A
  - No edges in A crosses the cut
- (u, v) be a light edge crossing cut $(S, V - S)$.
- Then, (u, v) is safe for A.

MST: optimal substructure

- MSTs satisfy the optimal substructure property: an optimal tree is composed of optimal subtrees
  - Let T be an MST of G with an edge (*u,v*) in the middle
  - Removing (*u,v*) partitions T into two trees T_1 and T_2
  - Claim: T_1 is an MST of $G_1 = (V_1,E_1)$, and T_2 is an MST of $G_2 = (V_2,E_2)$

Kruskal’s algorithm

- Starts with each vertex being its own component
- Repeatedly merges two components into one by choosing the light edge that connects them
- Scans the set of edges in monotonically increasing order by weight
- Uses a disjoint-set data structure to determine whether an edge connects vertices in different components.

Disjoint Sets Data Structure

- A disjoint-set is a collection $C = \{S_1, S_2, \dots, S_k\}$ of distinct dynamic sets
- Each set is identified by a member of the set, called representative.
- Disjoint set operations:
  - MAKE-SET(x): create a new set with only x
    - assume x is not already in some other set.
  - UNION(x,y): combine the two sets containing x and y into one new set.
    - A new representative is selected.
  - FIND-SET(x): return the representative of the set containing x.

Kruskal's Algorithm

```
Kruskal(G, w)
{
    A = ∅;
    for each v ∈ G.V
        Make-Set(v);
    sort G.E by non-decreasing order by weight w
    for each (u,v) ∈ G.E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            A = A ∪ {{u,v}};
            Union(u, v);
}
```

Kruskal’s Algorithm: Running Time

- Initialize A: O(1)
- First for loop: |V| MAKE-SETs
- Sort E: O(E lg E)
- Second for loop: O(E) FIND-SETs and UNIONs
- **O(V) +O (E α(V)) + O(E lg E)**
  - Since G is connected, $|E| \geq |V|-1 \Rightarrow$ O(E α(V)) + O(E lg E)
  - $\alpha(|V|)$ = O(lg V) = O(lg E)
  - Therefore, the total time is O(E lg E)
  - $|E| \leq |V|^2 \Rightarrow \lg |E|$ = O(2 lg V) = O(lg V)
  - Therefore, O(E lg V) time

Prim’s algorithm

- Build a tree *A* (A is always a tree)
  - Starts from an arbitrary “root” *r*.
  - At each step, find a light edge crossing the cut $(V_A, V - V_A)$, where V_A = vertices that *A* is incident on.
  - Add this light edge to *A*.
- GREEDY CHOICE:

  add min weight to A
- Use a priority queue *Q* to quickly find the light edge

Prim’s Algorithm

```
MST-Prim(G, w, r)
    for each u ∈ G.V
        u.key = ∞
        u.π = NIL
    r.key = 0
    Q = G.V
    while (Q not empty)
        u = ExtractMin(Q)
        for each v ∈ G.Adj[u]
            if (v ∈ Q and w(u,v) < v.key )
                v.π = u
                v.key = w(u,v)
```

Prim's Algorithm: running time

- We can use the BUILD-MIN-HEAP procedure to perform the initialization in lines 1–5 in $O(V)$ time
- EXTRACT-MIN operation is called $|V|$ times, and each call takes $O(\lg V)$ time, the total time for all calls to EXTRACT-MIN is $O(V \lg V)$

Running time (cont’d)

- The for loop in lines 8–11 is executed O(E) times altogether, since the sum of the lengths of all adjacency lists is 2 |E|.
  - Lines 9 -10 take constant time
  - line 11 involves an implicit DECREASE-KEY operation on the min-heap, which takes $O(\lg V)$ time
- Thus, the total time for Prim's algorithm is $O(V) + O(V \lg V) + O(E \lg V) = O(E \lg V)$
  - The same as Kruskal's algorithm

Single source shortest path problem

- Problem: given a weighted directed graph G, find the minimum-weight path from a given source vertex s to another vertex v
  - "Shortest-path" -> Weight of the path is minimum
  - Weight of a path is the sum of the weight of edges

Shortest path properties

- Optimal substructure property: any subpath of a shortest path is a shortest path
- In graphs with negative weight cycles, some shortest paths will not exist:
- Negative weight edges are ok for some cases
- Shortest paths cannot contain cycles

Initialization

- All the shortest-paths algorithms start with INIT-SINGLE-SOURCE

INIT-SINGLE-SOURCE*(G, s)*

for each vertex $v \in G.V$

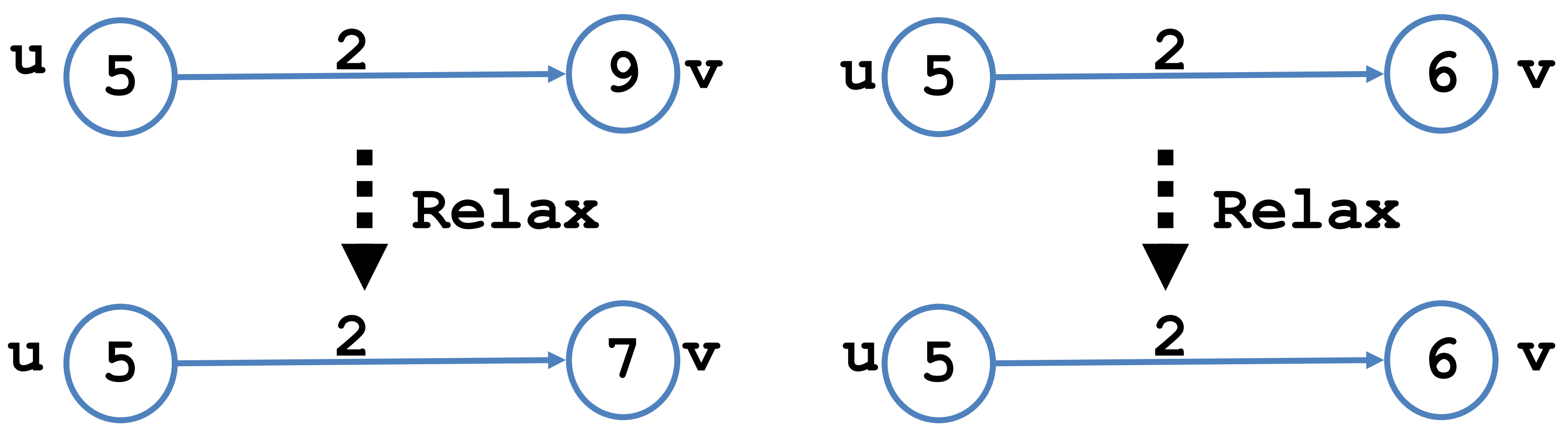
v.d = ∞

$v.\pi =$ NIL

$s.d =$ 0

Relaxation: reach v by u

```
Relax(u, v, w) {
   if (v.d > u.d + w(u,v))
                    v.d = u.d + w(u,v)
         v.π = u
}
```



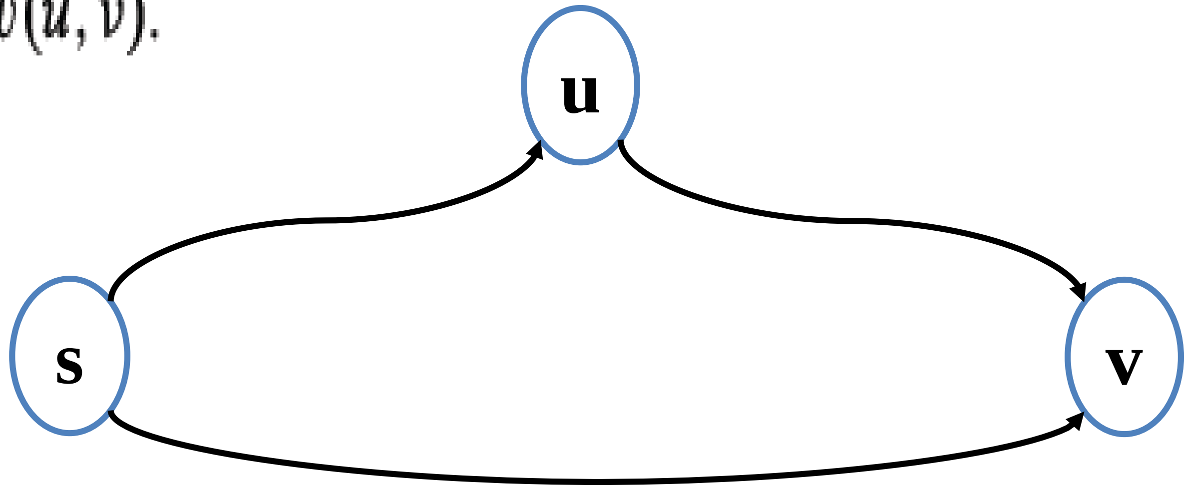
decrease by 2   unchanged

Properties of shortest paths

- Triangle inequality

For all $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Proof Weight of shortest path $s \leadsto v$ is $\leq$ weight of any path $s \leadsto v$. Path $s \leadsto u \to v$ is a path $s \leadsto v$, and if we use a shortest path $s \leadsto u$, its weight is $\delta(s, u) + w(u, v)$. ■



Upper-bound property

- Always have $v.d \geq \delta(s,v)$
  - Once $v.d = \delta(s,v)$, it never changes
- Proof: Initially, it is true: $v.d = \infty$
- Supposed there is vertex such that $v.d < \delta(s,v)$
- Without loss of generality, v is the first vertex for this happens
- Let u be the vertex that causes v.d to change
- Then $v.d = u.d + w(u,v)$
- So, $v.d < \delta(s,v) \leq \delta(s,u) + w(u,v) < u.d + w(u,v)$
- Then $v.d < u.d + w(u,v)$
- Contradict to $v.d = u.d + w(u,v)$

No-path property

- If $\delta(s,v) = \infty$, then $v.d = \infty$ always
- Proof: $v.d \geq \delta(s,v) = \infty \rightarrow v.d = \infty$

Convergence property

If $s \leadsto u \rightarrow v$ is a shortest path, $u.\mathbf{d} = \delta(s,u)$, and we call RELAX$(u,v,w)$, then $v.\mathbf{d} = \delta(s,v)$ afterward.

Proof After relaxation:

$$\begin{aligned} v.\mathbf{d} &\le u.\mathbf{d} + w(u,v) && \text{(RELAX code)} \\ &= \delta(s,u) + w(u,v) \\ &= \delta(s,v) && \text{(lemma—optimal substructure)} \end{aligned}$$

Since $v.\mathbf{d} \ge \delta(s,v)$, must have $v.\mathbf{d} = \delta(s,v)$. ∎

When the "if" condition is true, v.d = u.d + w(u, v)
When the "if" condition is false, v.d ≤ u.d + w(u, v)

Path relaxation property

Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from $s = v_0$ to v_k. If we relax, in order, $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, even intermixed with other relaxations, then $v_k.\mathrm{d} = \delta(s, v_k)$.

Proof Induction to show that $v_i.\mathrm{d} = \delta(s, v_i)$ after (v_{i-1}, v_i) is relaxed.

Basis: $i = 0$. Initially, $v_0.\mathrm{d} = 0 = \delta(s, v_0) = \delta(s, s)$.

Inductive step: Assume $v_{i-1}.\mathrm{d} = \delta(s, v_{i-1})$. Relax (v_{i-1}, v_i). By convergence property, $v_i.\mathrm{d} = \delta(s, v_i)$ afterward and $v_i.\mathrm{d}$ never changes. ■

Bellman-Ford algorithm

//Allows negative-weight edges

```
BellmanFord(G, w, s)
    INIT-SINGLE-SOURCE(G, s)
    for i=1 to |G.V|-1
        for each edge (u,v) ∈ G.E
            Relax(u, v, w);
    for each edge (u,v) ∈ G.E
        if (v.d > u.d + w(u,v))
            return "no solution";
```

Relaxation:
Make |V|-1 passes,
relaxing each edge

Test for solution
Under what condition
do we get a solution?

```
Relax(u,v,w): if (v.d > u.d + w(u,v))
                  v.d = u.d + w(u,v)
```

Running time

- Initialization: $\Theta(V)$
- Line 2-4 : $\Theta(E)$ * $|V|-1$ passes
- Line 5-7 : $O(E)$
- $O(VE)$

Dijkstra's Algorithm

- Assumes **no negative-weight edges**.
- Maintains a vertex set S whose shortest path from s has been determined.
- Repeatedly selects u in V–S with minimum Shortest Path estimate (greedy choice).
- Store V–S in priority queue Q.

```
DIJKSTRA(G, w, s)
Initialize-SINGLE-SOURCE(G, s);
S = ∅;
Q = G.V;
while Q ≠ ∅
    u = Extract-Min(Q);
    S = S ∪ {u};
    for each v ∈ G.Adj[u]
        Relax(u, v, w)
```

Dijkstra’s Running Time

- Extract-Min executed |V| time
- Decrease-Key executed |E| time
- Time = $|V|\ T_{\text{Extract-Min}} + |E|\ T_{\text{Decrease-Key}}$
- Time = O(VlgV) + O (ElgV) = O(ElgV)

Dynamic Programming (DP)

- Like divide-and-conquer, solve problem by combining the solutions to sub-problems.
- Divide-and-conquer vs. DP:
  - divide-and-conquer: Independent sub-problems
    - solve sub-problems independently and recursively, (→ so same sub-problems solved repeatedly)
  - DP: Sub-problems are dependent
    - sub-problems share sub-sub-problems
    - every sub-problem is solved just once
    - solutions to sub-problems are stored in a table and used for solving higher level sub-problems.

Overview of DP

- Not a specific algorithm, but a technique (like divide-and-conquer).
- Doesn’t really refer to computer programming
- Application domain of DP
  - Optimization problem: find a solution with the optimal (maximum or minimum) value

Matrix-chain multiplication problem

- Given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices
  - where for $i = 1, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$
  - fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.
- What is the minimum number of multiplications required to compute $A_1 \cdot A_2 \cdot \ldots \cdot A_n$?
- What order of matrix multiplications achieves this minimum? This is our goal !

Step 1: Find the structure of an optimal parenthesization

- Finding the optimal substructure and using it to construct an optimal solution to the problem based on optimal solutions to subproblems.

Both must be **Optimal** for sub-chain

$$((A_1 A_2 \cdots A_k)(A_{k+1} A_{k+2} \cdots A_n))$$

Then combine them for the original problem

- The key is to find k ; then, we can build the global optimal solution

Step 2: A recursive solution to define the cost of an optimal solution

- Define $m[i, j]$ = the minimum number of multiplications needed to compute the matrix $A_{i..j} = A_i A_{i+1} \cdots A_j$
- Goal: to compute $m[1, n]$
- Basis: m(i, i) = 0
  - Single matrix, no computation
- Recursion: How to define m[i, j] recursively?
  - $((A_i A_2 \cdots A_k)(A_{k+1} A_{k+2} \cdots A_j))$

Step2: Defining $m[i,j]$ Recursively

- Consider all possible ways to split A_i through A_j into two pieces: $(A_i \cdot \ldots \cdot A_k) \cdot (A_{k+1} \cdot \ldots \cdot A_j)$
- Compare the costs of all these splits:
  - best case cost for computing the product of the two pieces
  - plus the cost of multiplying the two products
  - Take the best one
  - $m[i,j] = \min_k\{ m[i,k] + m[k+1,j] + p_{i-1}p_kp_j \}$

Identify Order for Solving Subproblems

- Solve the subproblems (i.e., fill in the table entries) along the diagonal

	1	2	3	4	5
1	0				
2	n/a	0			
3	n/a	n/a	0		
4	n/a	n/a	n/a	0	
5	n/a	n/a	n/a	n/a	0

An example

	1	2	3	4
1	0	1200		
2	n/a	0	400	
3	n/a	n/a	0	10000
4	n/a	n/a	n/a	0

A1 is 30x1
A2 is 1x40
A3 is 40x10
A4 is 10x25
p0 = 30, p1 = 1
p2 = 40, p3 = 10
p4 = 25

m[1,2] = A1A2 : 30X1X40 = 1200,
m[2,3] = A2A3 : 1X40X10 = 400,
m[3,4] = A3A4: 40X10X25 = 10000

An example (cont'd)

	1	2	3	4
1	0	1200	700	
2	n/a	0	400	
3	n/a	n/a	0	10000
4	n/a	n/a	n/a	0

A1 is 30x1
A2 is 1x40
A3 is 40x10
A4 is 10x25
p0 = 30, p1 = 1
p2 = 40, p3 = 10
p4 = 25

$$m[i,j] = \min_k \{ m[i,k] + m[k+1,j] + p_{i-1}p_kp_j \}$$

m[1,3]: $i = 1$, $j = 3$, $k = 1, 2$
= min{ m[1,1]+m[2,3]+p0*p1*p3, m[1, 2]+m[3,3]+p0*p2*p3}
= min{0 + 400 + 30*1*10, 1200+0+30*40*10} = 700

An example (cont'd)

	1	2	3	4
1	0	1200	700	
2	n/a	0	400	650
3	n/a	n/a	0	10000
4	n/a	n/a	n/a	0

A1 is 30x1
A2 is 1x40
A3 is 40x10
A4 is 10x25
p0 = 30, p1 = 1
p2 = 40, p3 = 10
p4 = 25

$$m[i,j] = \min_k\{\ m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\ \}$$

m[2,4]: i = 2, j = 4, k = 2, 3
= min{ m[2,2]+m[3,4]+p1*p2*p4, m[2, 3]+m[4,4]+p1*p3*p4}
= min{0 + 10000 + 1*40*25, 400+0+1*10*25} = 650

An example (cont'd)

	1	2	3	4
1	0	1200	700	1400
2	n/a	0	400	650
3	n/a	n/a	0	10000
4	n/a	n/a	n/a	0

A1 is 30x1
A2 is 1x40
A3 is 40x10
A4 is 10x25
p0 = 30, p1 = 1
p2 = 40, p3 = 10
p4 = 25

$$m[i,j] = \min_k\{\ m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\ \}$$

m[1,4]: $i = 1$, $j = 4$, $k = 1, 2, 3$

= min{ m[1,1]+m[2,4]+p0*p1*p4, m[1,2]+m[3,4]+p0*p2*p4,
m[1,3]+m[4,4]+p0*p3*p4}

= min{0+650+30*1*25, 1200+10000+30*40*25, 700+0+30*10*25}

= 1400

Step 3: Keeping Track of the Order

- We know the cost of the cheapest order, but what is that cheapest order?
  - Use another array s[]
  - update it when computing the minimum cost in the inner loop
- After m[] and s[] are done, we call a recursive algorithm on s[] to print out the actual order

An example

	1	2	3	4
1	0	1		
2	n/a	0	2	
3	n/a	n/a	0	3
4	n/a	n/a	n/a	0

A1 is 30x1
A2 is 1x40
A3 is 40x10
A4 is 10x25
p0 = 30, p1 = 1
p2 = 40, p3 = 10
p4 = 25

m[1,2] = A1A2 : 30X1X40 = 1200, s[1,2] = 1
m[2,3] = A2A3 : 1X40X10 = 400, s[2,3] = 2
m[3,4] = A3A4: 40X10X25 = 10000, s[3,4] = 3

An example (cont'd)

	1	2	3	4
1	0	1	1	
2	n/a	0	2	
3	n/a	n/a	0	3
4	n/a	n/a	n/a	0

A1 is 30x1
A2 is 1x40
A3 is 40x10
A4 is 10x25
p0 = 30, p1 = 1
p2 = 40, p3 = 10
p4 = 25

m[1,3]: $i = 1$, $j = 3$, $k = 1, 2$
= min{ m[1,1]+m[2,3]+p0*p1*p3, m[1, 2]+m[3,3]+p0*p2*p3}
= min{0 + 400 + 30*1*10, 1200+0+30*40*10} = 700
m[1,3] is the minimum value when k = 1, so s[1,3] = 1

An example (cont'd)

	1	2	3	4
1	0	1	1	
2	n/a	0	2	3
3	n/a	n/a	0	3
4	n/a	n/a	n/a	0

A1 is 30x1
A2 is 1x40
A3 is 40x10
A4 is 10x25
p0 = 30, p1 = 1
p2 = 40, p3 = 10
p4 = 25

m[2,4]: i = 2, j = 4, k = 2, 3
= min{ m[2,2]+m[3,4]+p1*p2*p4, m[2, 3]+m[4,4]+p1*p3*p4}
= min{0 + 10000 + 1*40*25, 400+0+1*10*25} = 650
m[2,4] is the minimum value when k = 3, so s[2,4] = 3

An example (cont'd)

	1	2	3	4
1	0	1	1	1
2	n/a	0	2	3
3	n/a	n/a	0	3
4	n/a	n/a	n/a	0

A1 is 30x1
A2 is 1x40
A3 is 40x10
A4 is 10x25
p0 = 30, p1 = 1
p2 = 40, p3 = 10
p4 = 25

m[1,4]: i = 1, j = 4, k = 1, 2, 3
= min{ m[1,1]+m[2,4]+p0*p1*p4, m[1,2]+m[3,4]+p0*p2*p4,
m[1,3]+m[4,4]+p0*p3*p4}
= min{0+650+30*1*25, 1200+10000+30*40*25, 700+0+30*10*25}
= 1400
m[1,4] is the minimum value when k = 1, so s[1,4] = 1

Step 4: Using S to Print Best Ordering (cont'd)

	1	2	3	4
1	0	1	1	1
2	n/a	0	2	3
3	n/a	n/a	0	3
4	n/a	n/a	n/a	0

A1 A2 A3 A4
s[1,4] = 1 - > A1 (A2 A3 A4)
s[2,4] = 3 - > (A2 A3) A4
A1 (A2 A3 A4) -> A1 ((A2 A3) A4)

Step 3: Computing the optimal costs

```
MATRIX-CHAIN-ORDER(p)
  n = length[p] − 1
  Let m [1..n, 1..n] and s[1.. n-1, 2..n] be new tables
  for i = 1 to n
      m[i, i] = 0
  for l = 2 to n
      for i = 1 to (n − l + 1)
          j = i + l − 1
          m[i, j] = ∞
          for k = i to (j − 1)
              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
              if q < m[i, j]
                  m[i, j] = q
                  s[i, j] = k
  return m and s
```

Complexity: $O(n^3)$ Space: $\Theta(n^2)$

Step 4: Using S to Print Best Ordering

- $s[i,j]$ is the split position for $A_iA_{i+1}\ldots A_j \rightarrow A_i\ldots A_{s[i,j]}$ and $A_{s[i,j]+1}\ldots A_j$
- Call Print-Optimal-PARENS(s, 1, n)

```
Print-Optimal-PARENS (s, i, j)
  if (i == j) then
        print "A" + i            //+ is string concatenation
  else
        print "("
        Print-Optimal-PARENS (s, i,  s[i, j] )
        Print-Optimal-PARENS (s,  s[i, j]+1, j)
        Print ")"
```

16.3 Elements of dynamic programming

- **Optimal substructure**
  - a problem exhibits optimal substructure if an optimal solution to the problem contains within its optimal solutions to subproblems.
  - Example: Matrix-multiplication problem
- Overlapping subproblems
  - The space of subproblems is “small” in that a recursive algorithm for the problem solves the same subproblems over and over.
  - Total number of distinct subproblems is typically polynomial in input size
- Reconstructing an optimal solution

Optimal structure may not exist

- We cannot assume it when it is not there
- Consider the following two problems. in which we are given a directed graph G =(V,E) and vertices u, $v \in V$
  - P1: Unweighted shortest path (USP)
    - Find a path from u to v consisting of the fewest edges. Good for Dynamic programming.
  - P2: Unweighted longest simple path (ULSP)
    - A path is simple if all vertices in the path are distinct
    - Find a simple path from u to v consisting of the most edges. Not good for Dynamic programming.

Overlapping Subproblems

- Second ingredient: an optimization problem must have for DP is that the space of subproblems must be "small", in a sense that
  - A recursive algorithm solves the same subproblems over and over, rather than generating new subproblems.
  - The total number of distinct subproblems is polynomial in the input size
  - DP algorithms use a table to store the solutions to subproblems and look up the table in a constant time

Overlapping Subproblems (Cont'd)

- In contrast, a problem for which a divide-and-conquer approach is suitable when the recursive steps always generate new problems at each step of the recursion.
- Examples: Mergesort and Quicksort.
  - Sorting on smaller and smaller arrays (each recursion step work on a different subarray)

A Recursive Algorithm for Matrix-Chain Multiplication

RECURSIVE-MATRIX-CHAIN(p,i,j), called with(p,1,n)

1. **if** ($i == j$) **then return** 0

2. $m[i,j] = \infty$

3. **for** k**=** i to (j-1)

4. q **=** RECURSIVE-MATRIX-CHAIN(p,i,k)
   + RECURSIVE-MATRIX-CHAIN(p,k+1,j) + $p_{i-1}p_kp_j$

5. **if (**$q < m[i,j]$) **then** $m[i,j] = q$

6. **return** $m[i,j]$

The running time of the algorithm is $O(2^n)$.

The recursion tree

for k= i to (j-1)

q = RECURSIVE-MATRIX-CHAIN(p,i,k)

+ RECURSIVE-MATRIX-CHAIN(p,k+1,j) + $p_{i-1}p_kp_j$

RECURSIVE-MATRIX-CHAIN(p,1,4)

i =1, j = 4, k = 1, 2, 3 (i to j-1)

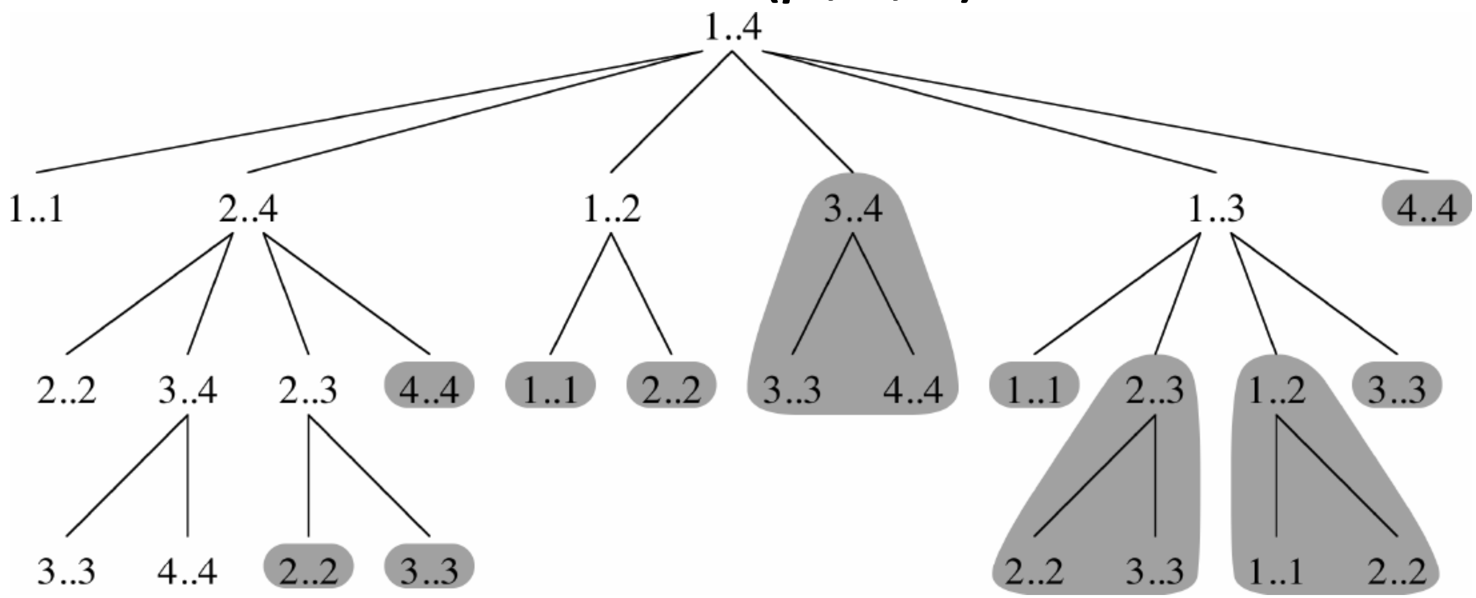
needs to solve (1, k) (k+1, 4)

k = 1 - > (1, 1) (2, 4)

k = 2 - > (1, 2) (3, 4)

K = 3 -> (1, 3) (4, 4)

Recursion tree of RECURSIVE-MATRIX-CHAIN(p,1,4)



- This divide-and-conquer recursive algorithm solves the overlapping problems over and over.
  - DP solves the same subproblems only once
  - The computations in darker color are replaced by table loop up in MEMOIZED-MATRIX-CHAIN(p,1,4).
- The divide-and-conquer is better for the problem which generates brand-new problems at each step of recursion.

General idea of Memoization

- A variation of DP
- Keep the same efficiency as DP
- But in a top-down manner.
- Idea:
  - When a subproblem is first encountered, its solution needs to be solved, and then is stored in the corresponding entry of the table.
  - If the subproblem is encountered again in the future, just look up the table to take the value.

Memoized Matrix Chain

```
MEMOIZED-MATRIX-CHAIN(p)
n ← length[p] − 1
for i ← 1 to n
      do for j ← i to n
              do m[i, j] ← ∞
return LOOKUP-CHAIN(p, 1, n)
```

```
LOOKUP-CHAIN(p,i,j)
if  m[i,j]<∞   then return m[i,j]
if  (i ==j)  then m[i,j] =0
else   for k= i to j-1
              q=LOOKUP-CHAIN(p,i,k)+
                     LOOKUP-CHAIN(p,k+1,j) + p_{i-1}p_kp_j
          if (q< m[i,j])  then m[i,j] = q
return m[i,j]
```

DP VS. Memoization

- MCM can be solved by DP or Memoized algorithm, both in $O(n^3)$
  - Total $\Theta(n^2)$ subproblems, with $O(n)$ for each.
- If all subproblems must be solved at least once, DP is better by a constant factor due to no recursive involvement as in memorized algorithm
- If some subproblems may not need to be solved, Memoized algorithm may be more efficient
  - since it only solve these subproblems which are definitely required.

Longest Common Subsequence (LCS)

- DNA analysis to compare two DNA strings
- DNA string: a sequence of symbols A,C,G,T
  - S =ACCGGTCGAGCTTCGAAT
- Subsequence of X is X with some symbols left out
  - Z =CGTC is a subsequence of X =ACGCTAC
- Common subsequence Z of X and Y: a subsequence of X and also a subsequence of Y
  - Z =CGA is a common subsequence of X =ACGCTAC and Y =CTGACA
- Longest Common Subsequence (LCS): the longest one of common subsequences
  - Z' =CGCA is the LCS of the above X and Y
- LCS problem: given $X = \langle x_1, x_2,\ldots, x_m \rangle$ and $Y = \langle y_1, y_2,\ldots, y_n \rangle$, find their LCS

LCS DP step 2: Recursive Solution

- What the theorem says:
  - If $x_m == y_n$, find LCS of $X_{m\text{-}1}$ and $Y_{n\text{-}1}$, then append x_m
  - If $x_m \neq y_n$, find (1) the LCS of $X_{m\text{-}1}$ and Y_n and (2) the LCS of X_m and $Y_{n\text{-}1}$; then, take which one is longer
- Overlapping substructure:
  - Both LCS of $X_{m\text{-}1}$ and Y_n and LCS of X_m and $Y_{n\text{-}1}$ will need to solve LCS of $X_{m\text{-}1}$ and $Y_{n\text{-}1}$ first
- $c[i,j]$ is the length of LCS of X_i and Y_j

$$c[i,j]=\begin{cases} 0 & \text{if } i = 0, \text{ or } j = 0 \\ c[i\text{-}1, j\text{-}1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{\, c[i\text{-}1,j], c[i,j\text{-}1] \,\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

LCS DP step 3: Computing the Length of LCS

$$c[i,j] = \begin{cases} 0 & \text{if } i=0, \text{ or } j=0 \\ c[i\text{-}1, j\text{-}1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{\, c[i\text{-}1,j], c[i,j\text{-}1] \,\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

- *c*[0..*m*, 0..*n*], where *c*[*i*,*j*] is defined as above.
  - *c*[*m*,*n*] is the answer (length of LCS)
- *b*[1..*m*, 1..*n*], where *b*[*i*,*j*] points to the table entry corresponding to the optimal subproblem solution chosen when computing *c*[*i*,*j*].
  - From *b*[*m*, *n*] backward to find the LCS.

LCS DP Algorithm

```
LCS-LENGTH(X, Y)
    m ← length[X]
    n ← length[Y]
    for i ← 1 to m
         do c[i, 0] ← 0
    for j ← 0 to n
         do c[0, j] ← 0
    for i ← 1 to m
         do for j ← 1 to n
                 do if x_i = y_j
                       then c[i, j] ← c[i − 1, j − 1] + 1
                            b[i, j] ← "↖"
                       else  if c[i − 1, j] ≥ c[i, j − 1]
                                then c[i, j] ← c[i − 1, j]
                                     b[i, j] ← "↑"
                                else  c[i, j] ← c[i, j − 1]
                                     b[i, j] ← "←"
    return c and b
```

LCS Example (0)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,6]

LCS Example (1)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

for i = 1 to m   c[i,0] = 0
for j = 1 to n   c[0,j] = 0

LCS Example (2)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**				
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (Xi == Yj)

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

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (3)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**		
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

if (Xi == Yj)

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

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (4)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

if (Xi == Yj)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (5)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1** →	**1**
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (Xi == Yj)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (6)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1				
3	C		0					
4	B		0					

if (Xi == Yj)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (7)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	
3	**C**	**0**					
4	**B**	**0**					

if (Xi == Yj)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (8)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**					
4	**B**	**0**					

if (Xi == Yj)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (10)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**			
4	**B**	**0**					

if (Xi == Yj)

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

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (11)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**		
4	**B**		**0**					

if (Xi == Yj)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (12)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**					

if (Xi == Yj)

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

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (13)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**	**1**				

if (Xi == Yj)

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

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (14)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**	**1**	**1**	**2**	**2**	

if (Xi == Yj)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (15)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	3

if (Xi == Yj)

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

else c[i,j] = max(c[i-1,j], c[i,j-1])

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Figure 15.8 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$—the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of X and Y. For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner; the path is shaded. Each "↖" on the path corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

Greedy Algorithms

- We have learned two design techniques
  - Divide-and-conquer
  - Dynamic Programming
- Now, the third → Greedy Algorithms
  - Optimization often goes through some choices
  - Make local best choices → hope to achieve global optimization
  - Many times, this works; Other times, does NOT!
    - Minimum spanning tree algorithms
  - We must carefully examine if we can apply this method

An activity-selection problem

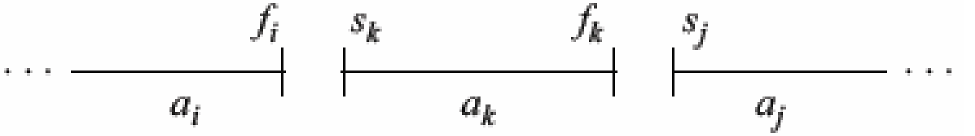
- Activity set $S = \{a_1, a_2, ..., a_n\}$
- n activities wish to use a single resource
- Each activity a_i has a **start time** s_i and a **finish time** f_i, where $0 \leq s_i < f_i < \infty$
- If selected, activity a_i take place during the half-open time interval $[s_i, f_i)$
- Activities a_i and a_j are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap
  - a_i and a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$

The greedy choice

- Intuition: Choose an activity that leaves the resource available for as many other activities as possible
- It must finish as early as possible: greedy
- Let $S_k = \{a_i \in S : s_i >= f_k\}$ be the set of activities that start after activity a_k finishes
- If we make the greedy choice of activity a_1 (i.e., a_1 is the first activity to finish), then S_1 remains as the only subproblem to solve.
  - **a_1 + S_1 , if S_1 is the optimal solution for others → a_1 must be in the optimal solution**
  - **Is this correct?**

Optimal substructure

- $S_{i\,j}$ is the subset of activities that can
  - start after activity a_i finishes
  - and finish before activity a_j starts
  - $S_{i\,j} = \{ a_k \in S: f_i \leq s_k < f_k \leq s_j \}$
  - $f_0 = 0$ and $s_{n+1} = \infty$. Then $S = S_{0,n+1}$, and the ranges for i and j are given by $0 \leq i, j \leq n+1$
- Define A_{ij} as the maximum set in S_{ij}
  - Selecting a_k in the optimal solutions generates two subproblems
  - $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$
  - $|A_{ij}| = |A_{ik}| + 1 + |A_{kj}|$



Converting a dynamic-programming solution to a greedy solution

- **Theorem 16.1** Consider any nonempty subproblem S_k, and let a_m be the activity in S_k with the earliest finish time: $f_m = \min \{ f_x : a_x \in S_k\}$. Then a_m is used in some maximum-size subset of mutually compatible activities of S_k
- Let A_k be the maximum-size subset of mutually compatible activities in S_k
- Let a_j be the activity in A_k with the earliest finish time
- If $a_j == a_m$, we are done.
- Otherwise, $A'_k = A_k - \{a_j\} \cup \{a_m\}$
- We have new A_k with a_m

An iterative greedy algorithm

GREEDY-ACTIVITY-SELECTOR(s, f)

```
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s_m ≥ f_k
        then A = A ∪ {a_m}
            k = m
return A
```

Ingredients of Greedy ALs

- **Greedy-choice property:** A global optimal solution can be achieved by making a local optimal choice.
  - **Without considering results of subproblems**
- **Optimal substructure:** An optimal solution to the problem within its optimal solution to subproblem

The End