เอกสารประกอบการอบรม

การลดเพื่อเอาชนะ

(Decrease and Conquer)

ค่ายคอมพิวเตอร์โอลิมปิก สอวน. ค่าย 2 2/2567
ศูนย์โรงเรียนสามเสนวิทยาลัย - มหาวิทยาลัยธรรมศาสตร์
ระหว่างวันที่ 10 มีนาคม – 26 มีนาคม 2568

โดย

สาขาวิชาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์และเทคโนโลยี
มหาวิทยาลัยธรรมศาสตร์

เค้าโครงการบรรยาย

- การลดเพื่อเอาชนะ (Decrease and Conquer)

ผศ.ดร.ฐาปนา บุญชู

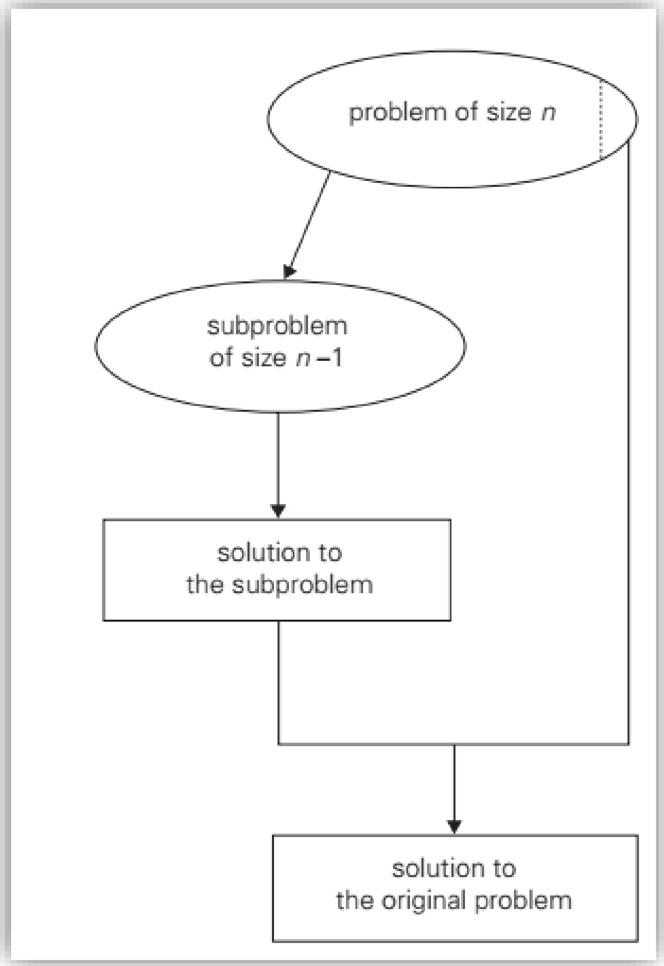
Decrease-and-Conquer

- **เทคนิคการลดเพื่อเอาชนะ (Decrease-and-conquer)** เป็นการอาศัยความสัมพันธ์ระหว่าง Solution ของ Instance ของปัญหาหนึ่ง กับ Solution ของ ตัวปัญหาหนึ่งที่เล็กกว่า
- เมื่อพบความสัมพันธ์ดังกล่าว เราสามารถใช้ประโยชน์จากความสัมพันธ์นั้นด้วย
  - **Top down:** การแก้ปัญหาในลักษณะ Recursive
  - **Bottom-up:** การแก้ปัญหาในลักษณะ Iterative (การเริ่มต้นจาก Solution ของปัญหาเล็กก่อนและสะสมขึ้นไปเรื่อย ๆ) อาจจะเรียกว่าเป็น Incremental approach.
- การลดเพื่อเอาชนะสามารถแบ่งได้ 3 แบบ
  - Decrease by a constant
  - Decrease by a constant factor
  - Variable size decrease

Decrease-and-Conquer

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0, \\ 1 & \text{if } n = 0, \end{cases}$$

- **Decrease by a constant**: ขนาดของปัญหาจะถูกลดลงด้วยจำนวนเท่า ๆ กันในแต่ละ Iteration ของ Algorithm
  - โดยทั่วไป ค่าคงที่ (Constant) จะมีค่าเท่ากับ 1 (แต่ก็อาจจะมีค่าอื่น ๆ ได้)
- ตัวอย่าง การคำนวณค่า Exponential a^n where $a \neq 0$ และ n เป็นเลขจำนวนเต็มที่ไม่ใช่จำนวนลบ (Nonnegative integer)
  - ความสัมพันธ์ระหว่าง Solution และ ขนาดของปัญหา n และ ขนาดปัญหาที่เล็กกว่า $n - 1$ สามารถเห็นได้ชัดเจนจาก $a^n = a^{n-1}a$
  - โดยฟังก์ชัน $f(n) = a^n$ สามารถคำนวณได้แบบ “Top down” (Recursive) หรือ “Bottom up” (การคูณเลข 1 ด้วย a จำนวน n ครั้ง)
  - ตัวอย่างดังกล่าวนี้ประสิทธิภาพอาจจะไม่ต่างจาก Brute force แต่จุดประสงค์ตัวอย่างนี้เพื่อนำเสนอกระบวนการคิดอีกรูปแบบหนึ่ง



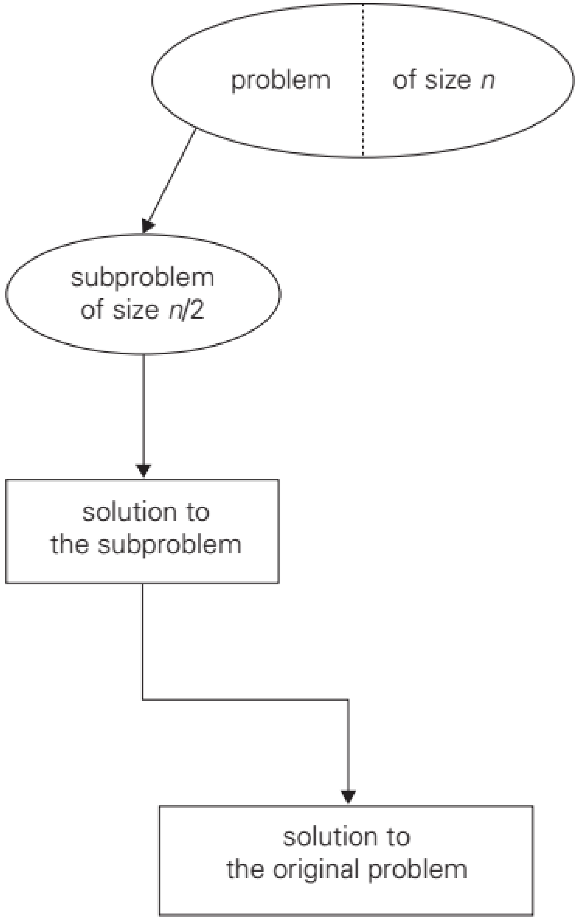
Decrease-and-Conquer

- **Decrease-by-a-constant-factor**: เป็นการลดขนาดของปัญหาลงขนาดจำนวนเท่าคงที่ (Constant factor) ในแต่ละ Iteration ของ Algorithm
  - โดย Constant factor ส่วนใหญ่จะมีค่า = 2
  - ตัวอย่าง ปัญหา a^n
    - ถ้าปัญหามีขนาด n ในการหาค่า a^n ขนาดของปัญหาที่มีขนาดเล็กลงครึ่งหนึ่งคือ $n/2$ ก็คือการคำนวณ $a^{n/2}$ จะเห็นได้ชัดว่าความสัมพันธ์ของปัญหา (Instances) 2 อันนี้คือ

$$a^n = \left(a^{\frac{n}{2}}\right)^2$$

    - จะได้ความสัมพันธ์ว่า (เปรียบเทียบกับ Brute force)

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases}$$



Guess time complexity?

$\Theta(\log n)$

Decrease-and-Conquer

- **Variable-size-decrease**: การลดของขนาดปัญหาแตกต่างกันไปในแต่ละ Iteration
  - ตัวอย่างคือ Euclid's algorithm ในการหาค่า หารร่วมมาก (ห.ร.ม.)

$$\boxed{\gcd(m, n) = \gcd(n, m \bmod n).}$$

  - จะเห็นว่าในแต่ละรอบของปัญหา ขนาดมันจะลดลงไปไม่เท่ากัน

แบบฝึกหัด

- ให้นักเรียนเขียนโปรแกรมเพื่อหาค่าของฟังก์ชัน $f(n) = a^n$ และเปรียบเทียบเวลาการรันโปรแกรมของการเขียนโปรแกรม 2 แบบต่อไปนี้
  - แบบที่ 1 ใช้วิธี Brute Force
  - แบบที่ 2 ใช้วิธี Decrease and Conquer
- 10 นาที

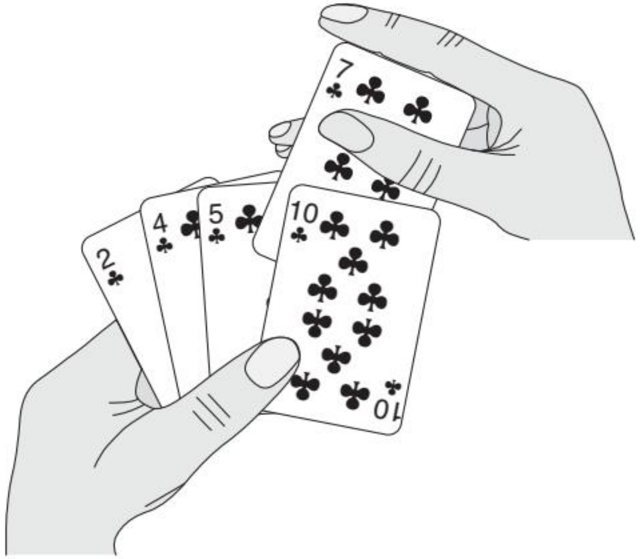
Problems

- Decrease by a Constant
    - Insertion sort
    - Topological sorting
    - Generating permutations, subsets
- Decrease by a Constant factor
    - Binary search
    - Fake-coin problems
    - Russian peasant multiplication
- Variable-Size-Decrease
    - Computing median and selection problem.
    - Interpolation Search
    - Euclid’s algorithm

Insertion Sort

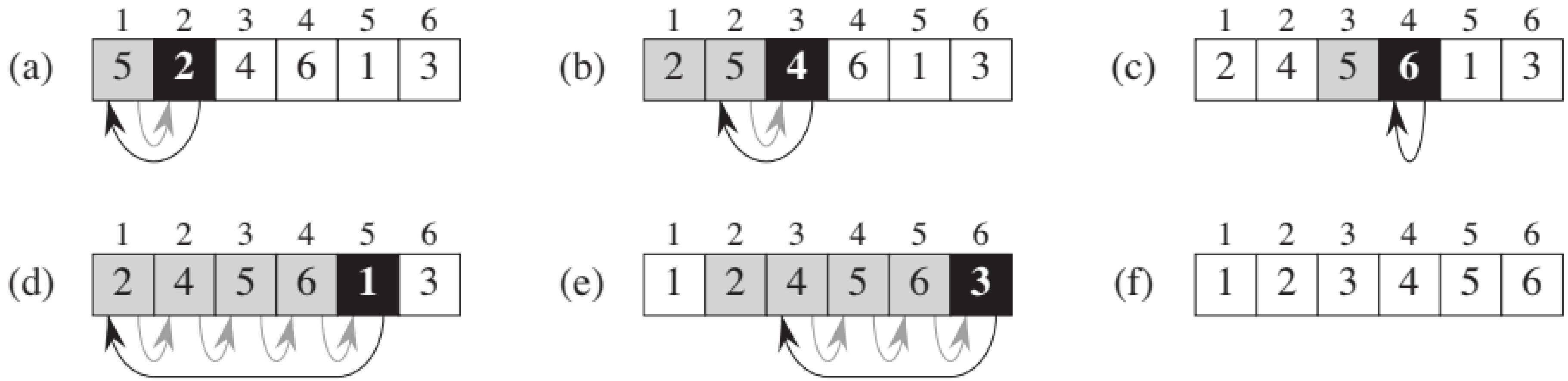
Insertion Sort

- **Insertion sort** เป็น Algorithm สำหรับการจัดเรียงข้อมูลที่ประสิทธิภาพ (เมื่อขนาดของปัญหาไม่ใหญ่มาก)
- การทำงานของ Insertion sort คล้ายวิธีการที่หลาย ๆ คนเรียงไพ่ในมือ
  - เริ่มจากมีไพ่คว่ำหน้าอยู่บนโต๊ะ
  - เราเริ่มหยิบไพ่จากเหล่านั้นขึ้นมาเรียงในมือเราทีละใบ**โดยเรียงตามลำดับ**
  - ทำจนกระทั่งไพ่บนโต๊ะไม่เหลือ



Insertion Sort Example

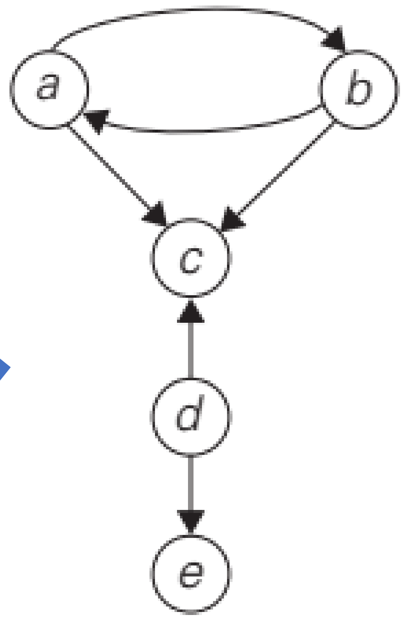
```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```



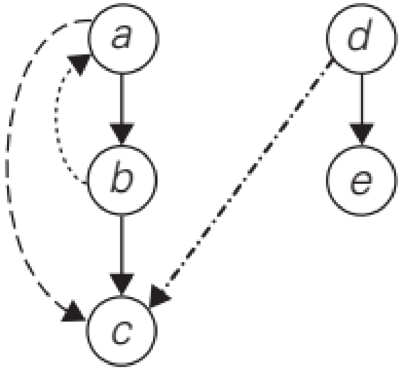
Topological Sorting

Topological Sorting

- Directed graph (Digraph) คืออะไร
  - เป็นกราฟที่มีการระบุทิศทางสำหรับทุก Edges
  - สามารถ Represent ด้วย Adjacency matrix หรือ Adjacency list



DAG: Directed Acyclic Graph?



Four types of edges possible in a DFS forest:

Tree edges:	ab, bc, de
Back edge :	ba
Forward edge:	ac
Cross edge:	dc

Topological Sorting

- ตัวอย่างของปัญหา: มีวิชา 5 วิชาคือ {C1, C2, C3, C4,C5} ที่นักศึกษาต้องเรียนให้ครบเพื่อจะได้ Degree
  - นักศึกษาจะลงเรียนวิชาทั้ง 5 นี้ให้ครบอย่างไรก็ได้ตราบเท่าที่มีการลงตาม Prerequisites ที่กำหนด
    - C1 และ C2 ไม่มี Prerequisites
    - C3 มี Prerequisites คือ C1 และ C2
    - C4 มี Prerequisites คือ C3
    - C5 มี Prerequisites คือ C3 และ C4
  - นักศึกษาลงได้เพียง 1 วิชาต่อ 1 เทอม
  - เราจะโมเดลปัญหานี้อย่างไร ?
    - Vertices -> Courses
    - Edge Directions -> Prerequisites

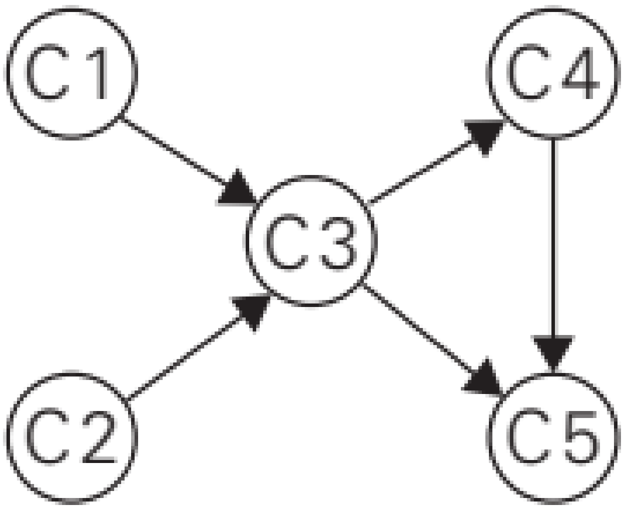
Topological Sorting

C1 และ C2 ไม่มี Prerequisites

C3 มี Prerequisites คือ C1 และ C2

C4 มี Prerequisites คือ C3

C5 มี Prerequisites คือ C3 และ C4



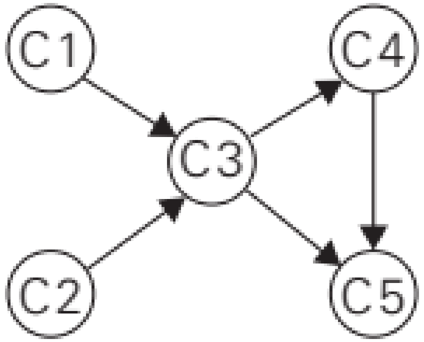
คำถามคือ จาก Digraph เราสามารถหา List ของทุก Vertices ที่ลำดับต้องสอดคล้องกับ Edges ในกราฟดังกล่าว (Vertex ที่อยู่ก่อนลูกศรจะต้องปรากฏอยู่ก่อน Vertex ที่อยู่ปลายลูกศรใน List เสมอ)

ปัญหานี้เราเรียกว่า ***topological sorting***.

นักเรียนคิดว่าหาก Graph เรามี Cycle เราจะสามารถทำ ***topological sorting*** ได้หรือไม่ ?

Topological Sorting

- มี 2 Algorithms
- **Algorithm 1**: ใช้ DFS Traversal และ Note Vertices ที่เกิด Dead-ends
    - หลังจากนั้นพิมพ์ลำดับการ Traverse ย้อนกลับหลังจะทำให้ได้ Solution สำหรับ Topological sorting
    - หากมีการพบ Back edge ทำให้ทราบได้ว่ากราฟดังกล่าวไม่ใช่ DAG ดังนั้นจะไม่สามารถหา Solution ได้



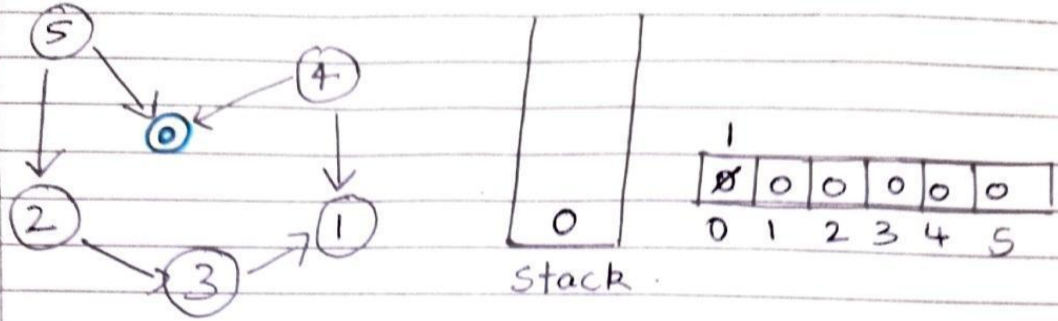
(a)

$C5_1$
$C4_2$
$C3_3$
$C1_4$ $C2_5$

(b)

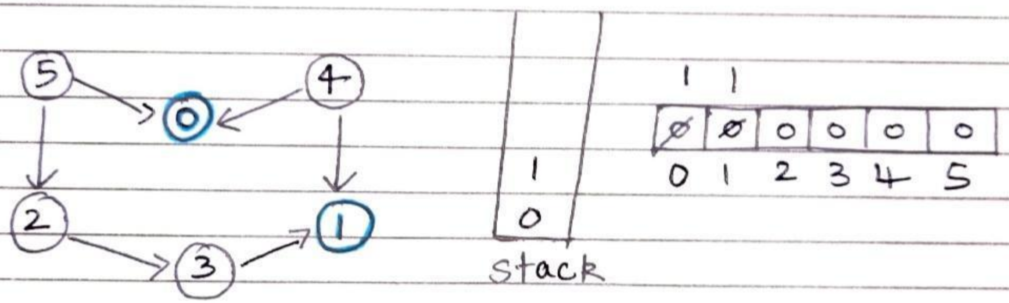
The popping-off order:
C5, C4, C3, C1, C2
The topologically sorted list:

(c)



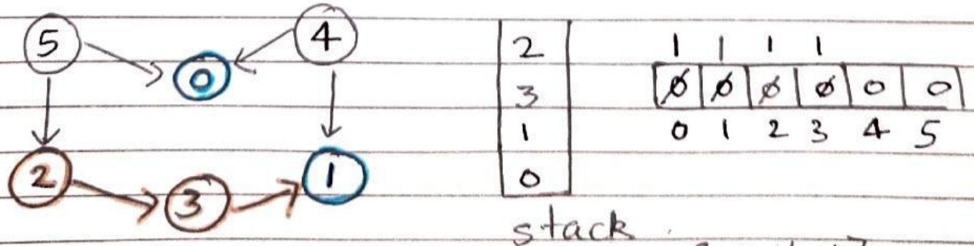
dfs(0)

x x

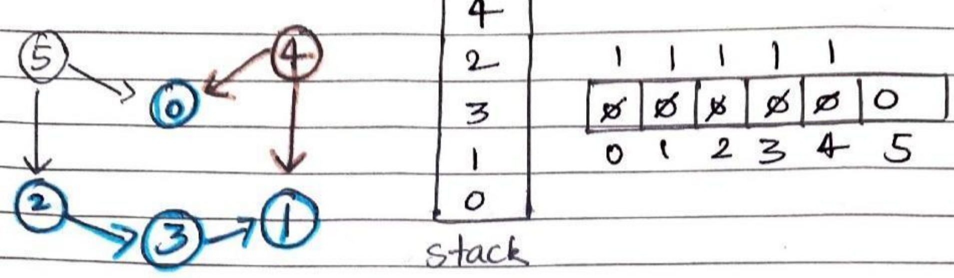


dfs(1)

x x



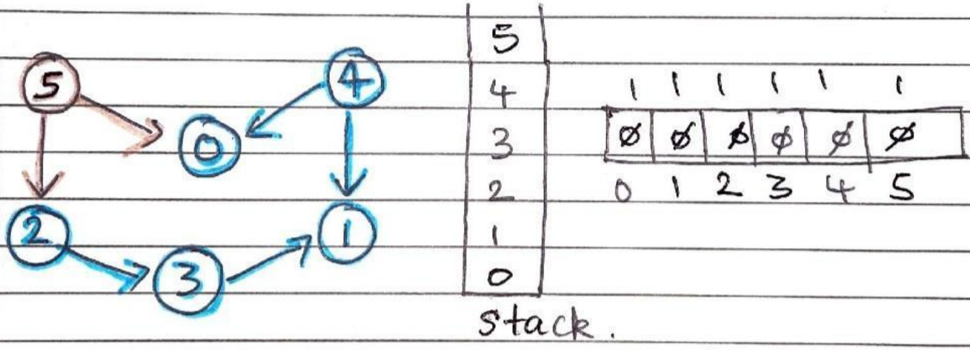
dfs(2) → dfs(3) $\xrightarrow{\times}$ dfs(1) [visited]



dfs(4)

x dfs(0) x dfs(1)

visited already.



dfs(5)

x dfs(2) x dfs(0)

visited already

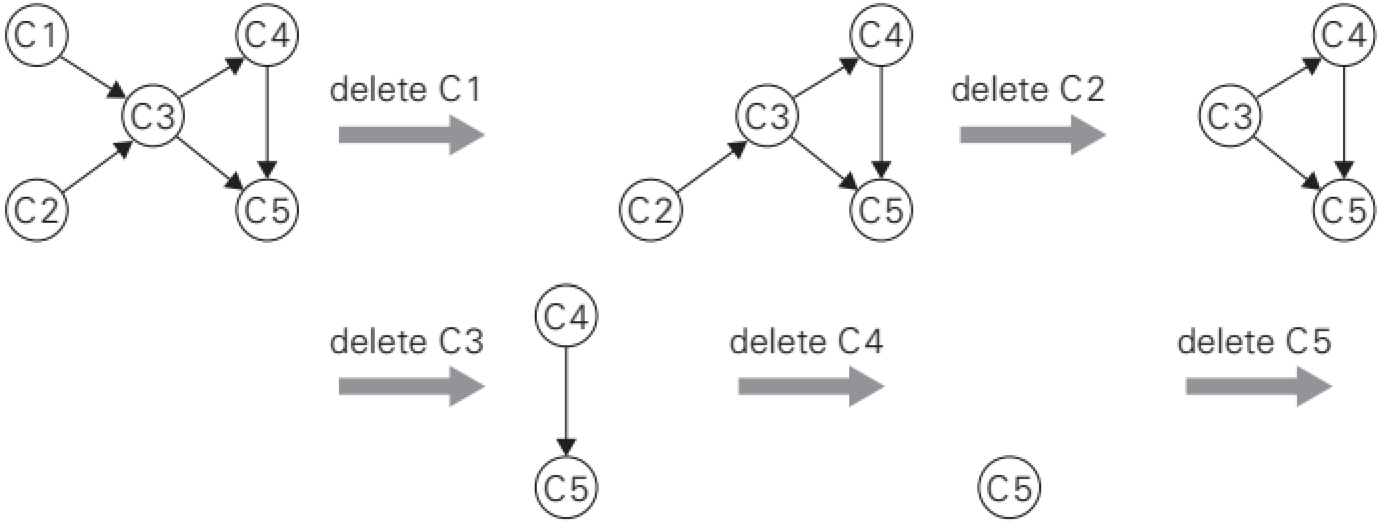
https://takeuforward.org/data-structure/topological-sort-using-dfs/

```cpp
#include <bits/stdc++.h>

using namespace std;

class Solution {
  void findTopoSort(int node, vector < int > & vis, stack < int > & st, vector < int > adj[]) {
    vis[node] = 1;

    for (auto it: adj[node]) {
      if (!vis[it]) {
        findTopoSort(it, vis, st, adj);
      }
    }
    st.push(node);
  }
  public:
    vector < int > topoSort(int N, vector < int > adj[]) {
      stack < int > st;
      vector < int > vis(N, 0);
      for (int i = 0; i < N; i++) {
        if (vis[i] == 0) {
          findTopoSort(i, vis, st, adj);
        }
      }
      vector < int > topo;
      while (!st.empty()) {
        topo.push_back(st.top());
        st.pop();
      }
      return topo;

    }
};
```

https://takeuforward.org/data-structure/topological-sort-using-dfs/

Topological Sorting

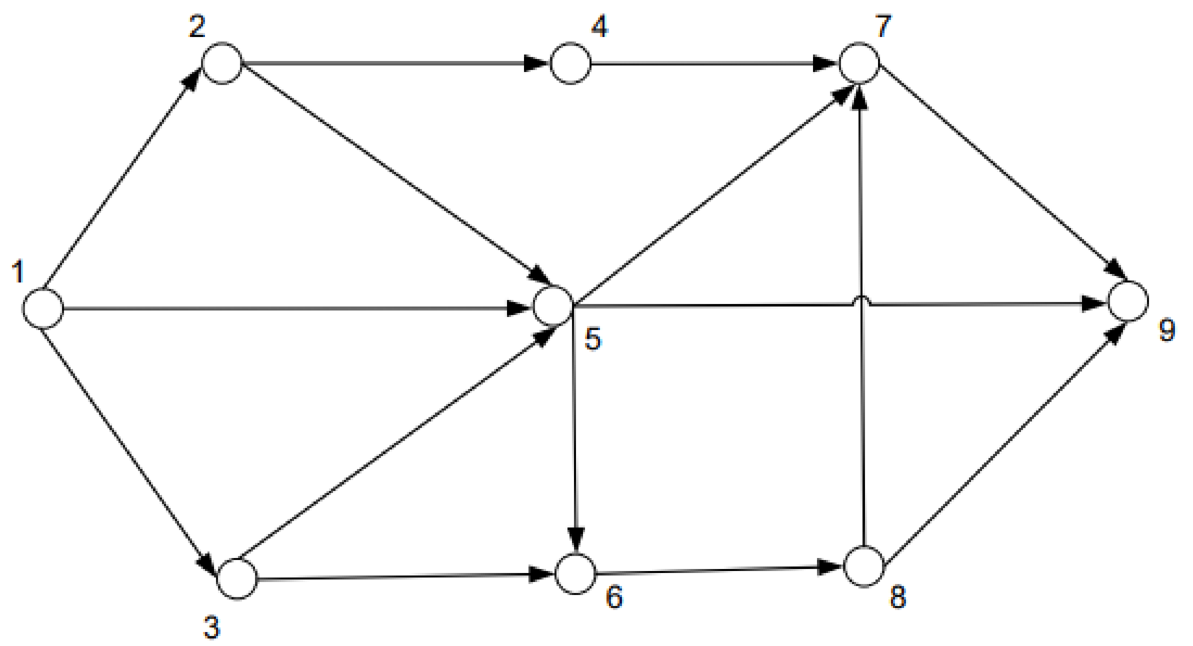
- มี 2 Algorithms
- **Algorithm 2**: พิจารณาใช้เทคนิค Decrease-(by one)-and-conquer โดยเริ่มจากเช็คว่า Vertex ไหนไม่มี Incoming edges ให้ลบทิ้งไปจากกราฟ (ต้องลบ Outgoing edges ของมันด้วย)



The solution obtained is C1, C2, C3, C4, C5

แบบฝึกหัด

- ให้นักศึกษาเขียนโปรแกรมสำหรับแก้ปัญหาTopological Sorting โดยใช้ Decrease and Conquer method (เลือก 1 วิธี)
- 15 นาที



Algorithms for Generating Combinatorial Objects

Combinatorial Objects

- สิ่งที่สำคัญในการจัดเรียงวัตถุคือ Permutation, Combination และ การหาทุก Subset ที่เป็นไปได้ของ set หนึ่ง
- นักคณิตศาสตร์สนใจว่าการจะนับการจัดเรียงวัตถุแต่ละแบบนั้นคำนวณออกมาอย่างไร ซึ่งทำให้เราทราบว่า จำนวนการจัดเรียงวัตถุนั้นมีการโตอย่าง **Exponential** หรือมากกว่า โดยเป็นฟังก์ชันกับขนาดของปัญหา
- ในบทนี้เราจะมาดู Algorithms ที่ช่วยในการ Generate การเรียงวัตถุเหล่านี้

Generating Permutation

Generating Permutations

- การ Generate Permutation
- สมมติว่า Set ที่เราต้องการทำการเรียงสับเปลี่ยนคือเซตจำนวนเต็ม 1 ถึง n นั่นคือ $\{a_1, \ldots, a_n\}$ (อาจจะมองเป็นดัชนีของวัตถุที่ต้องการเรียงสับเปลี่ยน)
- ถ้าปัญหาที่เล็กกว่าถูกแก้ เราสามารถได้ Solution ของปัญหาขนาดใหญ่กว่าได้โดยแทรก n ในทุก ๆ ตำแหน่งที่เป็นไปได้ของการเรียงสับเปลี่ยน n-1 วัตถุก่อนหน้า
- พิจารณารูปด้านล่าง

start	1			n-1 Solved!
insert 2 into 1 right to left	12	21		
insert 3 into 12 right to left	123	132	312	
insert 3 into 21 left to right	321	231	213	

Generating Permutations

- แต่ก็มีอีก Algorithm หนึ่งที่ช่วยเราสร้างลำกับการเรียงสับเปลี่ยนของ n วัตถุโดยที่**ไม่**ต้องจำเป็นต้อง Generate Permutation ของปัญหาที่เล็กกว่าก่อน
- สามารถทำได้โดยการพิจารณา **ทิศทาง** ของ Element k ในการเรียงสับเปลี่ยน
- Element k นี้เรียกว่าเป็น Mobile เมื่อมันลูกศรบนวัตถุนั้นชี้ไปยังตัวเลขติดกันที่ยังน้อยกว่า (พิจารณารูปด้านล่าง)

3 and 4 are mobile

$$\boxed{\overrightarrow{3}}\ \overleftarrow{2}\ \boxed{\overrightarrow{4}}\ \overleftarrow{1}.$$

Johnson-Trotter algorithm

$\Theta(n!)$

ALGORITHM *JohnsonTrotter*(n)

//Implements Johnson-Trotter algorithm for generating permutations
//Input: A positive integer n
//Output: A list of all permutations of $\{1, \ldots, n\}$
initialize the first permutation with $\overleftarrow{1}\,\overleftarrow{2} \ldots \overleftarrow{n}$
while the last permutation has a mobile element **do**
    find its largest mobile element k
    swap k with the adjacent element k's arrow points to
    reverse the direction of all the elements that are larger than k
    add the new permutation to the list

$$\overleftarrow{1}\,\overleftarrow{2}\,\overleftarrow{\mathbf{3}} \quad \overleftarrow{1}\,\overleftarrow{\mathbf{3}}\,\overleftarrow{2} \quad \overleftarrow{3}\,\overleftarrow{1}\,\overleftarrow{\mathbf{2}} \quad \overrightarrow{\mathbf{3}}\,\overleftarrow{2}\,\overleftarrow{1} \quad \overleftarrow{2}\,\overrightarrow{\mathbf{3}}\,\overleftarrow{1} \quad \overleftarrow{2}\,\overleftarrow{1}\,\overrightarrow{3}.$$

Find permutation using c++

```
void findPermutations(int a[], int n)
{

  // Sort the given array
  sort(a, a + n);

  // Find all possible permutations
  cout << "Possible permutations are:\n";
  do {
    display(a, n);
  } while (next_permutation(a, a + n));
}
```

https://www.geeksforgeeks.org/all-permutations-of-an-array-using-stl-in-c/

Exercise

- ใช้ STL of C++ เพื่อ Generating Permutation
- 5 นาที

Generating Subsets

Generating Subsets

- การ Generate subsets ก็สามารถใช้แนวคิดของ Decrease-by-one ได้เช่นเดียวกันโดย Sub set ของเซต $A = \{a_1, \ldots, a_n\}$ ที่เราต้องการจะหาจะแบ่งเป็น 2 กลุ่มคือ กลุ่มที่รวม n กับกลุ่มที่ไม่รวม n
  - กลุ่มแรก (ที่ไม่มี n) คือ Subsets ทุกอันของ $\{a_1, \ldots, a_{n-1}\}$
  - กลุ่มที่สองคือ (กลุ่มที่มี n) คือการเพิ่ม a_n ลงไปในแต่ละ Subset ของ $\{a_1, \ldots, a_{n-1}\}$.

n	**subsets** $\{a_1, a_2, a_3\}$ ***(in squashed order)***							
0	$\varnothing$							
1	$\varnothing$	$\{a_1\}$						
2	$\varnothing$	$\boxed{\{a_1\}}$	$\{a_2\}$	$\{a_1, a_2\}$				
3	$\varnothing$	$\{a_1\}$	$\{a_2\}$	$\{a_1, a_2\}$	$\{a_3\}$	$\{a_1, a_3\}$	$\{a_2, a_3\}$	$\{a_1, a_2, a_3\}$

bit strings	000	001	010	011	100	101	110	111
subsets	$\varnothing$	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$	$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$

Problems

- Decrease by a Constant
  - **Insertion sort**
  - **Topological sorting**
  - **Generating permutations, subsets**
- Decrease by a Constant factor
  - Binary search
  - Russian peasant multiplication
- Variable-Size-Decrease
  - Computing median and selection problem.
  - Interpolation Search
  - Euclid’s algorithm

Decrease-by-a-Constant-Factor Algorithms

Binary Search

Decrease-by-a-Constant-Factor Algorithms

- The most important and well-known of them is **binary search**.
- **Decrease-by-a-constant-factor** algorithms usually run in logarithmic time, and, being very efficient, do not happen often; a reduction by a factor other than two is especially rare.

Binary Search

Array A is assumed to be sorted.

$$\begin{array}{c} K \\ \updownarrow \end{array}$$

$$\underbrace{A[0]\dots A[m-1]}_{\substack{\text{search here if} \\ K<A[m]}} \quad A[m] \quad \underbrace{A[m+1]\dots A[n-1]}_{\substack{\text{search here if} \\ K>A[m]}}.$$

ALGORITHM *BinarySearch*$(A[0..n-1], K)$

```
//Implements nonrecursive binary search
//Input: An array A[0..n − 1] sorted in ascending order and
//        a search key K
//Output: An index of the array's element that is equal to K
//        or −1 if there is no such element
l ← 0;   r ← n − 1
while l ≤ r do
    m ← ⌊(l + r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ← m − 1
    else l ← m + 1
return −1
```

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	l						m						r
iteration 2								l		m			r
iteration 3								l,m	r				

Binary Search in C++ Standard Template Library (STL)

```
binary_search(startaddress,
              endaddress, valuetofind)
Parameters :
startaddress: the address of the first
              element of the array.
endaddress: the address of the next contiguous
            location of the last element of the array.
valuetofind: the target value which we have
             to search for.
Returns :
true if an element equal to valuetofind is found, else false.
```

```cpp
// CPP program to implement
// Binary Search in
// Standard Template Library (STL)
#include <algorithm>
#include <iostream>

using namespace std;

void show(int a[], int arraysize)
{
    for (int i = 0; i < arraysize; ++i)
        cout << a[i] << ",";
}

int main()
{
    int a[] = { 1, 5, 8, 9, 6, 7, 3, 4, 2, 0 };
    int asize = sizeof(a) / sizeof(a[0]);
    cout << "\nThe array is : \n";
    show(a, asize);

    cout << "\n\nLet's say we want to search for ";
    cout << "\n2 in the array So, we first sort the array";
    sort(a, a + asize);
    cout << "\n\nThe array after sorting is : \n";
    show(a, asize);
    cout << "\n\nNow, we do the binary search";
    if (binary_search(a, a + 10, 2))
        cout << "\nElement found in the array";
    else
        cout << "\nElement not found in the array";

    cout << "\n\nNow, say we want to search for 10";
    if (binary_search(a, a + 10, 10))
        cout << "\nElement found in the array";
    else
        cout << "\nElement not found in the array";

    return 0;
}
```

lower_bound

- **lower_bound(start_ptr, end_ptr, num):**
  - Returns pointer to the position of num if the container contains only one occurrence of num.
  - Returns a pointer to the first position of num if the container contains multiple occurrences of num.
  - Returns pointer to the position of a number **just higher than num**, if the container **does not contain an occurrence** of num which is the position of the number when inserted in the already sorted array and sorted again.
  - Subtracting the first position i.e vect.begin() from the pointer, returns the **actual index**.

upper_bound

- **upper_bound(start_ptr, end_ptr, num):**
  - Returns pointer to the position of **next higher number than num** if the container contains one occurrence of **num**.
  - Returns pointer to the first position of the next higher number than the last occurrence of num if the container contains multiple occurrences of num.
  - Returns pointer to position of next higher number than num if the container **does not contain an occurrence** of num.

```cpp
// C++ code to demonstrate the working of lower_bound()
#include <bits/stdc++.h>
using namespace std;

// Driver's code
int main()
{
    // initializing vector of integers
    // for single occurrence
    vector<int> arr1 = { 10, 15, 20, 25, 30, 35 };

    // initializing vector of integers
    // for multiple occurrences
    vector<int> arr2 = { 10, 15, 20, 20, 25, 30, 35 };

    // initializing vector of integers
    // for no occurrence
    vector<int> arr3 = { 10, 15, 25, 30, 35 };

    // using lower_bound() to check if 20 exists
    // single occurrence
    // prints 2
    cout << "The position of 20 using lower_bound "
            " (in single occurrence case) : ";
    cout << lower_bound(arr1.begin(), arr1.end(), 20)
                - arr1.begin();
```

```cpp
    cout << endl;

    // using lower_bound() to check if 20 exists
    // multiple occurrence
    // prints 2
    cout << "The position of 20 using lower_bound "
            "(in multiple occurrence case) : ";
    cout << lower_bound(arr2.begin(), arr2.end(), 20)
                - arr2.begin();

    cout << endl;

    // using lower_bound() to check if 20 exists
    // no occurrence
    // prints 2 ( index of next higher)
    cout << "The position of 20 using lower_bound "
            "(in no occurrence case) : ";
    cout << lower_bound(arr3.begin(), arr3.end(), 20)
                - arr3.begin();

    cout << endl;
}
```

Exercise

- เขียนโปรแกรมสำหรับค้นหาด้วย Binary Search
- Input: {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}, **search for 16 and 88**
- **(1) Implement ด้วยตนเอง**
- **(2) ใช้ STL**
  - **2.1 binary_search**
  - **2.2 lower_bound**
  - **2.3 upper_bound**

Russian Peasant Multiplication

(Multiply two numbers using bitwise operators)

Russian Peasant Multiplication

- Given two integers, write a function to multiply them **without using multiplication operator.**
- Idea:
  - **Double** the first number (a) and **halve** the second number (b) repeatedly till the second number **doesn't** become 1.
  - Whenever the second number become **odd**, we add the first number to result (result is initialized as 0)

		85	× 18	= 1530
0	1	85	18	18
1	0	42	36	
2	1	21	72	+ 72
3	0	10	144	
4	1	5	288	+ 288
5	0	2	576	
6	1	1	1152	+ 1152
				1530

https://www.cut-the-knot.org/Curriculum/Algebra/PeasantMultiplication.shtml

https://www.geeksforgeeks.org/russian-peasant-multiply-two-numbers-using-bitwise-operators/

Bitwise operators

https://www.geeksforgeeks.org/bitwise-operators-in-c-cpp/

- The **& (bitwise AND)** in C takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
- The **| (bitwise OR)** in C takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.
- The **^ (bitwise XOR)** in C takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
- The **<< (left shift)** in C takes two numbers, the left shifts the bits of the first operand, and the second operand decides the number of places to shift.
- The **>> (right shift)** in C takes two numbers, right shifts the bits of the first operand, and the second operand decides the number of places to shift.
- The **~ (bitwise NOT)** in C takes one number and inverts all bits of it.

```cpp
#include <iostream>
using namespace std;

// A method to multiply two numbers using Russian Peasant method
unsigned int russianPeasant(unsigned int a, unsigned int b)
{
    int res = 0; // initialize result

    // While second number doesn't become 1
    while (b > 0)
    {
        // If second number becomes odd, add the first number to result
        if (b & 1)
            res = res + a;

        // Double the first number and halve the second number
        a = a << 1;
        b = b >> 1;
    }
    return res;
}

// Driver program to test above function
int main()
{
    cout << russianPeasant(18, 1) << endl;
    cout << russianPeasant(20, 12) << endl;
    return 0;
}
```

https://www.geeksforgeeks.org/russian-peasant-multiply-two-numbers-using-bitwise-operators/

Problems

- Decrease by a Constant
  - **Insertion sort**
  - **Topological sorting**
  - **Generating permutations, subsets**
- Decrease by a Constant factor
  - **Binary search**
  - **Russian peasant multiplication**
- Variable-Size-Decrease
  - Computing median and selection problem.
  - Interpolation Search
  - Euclid’s algorithm

Variable-Size-Decrease Algorithms

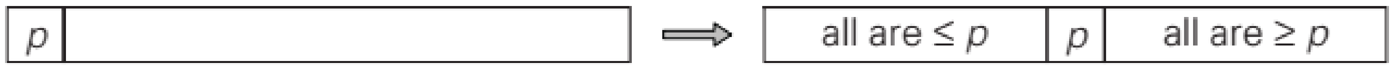
Computing median and selection problem.

Computing a Median and the Selection Problem

- The **selection problem** is the problem of finding the kth smallest element in a list of n numbers.
  - This number is called the $\boldsymbol{k^{th}}$ ***order statistic***.
- How to find the case that $k = 1$ and $k = n$?
- What if $k = \lceil n/2 \rceil$?
  - This is to find the **median.**
- Obviously, we can find the k^{th} smallest element in a list by sorting the list first and then selecting the k^{th} element in the output of a sorting algorithm. The time of such an algorithm is determined by the efficiency of the sorting algorithm used.
  - Thus, with a fast sorting algorithm such as mergesort (discussed in the next chapter), the algorithm's efficiency is in $O(n \log n)$.

Lomuto partitioning

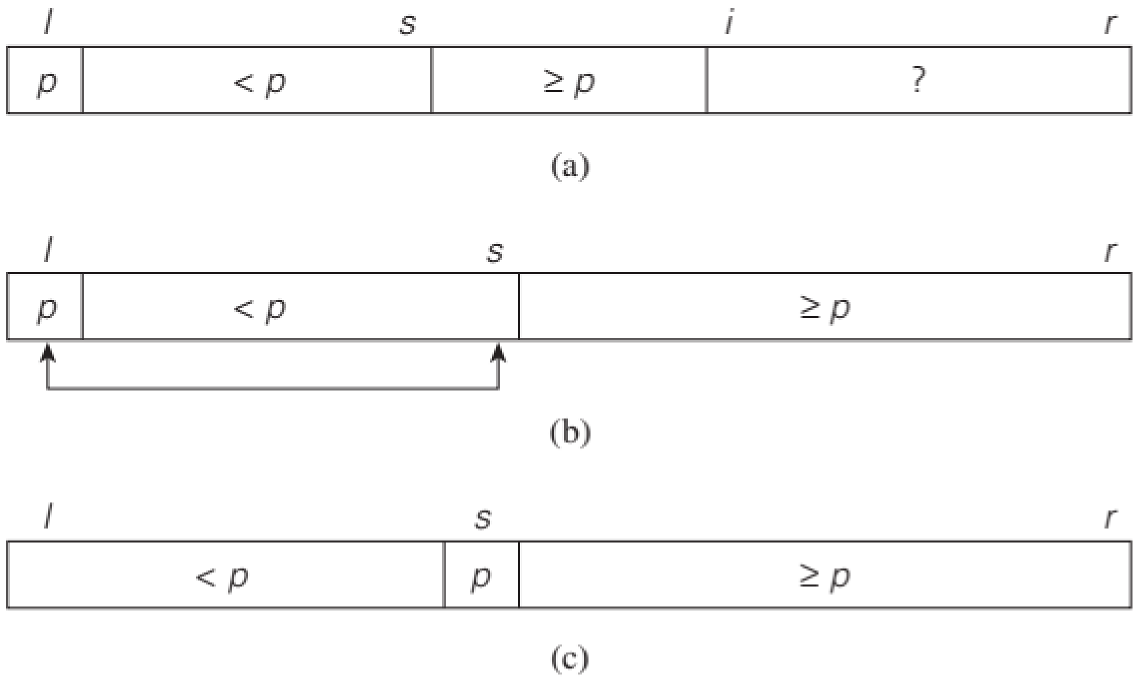
- Indeed, we can take advantage of the idea of partitioning a given list around some value p of, say, its first element.
- In general, this is a rearrangement of the list's elements so that the left part contains all the elements smaller than or equal to p, followed by the **pivot** p itself, followed by all the elements greater than or equal to p.



Lomuto partitioning

- We may think of an array—or, more generally, a subarray $A[l..r]$ $(0 \le l \le r \le n-1)$—under consideration as composed of <u>*three*</u> contiguous segments.

```
ALGORITHM  LomutoPartition(A[l..r])
    //Partitions subarray by Lomuto's algorithm using first element as pivot
    //Input: A subarray A[l..r] of array A[0..n − 1], defined by its left and right
    //       indices l and r (l ≤ r)
    //Output: Partition of A[l..r] and the new position of the pivot
    p ← A[l]
    s ← l
    for i ← l + 1 to r do
        if A[i] < p
            s ← s + 1;  swap(A[s], A[i])
    swap(A[l], A[s])
    return s
```



Quick Select

- How can we take advantage of a list partition to find the k^{th} smallest element in it?
  - Let S be the partition's split position i.e., the index of the array's element occupied by the pivot after partitioning.
  - If s = k — 1, pivot p itself is obviously the k^{th} smallest element, which solves the problem.
  - If s > k — 1, the kth smallest element in the entire array can be found as the k^{th} smallest element in the left part of the partitioned array.
  - If s < k — 1, it can be found as the (k — s)th smallest element in its right part.

ALGORITHM *Quickselect*$(A[l..r], k)$

//Solves the selection problem by recursive partition-based algorithm
//Input: Subarray $A[l..r]$ of array $A[0..n-1]$ of orderable elements and
// integer k $(1 \le k \le r - l + 1)$
//Output: The value of the kth smallest element in $A[l..r]$
$s \leftarrow$ *LomutoPartition*$(A[l..r])$ //or another partition algorithm
if $s = k - 1$ **return** $A[s]$
else if $s > l + k - 1$ *Quickselect*$(A[l..s-1], k)$
else *Quickselect*$(A[s+1..r], k - 1 - s)$

Quick Select

0	1	2	3	4	5	6	7	8
s	*i*							
4	1	10	8	7	12	9	2	15
	s	*i*						
4	1	10	8	7	12	9	2	15
	s						*i*	
4	1	10	8	7	12	9	2	15
		s					*i*	
4	1	2	8	7	12	9	10	15
		s						*i*
4	1	2	8	7	12	9	10	15
2	1	**4**	8	7	12	9	10	15

s	*i*				
8	7	12	9	10	15
	s	*i*			
8	7	12	9	10	15
	s				*i*
8	7	12	9	10	15
7	**8**	12	9	10	15

```cpp
// C++ program to demonstrate the use of std::nth_element
#include <algorithm>
#include <iostream>
using namespace std;

// Defining the BinaryFunction
bool comp(int a, int b) { return (a < b); }
int main()
{
    int v[] = { 3, 2, 10, 45, 33, 56, 23, 47 }, i;

    // Using std::nth_element with n as 6
    std::nth_element(v, v + 5, v + 8, comp);

    // Since, n is 6 so 6th element should be the same
    // as the sixth element present if we sort this array
    // Sorted Array
    /* 2 3 10 23 33 45 47 56 */
    for (i = 0; i < 8; ++i) {
        cout << v[i] << " ";
    }
    return 0;
}
```

https://www.geeksforgeeks.org/stdnth_element-in-cpp/

Interpolation Search

- The **Interpolation Search** is an improvement over Binary Search for instances, where the values in a sorted array are **uniformly distributed**.
- There are many different interpolation methods and one such is known as **linear interpolation**.
- This algorithm works in a way we search for a word in a **dictionary**.
- *pos* is a constant which is used to **narrow** the search space.
    - x is the key to be searched.

lo ==> Starting index in arr[]

hi ==> Ending index in arr[]

$$pos = lo + \left[\frac{(x - arr[lo]) * (hi - lo)}{(arr[hi] - arr[Lo])}\right]$$

Interpolation Search

```
int interpolationSearch(int arr[], int n, int x)
{
    // Find indexes of two corners
    int low = 0, high = (n - 1);
    // Since array is sorted, an element present
    // in array must be in range defined by corner
    while (low <= high && x >= arr[low] && x <= arr[high])
    {
        if (low == high)
        {if (arr[low] == x) return low;
        return -1;
        }
        // Probing the position with keeping
        // uniform distribution in mind.
        int pos = low + (((double)(high - low) /
            (arr[high] - arr[low])) * (x - arr[low]));

        // Condition of target found
        if (arr[pos] == x)
            return pos;
        // If x is larger, x is in upper part
        if (arr[pos] < x)
            low = pos + 1;
        // If x is smaller, x is in the lower part
        else
            high = pos - 1;
    }
    return -1;
}
```

Euclid's algorithm

- Greatest Common Division (GCD)

36 = 2 x 2 x 3 x 3
60 = 2 x 2 x 3 x 5

GCD = Multiplication of common factors
= 2 x 2 x 3
= 12

```
// C++ program to demonstrate
// Basic Euclidean Algorithm

#include <bits/stdc++.h>
using namespace std;

// Function to return
// gcd of a and b
int gcd(int a, int b)
{
    if (a == 0)
        return b;
    return gcd(b % a, a);
}

// Driver Code
int main()
{
    int a = 10, b = 15;

      // Function call
    cout << "GCD(" << a << ", " << b << ") = " << gcd(a, b)
         << endl;
    a = 35, b = 10;
    cout << "GCD(" << a << ", " << b << ") = " << gcd(a, b)
         << endl;
    a = 31, b = 2;
    cout << "GCD(" << a << ", " << b << ") = " << gcd(a, b)
         << endl;
    return 0;
}
```

Exercise

- 1. เขียนโปรแกรมหา GCD ด้วยวิธี Euclid
- 2. เขียนโปรแกรม Quick select
  - ด้วยขั้นตอนวิธีที่กำหนดให้
  - ด้วย STL

Problems

- Decrease by a Constant
  - **Insertion sort**
  - **Topological sorting**
  - **Generating permutations, subsets**
- Decrease by a Constant factor
  - **Binary search**
  - **Russian peasant multiplication**
- Variable-Size-Decrease
  - **Computing median and selection problem.**
  - **Interpolation Search**
  - **Euclid’s algorithm**

Optional

- Solve the problem **706B – Interesting drink**
- https://codeforces.com/problemset/problem/706/B

Fenwick (Binary Indexed) Tree

Fenwick (Binary Indexed) Tree

- Invented by Peter M. Fenwick in 1994.
- The **Fenwick Tree** is a useful data structure for implementing **dynamic cumulative frequency** tables.
- สมมติว่าเรามีตารางความถี่สะสม เราสามารถนำไปคำนวณ Range Sum Query (RSQ) ได้
  - เช่น
    - RSQ(1, 3) = **1** จำนวนคนสะสมที่ได้คะแนนตั้งแต่ 1 ถึง 2 คะแนน มีทั้งหมด **1** คน
    - RSQ(1, 6) = **7** จำนวนคนสะสมที่ได้คะแนนตั้งแต่ 1 ถึง 6 คะแนน มีทั้งหมด **7** คน
    - RSQ**(4, 6)** = จำนวนคนสะสมที่ได้คะแนนตั้งแต่ 4 ถึง 6 คะแนน มีทั้งหมด 6 คน
      - คำนวณจาก **RSQ(4, 6)** = **RSQ(1, 6) - RSQ(1, 3)** = 7 - 1 = 6

Index/ Score	Frequency f	Cumulative Frequency cf	Short Comment
0	-	-	Index 0 is ignored (as the sentinel value).
1	0	0	`cf[1] = f[1] = 0`, base case.
2	1	1	`cf[2] = cf[1]+f[2] = 0+1 = 1`.
3	0	1	`cf[3] = cf[2]+f[3] = 1+0 = 1`.
4	1	2	`cf[4] = cf[3]+f[4] = 1+1 = 2`.
5	2	4	`cf[5] = cf[4]+f[5] = 2+2 = 4`.
6	3	7	`cf[6] = cf[5]+f[6] = 4+3 = 7`.
7	2	9	`cf[7] = cf[6]+f[7] = 7+2 = 9`.
8	1	10	`cf[8] = cf[7]+f[8] = 9+1 = 10`.
9	1	11	`cf[9] = cf[8]+f[9] = 10+1 = 11`.
$10 = m$	0	$11 = n$	`cf[10] = cf[9]+f[10] = 11+0 = 11`.

Table 2.5: Example of a Cumulative Frequency Table

inclusion-exclusion principle

Fenwick (Binary Indexed) Tree

- ถ้าความถี่ (f[i] ใด ๆ) ไม่มีการเปลี่ยนแปลงเราสามารถคำนวณ **ตารางความถี่สะสม** ได้ใน O(m)
  - โดย cf[1] = f[1] และ $\forall i \in [2 \ldots m]$; cf[i] = cf[i-1] + f[i]
- **แต่ถ้าความถี่มีการเปลี่ยนแปลง (เพิ่ม/ลด/เปลี่ยนแปลงไปที่ค่าหนึ่ง) และ RSQ ถูกเรียกหลังจากนั้น เราควรต้องมีโครงสร้างข้อมูลที่เหมาะสมกว่า Static Array**

Fenwick (Binary Indexed) Tree

cout << 50 - (50&-(50)) << endl; 50 = $(110010)_2$
cout << 48 - (48&-(48)) << endl; 48 = $(110000)_2$
cout << 32 - (32&-(32)) << endl; 32 = $(100000)_2$
cout << 32 - (32&-(32)) << endl; 0 = $(000000)_2$

- **`LSOne(S)`** คือ **`((S) & -(S))`** ที่คำนวณ **First Least Significant One-bit** ใน **S**
  - `LSOne(90) = LSOne((1011010)`$_2$`) = (10)`$_2$` = 2.`
  - `LSOne(91) = LSOne((1011011)`$_2$`) = (1)`$_2$` = 1.`
- The Fenwick Tree is typically implemented as an array (<vector>).
  - The Fenwick Tree is a tree that is indexed by **the bits of its integer keys**.
  - These integer keys fall within the **fixed range [1..m]**—skipping index 0.
  - **m** can be 1M covering range [1…1M]
  - ในตารางตัวอย่าง integer keys คือ [1…m] และ m=10 โดยมีทั้งหมด 11 data points
- Fenwick Tree
  - Let **`ft`** be Fenwick Tree array.
  - The item at index **`i`** of Fenwick Tree **`ft`** is responsible for items in the range **`[(i-LSOne(i)+1)..i]`** of the frequency array **`f`**
  - **`ft[i]`** stores the cumulative frequency of items **`{i-LSOne(i)+1, i-LSOne(i)+2, i-LSOne(i)+3, .., i}`** of **`f`**

Fenwick (Binary Indexed) Tree

Operation: $O(\log m)$ rsq(j)

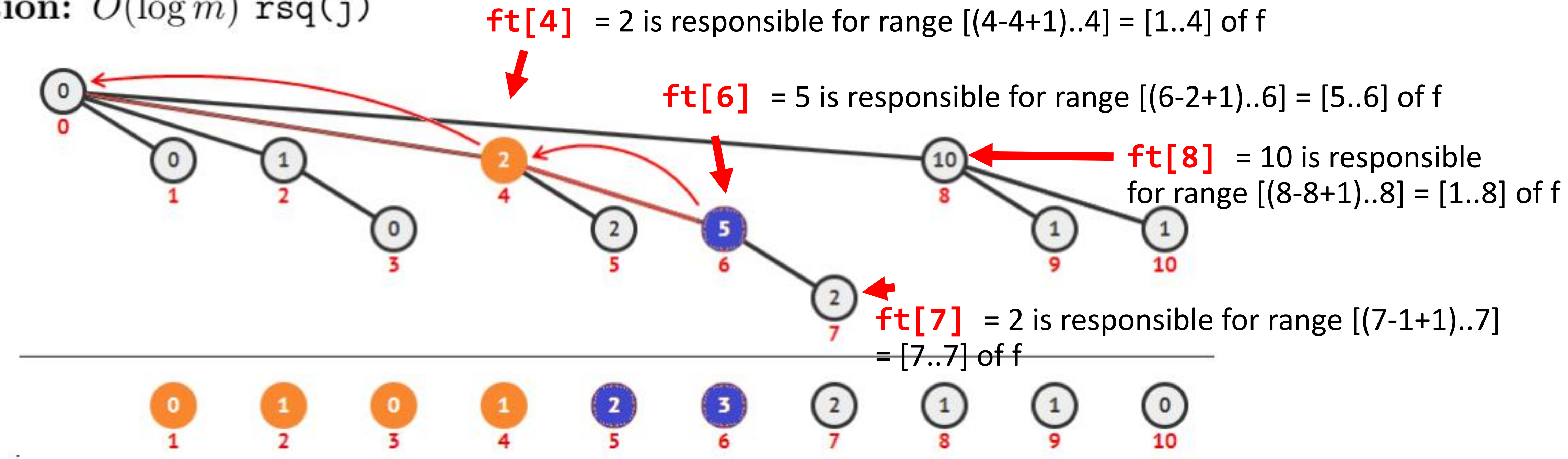


Figure 2.12: Example of rsq(6) on Fenwick (Interrogation/Query) Tree

- The value of **`ft[i]`** is shown inside the circle above index `i`.
- The range **`[i-LSOne(i)+1..i]`** is shown by the highlighted ranges

Fenwick (Binary Indexed) Tree

- If we want to obtain the cumulative frequency between [1..j]: rsq(j)
  - we simply add `ft[j]`, `ft[j’]`, `ft[j’’]`, . . . until index **`j` is 0**.
  - This sequence of indices is obtained via subtracting the **Least Significant One-bit** via the bit manipulation expression: `j’= j-LSOne(j)`
  - Iteration of this bit manipulation effectively *strips off* the least significant one-bit of `j` at each step.
  - As an integer `j` only has `O(log j)` bits, **rsq(j)** runs in **O(log m)** time when j = m
    - เช่น Unsigned Integer 0 to 4294967295
    - ประกอบไปด้วย 32 bits (Log_2 ???)

Fenwick (Binary Indexed) Tree

- Operation: **O(log m) rsq(i, j)**
  - To compute rsq(4, 6)
    - we can simply return rsq(6)-rsq(3) = (5+2)-(0+1) = 7-1 = 6.
    - this operation runs in O(2 x log j) ≈ O(log m) time when j = m.
- Operation: **O(log m) update(i, v)**
  - When updating the value of the item at index `i` by **adding its value by v** (note that v can be either positive or negative), i.e., by calling **`update(i, v)`**, we have to update `ft[i]`, `ft[i']`, `ft[i'']`,… until this index exceeds `m` because all these indices are affected.
    - `i' = i+LSOne(i)`
    - The operation **`update(i, v)`** will take at most O(log m) steps until $i > m$ even if $i = 1$ at the beginning.

Fenwick (Binary Indexed) Tree

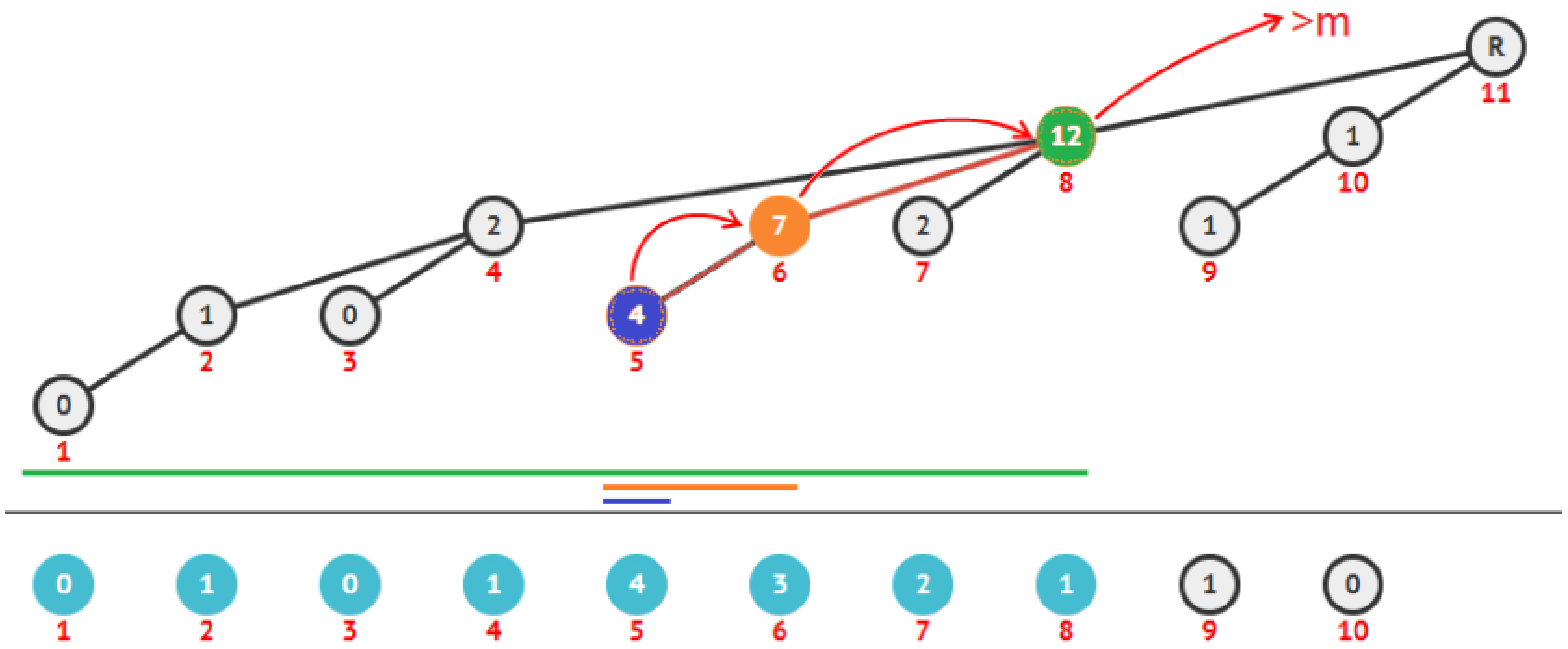


Figure 2.14: Example of `update(5, 2)` on Fenwick (Updating) Tree

update(5, 2) will affect (add +2 to) ft at indices i = 5_{10} = $(101)_2$, i' = $(101)_2$ + $(001)_2$ = $(110)_2$ = 6_{10}, and i'' = $(110)_2$ + $(010)_2$ = $(1000)_2$ = 8_{10} via the expression given above.

Implementation

Basic Implementation

- This basic version assumes that the keys are integers within range [1..m].

```
#define LSOne(S) ((S) & -(S))                    // the key operation

typedef vector<int> vi;

class FenwickTree {                              // index 0 is not used
private:
  vi ft;
public:
  FenwickTree(int m) { ft.assign(m+1, 0); }      // create empty FT
  int rsq(int j) {                               // returns RSQ(1, j)
    int sum = 0;
    for (; j; j -= LSOne(j))
      sum += ft[j];
    return sum;
  }
  int rsq(int i, int j) { return rsq(j) - rsq(i-1); } // inc/exclusion
  // updates value of the i-th element by v (v can be +ve/inc or -ve/dec)
  void update(int i, int v) {
    for (; i < (int)ft.size(); i += LSOne(i))
      ft[i] += v;
  }
};
```

Operation: $O(n + m)$ build(frequency-array f)

O(n + m) build(frequency-array f)

- We can build Fenwick Tree from an array of raw data that contains *n* items,
- do one linear $O(n)$ pass to create an array of frequencies with *m* keys/integer indices,
- and then call `update(i, f[i])`.

Improved:

- After having the array of frequencies that have m keys/integer indices,
- we simply set `ft[i] += f[i]` and then check if its parent in the updating tree of Fenwick Tree is still within range.
    - If it is, we update its parent too.
- This build is slightly **faster**, i.e., in O(n + m) operations as we only do the necessary updating work.

Basic Implementation

```
void build(const vll &f) {
  int m = (int)f.size()-1;                          // note f[0] is always 0
  ft.assign(m+1, 0);
  for (int i = 1; i <= m; ++i) {                    // O(m)
    ft[i] += f[i];                                  // add this value
    if (i+LSOne(i) <= m)                            // i has parent
      ft[i+LSOne(i)] += ft[i];                      // add to that parent
  }
}
```

Fenwick (Binary Indexed) Tree

- $O(\log^2 m)$ **select(rank k)**
  - Find the smallest index/key `i` so that the cumulative frequency in the range [1..i] >= k.
  - Example: there are at least k = 7 students covered in the range [1..i].
    - index/score = 6 in this case.

```
int select(ll k) {                          // O(log^2 m)
  int lo = 1, hi = ft.size()-1;
  for (int i = 0; i < 30; ++i) {            // 2^30 > 10^9; usually ok
    int mid = (lo+hi) / 2;                  // BSTA
    (rsq(1, mid) < k) ? lo = mid : hi = mid; // See Section 3.3.1
  }
  return hi;
}
```

Exercise

- ให้นักเรียนเขียนโปรแกรมเพื่อสร้าง Fenwick (Binary Indexed) Tree
- ใช้ข้อมูล test ในตัวอย่างที่เรียน

Interesting Question

- https://www.geeksforgeeks.org/counting-triangles-in-a-rectangular-space-using-2d-bit/

References

- Introduction to the Design and Analysis of Algorithms (3rd Edition)
- Competitive Programming 3 by Steven Halim