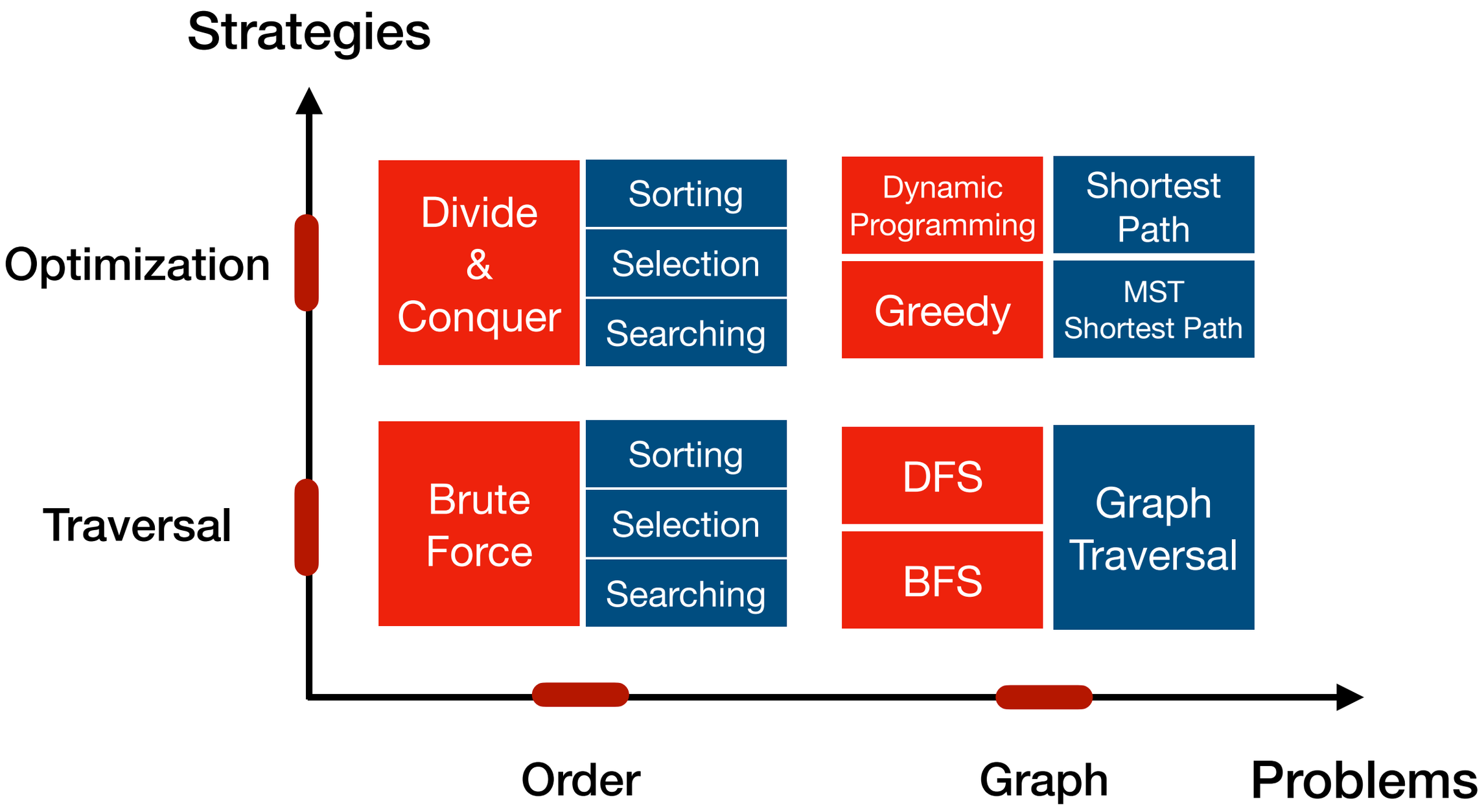
Introduction to

Algorithm Design and Analysis

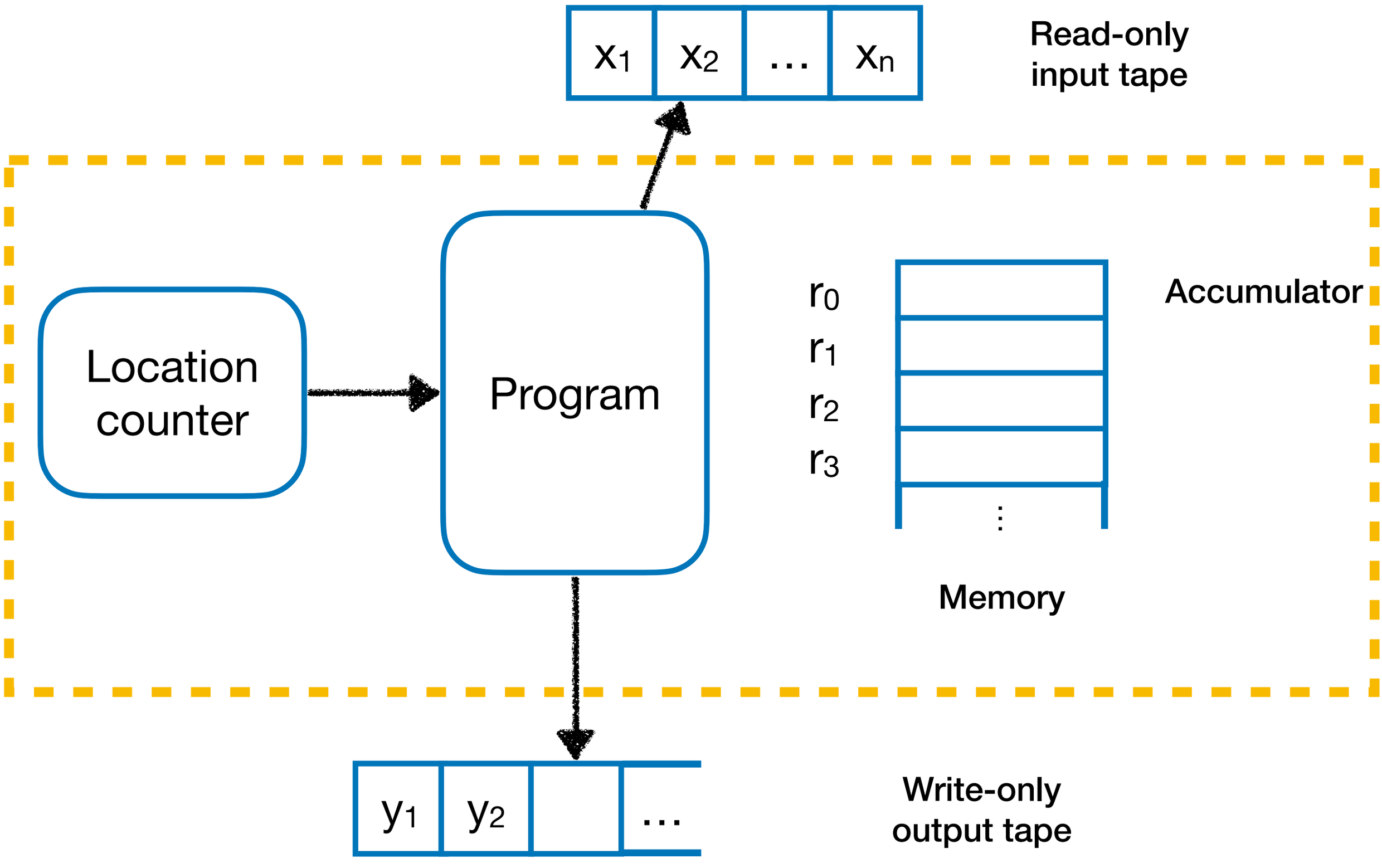
Summary

Jingwei Xu
http://ics.nju.edu.cn/~xjw
Institute of Computer Software
Nanjing University

Syllabus


Strategies
Optimization
Traversal
Order
Graph
Problems
Divide & Conquer
Sorting
Selection
Searching
Dynamic Programming
Greedy
Shortest Path
MST
Shortest Path
Brute Force
Sorting
Selection
Searching
DFS
BFS
Graph Traversal

RAM Model

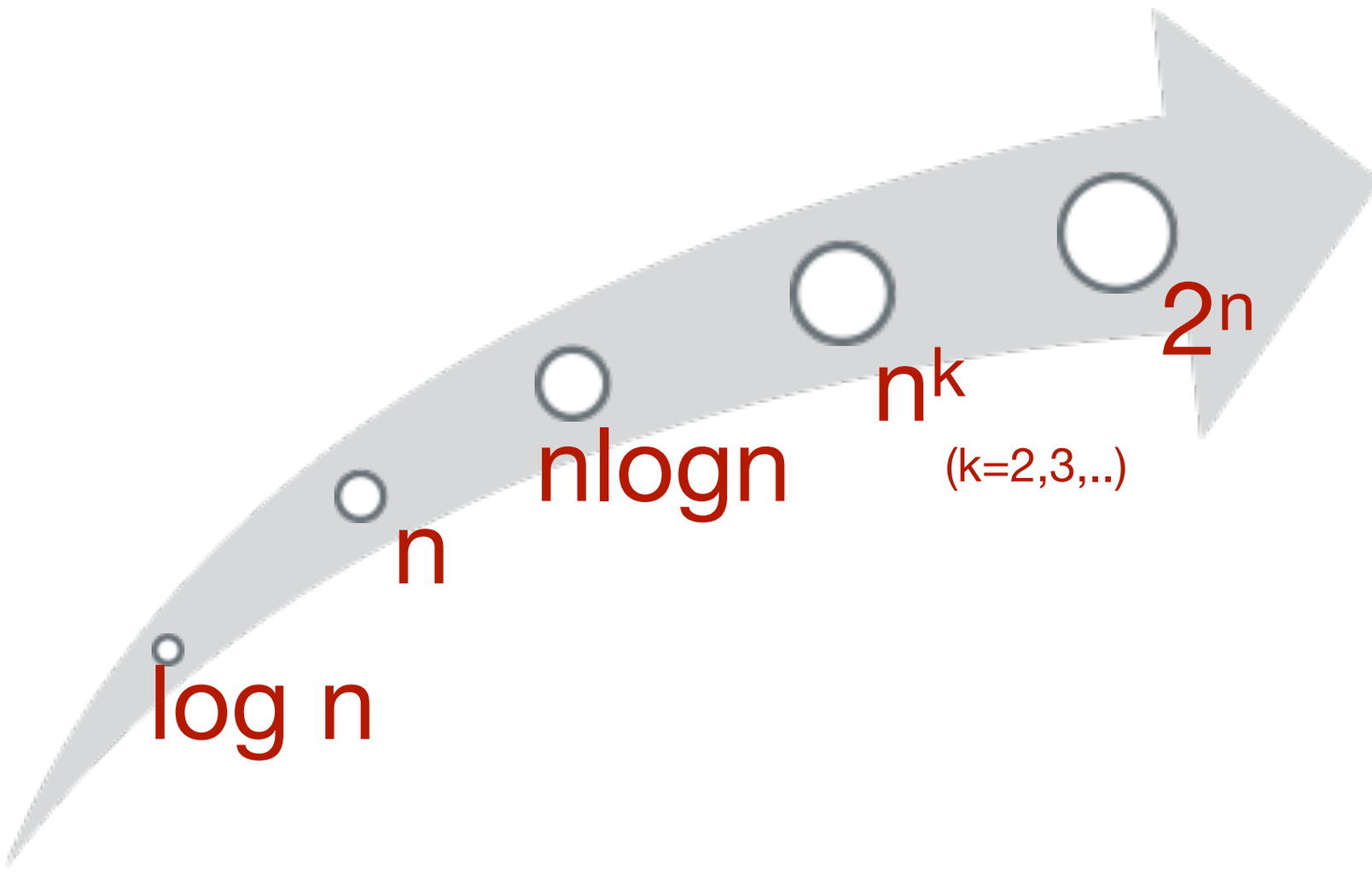

x_1 x_2 … x_n
Read-only input tape
Location counter
Program
r_0
r_1
r_2
r_3
Accumulator
Memory
y_1 y_2 …
Write-only output tape

The RAM Model of Computation

- **Each simple operation takes one time step**
  - E.g., key comparison, +/-, memory access, …
- **Non-simple operations should be decomposed**
  - Loop
  - Subroutine
- **Memory**
  - Memory access is simple operation
  - Unlimited memory

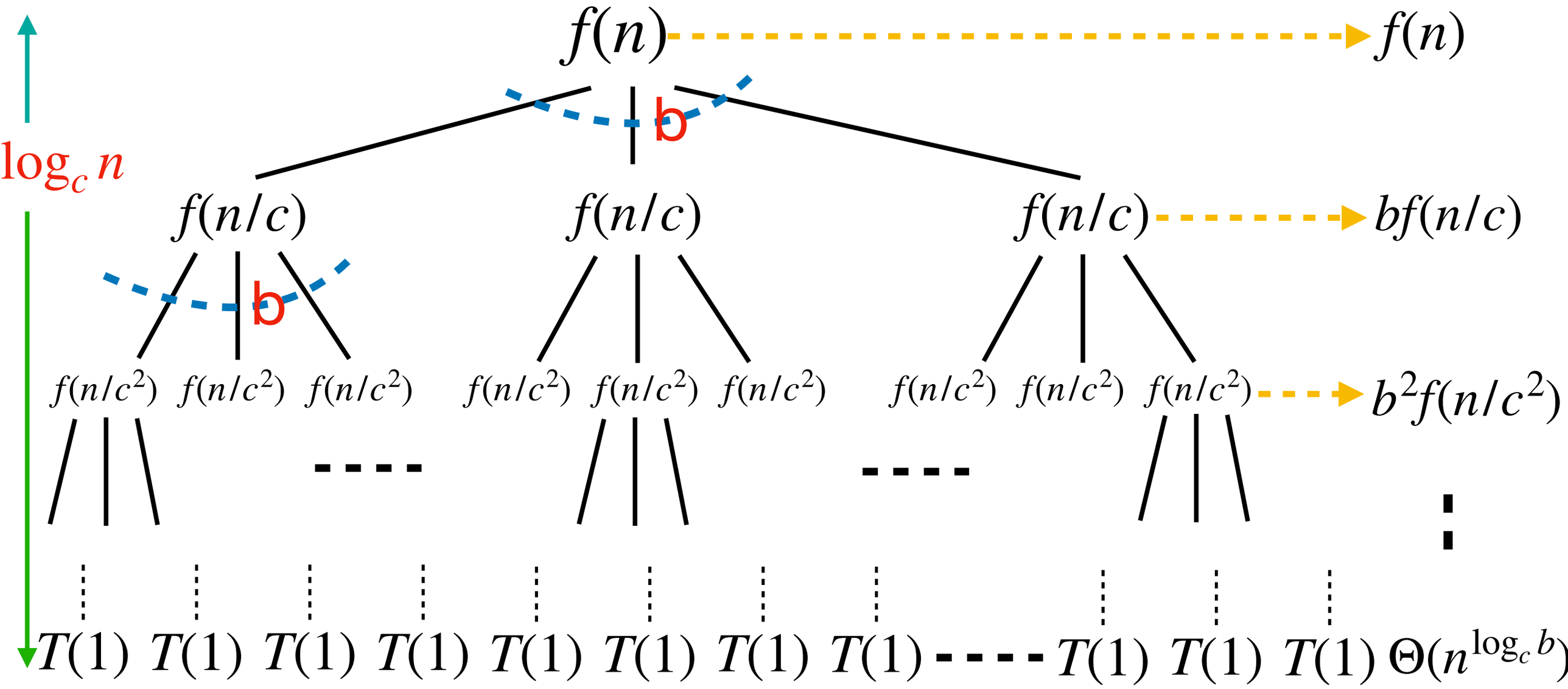
Tradeoff: accuracy v.s. ease of use

Asymptotic Growth Rate



Recursion Tree for

$$T(n) = bT(n/c) + f(n)$$



Note: $b^{\log_c n} = n^{\log_c b}$

Total?

Master Theorem

- **Loosening the restrictions on f(n)**
  - Case 1: $f(n) \in \mathrm{O}(n^{E-\varepsilon}), (\varepsilon > 0)$, then:

$$T(n) \in \Theta(n^E)$$

  - Case 2: $f(n) \in \Theta(n^E)$, as all node depth contribute about equally:

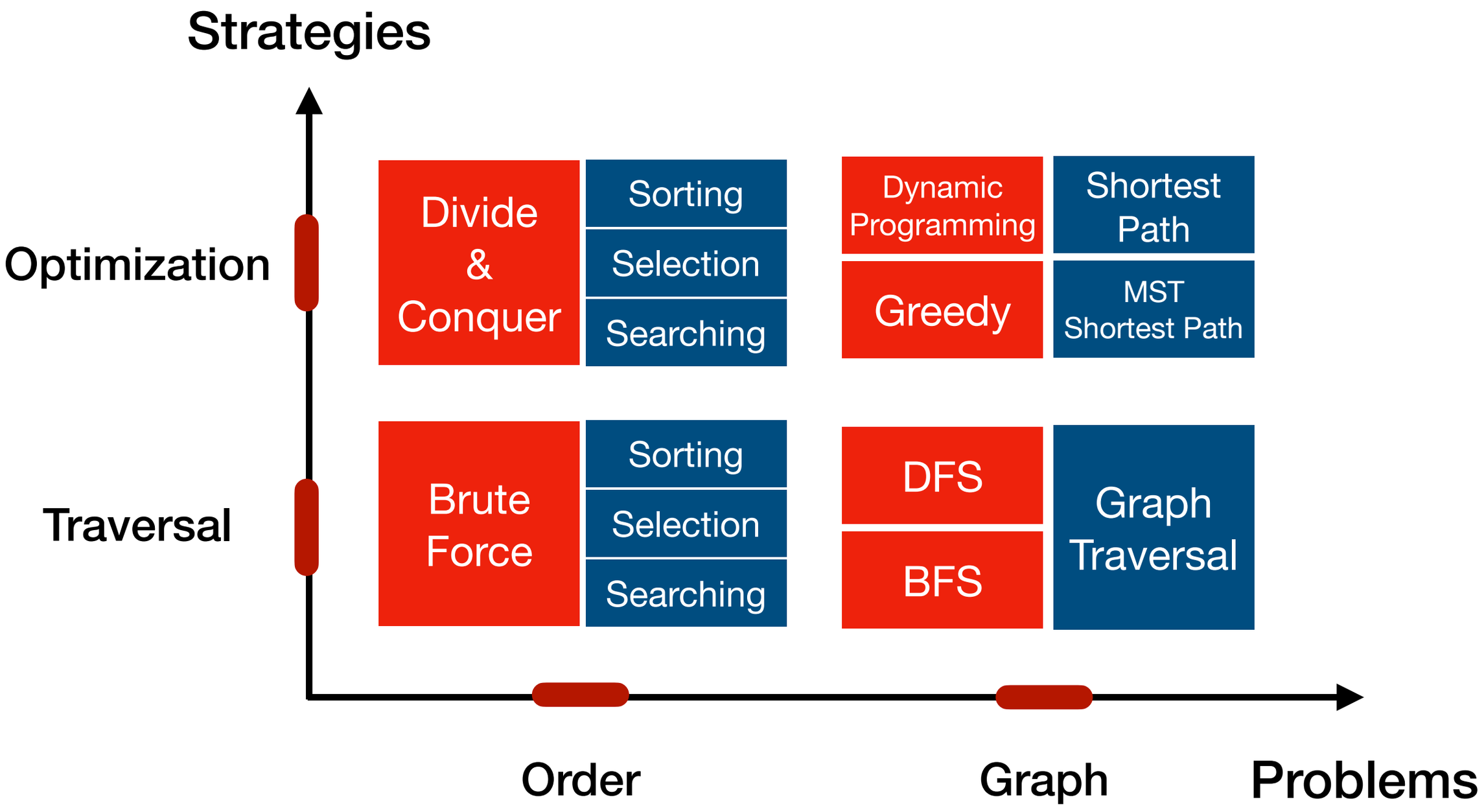
$$T(n) \in \Theta(f(n)\log(n))$$

  - Case 3: $f(n) \in \Omega(n^{E+\varepsilon}), (\varepsilon > 0)$, and of $bf(n/c) \leq \theta f(n)$ for some constant θ<1 and all sufficiently large n, then:

$$T(n) \in \Theta(f(n))$$

The positive ε is critical, resulting gaps between cases as well.

Syllabus


Strategies
Optimization
Traversal
Order
Graph
Problems
Divide & Conquer
Sorting
Selection
Searching
Dynamic Programming
Greedy
Shortest Path
MST
Shortest Path
Brute Force
Sorting
Selection
Searching
DFS
BFS
Graph Traversal

The Sorting Problem

- **Sorting**
  - E.g., sort all the students according to their GPA
- **Assumptions for analysis of sorting**
  - What to sort?
    - Problem size n: elements a1,a2,…,an with no identical keys
  - In which order to sort?
    - Sort in increasing order
  - What are the inputs likely to be?
    - Each possible input appears with the same probability

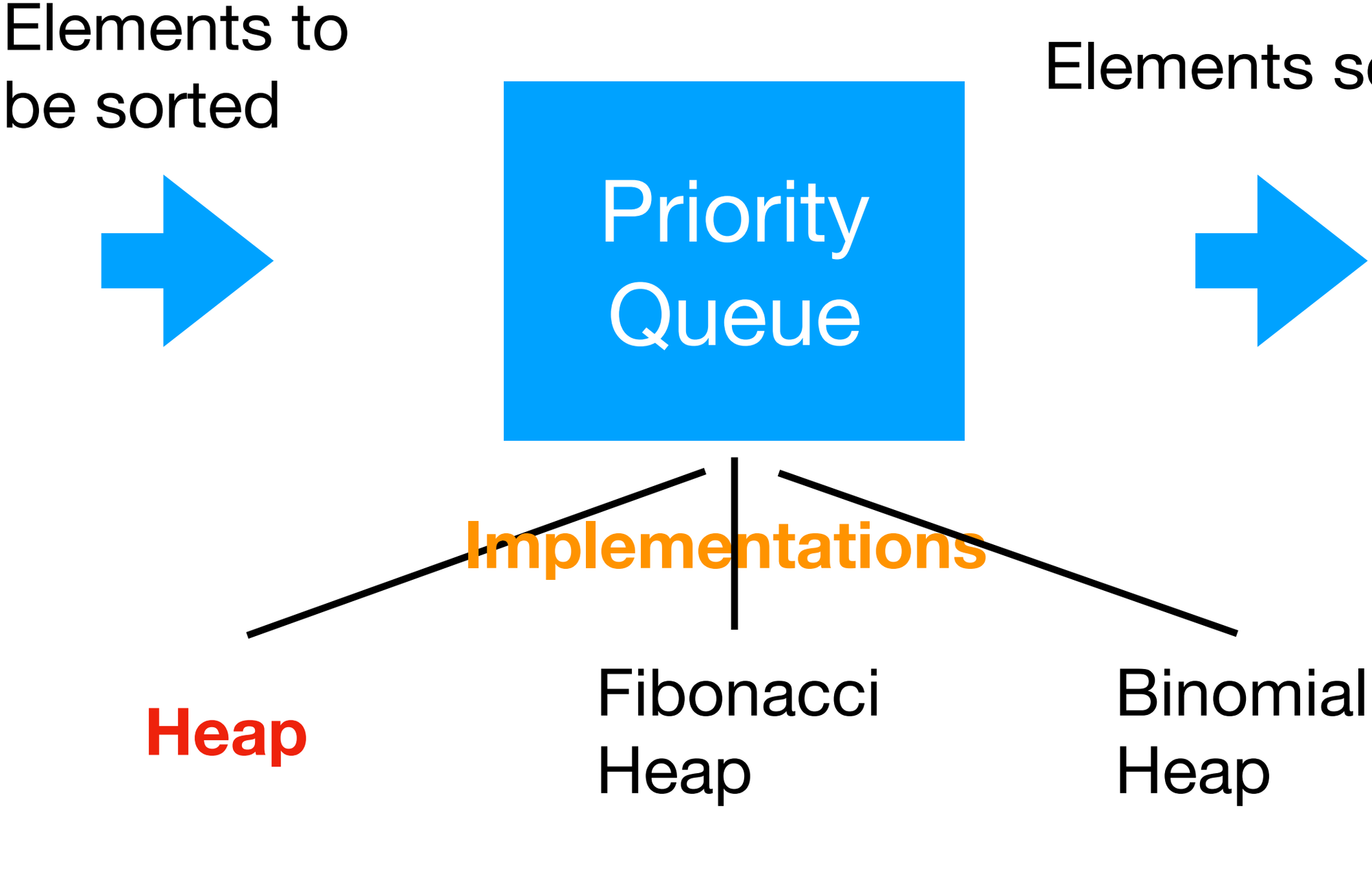
Comparison-Based Sorting

- **Sorting a number of keys**
  - The class of "algorithms that sort by comparison of keys"
- **Critical operation**
  - Comparison between two keys
  - No other operations are allowed for sorting
- **Amount of work done**
  - The number of critical operations (key comparisons)

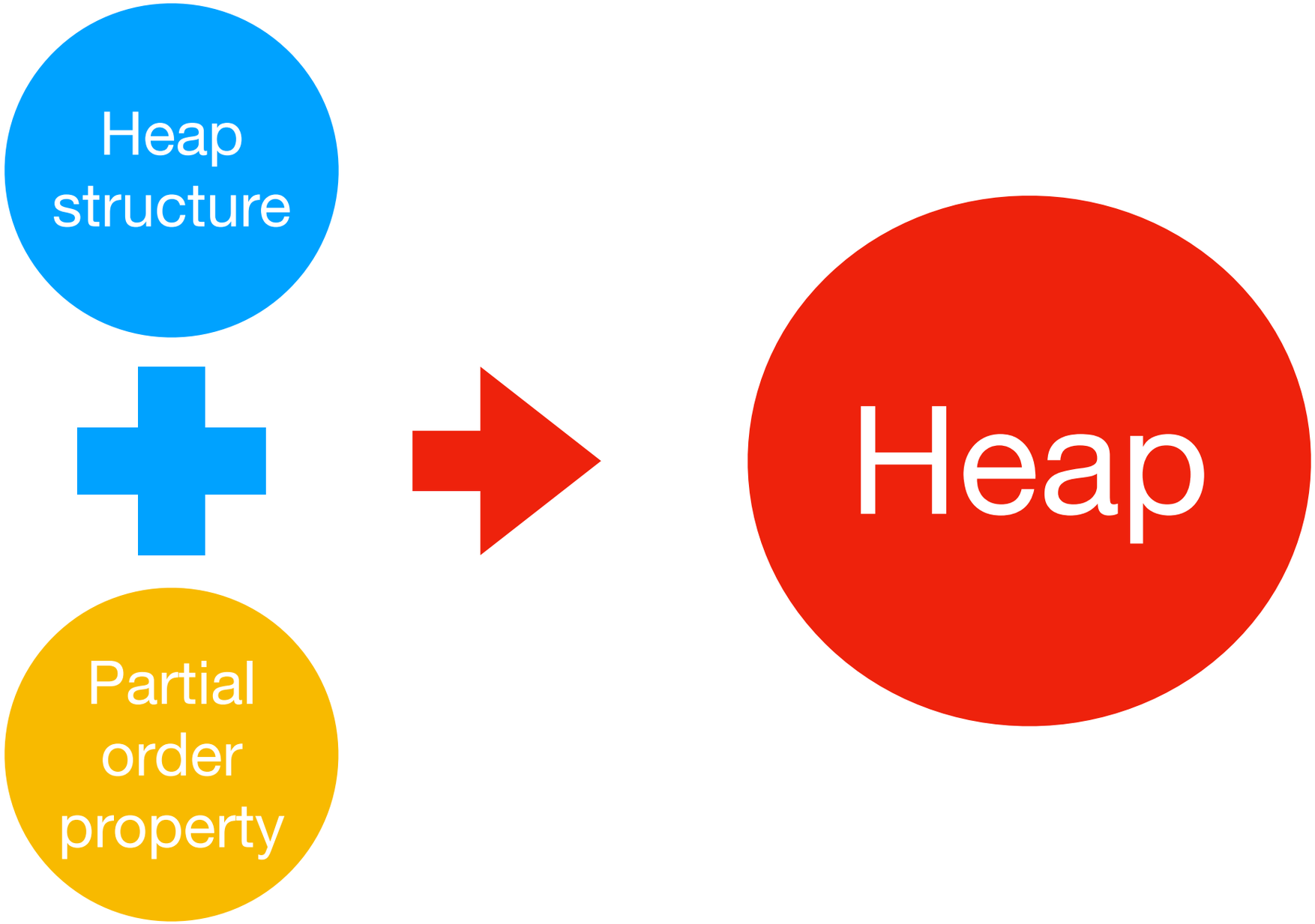
HeapSort

- Heap
- HeapSort
- FixHeap
- ConstructHeap

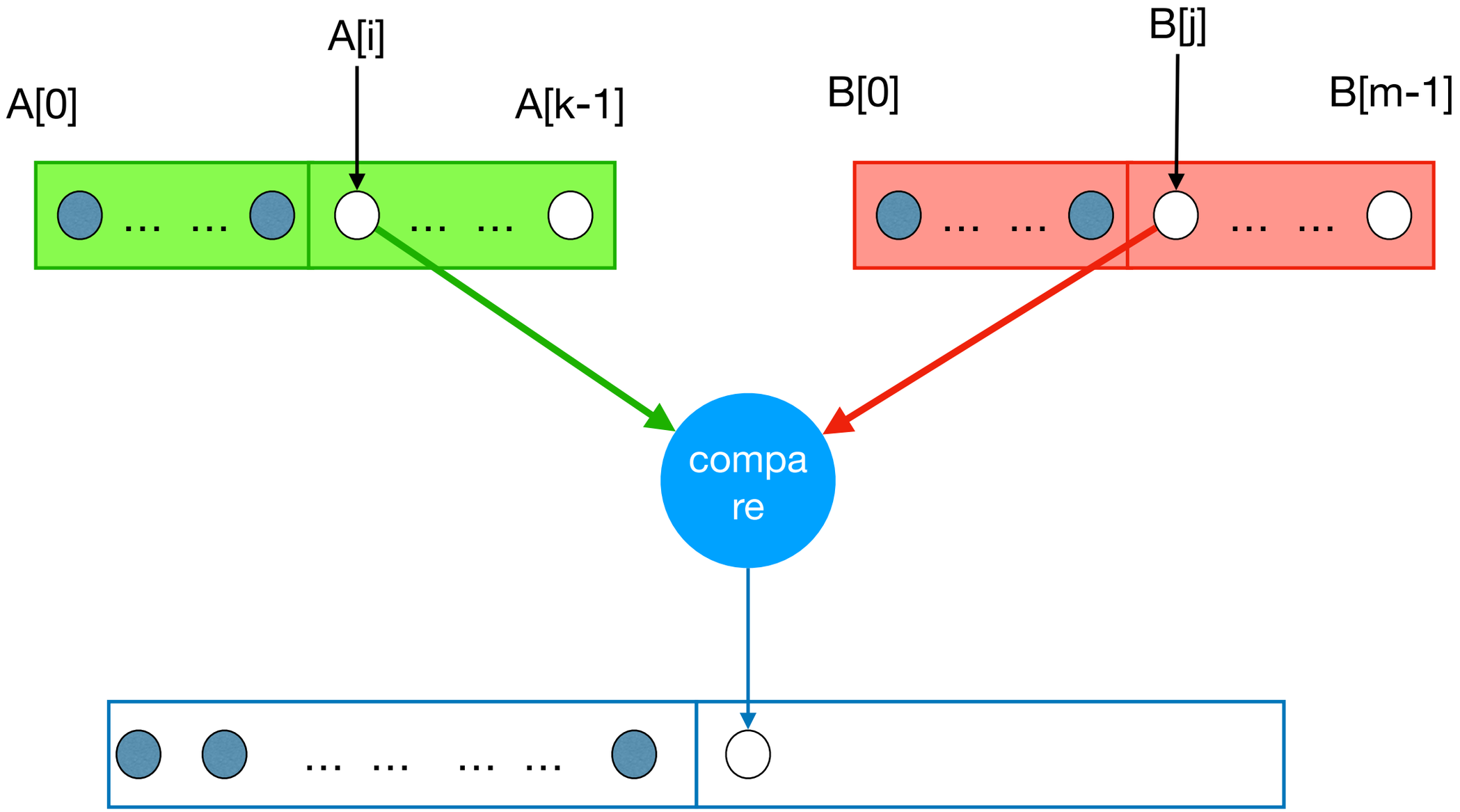
How HeapSort Works



Heap: an Implementation of Priority Queue



Merging Sorted Arrays



Inversion and Sorting

- **An unsorted sequence E:**
  - $\{x_1, x_2, x_3, \ldots, x_{n-1}, x_n\} = \{1, 2, 3, \ldots, n-1, n\}$
- **$\langle x_i, x_j \rangle$ is an inversion if $x_i > x_j$, but $i < j$**
- **Sorting ≡Eliminating inversions**
  - All the inversions must be eliminated during the process of sorting

Eliminating Inverses: Worst Case

- Local comparison is done between two adjacent elements
- At most one inversion is removed by a local comparison
- There do exist inputs with n(n-1)/2 inversions, such as (n, n-1, …, 3, 2, 1)
- The worst-case behavior of any sorting algorithm that remove at most one inversion per key comparison must in $\Omega(n^2)$

MergeSort: the Strategy

- **Easy division**
  - No comparison is conducted during the division
    - Minimizing the size difference between the divided subproblems
- **Merging two sorted subranges**
  - Using Merge

More than Sorting (QS)

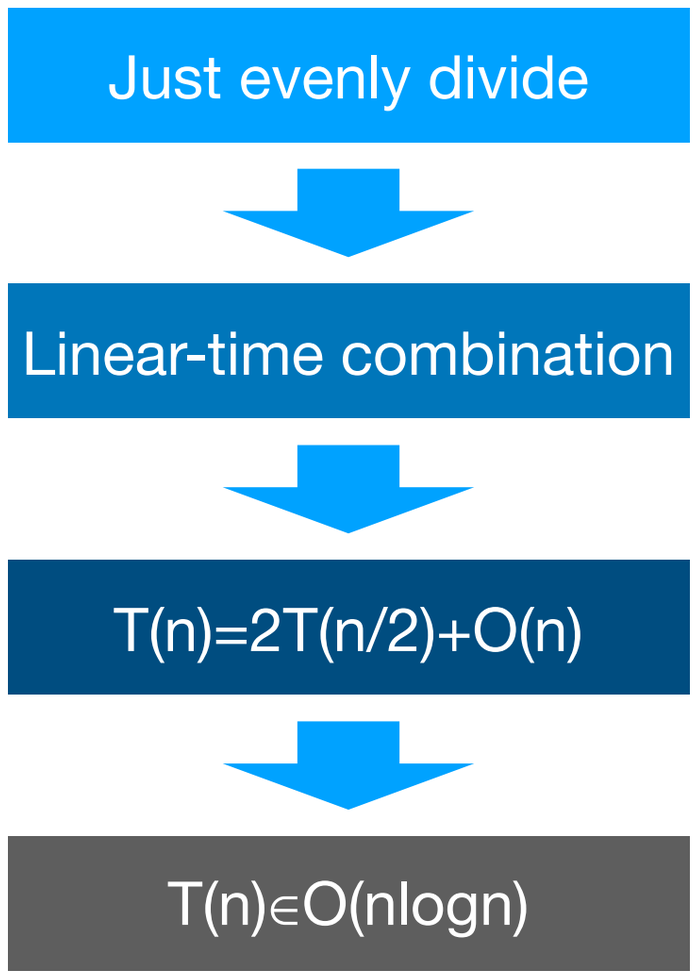
- **QuickSort Partition**
  - O(n)
- **Bolts and nuts**
  - O(nlogn)
- **k-Sorted**
  - O(nlogk)

Not only for Sorting (Heap)

- **Eg1: how to find the k^{th} max element?**
  - The cost should be f(k)
- **Eg2: how to find the first k elements?**
  - In sorted order?
- **Eg3: how to merge k sorted lists?**
- **Eg4: how to find the median dynamically?**
- **...**

The MergeSort D&C

- Max-sum subsequence
- Maxima on a plane
- Finding the frequent element
- Integer/matrix multiplication
- …

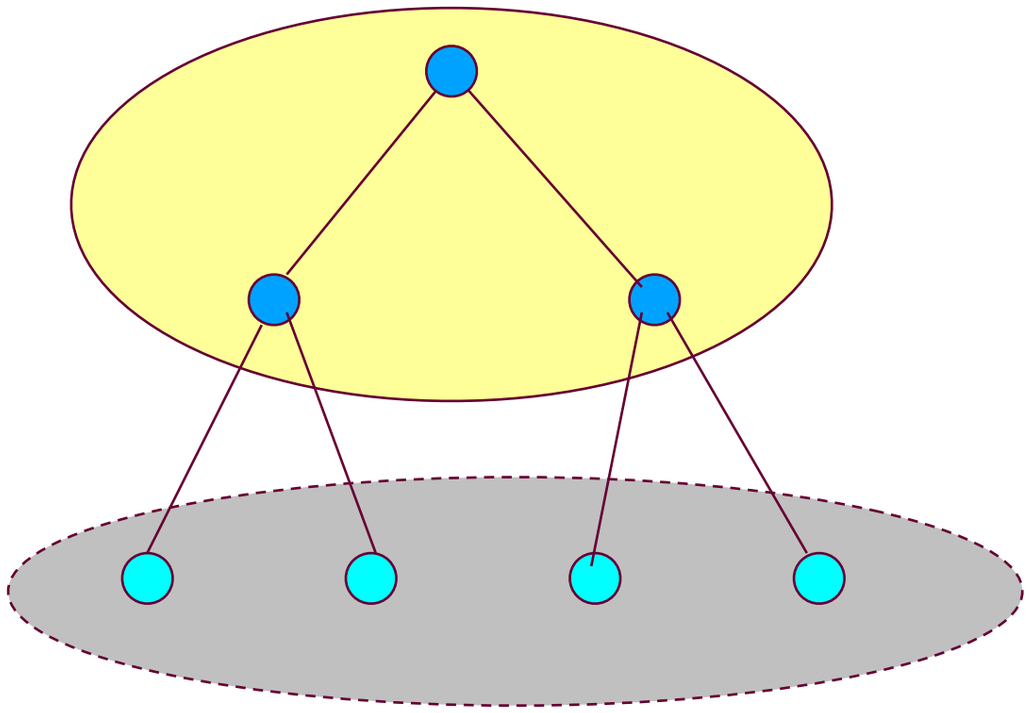


Lower Bounds for Comparison-based Sorting

- **Upper bound, e.g., worst-case cost**
  - For any possible input, the cost of the specific algorithm A is no more than the upper bound
    - Max{cost(i) | i is an input}
- **Lower bound, e.g., comparison-based sorting**
  - For any possible (comparison-based) sorting algorithm A, the worst-case cost is no less than the lower bound
    - Min{worst-case(a) | a is an algorithm}

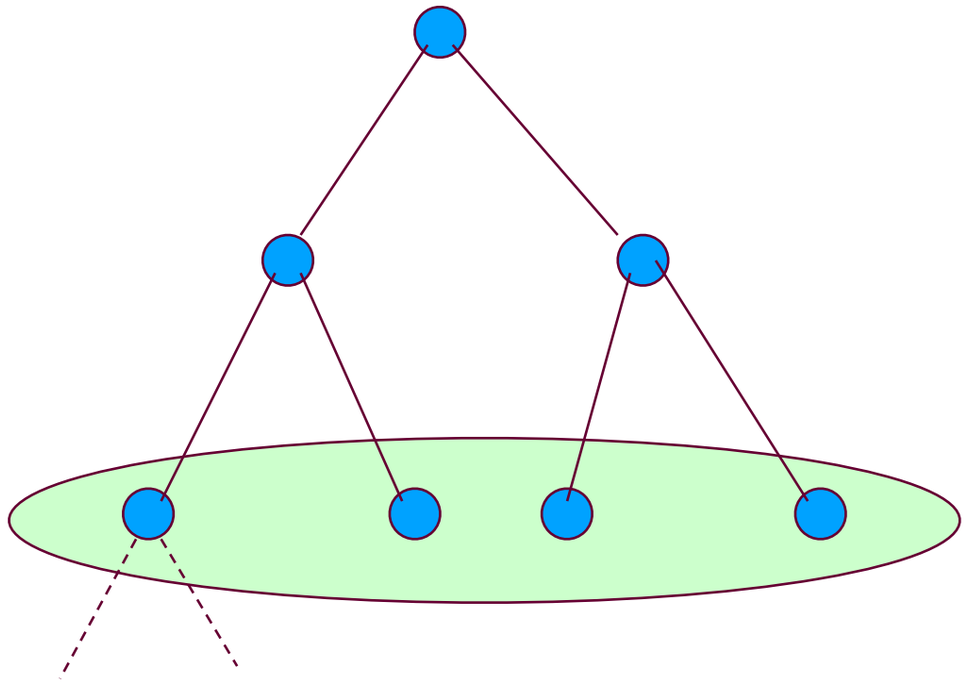
2-Tree

- 2-Tree



external nodes
no child any type

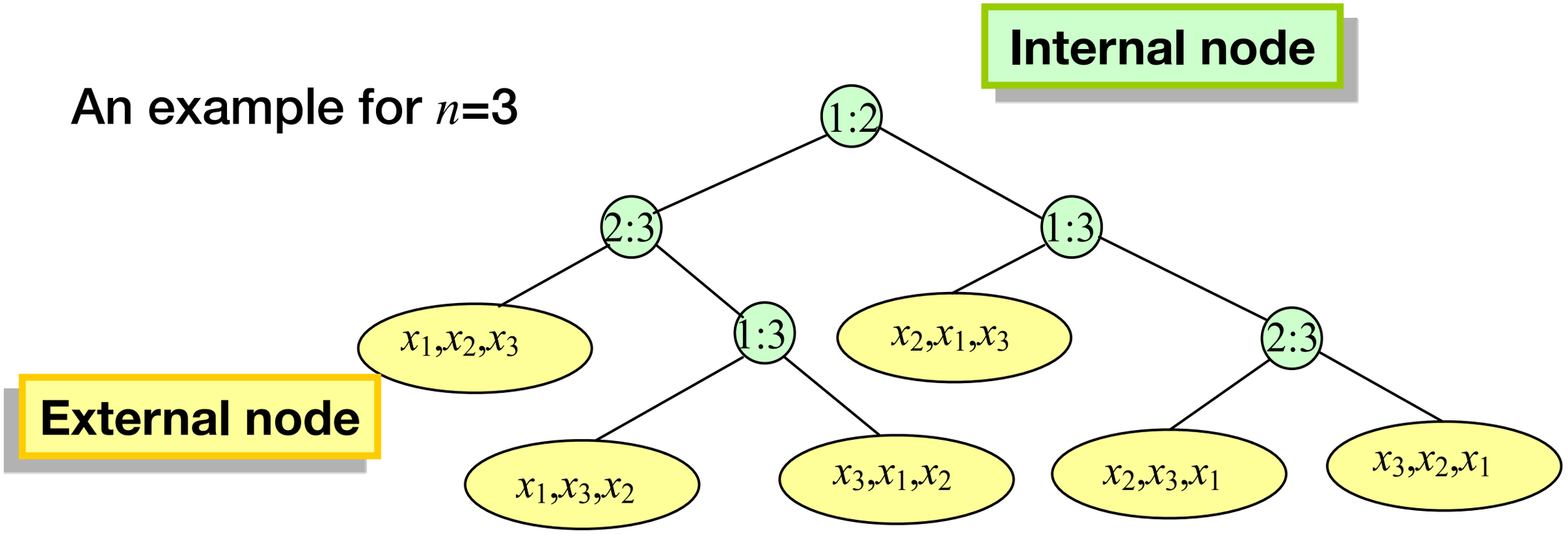
- **Common Binary Tree**



Both left and right children of these nodes are empty tree

Decision Tree for Sorting

An example for n=3



- Decision tree is a 2-tree (Assuming no same keys)
- The action of Sort on a particular input corresponds to following on path in its decision tree from the root to a leaf associated to the specific output

Characterizing the Decision Tree

- For a sequence of n distinct elements, there are n! different permutation
  - So, the decision tree has at least n! leaves, and exactly n! leaves can be reached from the root.
  - So, for the purpose of lower bounds evaluation, we use trees with exactly n! leaves.
- The number of comparison done in the worst case is the height of the tree.
- The average number of comparison done is the average of the lengths of all paths from the root to a leaf.

Lower Bound for Worst Case

- ***Theorem***: Any algorithm to sort n items by comparisons of keys must do at least $\lceil \lg n! \rceil$, or approximately $\lceil n\lg n - 1.443n \rceil$, key comparisons in the worst case.
  - Note: Let $L=n!$, which is the number of leaves, then $L \le 2^h$, where h is the height of the tree, that is $h \ge \lceil \lg L \rceil = \lceil \lg n! \rceil$
  - For the asymptotic behavior:

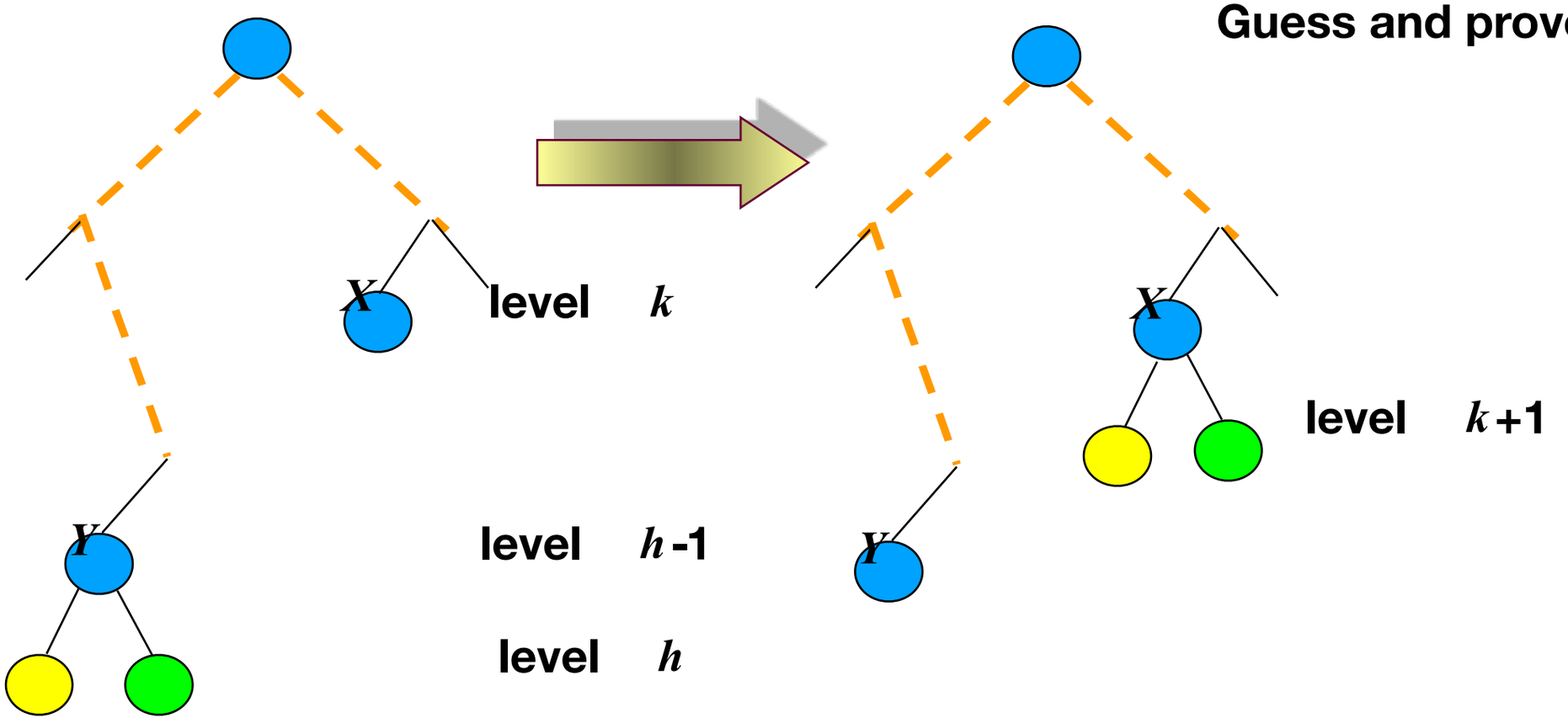
$$\lg(n!) \ge \lg[n(n-1)\ldots\left(\left\lceil \frac{n}{2} \right\rceil\right)] \ge \lg\left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2}\lg\left(\frac{n}{2}\right) \in \Theta(n\lg n)$$

derived using: $\lg n! = \sum_{j=1}^{n} \lg(j)$

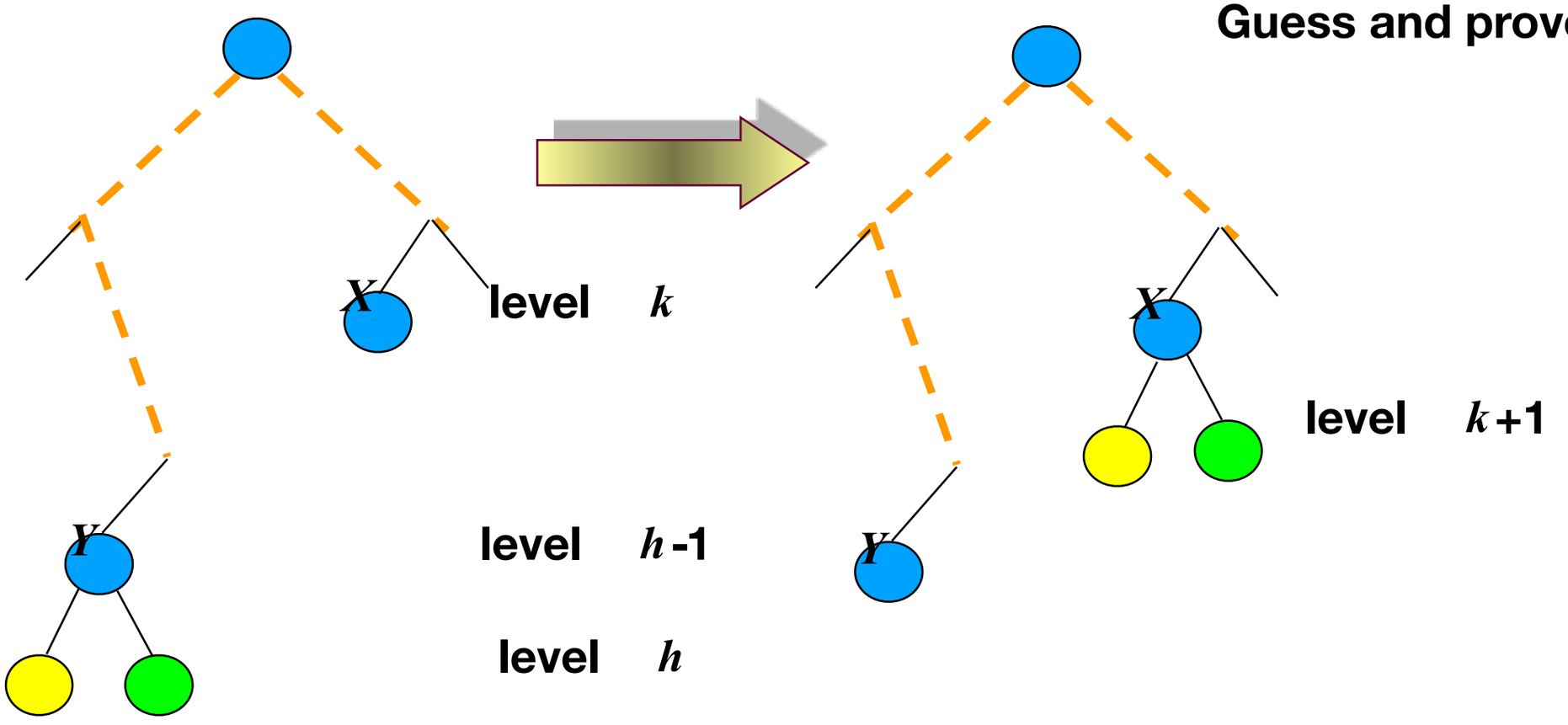
External Path Length (EPL)

- The EPL of a 2-tree t is defined as follows:
  - [Base case] 0 for a single external node
  - [Recursion] *t* is non-leaf with sub-trees *L* and *R*, then the sum of:
    - the external path length of *L*;
    - the number of external node of *L*;
    - the external path length of *R*;
    - the number of external node of *R*;

More Balanced 2-tree, Less EPL



More Balanced 2-tree, Less EPL



Assuming that h-k>1, when calculating epl, h+h+k is replaced by (h-1)+2(k+1). The net change in epl is k-h+1<0, that is, the epl decreases.

So, more balanced 2-tree has smaller epl.

Properties of EPL

- **Let t is a 2-tree, then the epl of t is the sum of the paths from the root to each external node.**
- $epl \geq m\lg(m)$, where m is the number of external nodes in t
  - $epl = epl_L + epl_R + m \geq m_L\lg(m_L) + m_R\lg(m_R) + m$,
  - note $f(x)+f(y) \geq 2f((x+y)/2)$ for $f(x)=x\lg x$
  - so,

    $epl \geq 2((m_L+m_R)/2)\lg((m_L+m_R)/2)+m = m(\lg(m)-1)+m$

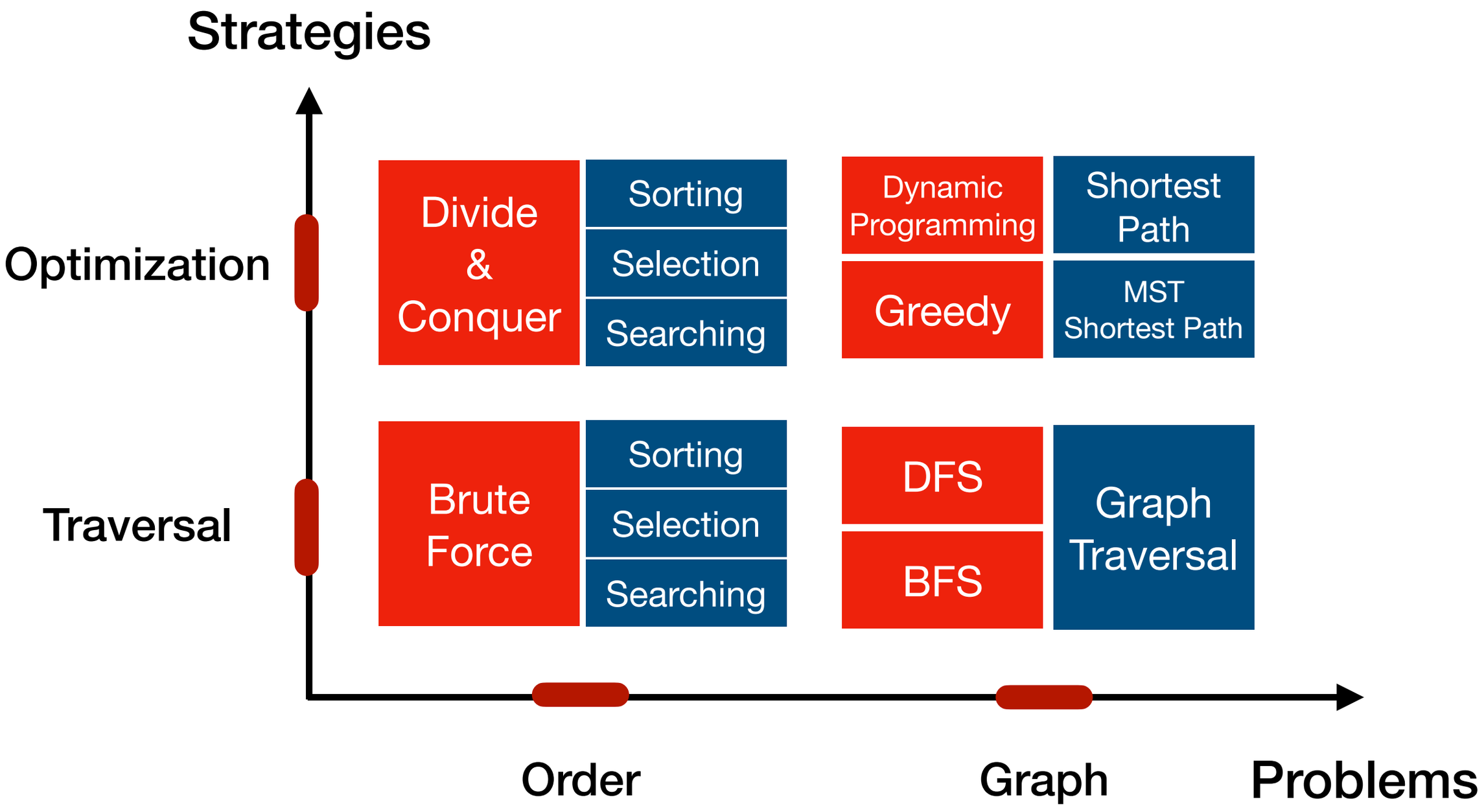
    $=m\lg m$.

Lower Bound for Average Behavior

- Since a decision tree with L leaves is a 2-tree, the average path length from the root to a leaf is $\frac{epl}{L}$
  - Recall that $epl \geq L\lg(L)$.

- **Theorem**: The average number of comparison done by an algorithm to sort n items by comparison of keys is at least $\lg(n!)$, which is about $n\lg n - 1.443n$.

Syllabus


Strategies
Optimization
Traversal
Order
Graph
Problems
Divide & Conquer
Sorting
Selection
Searching
Dynamic Programming
Greedy
Shortest Path
MST
Shortest Path
Brute Force
Sorting
Selection
Searching
DFS
BFS
Graph Traversal

The Selection Problem

- **Problem Definition**
  - Suppose E is an array containing n elements with keys from some linearly order set, and let k be an integer such that 1<=k<=n. The selection problem is to find an element with the k^{th} smallest key in E.

- **Special cases**
  - Find the max/min -k=n or k=1
  - Find the median (k=n/2)

Selection
v.s.
Searching

Adversary Strategy

Status of keys x and y Compared by an algorithm	Adversary response	New status	Units of new information
N,N	$x>y$	W,L	2
W,N or WL,N	$x>y$	W,L or WL,L	1
L,N	$x<y$	L,W	1
W,W	$x>y$	W,WL	1
L,L	$x>y$	WL,L	1
W,L or WL,L or W,WL	$x>y$	No change	0
WL,WL	Consistent with Assigned values	No change	0

The principle: let the key win if it never lose, or, let the key lose if it never win, and change one value if necessary.

Lower Bound by the Adversary Argument

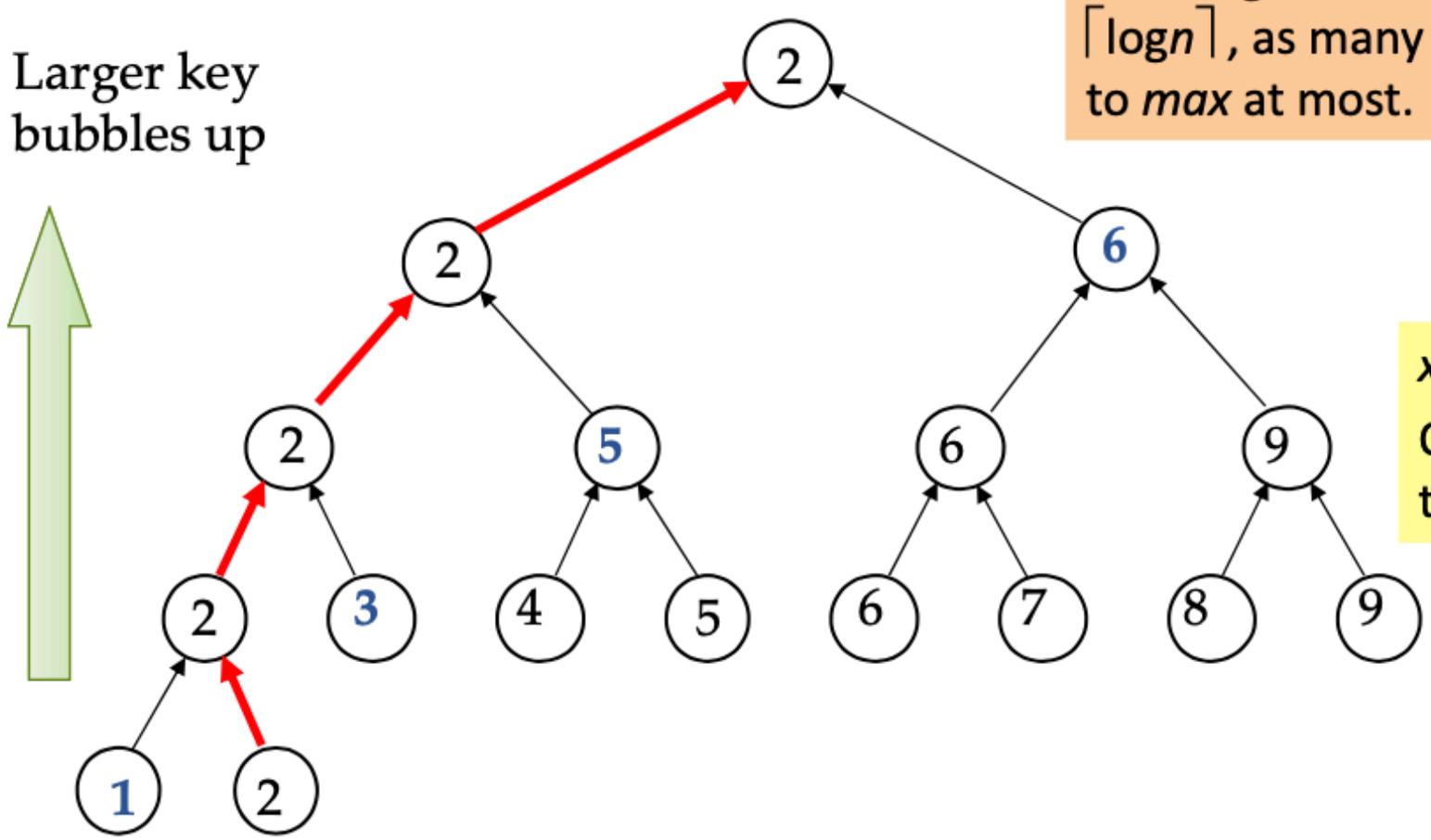
- **Construct an input to force *the* algorithm to do more comparisons as possible**
  - To give away as few as possible units of new information with each comparison.
    - It can be achieved that 2 units of new information are given away only when the status is N,N.
    - It is ***always*** possible to give adversary response for other status so that at most one new unit of information is given away, ***without any inconsistencies***.
- **So, the *Lower Bound* is *n*/2+n-2(for even *n*)**

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

Find the 2nd Largest Key

- **Brute force - using FindMax twice**
  - Need 2n-3 comparisons.
- **For a better algorithm**
  - Collect some useful information from the first FindMax
- **Observations**
  - The key which loses to a key other than max cannot be the 2nd largest key.
  - To check “whether you lose to max?”

Tournament for the 2nd Largest Key



The length of the longest path is $\lceil \log n \rceil$, as many as those compared to *max* at most.

x_2 is *max*

Only x_1, x_3, x_5, x_6 may be the second largest key.

Analysis of Finding the 2nd

- Any algorithm that finds *secondLargest* must also find *max* before. (*n*-1)

- The *secondLargest* can only be in those which lose directly to *max*.

- On its path along which bubbling up to the root of tournament tree, *max* beat $\lceil \log n \rceil$ keys at most.

- Pick up secondLargest $(\lceil \log n \rceil - 1)$

- Totalcost: $n + \lceil \log n \rceil - 2$

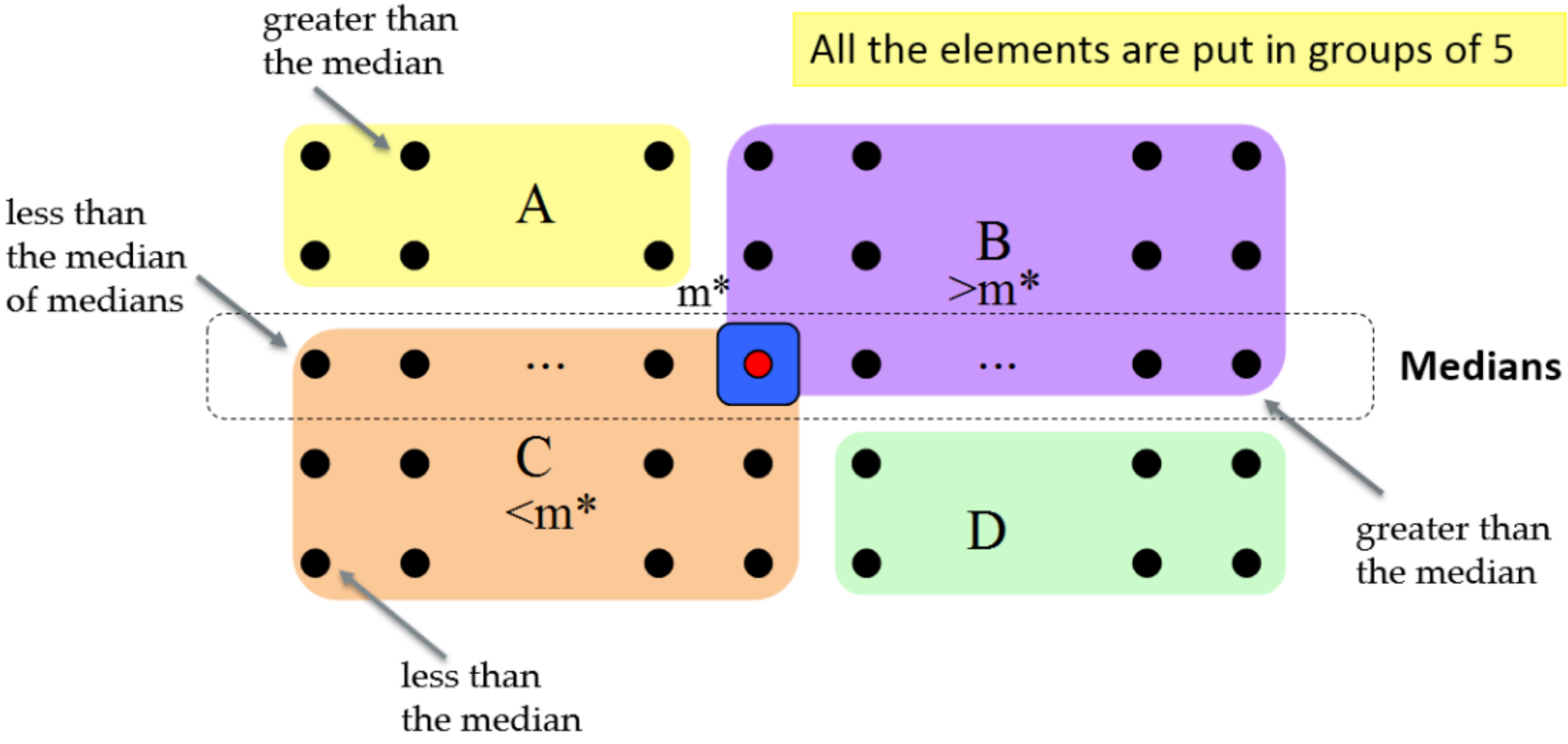
Lower Bound by Adversary

- **Theorem**
  - Any algorithm (that works by comparing keys) to find the second largest in a set of *n* keys must do at least $n + \lceil \log n \rceil - 2$ comparisons in the worst case.

- **Proof**
  - There is an adversary strategy that can force any algorithm that finds *secondLargest* to compare *max* to $\lceil \log n \rceil$ distinct keys.

.

Partition improved: the Strategy



Constructing the Partition

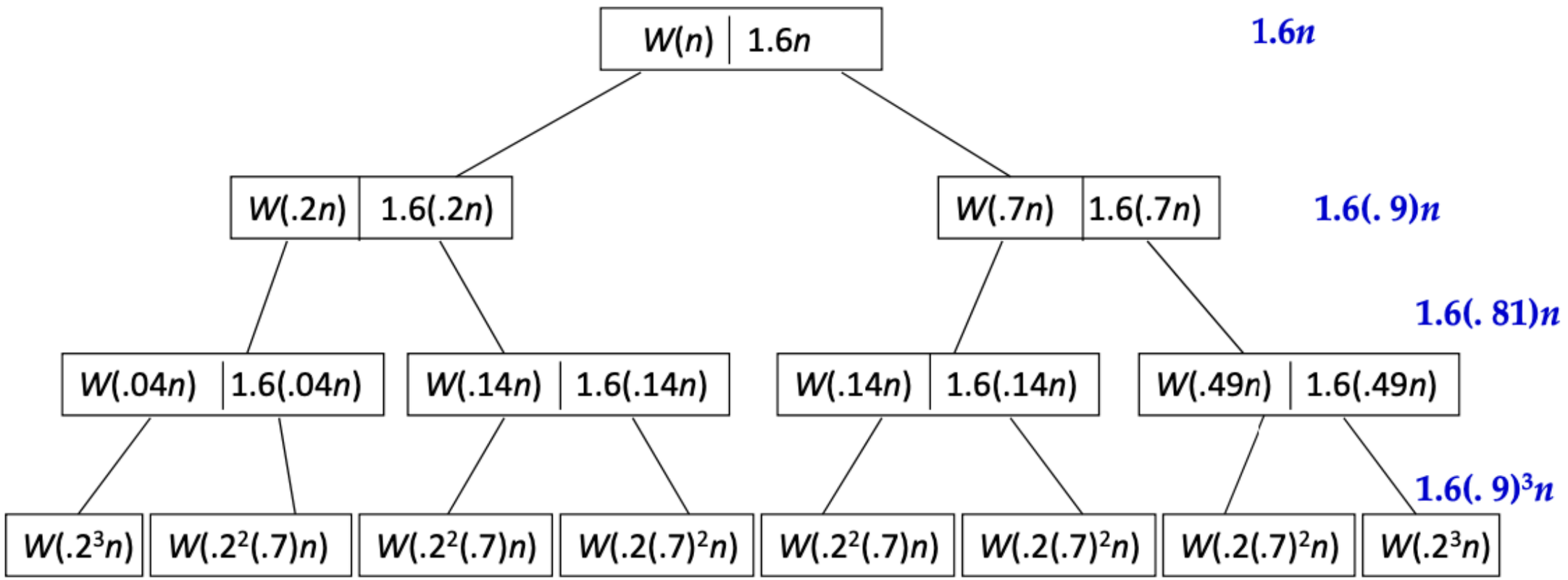
- **Find the *m**, the median of medians of all the groups of 5, as illustrated previously.**

- **Compare each key in sections A and D to *m**, and**

  - Let $S_1 = C \cup \{x \mid x \in A \cup D and x < m^*\}$

  - Let $S_2 = B \cup \{x \mid x \in A \cup D and x > m^*\}$ (*m** is to be used as the pivot for the partition)

Worst Case Complexity of Select



Note: Row sums is a decreasing geometric series, so

$$W(n) \in \Theta(n)$$

Relation to Median

- **Observation**
  - Any algorithm of selection must know the relation of every element to the *median*.

The adversary makes you wrong in either case

Crucial Comparison

- **A crucial comparison**
  - Establishing the relation of some *x* to the median.

- **Definition (for a comparison involving a key *x*)**
  - Crucial comparison for *x*: the first comparison where *x*>*y*, for some *y*=>median, or *x*<*y* for some *y*<=median
  - Non-crucial comparison: the comparison between *x* and *y* where *x*>median and *y*<median, or vise versa

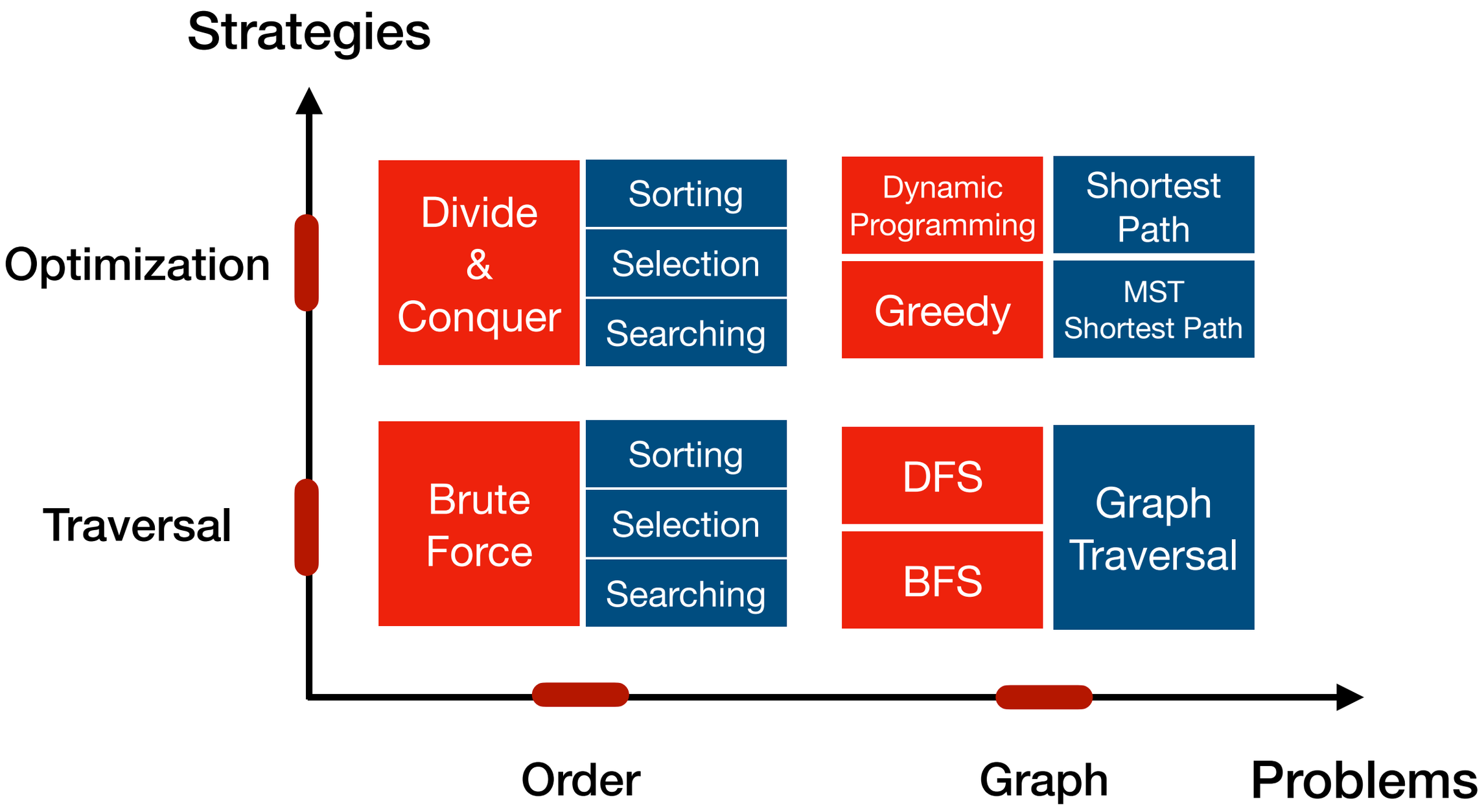
Adversary for Lower Bound

- **Status of the key during the running of the Algorithm:**
  - L: Has been assigned a value larger than median
  - S: Has been assigned a value smaller than median
  - N: Has not yet been in a comparison
- **Adversary rule:**

Comparands	Adversary's action
N,*N*	one *L*, the another *S*
L,*N* or *N*,*L*	change *N* to *S*
S,*N* or *N*,*S*	change *N* to *L*

(In all other cases, just keep consistency)

Syllabus


Strategies
Optimization
Traversal
Order
Graph
Problems
Divide & Conquer
Sorting
Selection
Searching
Dynamic Programming
Greedy
Shortest Path
MST
Shortest Path
Brute Force
Sorting
Selection
Searching
DFS
BFS
Graph Traversal

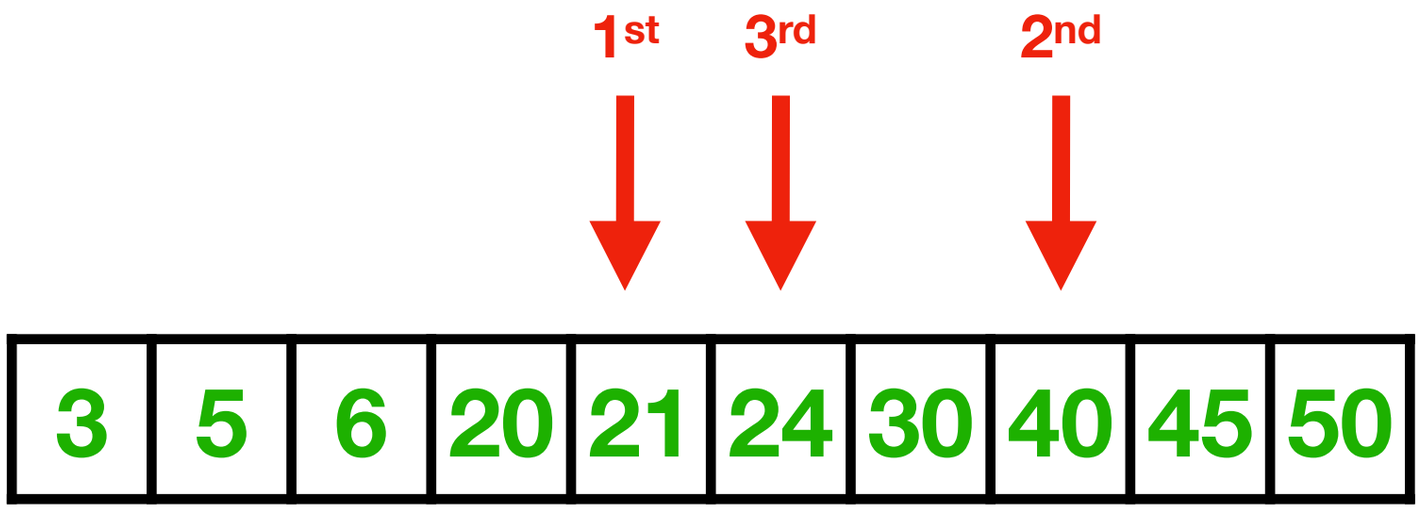
The Searching Problem

- **Essential of searching**
  - How to organize the data to enable efficient search
  - logn search
    - Each search cuts off half of the search space
      - How to organize the data to enable logn search
- **logn search techniques**
  - Warmup
    - Binary search over sorted sequences
  - Balanced Binary Search Tree (BST)
    - Red-black tree

Binary Search by Example

- **Binary search for “24”**
  - Divide the search space
  - Cut off half the space after each search

The sequence is already sorted

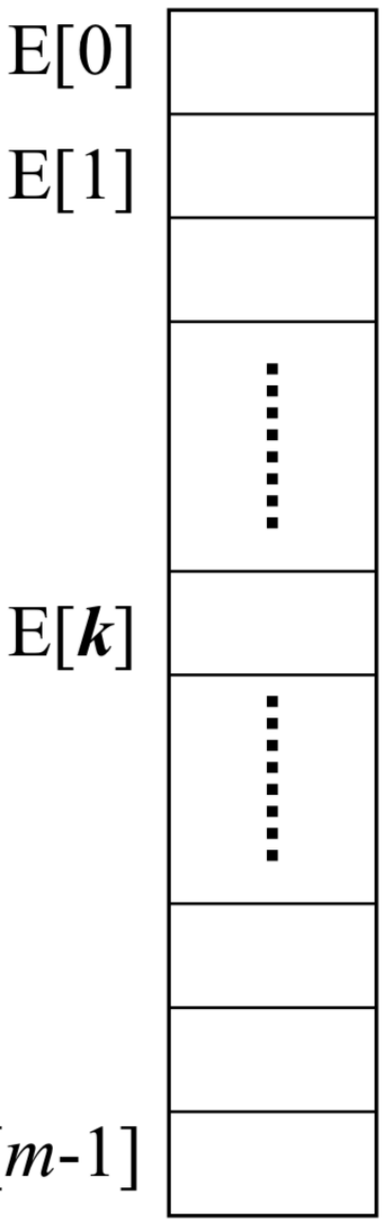


Cost for Searching

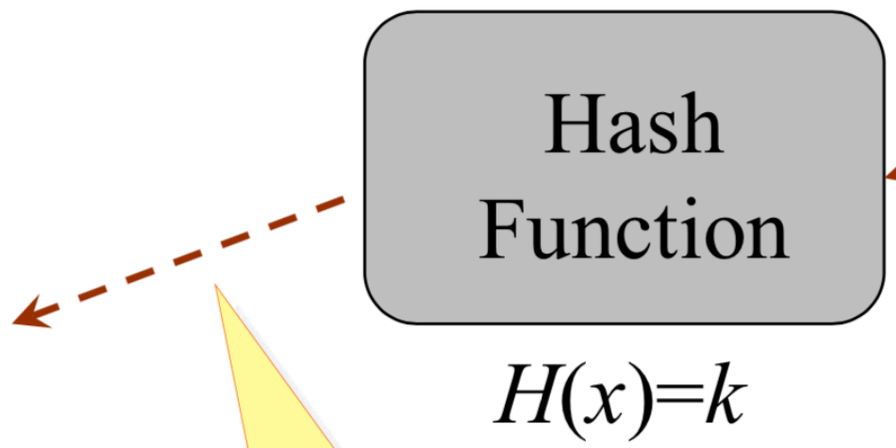
- **Brute force**
  - O(n)
- **Balanced BST**
  - O(logn)
- **Hashing - almost constant time**
  - O(1+α)
- **“Mission impossible”**
  - O(1)

Hashing: the Idea

Hash Table (in feasible size)

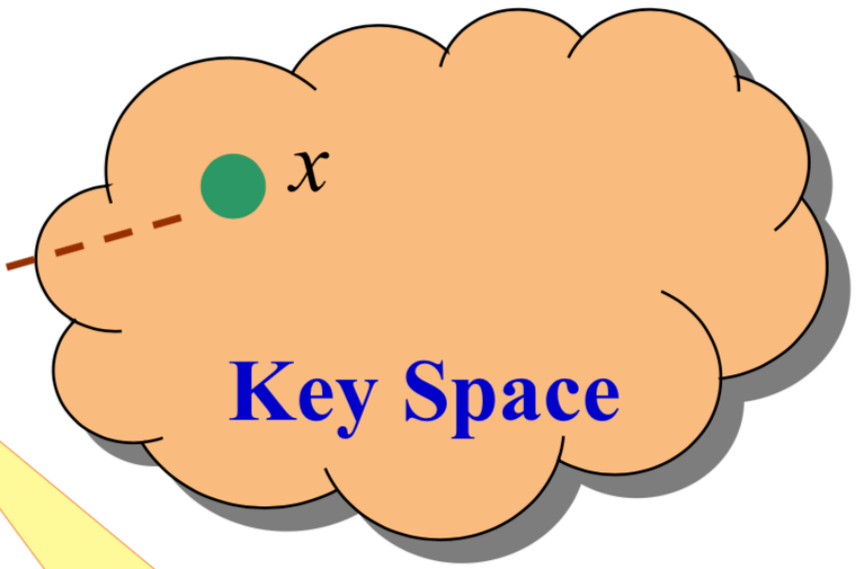


- ***Index distribution***
- ***Collision handling***



A calculated array index for the key

quite large, but only a small part is used in an application



Value of a specific key

Closed Address - Analysis

- **Assumption - simple uniform hashing**
  - For j = 0, 1, 2, …, m-1, the average length of the list at E[j] is n/m.
- **The average cost for an unsuccessful search**
  - Any key that is not in the table is equally likely to hash to any of the m address.
  - Total cost Θ(1+n/m)
    - The average cost to determine that the key is not in the list e[h(k)] is the cost to search to the end of the list, which is n/m.

Closed Address - Analysis

- **For successful search** (assuming that x_i is the ith element inserted into the table, i = 1, 2, …, n)
  - For each i, the probability of that x_i is searched is 1/n.
  - For a specific x_i, the number of elements examined in a successful search is t+1, where t is the number of elements inserted into the same list as x_i, after x_i has been inserted

$$\frac{1}{n}\sum_{i=1}^{n}(1+t)$$

- **How to compute t?**
  - Consider the construction process of the hash table.

Closed Address - Analysis

- **For successful search:** (assuming that x_i is the i^{th} element inserted into the table, i = 1, 2, …, n)
  - For each i, the probability of that x_i is searched is 1/n.
  - For a specific x_i, the number of elements examined in a successful search is t+1, where t is the number of elements inserted into the same list as x_i, after x_i has been inserted. And for any j, the probability of that x_j is inserted into the same list of x_i is 1/m. So, the cost is:

Cost for computing hashing

$$1 + \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n}\frac{1}{m}\right)$$

Expected number of elements in front of the searched one in the same linked list.

Closed Address: Analysis

- **The average cost of a successful search:**
  - Define $\alpha = n/m$ as load factor,
  - The average cost of a successful search is:

$$\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) = 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) = 1+\frac{1}{nm}\sum_{i=1}^{n-1} i$$

$$= 1+\frac{n-1}{2m} = 1+\frac{\alpha}{2}-\frac{\alpha}{2n} = \Theta(1+\alpha)$$

Number of elements in front of the searched one in the same linked list

Collision Handling: Open Address

- **All elements are stored in the hash table**
  - No linked list is used
  - The load factor α cannot be larger than 1
- **Collision is settled by “rehashing”**
  - A function is used to get a new hashing address for each collided address
    - The hash table slots are probed successively, until a valid location is found.
- **The probe sequence can be seen as a permutation of (0,1,2,…,m-1)**

Analysis for Open Address hashing

- **The average number of probes in an unsuccessful search is at most 1/(1-α) (α=n/m<1)**

  - Assuming uniform hashing

The probability of the first probed position being occupied is $\frac{n}{m}$, and that of the $j^{th}(j > 1)$ position occupied is $\frac{n-j+1}{m-j+1}$. So the probability of the number of probes no less than i will be:

$$\frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdot \cdots \cdot \frac{n-i+2}{m-i+2} \leq (\frac{n}{m})^{i-1} = \alpha^{i-1}$$

The the average number of probe is: $\sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha}$

See [CLRS] p.1199, C.25

Analysis for Open Address Hashing

- **The average cost of probes in an successful search is at most** $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$ **(α=n/m<1)**
  - Assuming uniform hashing

To search for the $(i+1)^{th}$ inserted element in the table, the cost is the same as that for inserting it when there are just i elements in the table. At that time, $\alpha = \frac{i}{m}$. So the cost is $\frac{1}{1-\frac{i}{m}} = \frac{m}{m-i}$.
So the average cost for a successful search is:

For your reference:
Half full: 1.387;
90% full: 2.559

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} = \frac{1}{\alpha}\sum_{i=m-n+1}^{m}\frac{1}{i}$$

$$\leq \frac{1}{\alpha}\int_{m-n}^{m}\frac{dx}{x} = \frac{1}{\alpha}\ln\frac{m}{m-n} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

Array Doubling

- **Cost for search in a hash table is Θ(1+α)**
  - If we can keep α constant, the cost will be Θ(1)
- **What if the hash table is more and more loaded?**
  - Space allocation techniques such as array doubling may be needed
- **The problem of "unusually expensive" individual operation**

Looking at the Memory Allocation

```
hashingInsert(HASHTABLE H, ITEM x)
    integer size = 0, num = 0;
    if size = 0 then allocate a block of size 1; size = 1;
    if num = size then
        allocate a block of size 2size;
        move all item into new table;
        size = 2size;
    insert x into the table;
    num = num + 1;
return
```

Insertion with expansion: cost size

Elementary insertion: cost 1

Worst-case Analysis

- **For n execution of insertion operations**
  - A bad analysis: the worst case for one insertion is the case when expansion is required, up to n
  - So, the worst case cost is in $O(n^2)$

- **Note the expansion is required during the i^{th} operation only if $i=2^k$, and the cost of the i^{th} operation**

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is exactly the power of 2} \\ 1 & otherwise \end{cases}$$

So the total cost is: $\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j < n + 2n = 3n$

Amortized Analysis - Why?

- **Unusually expensive operations**
  - E.g., Insert-with-array-doubling
- **Relation between expensive and usually operations**
  - Each piece of the doubling cost corresponds to some previous insert

Amortized Analysis - How?

- **Amortized equation:**
  - amortized cost = actual cost + accounting cost
- **Design goal for accounting cost**
  - In any legal sequence of operations, the sum of the accounting costs is nonnegative
  - The amortized cost of each operation is fairly regular, in spite of the wide fluctuate possible for the actual cost of individual operations

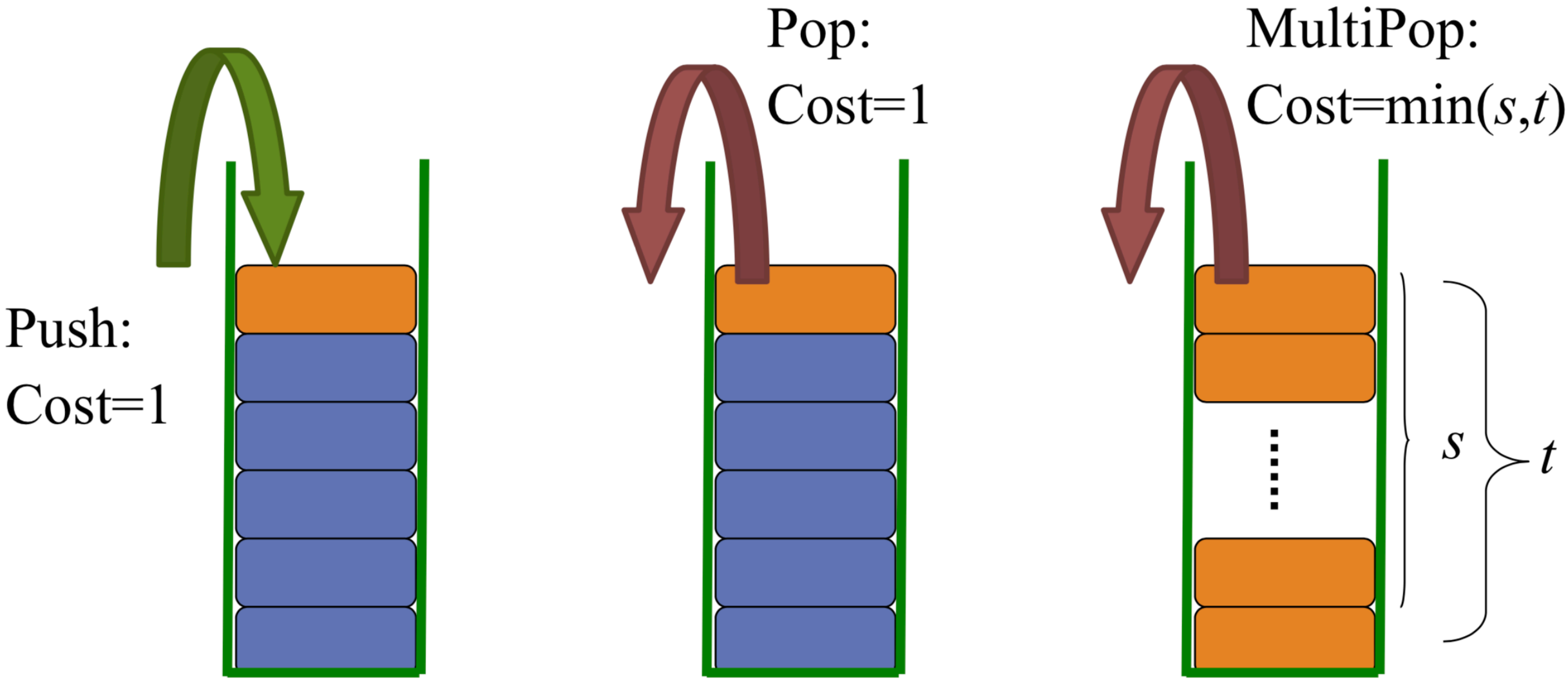
Array Doubling

- **Why non-negative accounting cost?**
  - For any possible sequence of operations?

	Amortized	Actual	Accounting
Insert (normal)	3	1	2
Insert (doubling)	3	k+1	-k+2

K is the number of elements upon doubling

Multi-pop Stack



Amortized cost: push:2; pop, multipop: 0

Binary Counter

0	0 0 0 0 0 0 0 0	0
1	0 0 0 0 0 0 0 1	1
2	0 0 0 0 0 0 1 0	3
3	0 0 0 0 0 0 1 1	4
4	0 0 0 0 0 1 0 0	7
5	0 0 0 0 0 1 0 1	8
6	0 0 0 0 0 1 1 0	10
7	0 0 0 0 0 1 1 1	11
8	0 0 0 0 1 0 0 0	15
9	0 0 0 0 1 0 0 1	16
10	0 0 0 0 1 0 1 0	18
11	0 0 0 0 1 0 1 1	19
12	0 0 0 0 1 1 0 0	22
13	0 0 0 0 1 1 0 1	23
14	0 0 0 0 1 1 1 0	25
15	0 0 0 0 1 1 1 1	26
16	0 0 0 1 0 0 0 0	31

Cost measure: bit flip

amortized cost:

set 1: 2

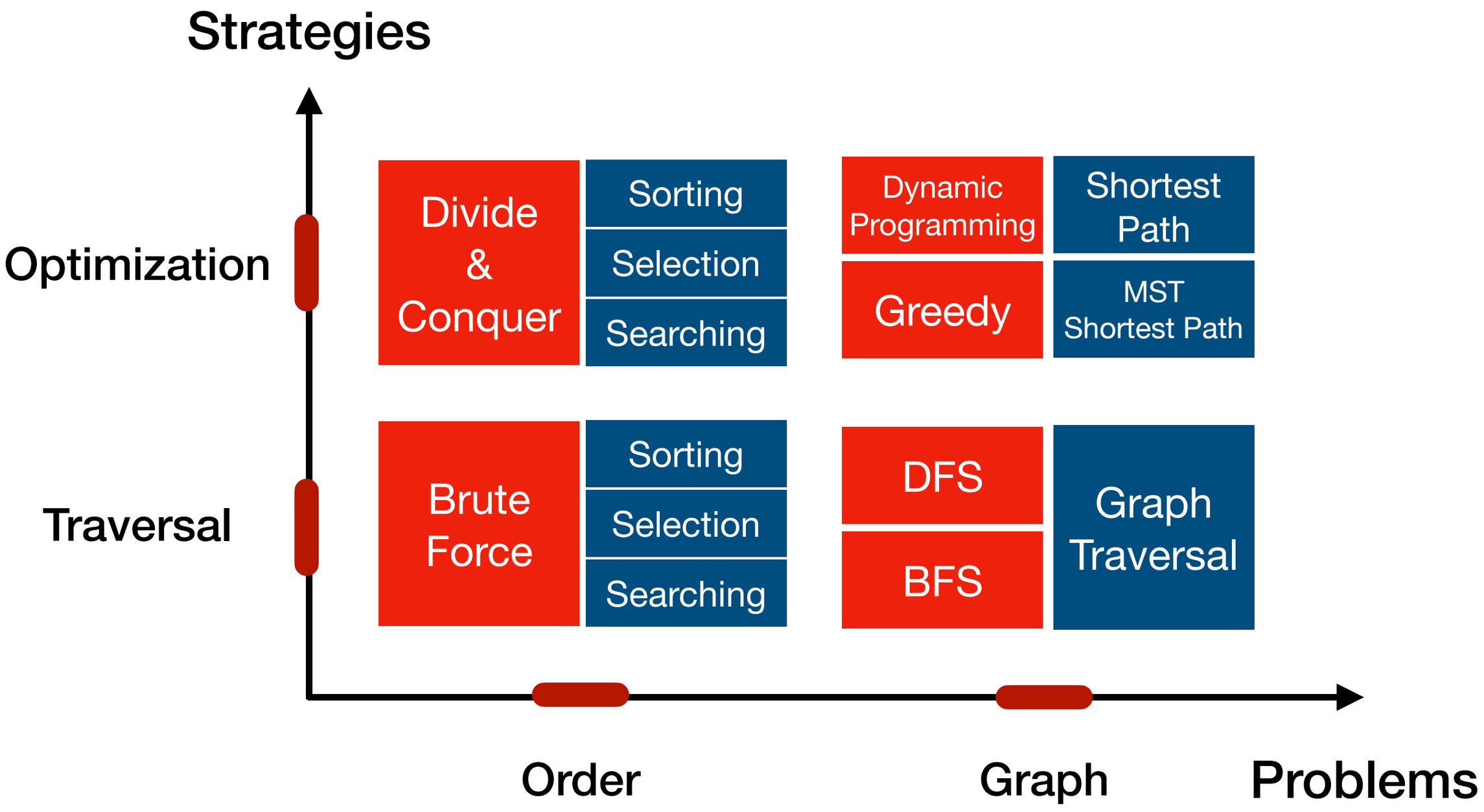
set 0: 0

Binary Counter

- **Why non-negative accounting cost?**
  - For any possible sequence of operations?

	Amortized	Actual	Accounting
Set 1	2	1	1
Set 0	0	1	-1

Syllabus


Strategies
Optimization
Traversal
Order
Graph
Problems
Divide & Conquer
Sorting
Selection
Searching
Dynamic Programming
Greedy
Shortest Path
MST
Shortest Path
Brute Force
Sorting
Selection
Searching
DFS
BFS
Graph Traversal

Course Contents

optimization problems

greedy

DP

BFS
DFS

DFS
(directed)

DFS
(undirect
ed)

MST

Path
(single
source)

Path
(all-pair)

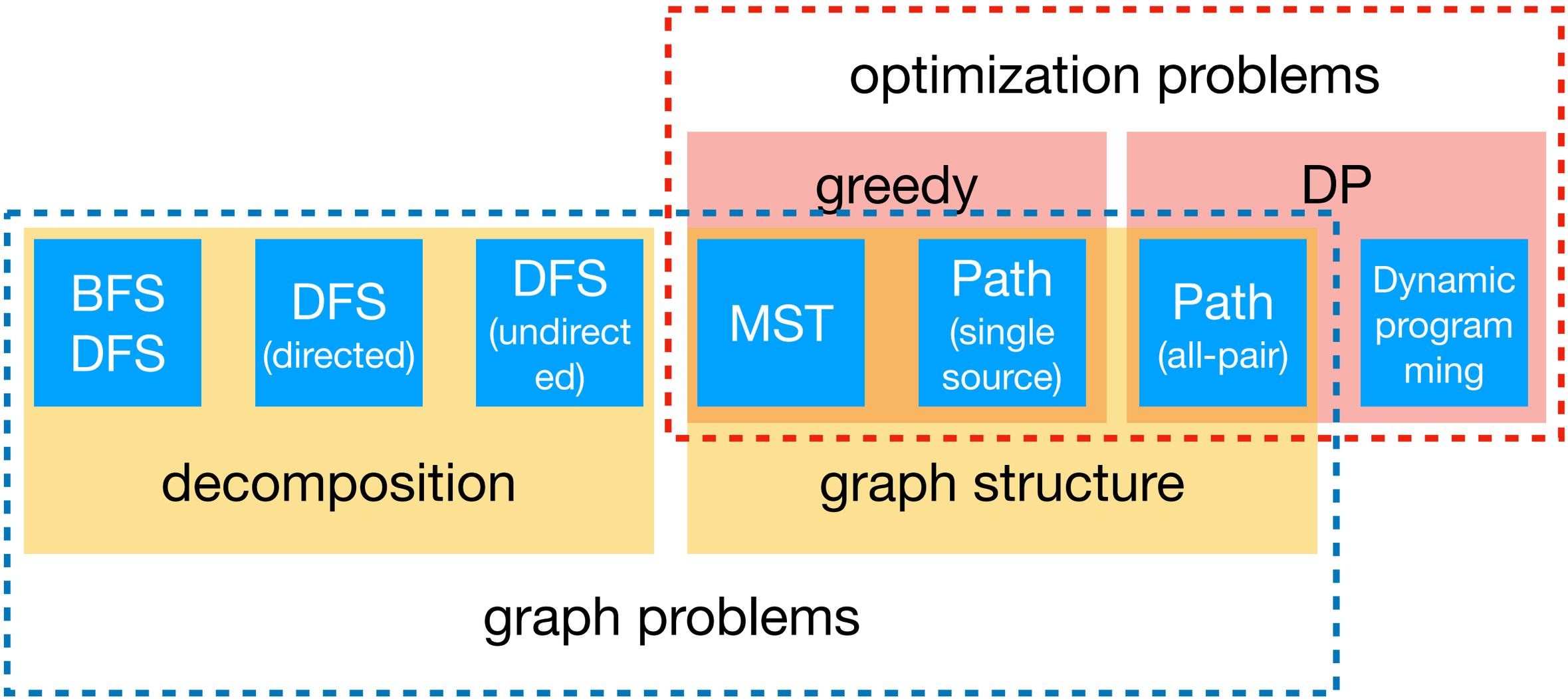
Dynamic
program
ming

decomposition

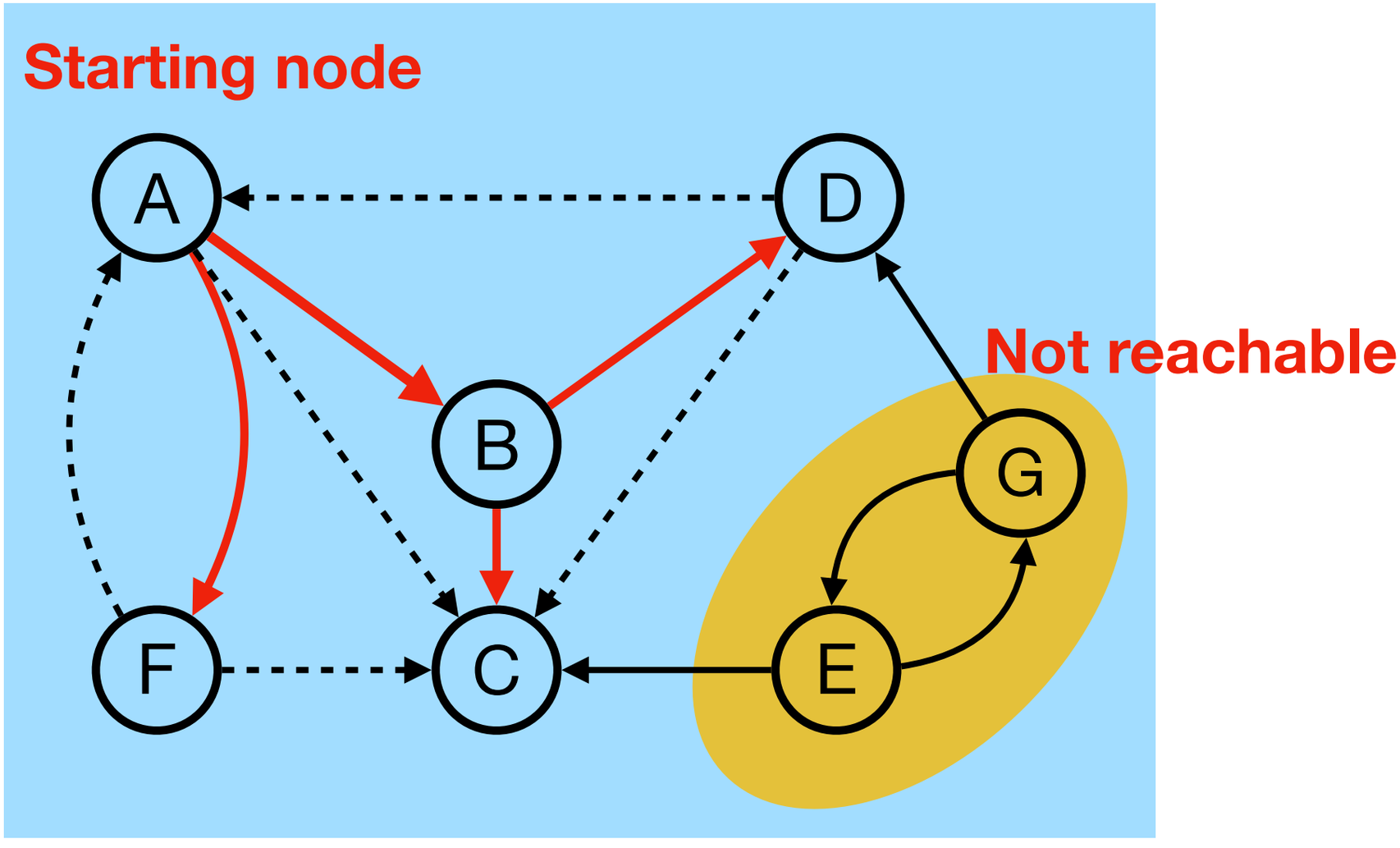
graph structure

graph problems

Course Contents

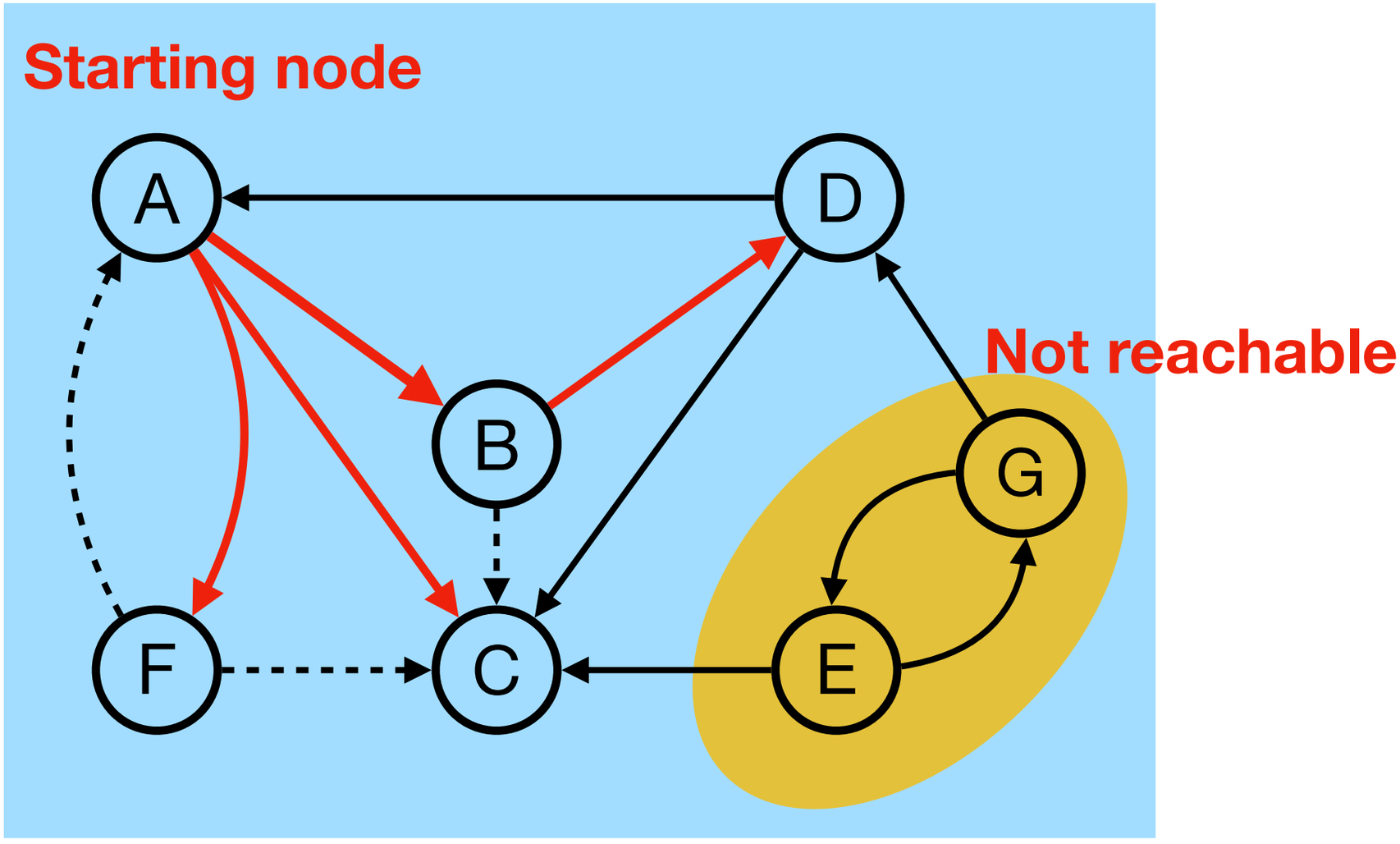


Graph Traversal



Depth-First Search

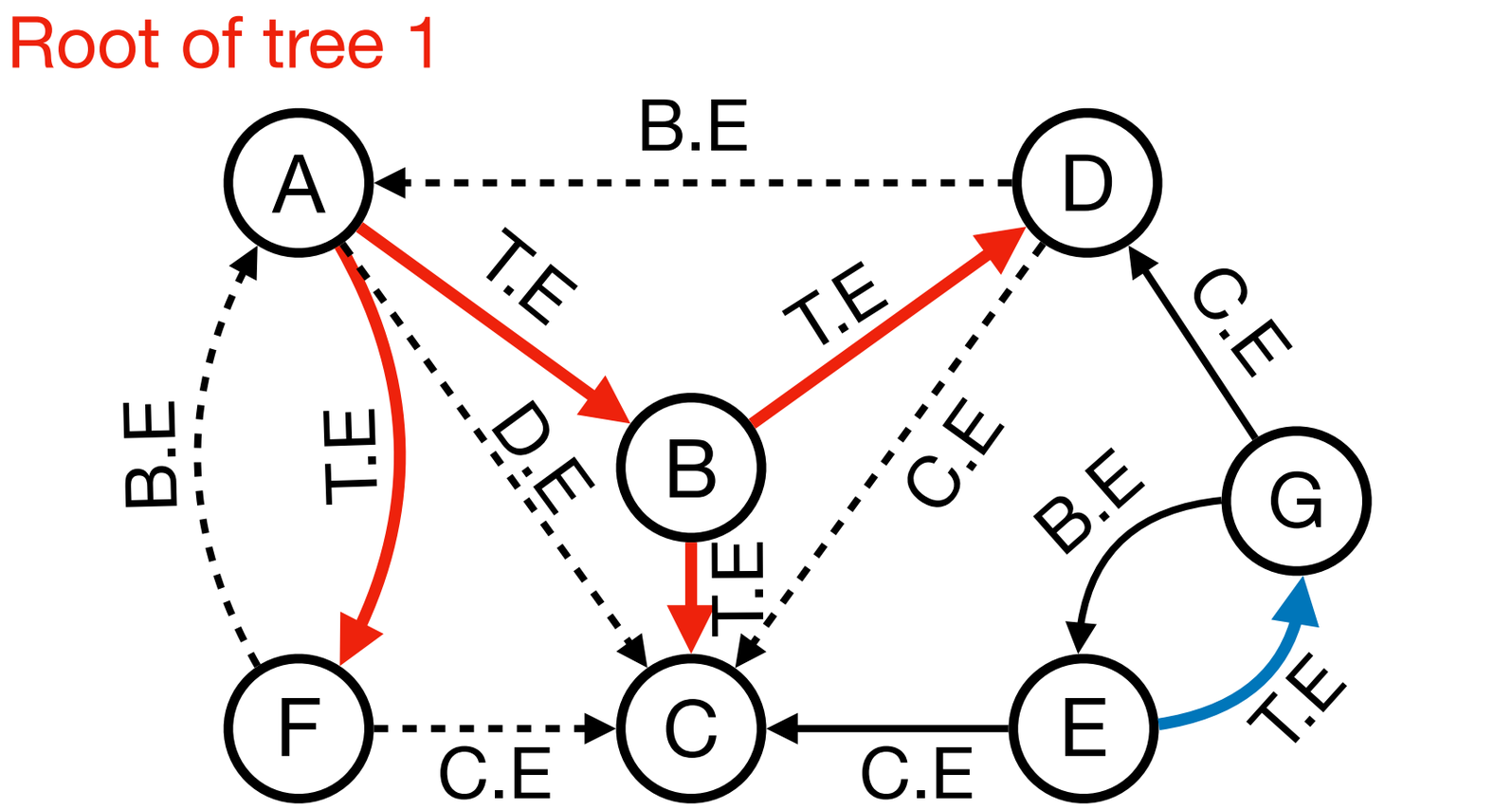
Graph Traversal



Breadth-First Search

Depth-first Search Trees

DFS forest={(DFS tree1), (DFS tree2)}



T.E: tree edge
B.E: back edge
D.E: descendant edge
C.E: cross edge

A finished vertex is never revisited, such as C.

Depth-First Search — Generalized Skeleton

- Input: Array adjVertices for graph G
- Output: Return value depends on application
- int dfsSweep(intList[], adjVertices, int n, …)

```
int ans;
<Allocate color array and initialize to white>
for each vertex v of G, in some order
    if(color[v]==white)
        int vAns=dfs(adjVertices, color, v, …);
        <Process vAns>
    // continue loop
return ans;
```

Depth-First Search — Generalized Skeleton

```
int dfs(intList[] adjVertices, int[] color, int v, …)
    int w;
    intList remAdj;
    int ans;
    color[v]=gray;
    <Preorder processing of vertex v>
    remAdj=adjVertices[v];
    while(remAdj != nil)
        w=first(remAdj);
        if(color[w]==white)
            <Exploratory processing for tree edge vw>
            int wAns=dfs(adjVertices, color, w, …)
            <Backtrack processing for tree edge vw, using wAns>
        else
            <Checking for nontree edge vw>
        remAdj=rest(remAdj);
    <Postorder processing of vertex v, including final computation of ans>
    color[v]=black;
    return ans;
```

If partial search is used for a application, tests for termination may be inserted here.

Specialized for connected components:
- Parameter added
- Preorder processing inserted - cc[v] =ccNum

DFS v.s. BFS Search

- **Processing opportunities for a node**
  - Depth-first: 2
    - At discovering
    - At finishing
  - Breadth-first: only 1, when de-queued
  - At the second processing opportunity for the DFS, the algorithm can make use of information about the descendants of the current node.

Time Relation on Changing Color

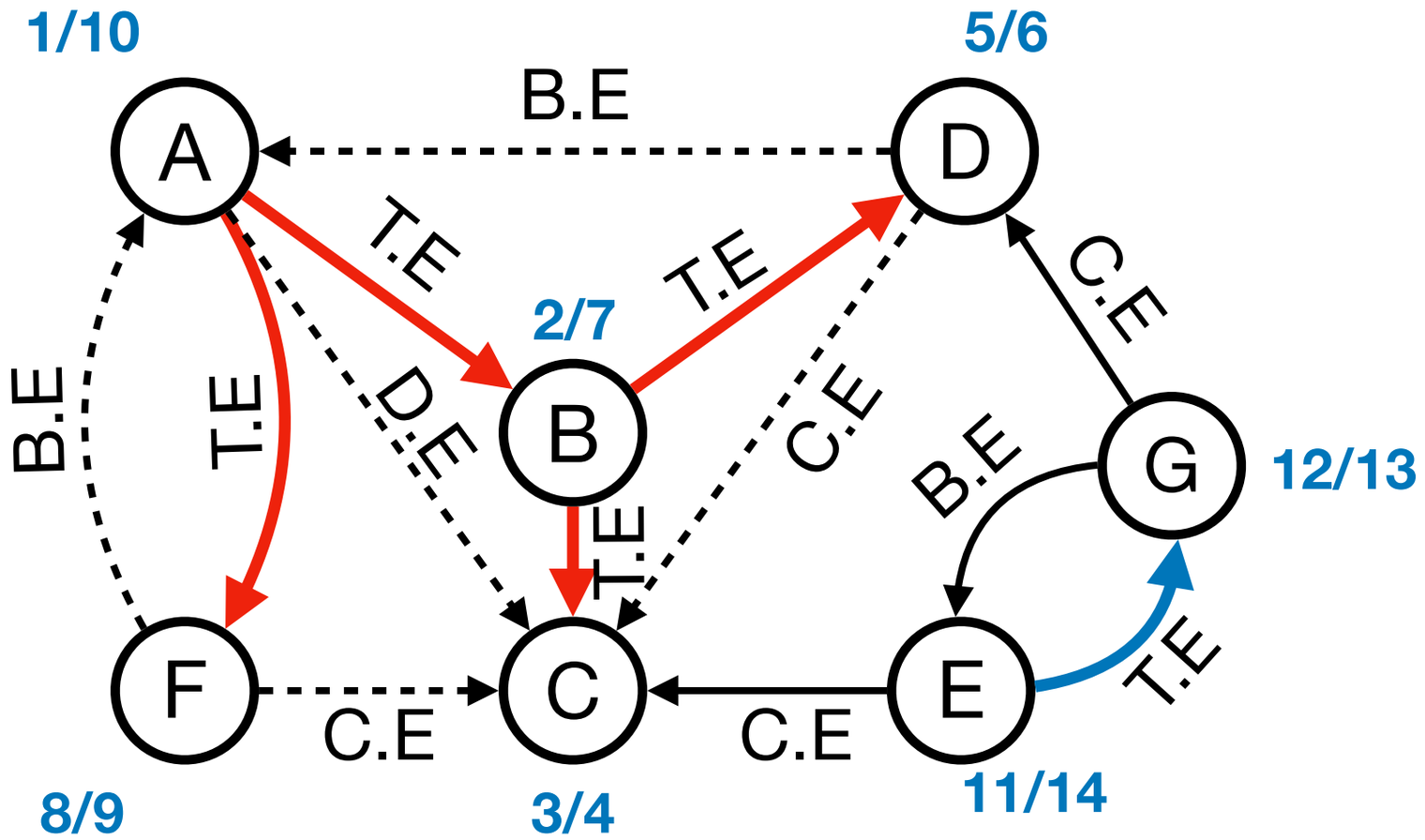
- **Keeping the order in which vertices are encountered for the first or last time**

  - A global integer time: 0 as the initial value, incremented with each color changing for *any* vertex, and the final value is 2*n*
  - Array *discoverTime*: the *i* th element records the time vertex v_i turns into gray
  - Array *finishTime*: the *i* th element records the time vertex v_i turns into black
  - The active interval for vertex *v,* denoted as *active*(*v*), is the duration while *v* is gray, that is:

.

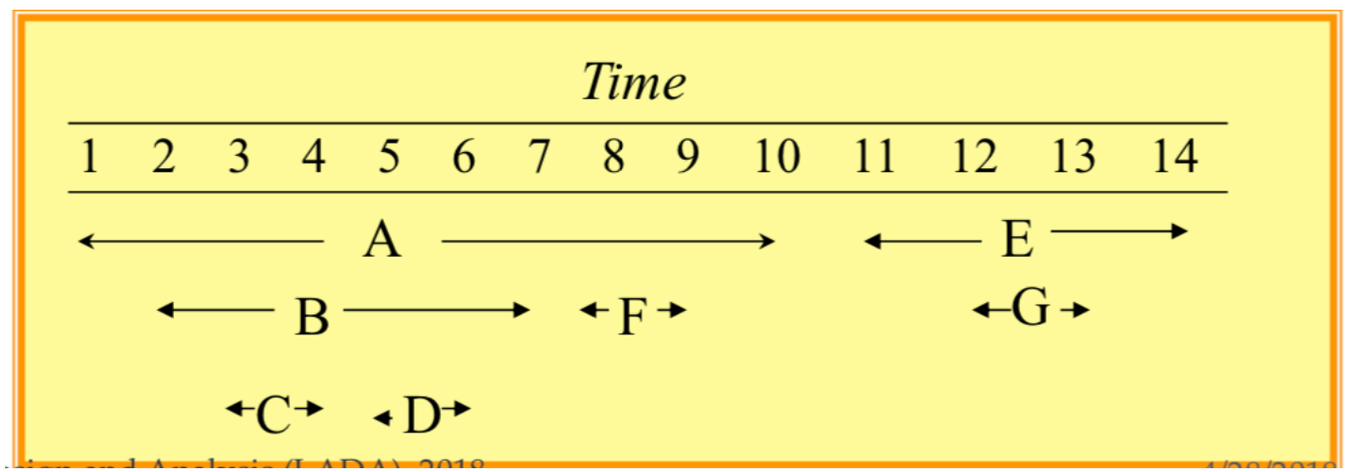
discoverTime[v], …, finishTime[v]

Active Interval



T.E: tree edge
B.E: back edge
D.E: descendant edge
C.E: cross edge

The relations are summarized in the next frame

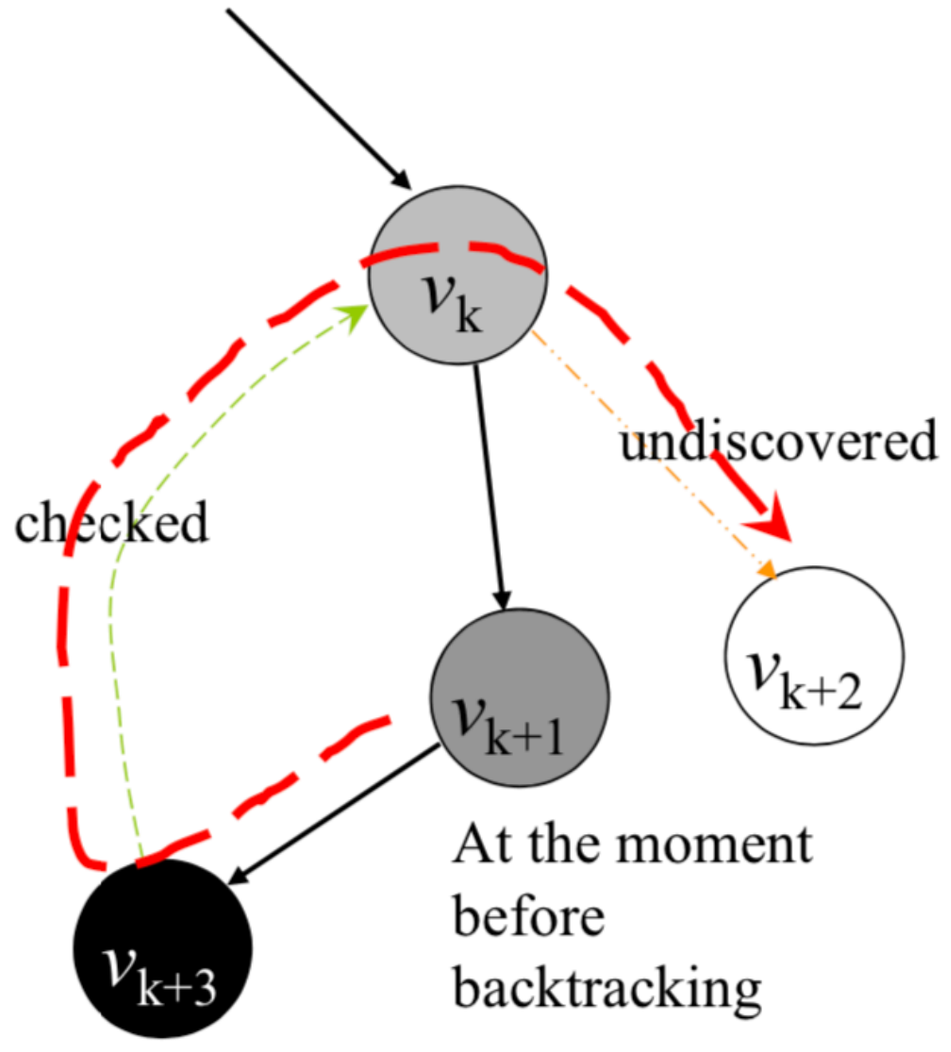


Ancestor and Descendant

- That *w* is a descendant of *v* in the DFS forest means that there is a direct path from *v* to *w* in some DFS tree.

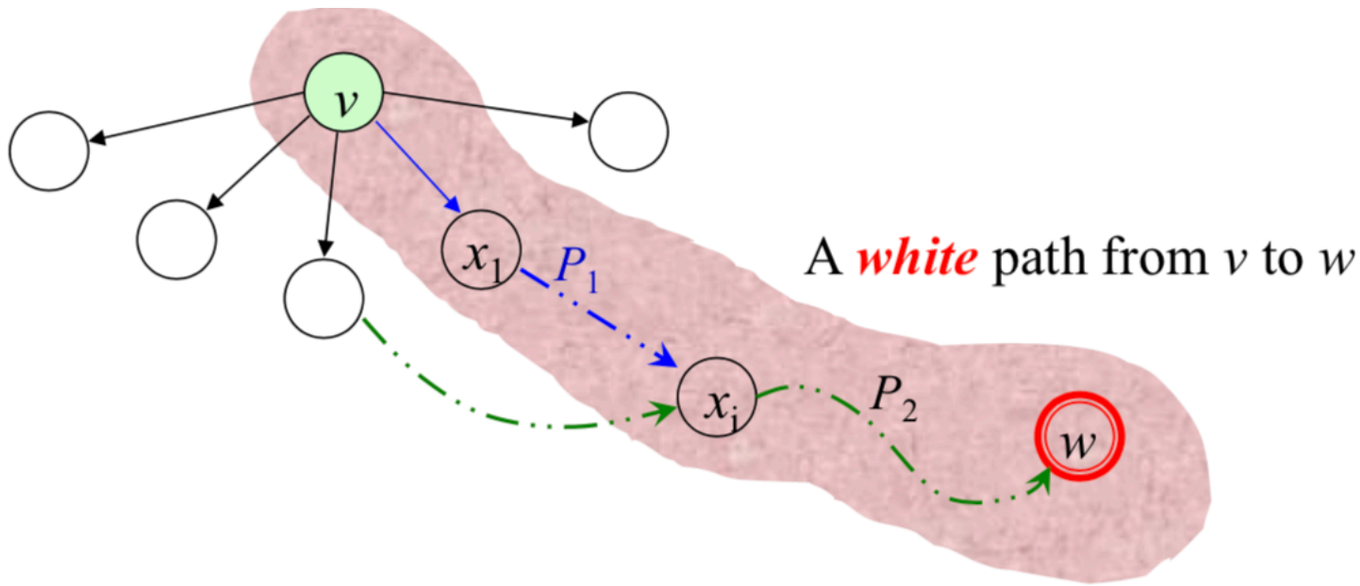
- The path is also a path in *G*.

- However, if there is a direct path from *v* to *w* in G, is *w* necessarily a descendant of *v* in *the* DFS forest?



DFS Tree Path

- [White Path Theorem] w is a descendant of v in a DFS tree iff. At the time v is discovered(just to be changing color into gray), there is a path in G from v to w consisting entirely of white vertices.

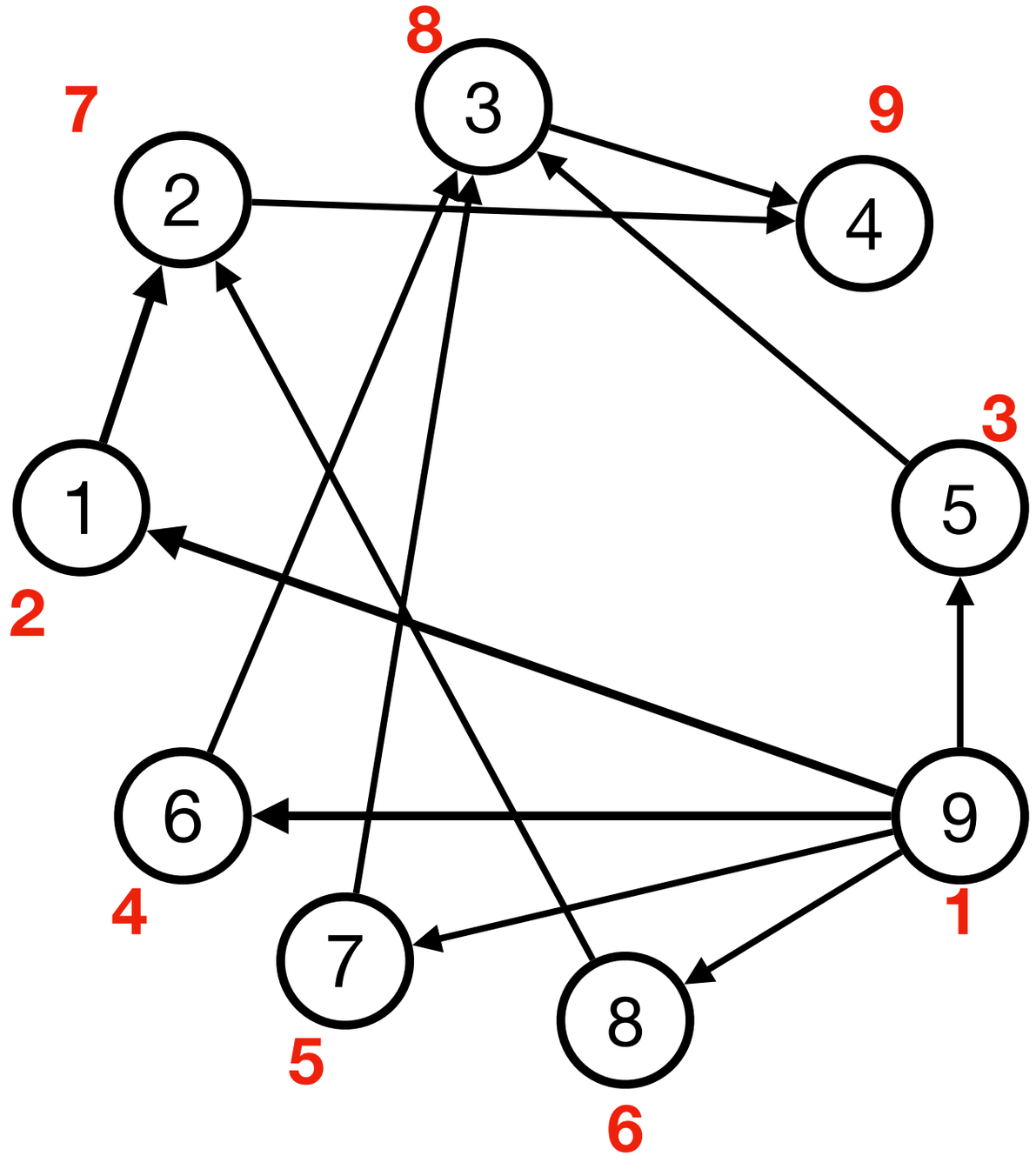


Proof of White Path Theorem

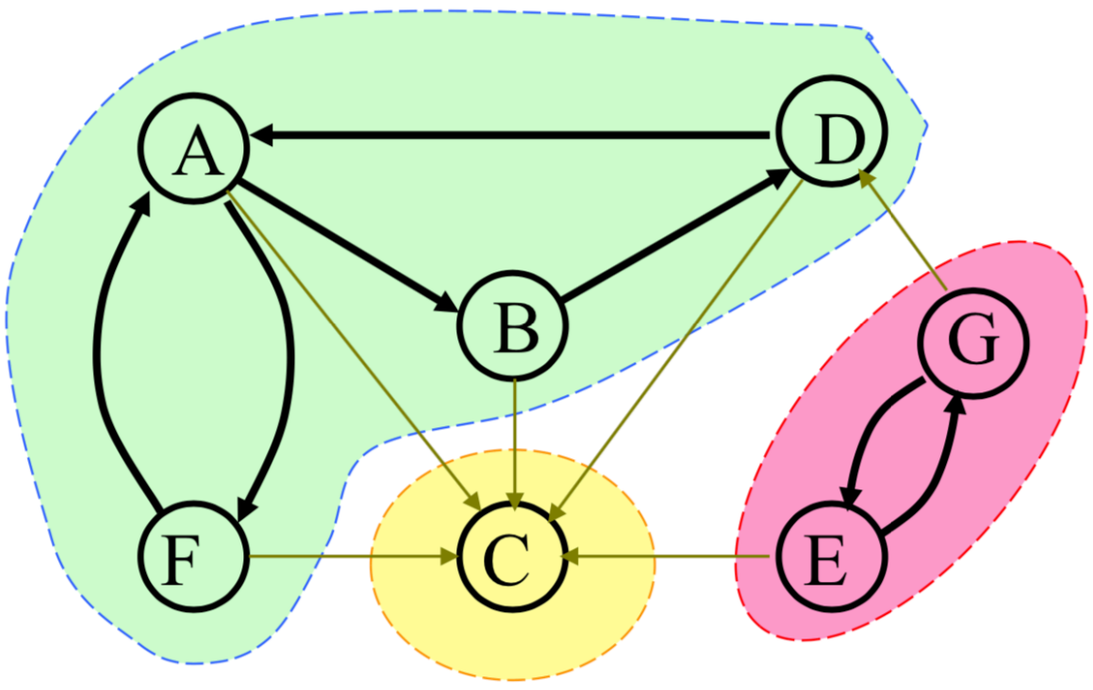
- **Proof**
  - => all the vertices in the path are descendants of v.
  - <= by induction on the length k of a white path from v to w.
    - When k=0, v=w.
    - For k>0, let *P*=(v, x_1,x_2...x_k=w). There must be some vertex on *P* which is discovered during the active interval of v, e.g. x_1, Let x_i is earliest discovered among them. Divide *P* into P_1 from v to x_i, and P_2 from x_i to w. P_2 is a white path with length less than *k*, so, by inductive hypothesis, w is a descendant of x_i. Note: *active*(x_i)⊆*active*(v), so x_i is a descendant of v. By transitivity, w is a descendant of v.

Topological Order for G=(V, E)

- **Topological number**
  - An assignment of distinct integer 1,2,…,n to the vertices of V
  - For every vw∈E, the topological number of v is less than that of w.
- **Reverse topological order**
  - Defined similarly (“greater than”)

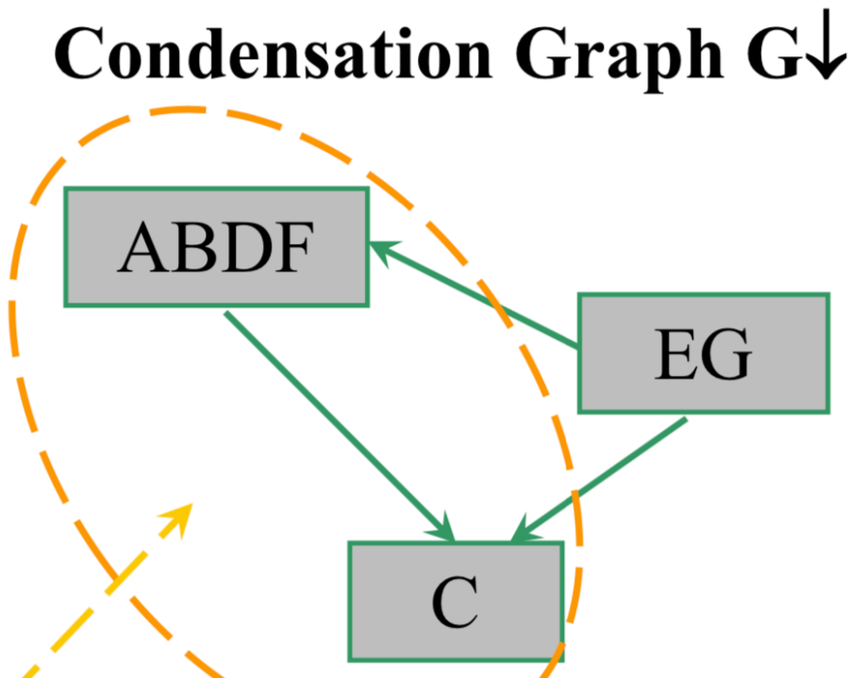


SCC: Strongly Connected Component



Graph G

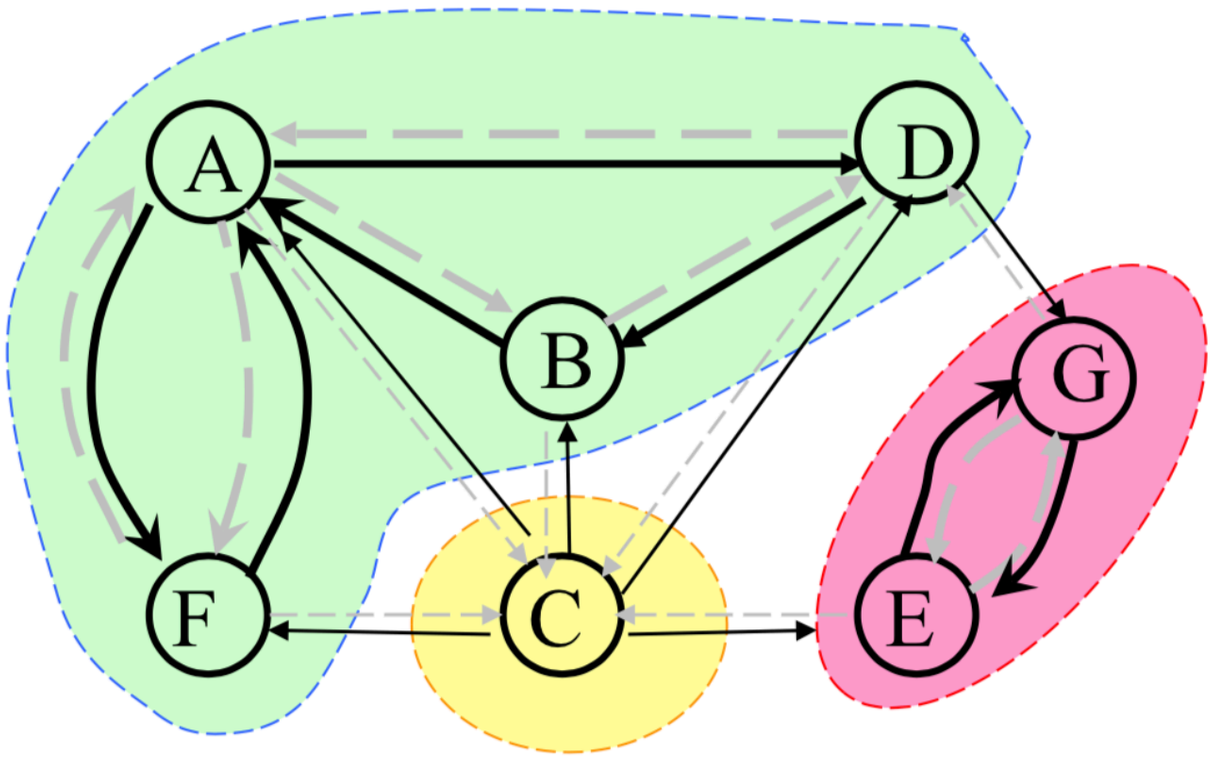
3 Strongly Connected Components



It's acyclic, ***Why?***

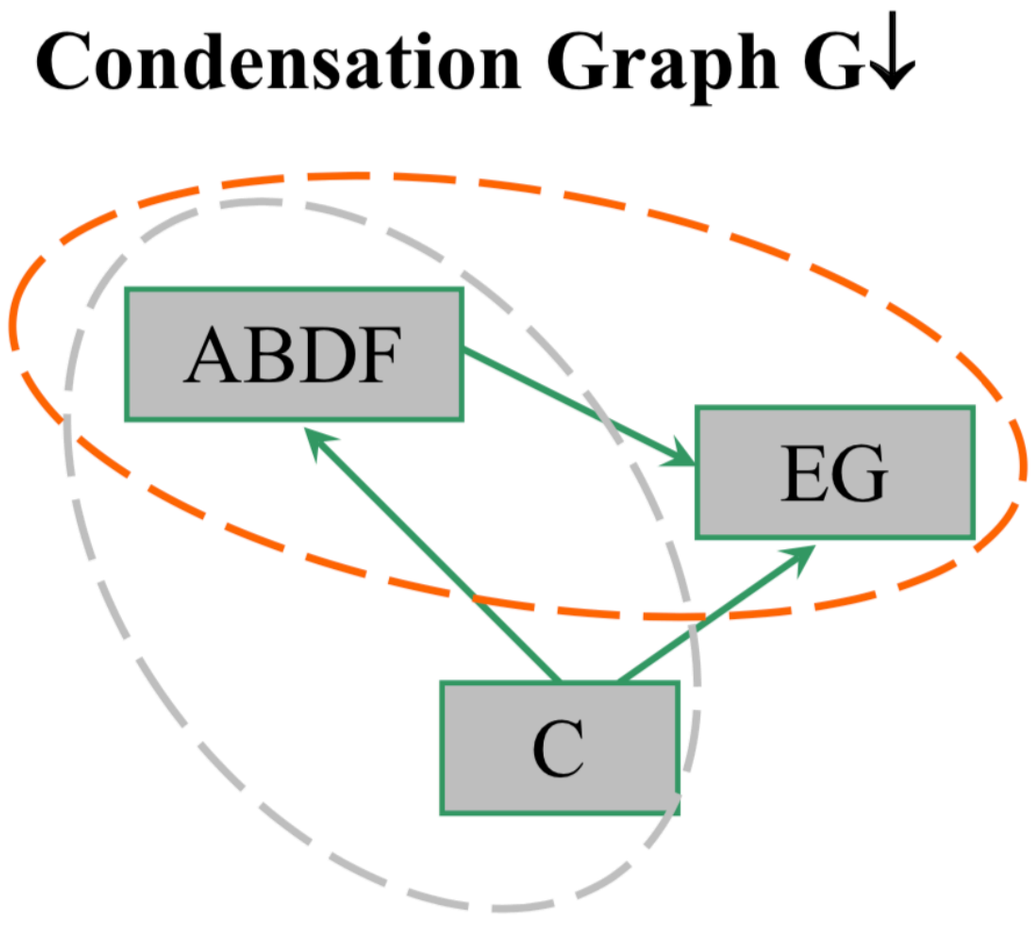
Note: two SCC in one DFS tree

Transpose Graph



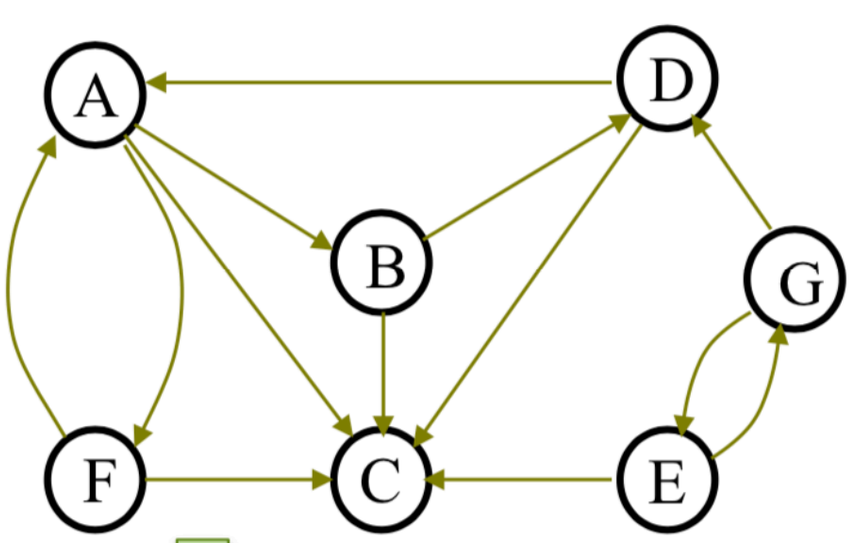
Tranpose Graph G^T

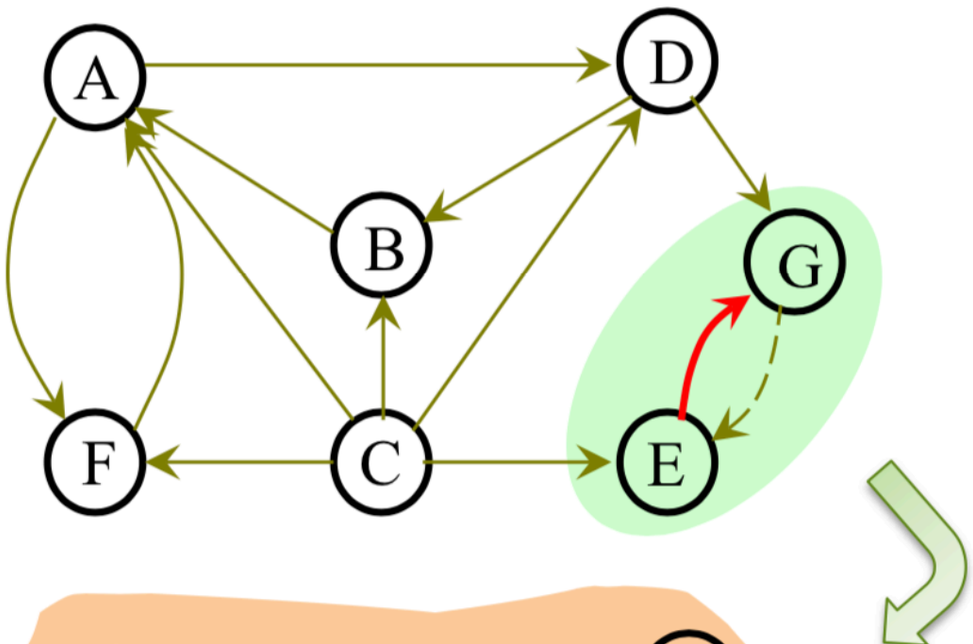
Connected Components unchanged according to vertices



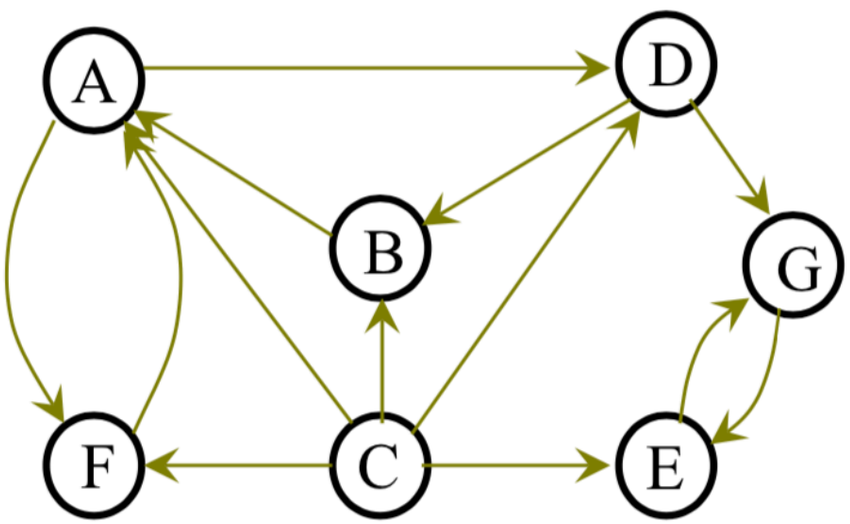
But, DFS tree changed

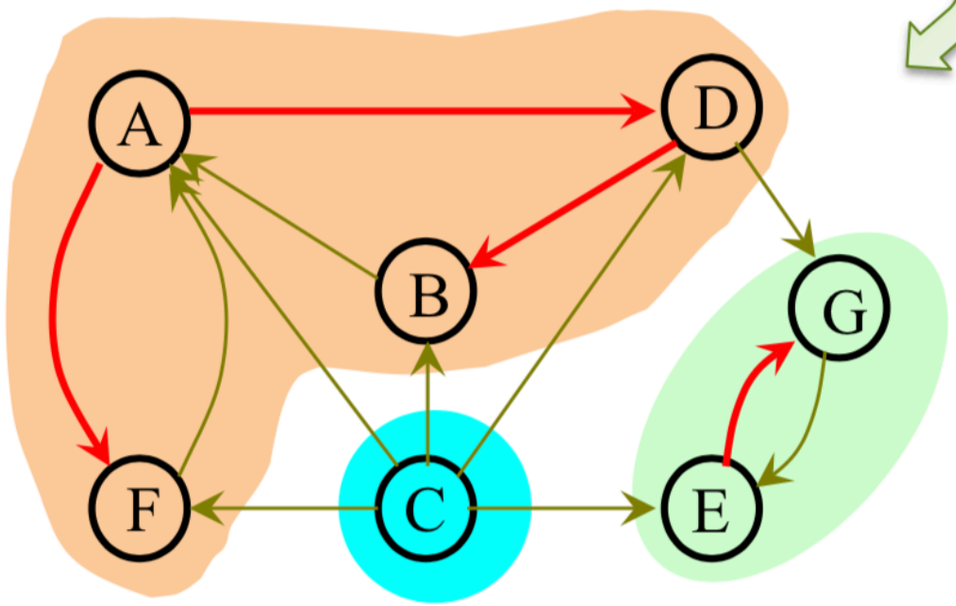
SCC - An Example





transposed





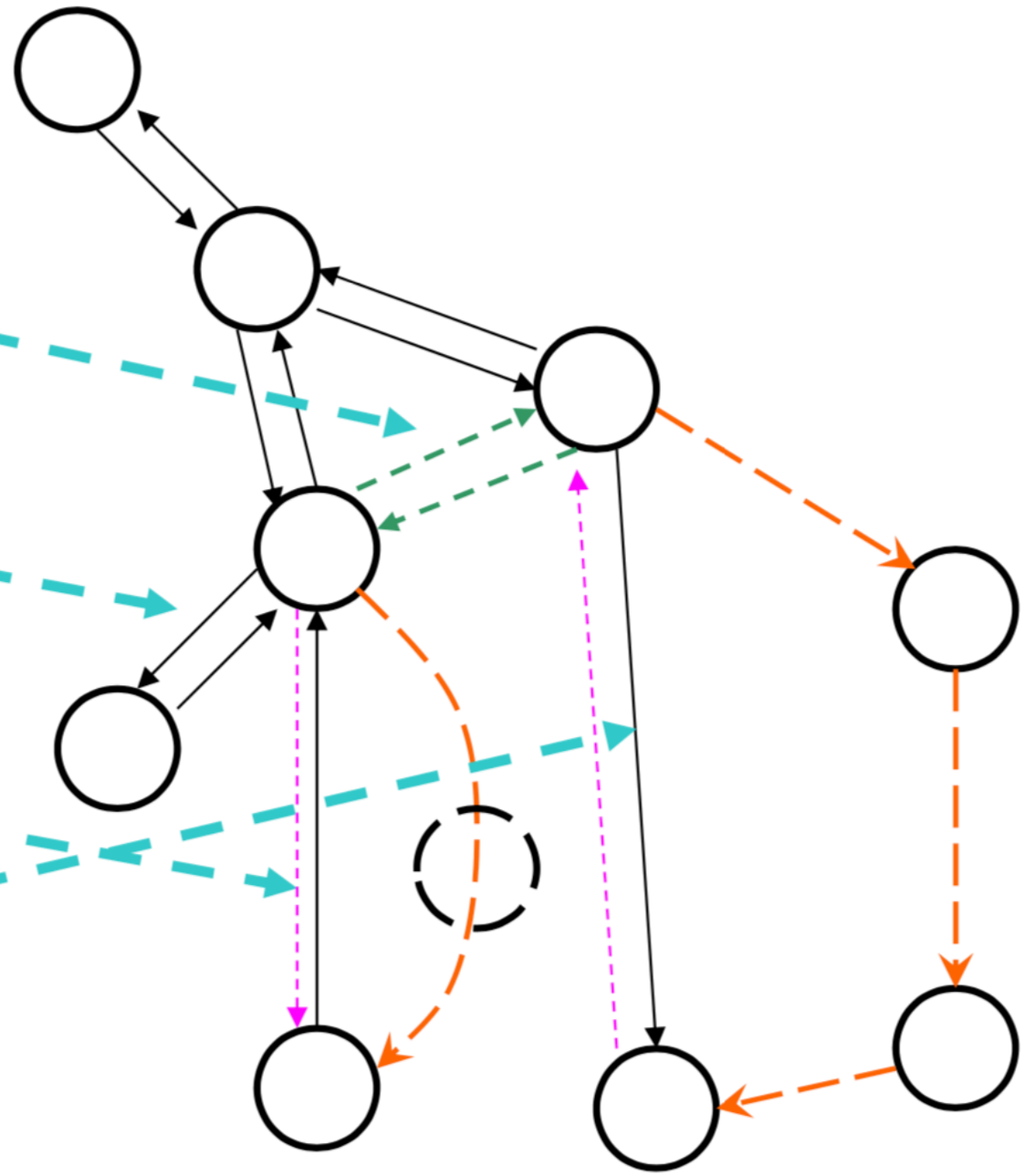
C D B F A G E ← - - - - push/pop

What is Different for "Undirected"

- **Characteristics of undirected graph traversal**
  - One edge may be traversed for two times in opposite directions.
- **For an undirected graph, DFS provides an orientation for each of its edges**
  - Oriented in the direction in which they are first encountered.

Edges in DFS

- **Cross edge**
  - Not existing
- **Back edge**
  - Back to the direct parent: second encounter
  - Otherwise: **first encounter**
- **Forward edge**
  - Always second encounter, ***and first time as back edge***



Modifications to the DFS Skeleton

- All the second encounter are bypassed.

- So, the only substantial modification is for the possible back edges leading to an ancestor, but not direct parent.

- We need know the parent, that is, the direct ancestor, for the vertex to be processed.

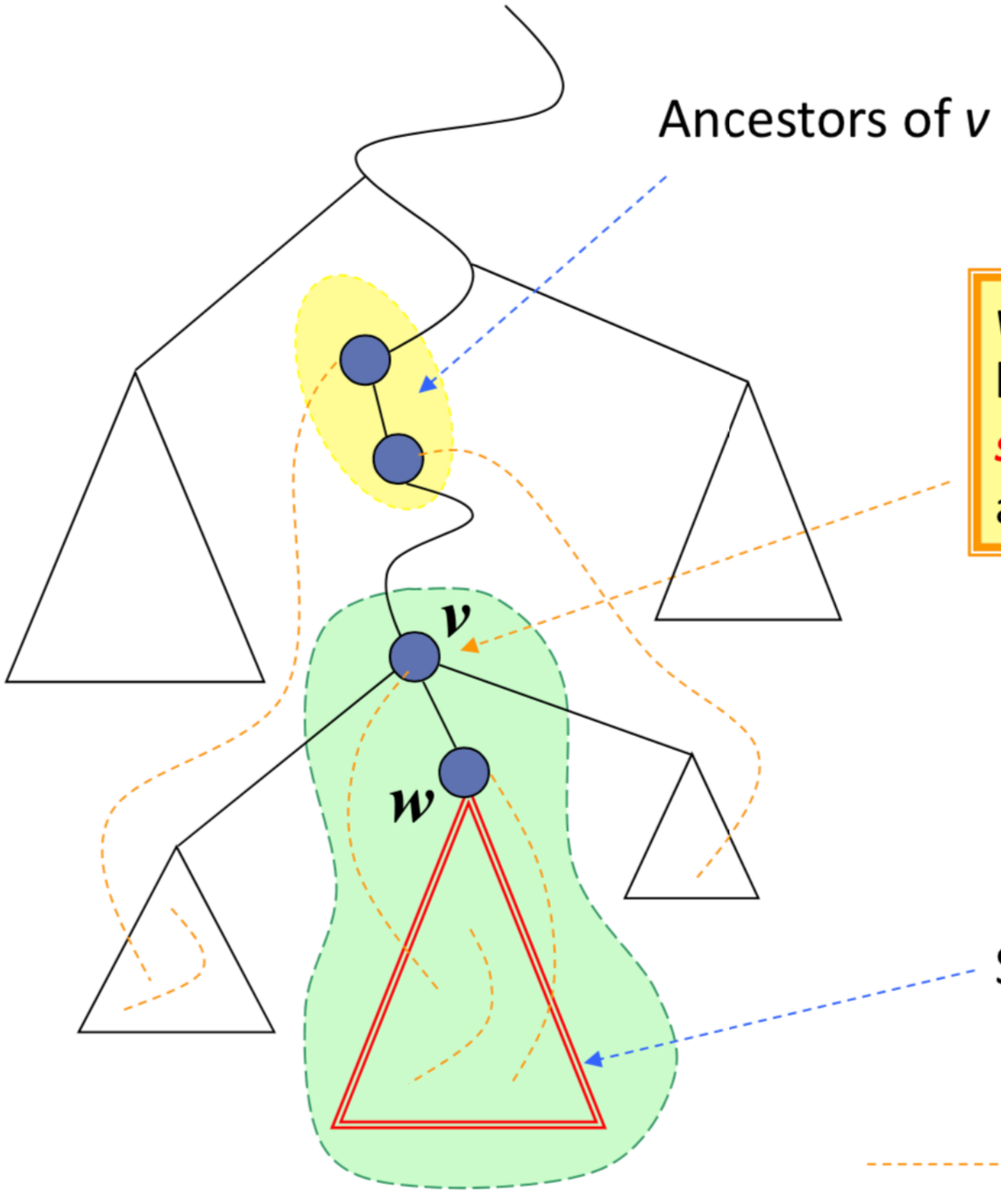
Biconnected Graph

- **Being connected**
  - Tree: acyclic, least (cost) connected
  - Node/edge connected: fault-tolerant connection
- **Articulation point (2-node connected)**
  - v is an articulation point if deleting v leads to disconnection
- **Bridge (2-edge connected)**
  - uv is a bridge if deleting uv leads to disconnection

Definition Transformation

- **“Short definition”**
  - Deleting v leads to disconnection
- **“Long definition”**
  - If there exist nodes w and x, such that v is in every path from w to x (w and x are vertices different from v)
- **“Long definition” or “DFS definition”**
  - No back edges linking any vertex in some w-rooted subtree and any ancestor of v

Articulation Point Algorithm



v is an articulation point iff no back edges linking any vertex in ***some*** *w*-rooted subtree and any ancestor of *v.*

If *v* is the articulation point farthest away from the root on *the* branch, then one bicomponent is detected.

Updating the value of back

- **v first discovered**
  - back=discoverTime(v)
- **Trying to explore, but a back edge vw from v encountered**
  - back=min(back, discoverTime(w))
- **Backtracking from w to v**
  - back=min(back, wback)

The back value of v is the smallest discover time a back edge “sees” from any subtree of v.

Keeping the Track of Backing

- **Tracking data**
  - For each vertex v, a local variable back is used to store the required information, as the value of discoverTime of some vertex.

- **Testing for bicomponent**
  - At backtracking from w to v, the condition implying a bicomponent is:
    - wBack ≥ discoverTime(v)

    (where wback is the returned back value for w)

When back is no less than the discover time of v, there is at least one subtree of v connected to other part of the graph only by v.

Other Traversal Problems

- **Orientation of an undirected graph**
  - Give each edge a direction
  - Satisfying pre-specified constraints
    - E.g., the “in-degree of each vertex is at least 1”
- **Possible or not?**
  - If possible, how to?
- **As for “in-degree ≥ 1”**
  - Orientation possible iff. the graph has at least a circle
    - Find the end point of some back edge
    - A second DFS from this end point

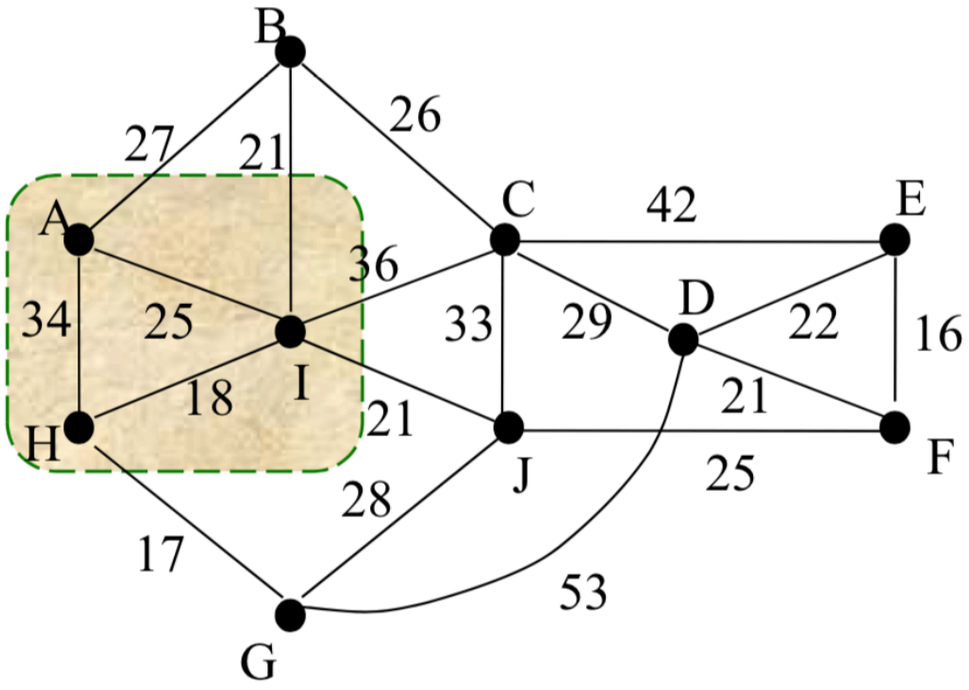
Greedy Strategy

- Expanding the partial solution step by step

- In each step, a selection is made from a set of candidates. The choice made must be:
  - [Feasible] it has to satisfy the problem's constraints
  - [Locally optimal] it has to be the best local choice among all feasible choices on the step
  - [Irrevocable] the choice cannot be revoked in subsequent steps

```
set greedy(set candidate)
    set S=∅;
    while not solution(S) and    candidate≠∅
        select locally optimizing x from candidate;
        candidate=candidate-{x};
        if feasible(x) then S=S∪{x};
    if solution(S) then return S
            else return ("no solution")
```

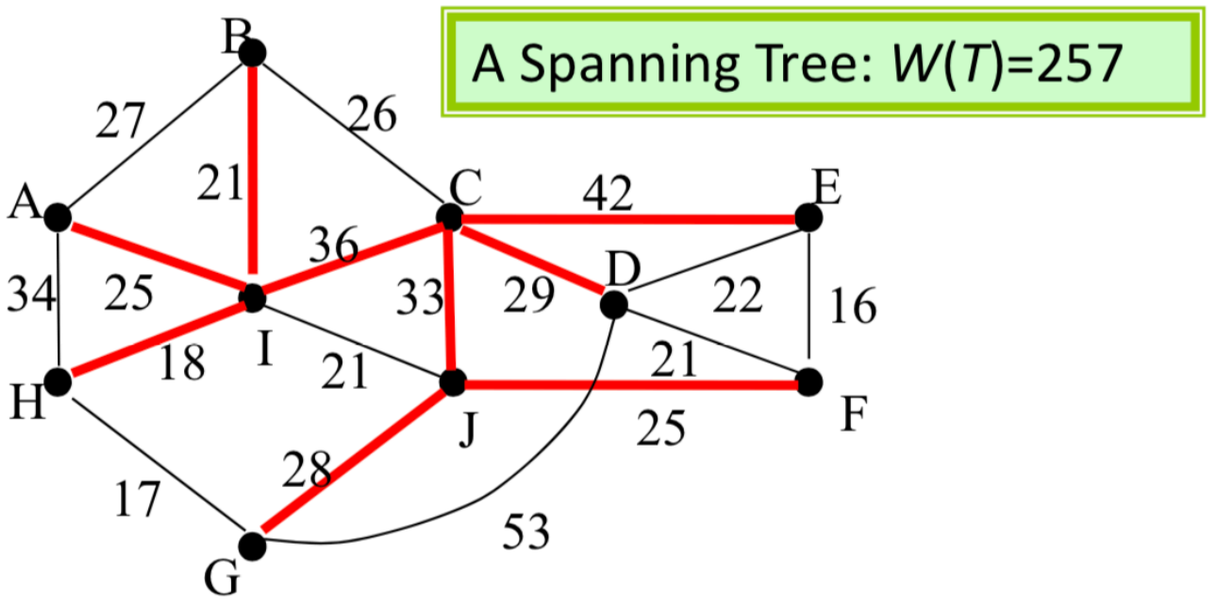
Weighted Graph and MST

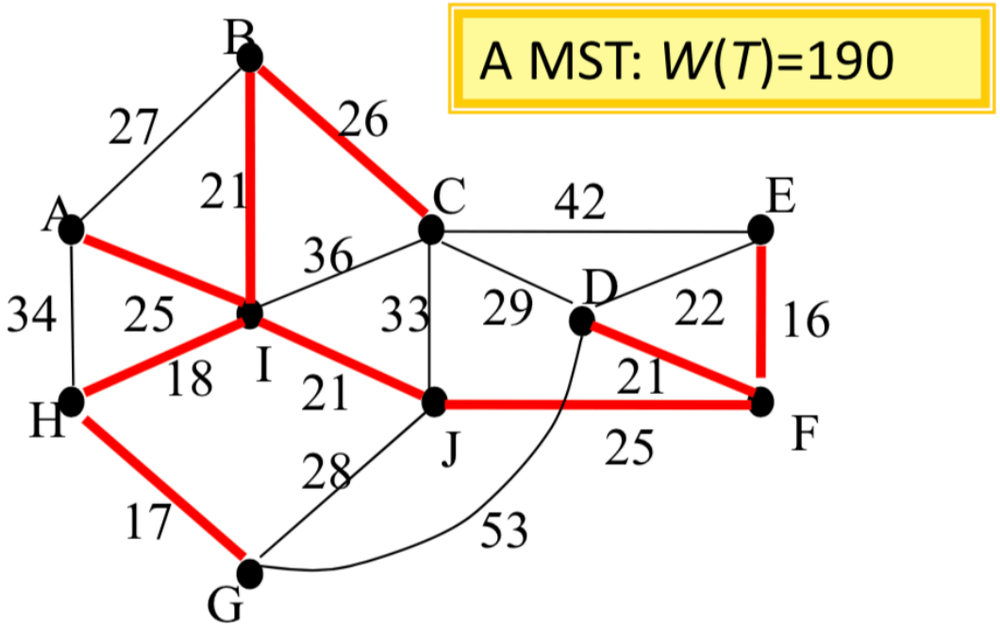


A weighted graph

The nearest neighbor of vertex ***I*** is ***H***

The nearest neighbor of shaded subset of vertex is ***G***

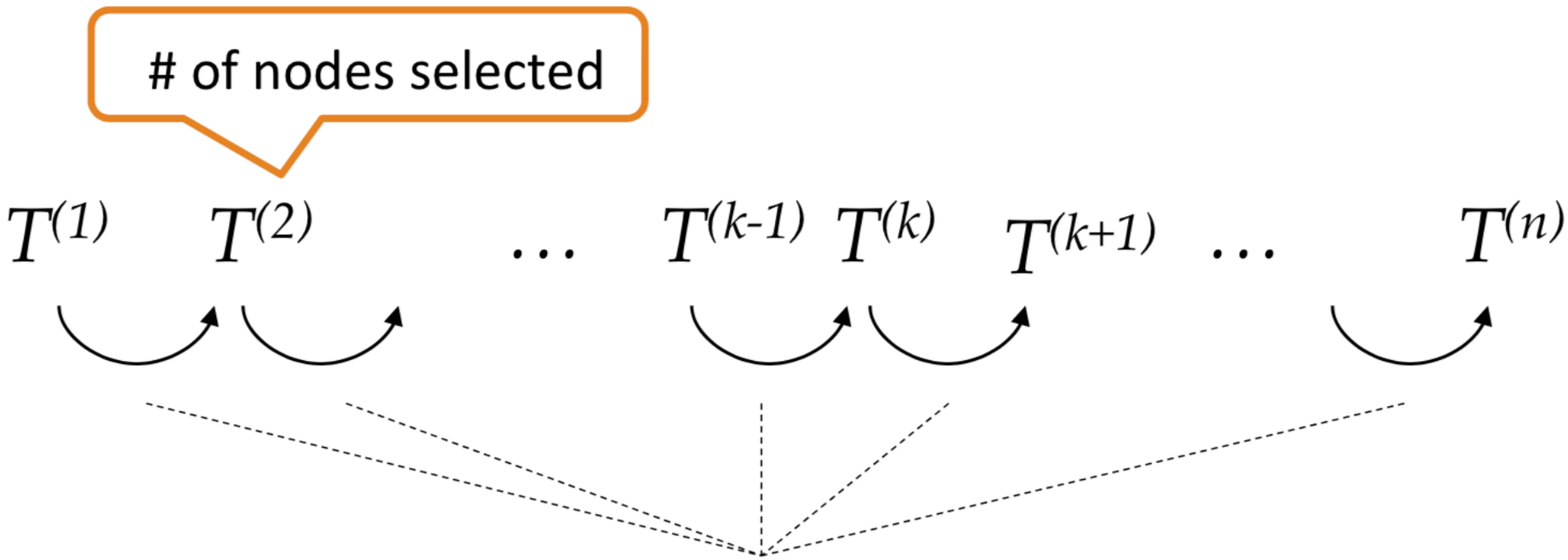




Greedy Algorithms for MST

- **Prim's algorithm:**
  - Difficult selecting: "best local optimization means no cycle and small weight under limitation"
  - Easy checking: doing nothing
- **Kruskal's algorithm:**
  - Easy selecting: smallest in primitive meaning
  - Difficult checking: no cycle

Correctness: How to Prove



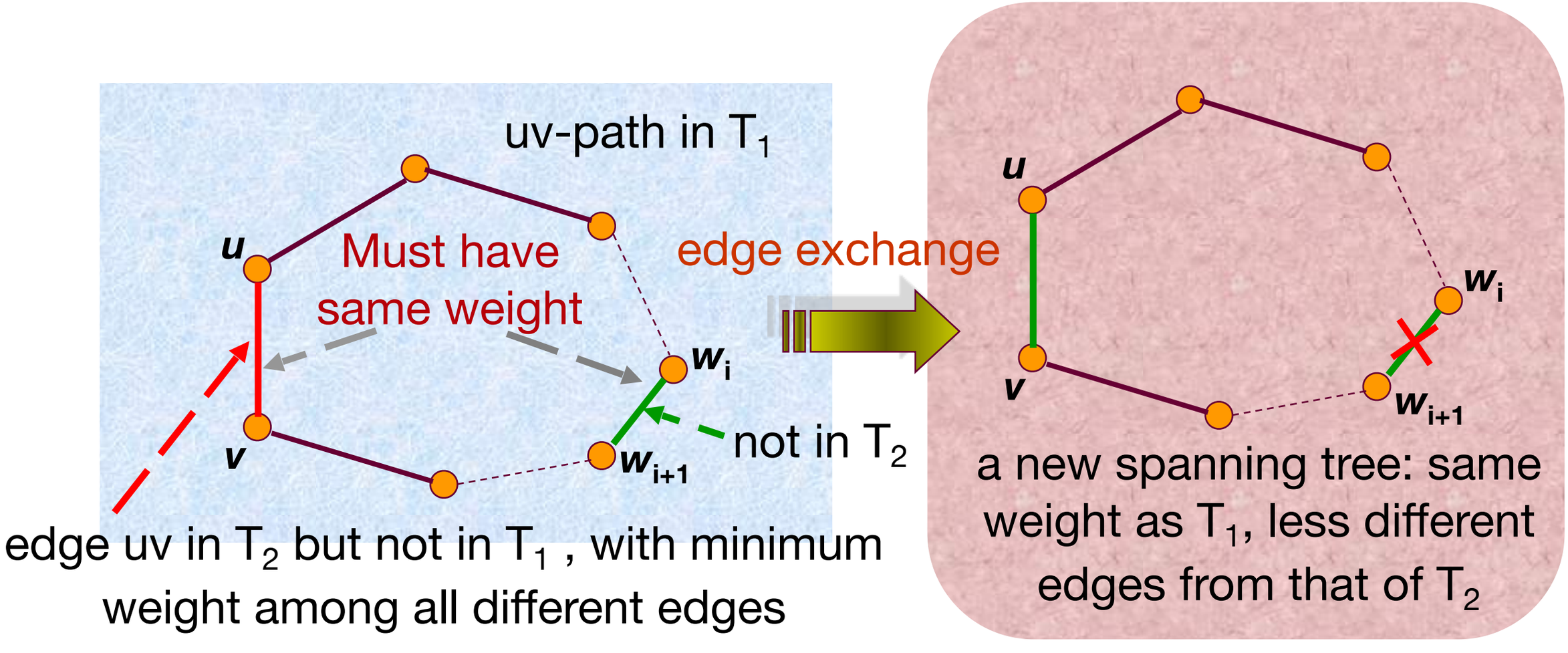
Invariance: MST

- Spanning tree
- Min weight

Definition transformation

Minimum Spanning Tree Property

- A spanning tree T of a connected, weighted graph has MST property if and only if for any non-tree edge uv, T ∪ {uv} contain a cycle in which uv is one of the maximum-weight edge.
- All the spanning trees having MST property have the same weight.



MST Property and Minimum Spanning Tree

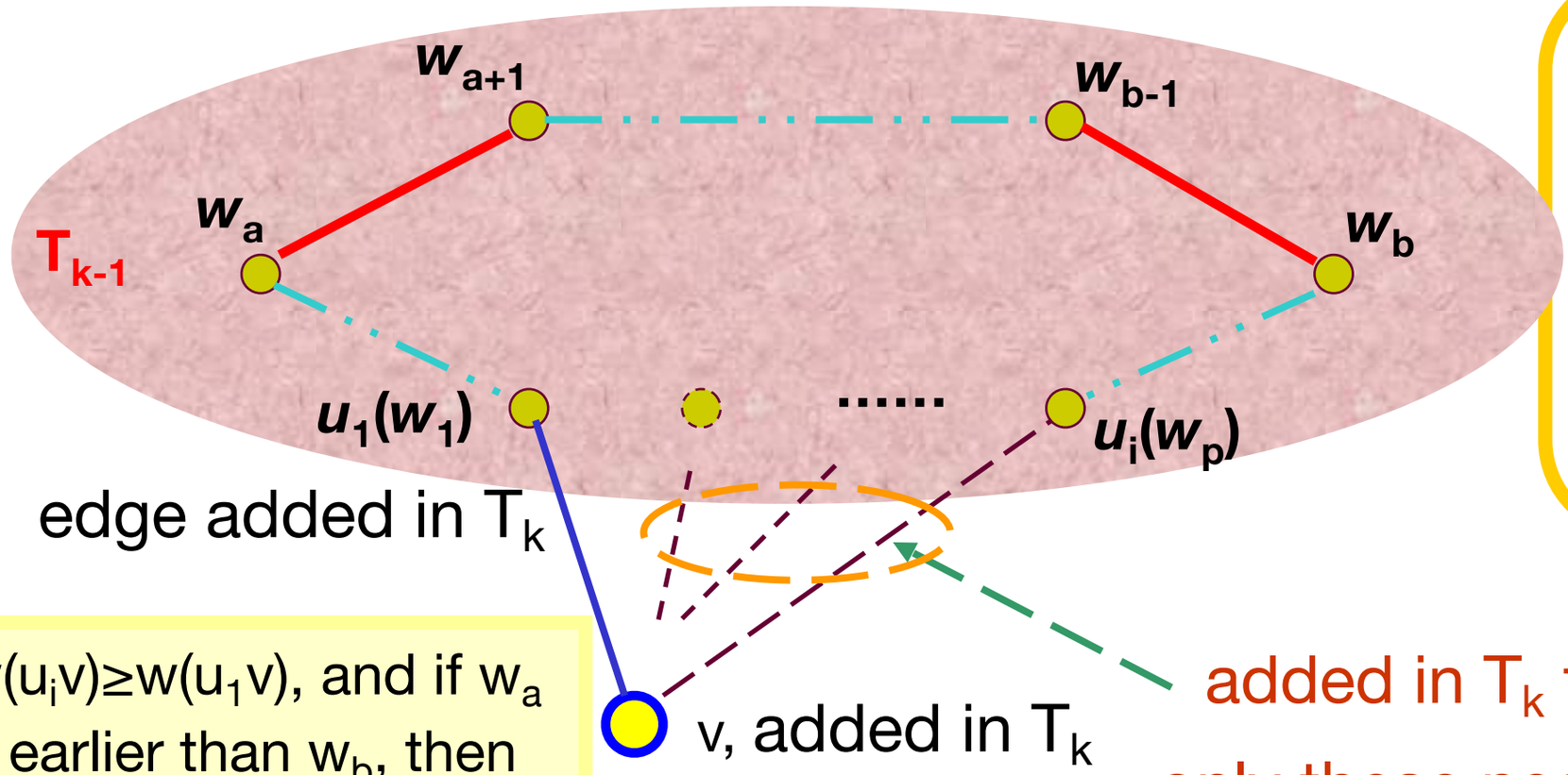
- **In a connected, weighted graph G={V,E,W}, a tree T is a minimum spanning tree if and only if T has the MST property.**

- **Proof**
  - => For a minimum spanning tree T, if it doesn't has MST property. So, there is a non-tree edge uv, and T ∪ {uv} contain an edge xy with weight larger than that of uv. Substituting uv for xy results a spanning tree with less weight than T. Contradiction.

  - <= As claimed above, any minimum spanning tree has the MST property. Since T has MST property, it has the same weight as any minimum spanning tree, i.e. T is a minimum spanning tree as well.

Correctness of Prim's Algorithm

- Let T_k be the tree constructed after the k^{th} step of Prim's algorithm is executed. Then T_k has the MST property in G_k, the subgraph of G induced by vertices of T_k.

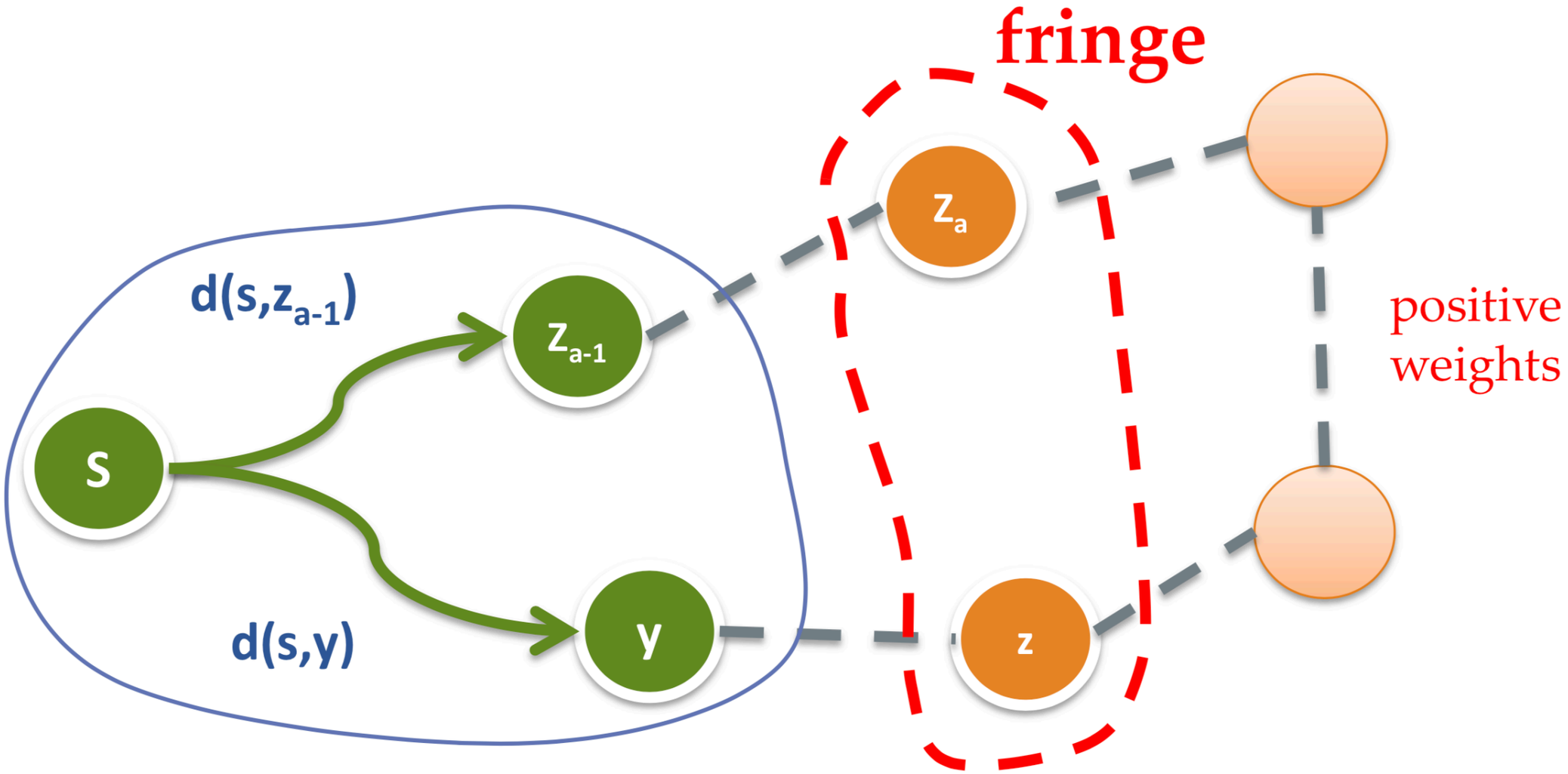


assumed first and last edges with larger weight than $w(u_iv)$, resulting contradictions.

Note: $w(u_iv) \geq w(u_1v)$, and if w_a added earlier than w_b, then w_aw_{a+1} and $w_{b-1}w_b$ added later than any edges in u_1w_a-path, and v as well

Correctness of Dijkstra Algorithm

- $W(s->y->z) < W(s->z_{a-1}->z_a->z)$



The Dijkstra Skeleton

- **Single-source shortest path (SSSP)**
- **SSSP + node weight constraint**
  - E.g. in routing
    - Each router has its cost (node cost)
    - Each route has its cost (edge cost)
- **SSP + capacity constraint**
  - The “pipe problem”
    - Maximize the min edge weight
  - The “electric vehicle problem”
    - Minimize the max edge weight

“Dijkstra Skeleton”

All-pairs Shortest Paths

- **For all pair of vertices in a graph, say, u, v:**
  - Is there a path from u to v?
  - What is the shortest path from u to v?
- **Reachability as a (reflexive) transitive closure of the adjacency relation**
  - Which can be represented as a bit matrix

Change the Order: The Warshall Algorithm

- void simpleTransitiveClosure(**boolean**[][] A, **int** n, **boolean**[][] R)
- **int** i,j,k;
- Copy A to R;
- Set all main diagonal entries, r_{ii}, to true;
- ~~**while**(any entry of R changed during one complete pass)~~
- **for**(k=1; k≤n; i++)
- **for**(i=1; i≤n; j++)
- **for**(j=1; j≤n; j++)
- $r_{ij}=r_{ij}\vee(r_{ik}\wedge r_{kj})$

k Varys in the outmost loop

Note: “false to true” can not be reversed

Correctness of the Warshall Algorithm

- If $r_{ij}^{(k)}$=true, then there is a $(s_i, s_j)^{(k)}$ path
- Proof
  - If $r_{ij}^{(0)}$=true, then there is $(s_i,s_j)^{(0)}$ path
  - If r_{ij} first become true in round k, then
    - $r_{ik}^{(k-1)}$ = true, $r_{kj}^{(k-1)}$ = true
  - We have a "s_i->s_k->s_j" path
    - Intermediate nodes in {1, 2, ..., k-1} ∪ {k}

All-pairs Shortest Paths

- **Floyd algorithm**
  - Only slight changes on Washall's algorithm.

```
Void allPairsShortestPaths(float [][] W, int n, float [][] D)
    int i, j, k;
    Copy W into D;
    for (k=1; k≤n; k++)
        for (i=1; i≤n; i++)
            for (j=1; j≤n; j++)
                D[i][j] = min (D[i][j], D[i][k]+D[k][j]);
```

All-pairs Shortest Paths

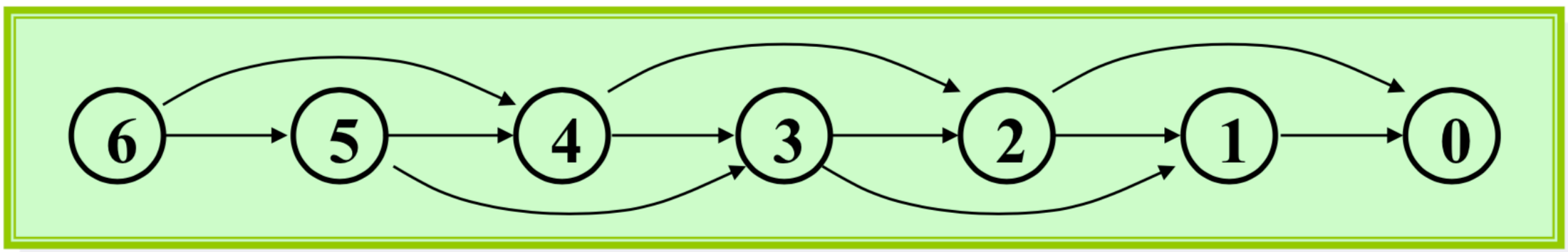
- **Construction of the routing table**
  - Forward, backward
- **APSP + capacity constraints**
  - The pipeline problem
  - The electric vehicle problem

Floyd algorithm => Floyd skeleton

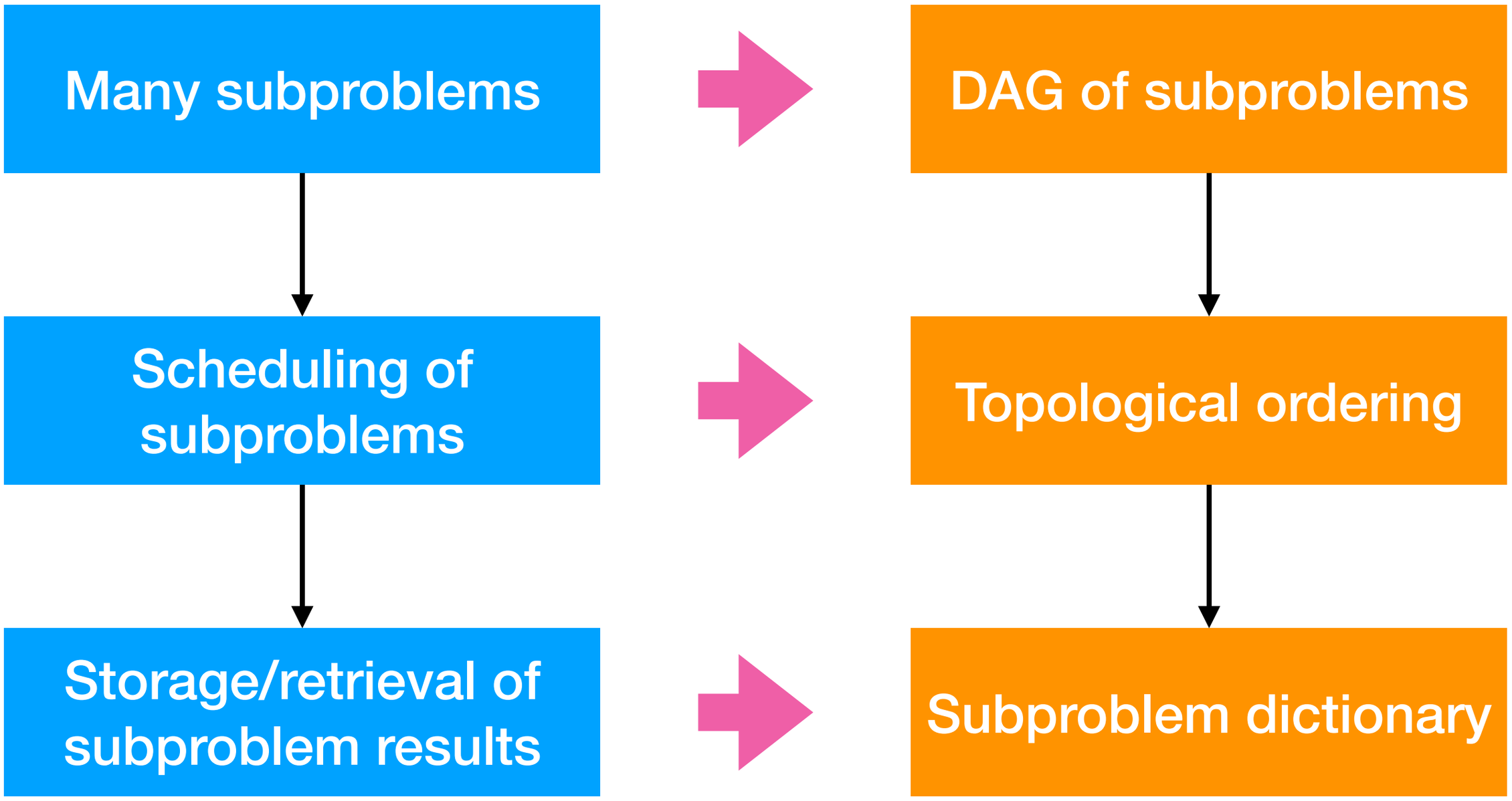
Subproblem Graph

- **The subproblem graph for a recursive algorithm A of some problem is defined as:**
  - vertex: the instance of the problem
  - 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)



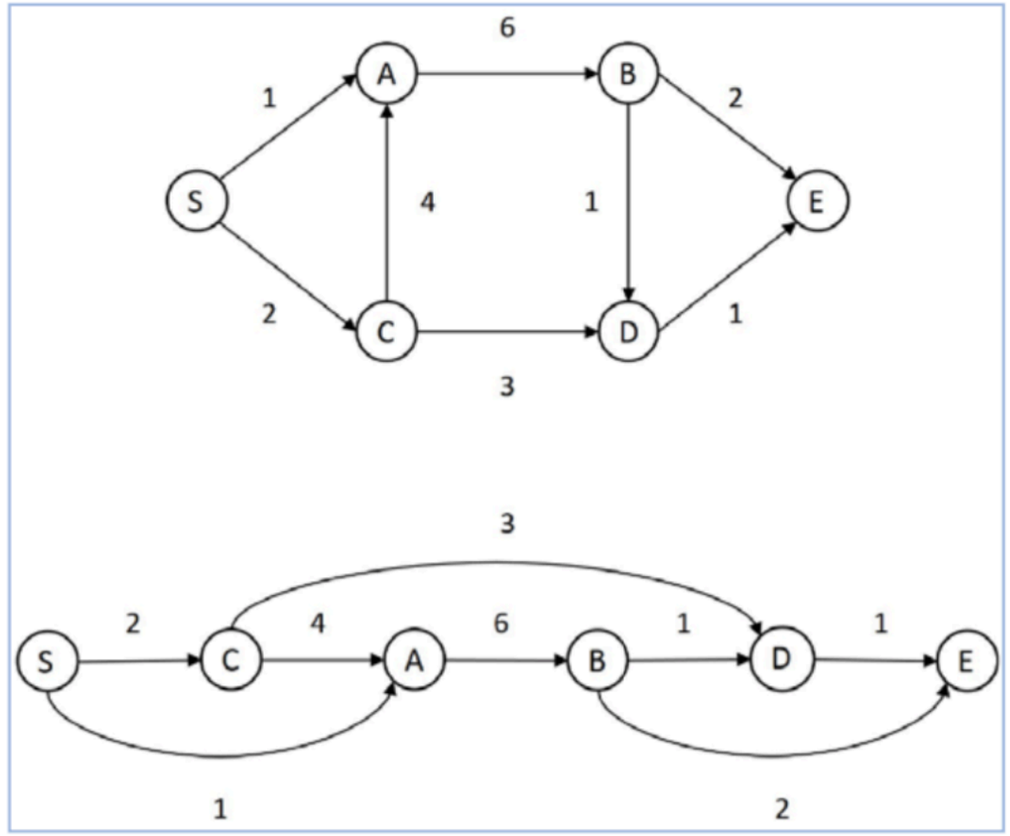
DP: New Concept Recursion



SSSP over a DAG

- **Subproblems**
  - One problem for each node
    - dis[1..n]
- **Dynamic programming**
  - Topological ordering of nodes in a DAG
- **More than SSSP**
  - As long as the recursion succeeds

$$D.dis = min\{\, B.dis + 1, C.dis + 3 \,\}$$



Edit Distance

- **You can edit a word by**
  - **I**nsert, **D**elete, **R**eplace
- **Edit distance**
  - Minimum number of edit operations
- **Problem**
  - Given two strings, compute the edit distance

The edit distance is 4

F O O D
M O N E Y

4 op: **R R I R**

3 op: not possible

“BF” Recursion

- **EditDis(i,j)**
  - Base case:
    - If i=0, EditDis(i,j)=j
    - If j=0, EditDis(i,j)=i
  - Recursion:

$$EditDis(A[1..m], B[1..n]) = min \begin{cases} EditDis(A[1..m-1], B[1..n]) + 1 \\ EditDis(A[1..m], B[1..n-1]) + 1 \\ EditDis(A[1..m-1], B[1..n-1]) + I\{A[m] \neq B[n]\} \end{cases}$$

Example

algorithm

vs.

altruistic

		A	L	G	O	R	I	T	H	M
	0 →	1 →	2 →	3 →	4 →	5 →	6 →	7 →	8 →	9
A	1	0 →	1 →	2 →	3 →	4 →	5 →	6 →	7 →	8
L	2	1	0 →	1 →	2 →	3 →	4 →	5 →	6 →	7
T	3	2	1	1 →	2 →	3 →	4 →	4 →	5 →	6
R	4	3	2	2	2	2 →	3 →	4 →	5 →	6
U	5	4	3	3	3	3	3 →	4 →	5 →	6
I	6	5	4	4	4	4	3 →	4 →	5 →	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	4 →	5 →	6
I	9	8	7	7	7	7	6	5	5 →	6
C	10	9	8	8	8	8	7	6	6	6

Highway Restaurants

- **The recursion**
  - P(j): the max profit achievable using only first j locations
    - P(0)=0
- prev[j]: largest index before j and k miles away

$$P(j) = \max(p_j + P(prev[j]), P(j-1))$$

Words into Lines

- **Words into lines**
  - Word-length w_1, w_2, …, w_n and line-width: W
- **Basic constraint**
  - If w_1, w_2, …, w_j are in one line, then $w_i+w_{i+1}+\ldots+w_j \leq W$
- **Penalty for one line: some function of X. X is:**
  - 0 for the last line in a paragraph, and
  - $W-(w_i+w_{i+1}+\ldots+w_j)$ for other lines
- **The problem**
  - How to make the penalty of the paragraph, which is the sum of the penalties of individual lines, minimized

Problem Decomposition

- **Representation of subproblem: a pair of indexes (i,j), breaking words i through j into lines with minimum penalty.**
- **Two kinds of subproblem**
  - (k,n): the penalty of the last line is 0
  - all other subproblems
- **For some k, the combination of the optimal solution for (1,k) and (k+1, n) gives a optimal solution for (1,n)**
- **Subproblem graph**
  - About n^2 vertices
  - Each vertex (i,j) has an edge to about j-i other vertices, so, the number of edges is in $\Theta(n^3)$

One-dimension Recursion

- **One-dimension problem space**
  - (1,n), (2,n), …, (n,n)

Subproblem (i,n)

Algorithm: lineBreak(w, W, i, n, L) — the current line

if $w_i + w_{i+1} + \cdots + w_n \leq W$ **then**

  <Put all words on line L, set penelty to 0> ;

else

  for $k = 1; w_i + \cdots + w_{i+k-1} \leq W; k++$ **do**

    $X = W - (w_i + \cdots + w_{i+k-1})$;

    $kPenalty = lineCost(X) + \text{lineBreak}(w, W, i + k, n, L + 1)$;

    <Set penalty always to the minimum $kPenalty$> ;

    <Updating k_{min}, which records the k part that produced the minimum penalty> ;

    <Put words i through $i + k_{min} - 1$ on line L> ;

return *penalty* ;

Dependency of Subproblems

- **c[i,0] is 0 for all i**
- **When we are to pay an amount j using coins of denominations 1 to i, we have two choices:**
  - No coins of denomination i is used: c[i-1, j]
  - One coins of denomination i is used: $1+c[i,j-d_i]$
- **So, $c[i,j]=\min(c[i-1,j], 1+c[i,j-d_i])$**

Other DP Problems

- **Text string problems**
  - Longest common subsequence, …
  - Variations of standard text string problems, …
- **One dimensional problems**
  - Arrangements along a straight line, …
- **Graph problems**
  - Vertex cover, …
- **Hard problems**
  - Knapsack problems and variations, …

Bowling Problem

- **Given n pins labeled 0,1,…,n-1**
- **Pin i has value v_i**
- **Two ball hit mode:**
  - Hit 1 pin I, get v_i points
  - Hit pin i and i+1, get v_i * v_{I+1} points
- **Problem: throw zero or more balls to maximize the total points**

Coin Picking Game

- **Given sequence of n coins of value v_0, v_1,…, V_{n-1}**
- **Two players take turns**
  - In a turn, take the first or last coin among the remaining coins
- **Goal: maximize total value of your taken coins (you go first)**

Arithmetic Parenthesization

- Input: an arithmetic expression: a_0, e_1, a_1, e_2,…, e_{n-1}, a_{n-1}
- Output: place parentheses to maximize the evaluated expression

Rod Cutting

- Given: a rod of length L and value(l) of rod of length l for all l in {1,2,3,…,L}
- Goal: cut the rod to maximize the value of cut rod pieces

Subset Sum

- Input: Sequence of n positive integers A={a_0, a_1, ..., a_n-1}
- Output: is there a subset of A that sums exactly to T?

Principle of optimality

- **Given an optimal sequence of decisions, each subsequence must be optimal by itself.**
  - Positive example: shortest path
  - Counterexample: longest (simple) path
- **DP relies on the principle of optimality**
  - The optimal solution to any nontrivial instance of a problem is a combination of optimal solutions to some of its sub-instances.
  - It is often not obvious which sub-instances are relevant to the instance under consideration.

Principle of optimality

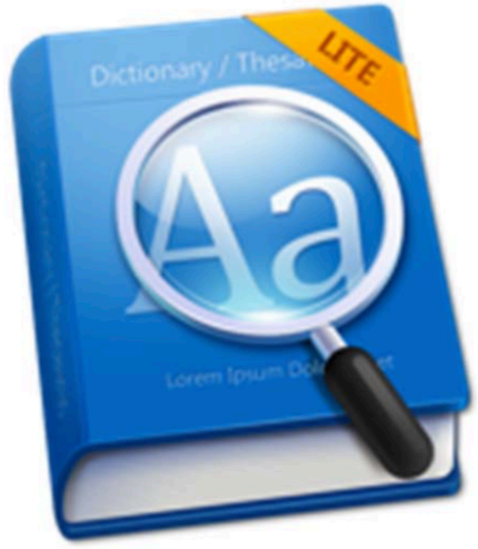
- **Given an optimal sequence of decisions, each subsequence must be optimal by itself.**
  - Positive example: shortest path
  - Counterexample: longest (simple) path
- **DP relies on the principle of optimality**
  - The optimal solution to any nontrivial instance of a problem is a combination of optimal solutions to some of its sub-instances.
  - It is often not obvious which sub-instances are relevant to the instance under consideration.

Optimal Substructure

Elements of Dynamic Programming

- **Symptoms of DP**
  - Overlapping subproblems
  - Optimal substructure
- **How to use DP**
  - “Brute force” recursion
    - Overlapping subproblems
  - “Smart” programming
    - Topological ordering of subproblems

DP Dictionary



Syllabus

Model of Computation

Algorithm Design & Analysis Techniques

Computation Complexity

Decision Problem

- **Statement of a decision problem**
  - Part 1: instance description defining the input
  - Part 2: question stating the actual yes-or-no question
- **A decision problem is a mapping from all possible inputs into the set {yes , no}**

Optimization v.s. Decision

- Usually, an optimization problem can be rephrased as a decision problem.

- For some cases, it can be proved that the decision problem can be solved in polynomial time if and only if the corresponding optimization problem can.

- We can make the statement that if the decision problem cannot be solved in polynomial time then the corresponding optimization problem cannot either.

The Class P

- A polynomially bounded algorithm is one with its worst-case complexity bounded by a polynomial function of the input size.

- A polynomially bounded problem is one for which there is a polynomially bounded algorithm.

- The class P is the class of decision problems that are polynomially bounded.

Nondeterministic Algorithm

```
void nondetA(String input)
    String s=genCertif();
    Boolean CheckOK=verifyA(input,s);
    if (checkOK)
        Output “yes”;
    return;
```

Phase 1 Guessing: generating **arbitrarily** “certificate”, i.e. proposed solution

The algorithm may behave differently on the same input in different executions: “yes” or “no output”.

Phase 2 Verifying: determining if *s* is a valid description of a object for answer, and satisfying the criteria for solution

Nondeterministic Algorithm

- For a particular decision problem with input x:
  - The answer computed by a nondeterministic algorithm is defined to be yes if and only if there is some execution of the algorithm that gives a yes output.
  - The answer is no, if for all s, there is no output.

The Class NP

- A polynomial bounded nondeterministic algorithm is one for which there is a (fixed) polynomial function *p* such that for each input of size *n* for which the answer is *yes* , there is some execution of the algorithm that produces a *yes* output in at most *p*(*n*) steps.

- The class NP is the class of decision problems for which there is a polynomial bounded nondeterministic algorithm.

The Class NP

- **NP means Non-deterministic P**
  - From “deterministic” to “non-deterministic”
  - From “solve a problem” to “verify the answer of a problem”
- **What does NP indicate?**
  - Harder problems
  - Not too hard
    - At least, you can quickly understand the answer

SAT

An example of propositional conjunctive normal form (CNF) is like this:

$$(p \vee q \vee s) \wedge (\bar{q} \vee r) \wedge (\bar{p} \vee r) \wedge (\bar{r} \vee s) \wedge (\bar{p} \vee \bar{s} \vee \bar{q})$$

Satisfiability Problem

Given a CNF formula, is there a truth assignment that satisfies it?

In other words, is there a assignment for the set of propositional variable in the CNF, such that the value of the formula is **true**.

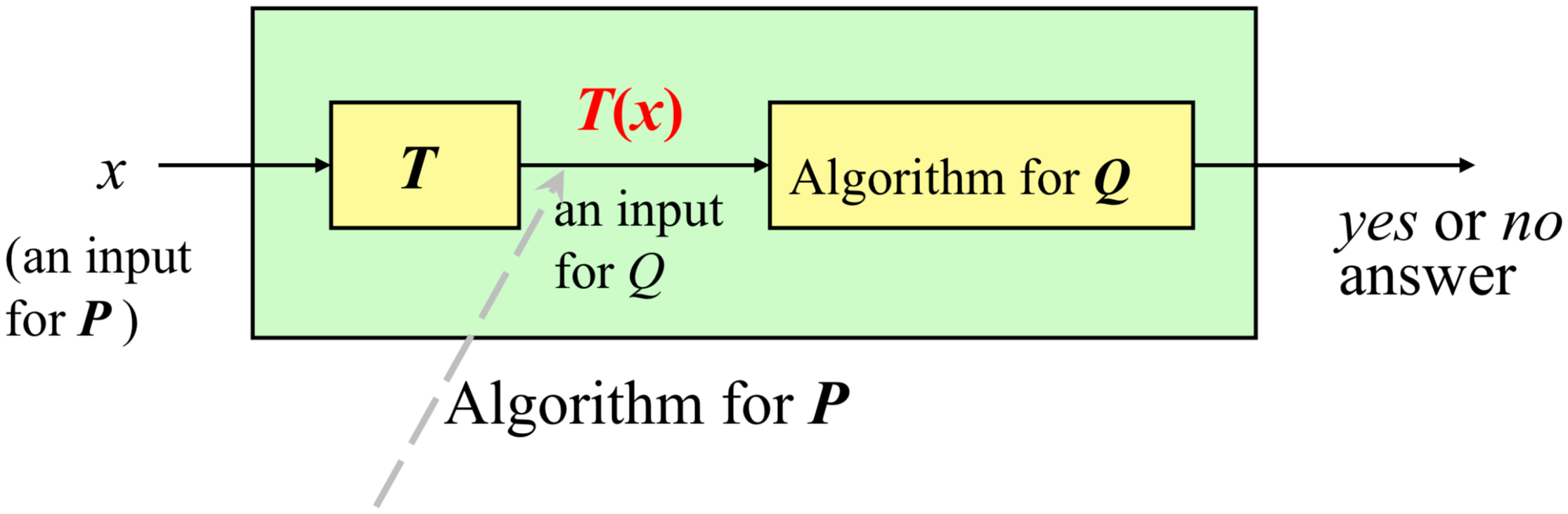
```
void nondetSat(E, n)
  boolean p[ ];
  for int i =1 to n do
     p[i ]= genCertif(true, false);
  if E(p[1], p[2], ..., p[n])=true
     then Output("yes");
```

So, the problem is in ***NP***

Relation between P and NP

- **An deterministic algorithm for a decision problem is a special case of a nondeterministic algorithm, which means: P ⊆ NP**
  - The deterministic algorithm is looked as the phase 2 of a nondeterministic one, which always ignore the *s* the phase 1 has written.
- **Intuition implies that NP is a much larger set than P.**
  - The number of possible *s* is exponential in *n*.
  - No one problem in NP has been proved not in P.

Reduction



The correct answer for ***P*** on x is *yes* **if and only if** the correct answer for ***Q*** on $T(x)$ is *yes*.

Polynomial Reduction

- **Let *T* be a function from the input set for a decision problem *P* into the input set for *Q*. *T* is a polynomial reduction from *P* to *Q* if:**
  - *T* can be computed in polynomial bounded time
  - x is a *yes* input for $P \rightarrow$ $T(x)$ is a *yes* input for Q
  - x is a *no* input for $P \rightarrow$ $T(x)$ is a *no* input for Q

An example:

***P*: Given a sequence of Boolean values, does at least one of them have the value true?**

***Q*: Given a sequence of integers, is the maximum of them positive?**

$T(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$, where: y_i=1 if x_i=true, and y_i=0 if x_i=false

Relation of Reducibility

- Problem *P* is polynomially reducible to *Q* if there exists a polynomial reduction from *P* to *Q*, denoted as: $P \leq_P Q$
- If $P \leq_P Q$ and *Q* is in P, then *P* is in P
  - The complexity of *P* is the sum of *T*, with the input size *n*, and *Q*, with the input size *p*(*n*), where *p* is the polynomial bound on *T*,
  - So, the total cost is: $p(n)+q(p(n))$, where *q* is the polynomial bound on *Q*.

(If $P \leq_P Q$, then Q is at least as “hard” to solve as P)

NP-complete Problems

- **A problem *Q* is NP-hard if every problem *P* in NP is reducible to *Q*, that is $P \leq_P Q$.**
  - (which means that *Q* is at least as hard as any problem in NP)
- **A problem *Q* is NP-complete if it is in NP and is NP-hard.**
  - (which means that *Q* is at most as hard as to be solved by a polynomially bounded nondeterministic algorithm)

Example of an NP-hard problem

- **Halt problem: Given an arbitrary deterministic algorithm *A* and an input *I*, does *A* with input *I* ever terminate?**

  - A well-known **undecidable** problem, of course not in NP .

  - Satisfiability problem is reducible to it.

    - Construct an algorithm *A* whose input is a propositional formula *X*. If *X* has *n* variables then *A* tries out all 2^n possible truth assignments and verifies if *X* is satisfiable. If it is satisfiable then *A* stops. Otherwise, *A* enters an infinite loop.

    - So, *A* halts on *X* iff. *X* is satisfiable.

Procedure for NP-Completeness

- **Knowledge: *P* is *NPC***
- **Task: to prove that *Q* is *NPC***
- **Approach: to reduce *P* to *Q***

- For any $R \in NP$, $R \leq_P P$
- Show $P \leq_P Q$
- Then $R \leq_P Q$, by transitivity of reductions
- Done. *Q* is *NP*-complete (given that Q has been proven in NP)

First Known NPC Problem

- **Cook's theorem:**
  - The SAT problem is NP-complete.
- **Reduction as tool for proving NP-completeness**
  - Since *CNF-SAT* is known to be *NP*-hard, then all the problems, to which *CNF-SAT* is reducible, are also *NP*-hard. So, the formidable task of proving *NP*- complete is transformed into relatively easy task of proving of being in NP.

Thank you!

Q & A