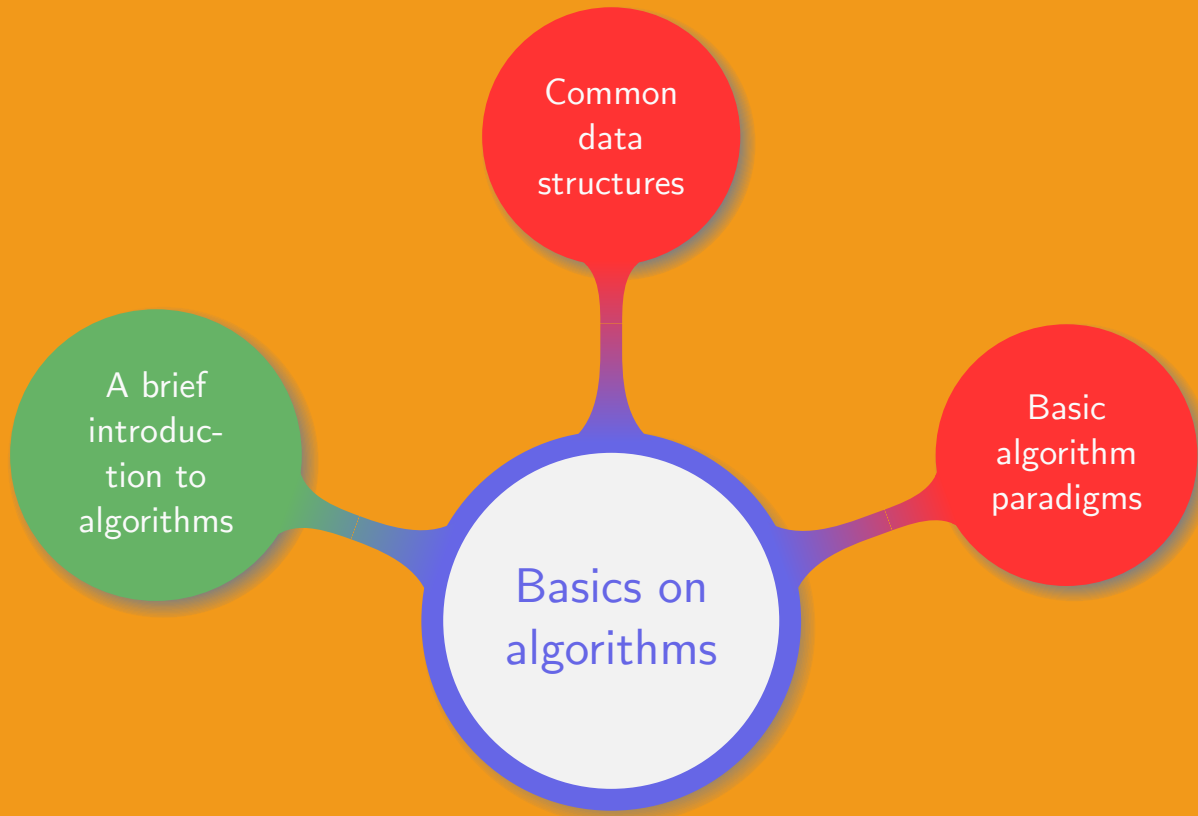


Introduction to Algorithms

Manuel – Fall 2020



An Algorithm is a recipe telling the computer how to solve a problem

An Algorithm is a recipe telling the computer how to solve a problem

Example. Detail an algorithm to prepare a jam sandwich

Actions: cut, listen, spread, sleep, read, take, eat, dip

Things: knife, guitar, bread, honey, jam jar, sword

An Algorithm is a recipe telling the computer how to solve a problem

Example. Detail an algorithm to prepare a jam sandwich

Actions: cut, listen, spread, sleep, read, take, eat, dip

Things: knife, guitar, bread, honey, jam jar, sword

Algorithm. (*Sandwich making*)

Input : 1 bread, 1 jamjar, 1 knife

Output: 1 jam sandwich

- 1 take the knife and cut 2 slices of bread;
 - 2 dip the knife into the jamjar;
 - 3 spread the jam on the bread, using the knife;
 - 4 assemble the 2 slices together, jam on the inside;
-

jam 不夠多}

An algorithm systematically solves a well-defined problem:

- The *Input* is clearly expressed
- The *Output* solves the problem
- The *Algorithm* provides a precise step-by-step procedure starting from the Input and leading to the Output

An algorithm systematically solves a *well-defined* problem:

- The *Input* is clearly expressed
- The *Output* solves the problem
- The *Algorithm* provides a precise step-by-step procedure starting from the Input and leading to the Output

Algorithms can be described using one of the three following ways:

- English
- Pseudocode 考试中、
- Programming language

Algorithm. (*Insertion Sort*)

Input : a_1, \dots, a_n , n unsorted elements

Output: the a_i , $1 \leq i \leq n$, in increasing order

```
1 for  $j \leftarrow 2$  to  $n$  do
2    $i \leftarrow 1$ ;
3   while  $a_j > a_i$  do  $i \leftarrow i + 1$ ;
4    $m \leftarrow a_j$ ;
5   for  $k \leftarrow 0$  to  $j - i - 1$  do  $a_{j-k} \leftarrow a_{j-k-1}$ ;
6    $a_i \leftarrow m$ 
7 end for
8 return  $(a_1, \dots, a_n)$ 
```

Example. A robot arm solders chips on a board in n contact points. We want to minimize the time to attach a chip to the board, knowing that

- The arm moves at constant speed;
- Once a chip has been attached another one is soldered;

Example. A robot arm solders chips on a board in n contact points. We want to minimize the time to attach a chip to the board, knowing that

- The arm moves at constant speed;
- Once a chip has been attached another one is soldered;

Defining the Input and Output:

- Input: a set S of n points in the plane
- Output: the shortest path visiting all the points in S

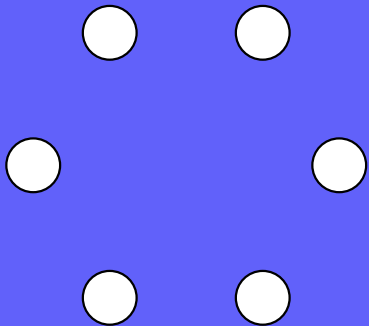
Algorithm. (*Nearest neighbor*)

Input : a set $S = \{s_0, \dots, s_{n-1}\}$ of n points in the plane

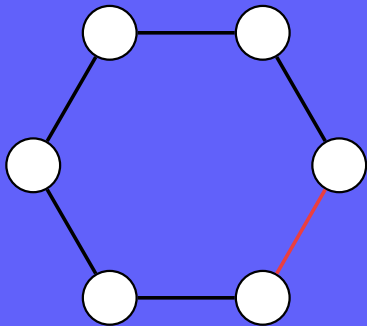
Output: the shortest cycle visiting all the points in S

```
1  $p_0 \leftarrow s_0$ ;  
2 for  $i \leftarrow 1$  to  $n - 1$  do  
3   |  $p_i \leftarrow$  closest unvisited neighbor to  $p_{i-1}$ ;  
4   | Visit  $p_i$ ;  
5 end for  
6 return  $\langle p_0, \dots, p_{n-1} \rangle$ 
```

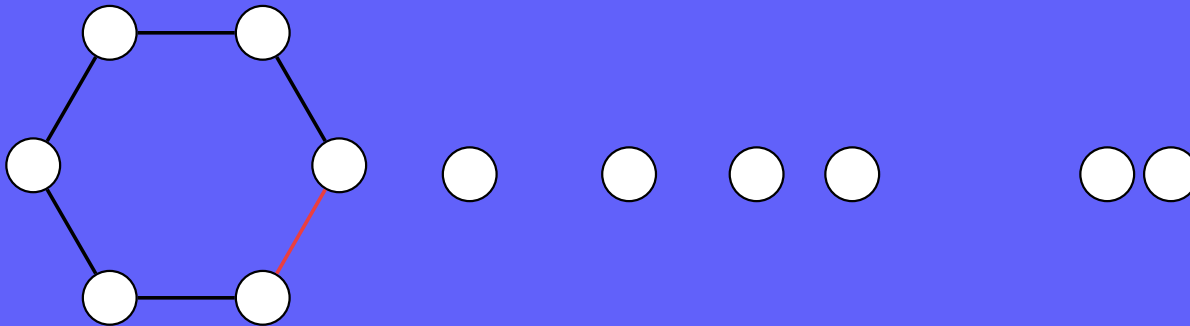
How does the nearest neighbor algorithm (1.7) perform in the following three cases?



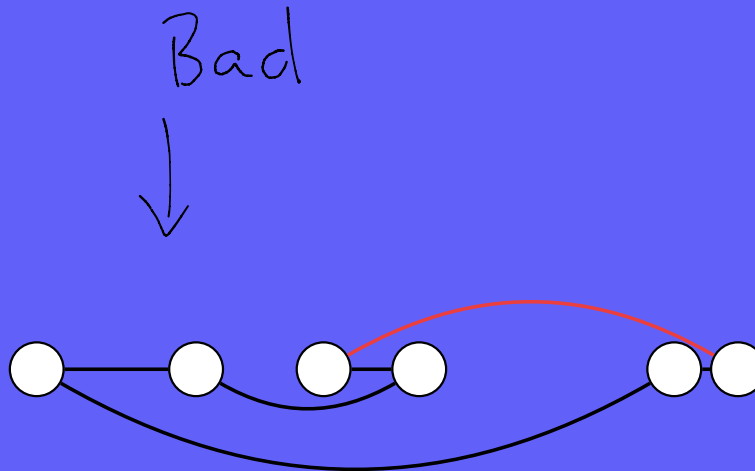
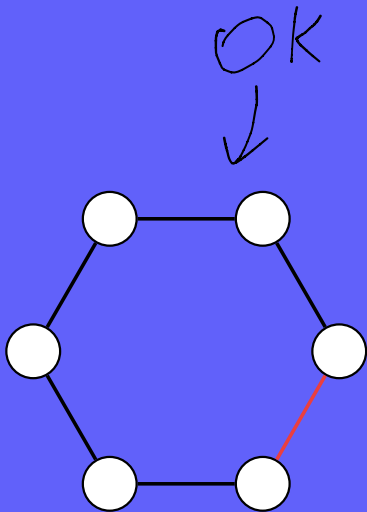
How does the nearest neighbor algorithm (1.7) perform in the following three cases?



How does the nearest neighbor algorithm (1.7) perform in the following three cases?

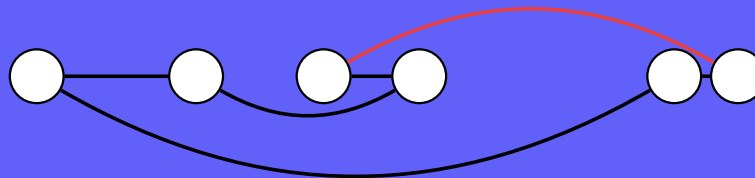
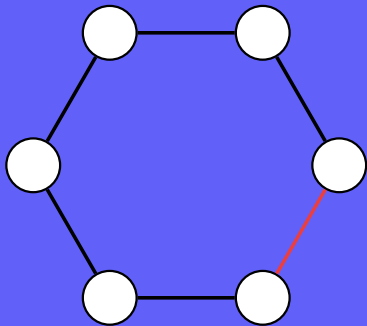


How does the nearest neighbor algorithm (1.7) perform in the following three cases?

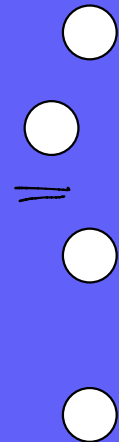


↓
換 starting point
(1st Bad)

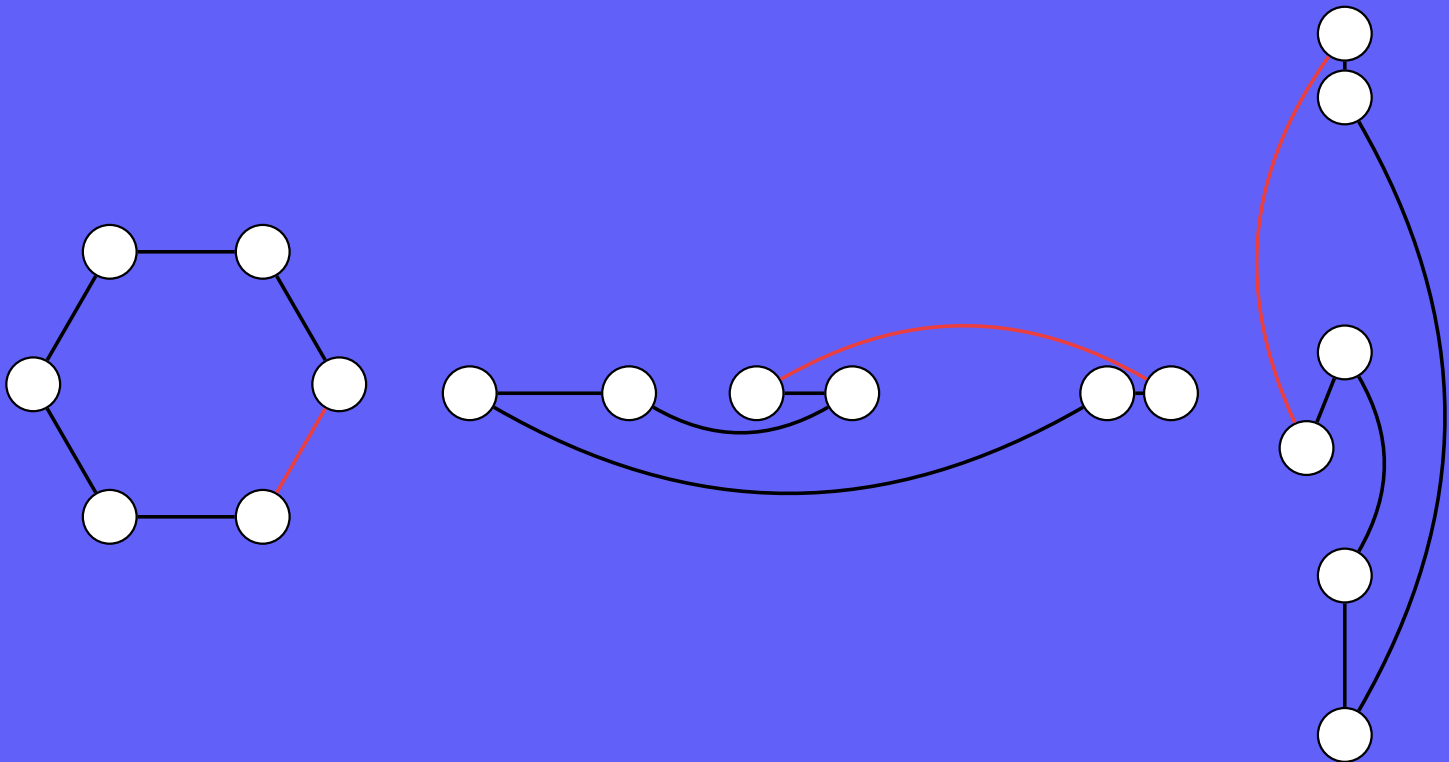
How does the nearest neighbor algorithm (1.7) perform in the following three cases?



见此
(选最左没用)



How does the nearest neighbor algorithm (1.7) perform in the following three cases?



Algorithm. (*Closest pair*)

Input : a set S of n points in the plane

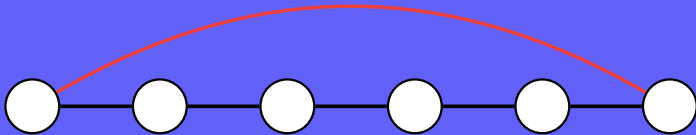
Output: the shortest cycle visiting all the points in S

```
1 for  $i \leftarrow 1$  to  $n - 1$  do
2    $d \leftarrow \infty$ ;
3   foreach pair of end points  $\langle s, t \rangle$  from distinct vertex chains do
4     if  $\text{dist}(s, t) \leq d$  then
5        $s_m \leftarrow s$ ;  $t_m \leftarrow t$ ;  $d \leftarrow \text{dist}(s, t)$ ;
6     end if
7   end foreach
8   Connect  $s_m$  and  $t_m$  by an edge;
9 end for
10 return all the points starting from one of the two end points
```

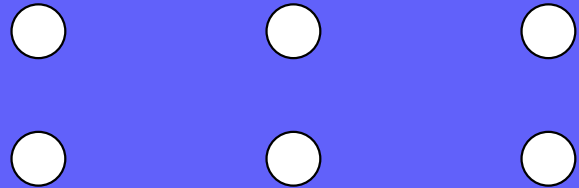
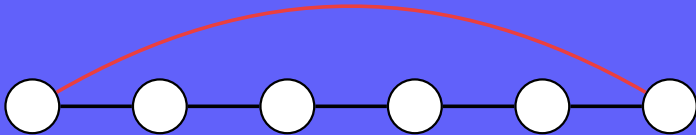
Applying the closest pair algorithm (1.9) to the following vertices arrangement yields the two graphs:



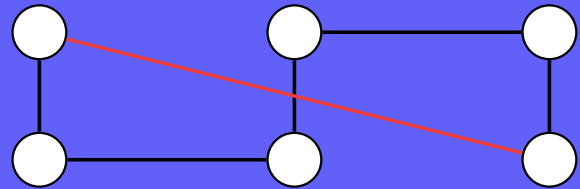
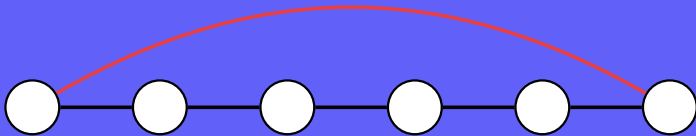
Applying the closest pair algorithm (1.9) to the following vertices arrangement yields the two graphs:



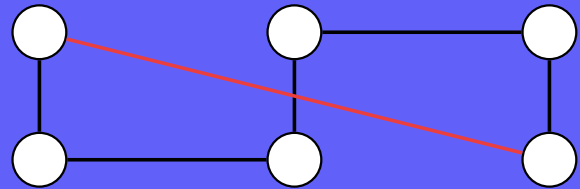
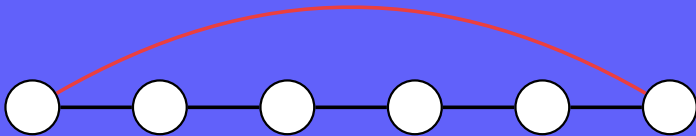
Applying the closest pair algorithm (1.9) to the following vertices arrangement yields the two graphs:



Applying the closest pair algorithm (1.9) to the following vertices arrangement yields the two graphs:



Applying the closest pair algorithm (1.9) to the following vertices arrangement yields the two graphs:



Possible strategy to ensure the most optimal path:

- Enumerate all the possible paths
- Select the one that minimizes the total length

For only 20 vertices $20!$ paths have to be explored

A difference:

- Algorithm: always output a correct result
- **Heuristic**: idea serving as a guide to solve a problem with no guarantee of always providing a good solution

Correctness and efficiency:

- An algorithm working on a set of input does not imply it will work on all instances
- Efficient algorithm totally solving a problem might not exist

Common traps when defining the Input and Output:

- Are they **precise** enough?
- Can all the Input be easily and efficiently generated?
- Could there be any confusion on the expected Output?

Example. For an Output, what does it mean to “find the best route”?

Common traps when defining the Input and Output:

- Are they precise enough?
- Can all the Input be easily and efficiently generated?
- Could there be any confusion on the expected Output?

Example. For an Output, what does it mean to “find the best route”?
The shortest in distance, the fastest in time, or the one minimizing the number of turns?

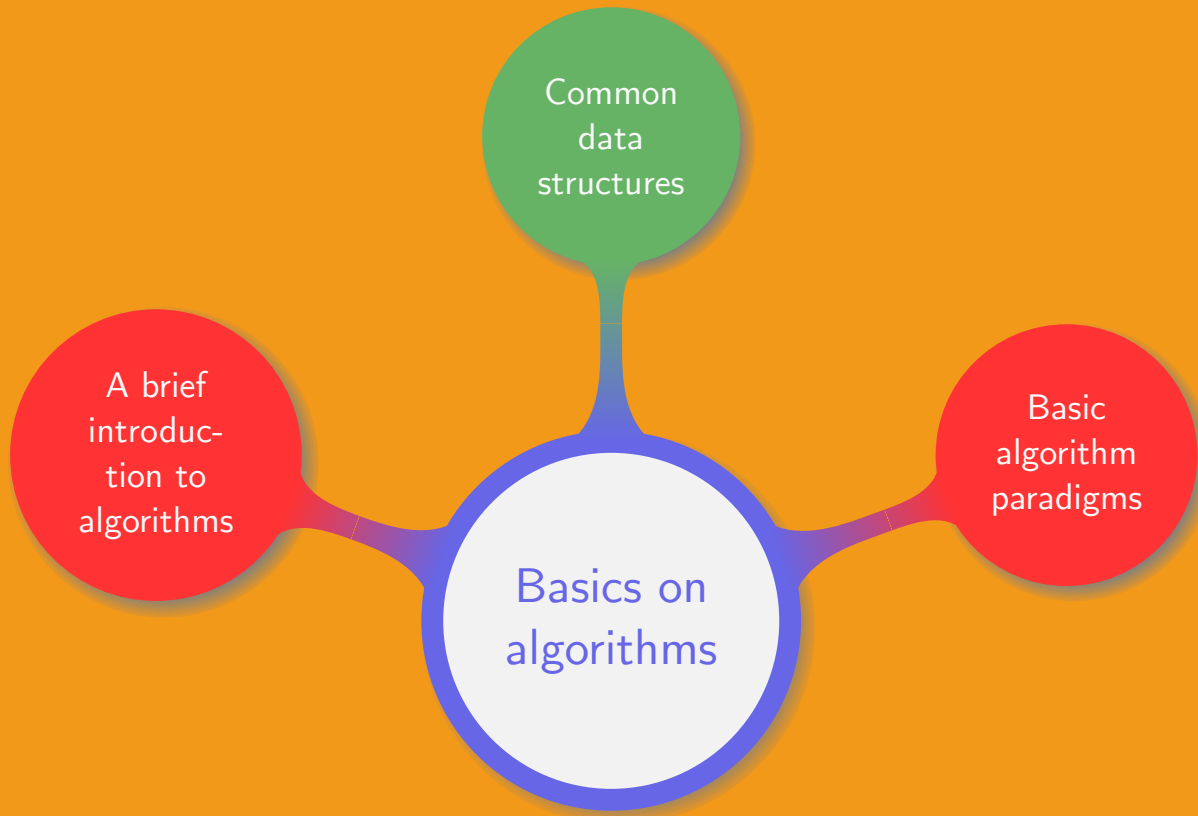
Input and Output must be expressed in simple, precise, and clear terms

考前周全

Finding good counter-examples:

- Seek simplicity: make it clear why the algorithm fails
- Think small: algorithms failing for large Input often fail for smaller one
- Test the extremes: study special cases, e.g. inputs equal, tiny, huge...
- Think exhaustively: test whether all the possible cases are covered by the algorithm
- Track weaknesses: check if the underlying idea behind the algorithm has any “unexpected” impact on the output

VE471 数值分析 numerical analysis



Data structures can be split into two main categories:

- **Continuous:** a single piece of memory, e.g. array, matrices, hash tables
- **Linked** data structures: distinct chunks of memory connected together, e.g. linked list, trees, graph adjacency lists

Choosing an appropriate data structure is of a major importance

OCaml

虽然有loop/array, 但FP中其实没有

Each element can be efficiently located using its index:

- Constant access time: each index maps to a memory address
- Space efficiency: no space wasted with links or information on the data
- Memory locality: data is contiguous so *cache* can be used to speed up successive data accesses

*The **size cannot be easily adjusted** during the program's execution*

A linked structure is composed of nodes. Each one contains:

- One or more fields on data
- A pointer to at least another node

The most common operations are:

- Search: find an item in the list
- Insert: add an item to the list
- Delete: remove an item from the list

Search can be implemented either iteratively or recursively

Linked structure

- Overflow only occurs when memory is full
- Insertion/deletion are simple and fast
- Moving pointers is faster than moving the actual data

Array

- No extra space wasted for the pointer field
- Efficient random access is possible
- Better memory locality and cache performance

Common data structures allowing the storage and retrieval of data independently of the content:

- Stack:
 - LIFO order
 - Simple to implement and very efficient
- Queue:
 - FIFO order
 - Minimize the maximum waiting time
 - Trickier to implement than stacks

Both can be implemented using either linked lists or arrays

Data type allowing access by content. Primary operations:

- Search: search a value in a given dictionary
- Insert: add an element to the dictionary
- Delete: remove an element from the dictionary

Most common operations:

- Max/Min: retrieve the largest/smallest element from the dictionary
- Predecessor/Successor: retrieve the element just before/after a given element; before/after are defined with respect to a sorted order

Let n be the number of elements in the array

Operation	Unsorted array	Sorted array
search(D,k)	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
insert(D,k)	$\mathcal{O}(1)$	$\mathcal{O}(n)$
delete(D,k)	$\mathcal{O}(1)^*$ ☆	$\mathcal{O}(n)$
predecessor(D,k)	$\mathcal{O}(n)$	$\mathcal{O}(1)$
successor(D,k)	$\mathcal{O}(n)$	$\mathcal{O}(1)$
minimum(D)	$\mathcal{O}(n)$	$\mathcal{O}(1)$
maximum(D)	$\mathcal{O}(n)$	$\mathcal{O}(1)$

* Assuming a pointer to the key k is given how to get $\mathcal{O}(1)$?

先删除，再和最后一个换一下。

Dictionary using linked structures

Let n be the number of elements in the list

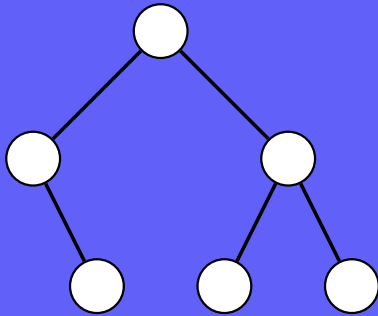
Operation	Unsorted		Sorted	
	Single	Double	Single	Double
search(D,k)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
insert(D,k)	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
delete(D,k)	$\mathcal{O}(n)^*$	$\mathcal{O}(1)$	$\mathcal{O}(n)^*$	$\mathcal{O}(1)$
predecessor(D,k)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)^*$	$\mathcal{O}(1)$
successor(D,k)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
minimum(D)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
maximum(D)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)^\dagger$	$\mathcal{O}(1)$

* Why are singly linked lists slower?

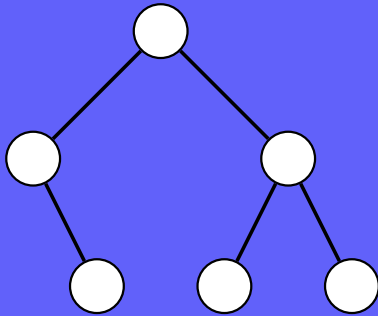
单向.

† **How** to achieve $\mathcal{O}(1)$ for singly sorted lists?

加一个结构.

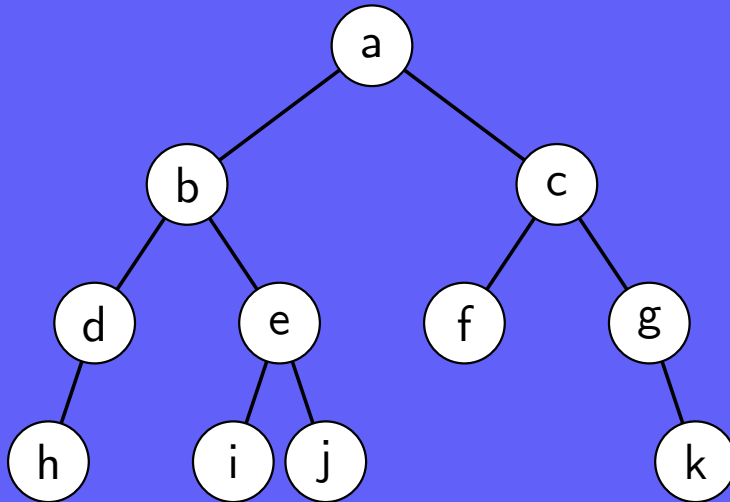


- Based on doubly linked lists
- First object is the root of the tree
- Second object is a left child if it precedes the root and a right child if it succeeds it
- Third and further object are sorted along the tree following a similar pattern



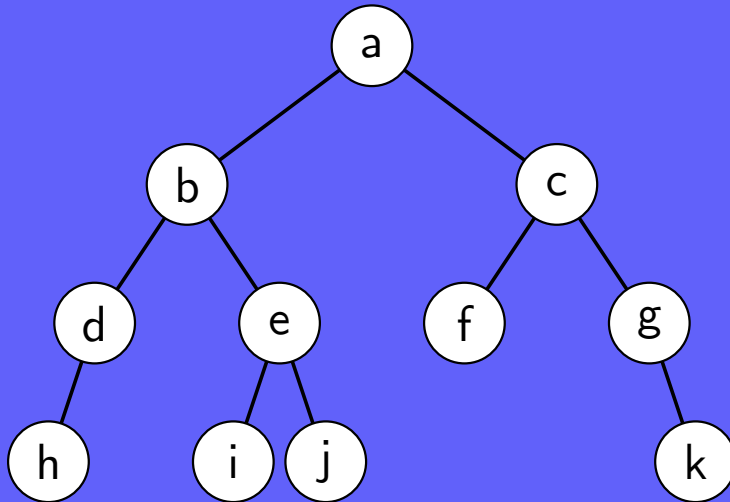
- Based on doubly linked lists
 - First object is the root of the tree
 - Second object is a left child if it precedes the root and a right child if it succeeds it
 - Third and further object are sorted along the tree following a similar pattern
-
- The three primary dictionary operations take $\mathcal{O}(h)$, with h the height of the tree
 - Binary search trees balance the search time and flexible update 否则会太偏
 - How to handle deletion? 见VE281详细

Binary search tree traversal



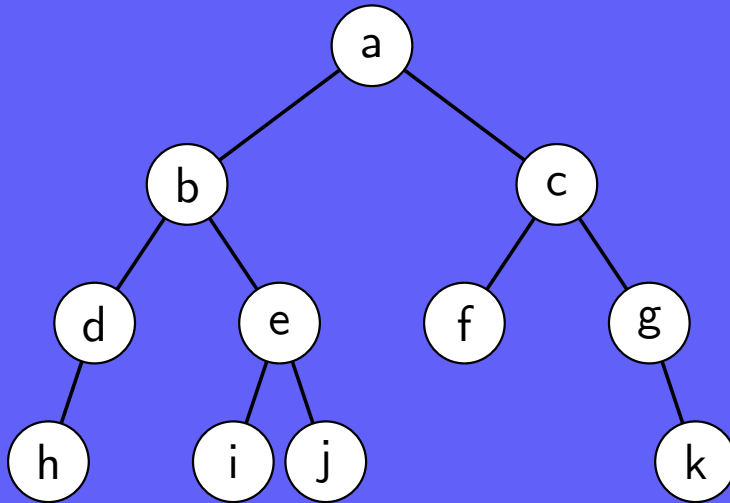
- Preorder traversal:

Binary search tree traversal



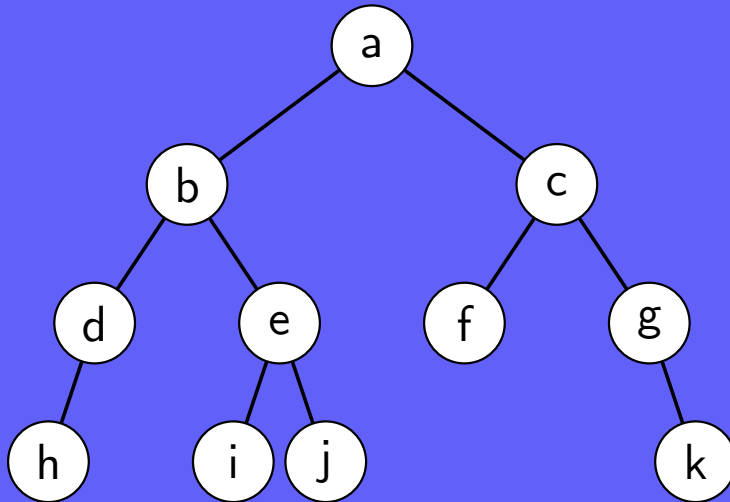
- Preorder traversal:
a, b, d, h, e, i, j, c, f, g, k
- Inorder traversal:

Binary search tree traversal



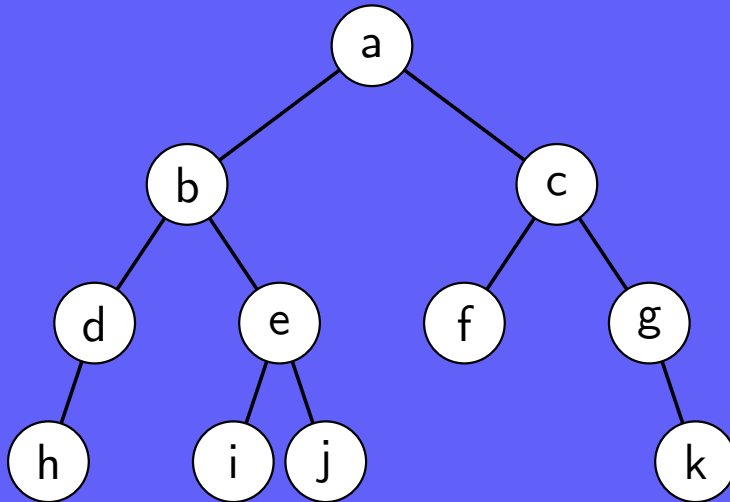
- Preorder traversal:
a, b, d, h, e, i, j, c, f, g, k
- Inorder traversal:
h, d, b, i, e, j, a, f, c, g, k
- Postorder traversal:

Binary search tree traversal



- Preorder traversal:
a, b, d, h, e, i, j, c, f, g, k
- Inorder traversal:
h, d, b, i, e, j, a, f, c, g, k
- Postorder traversal:
h, d, i, j, e, b, f, k, g, c, a

Binary search tree traversal



- Preorder traversal:
a, b, d, h, e, i, j, c, f, g, k
- Inorder traversal:
h, d, b, i, e, j, a, f, c, g, k
- Postorder traversal:
h, d, i, j, e, b, f, k, g, c, a

How to implement inorder tree traversal?

田 recursion.

Primary operations for priority queues:

- Insert: add an element to the queue
- Find min/max: return the last/first element in the queue
- Delete min/max: remove the last/first element in the queue

Operation	Array		Balanced tree
	Unsorted	Sorted	
<code>insert(Q,x)</code>	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
<code>find_min(Q)</code>	$\mathcal{O}(1)^*$	$\mathcal{O}(1)$	$\mathcal{O}(1)^*$
<code>delete_min(Q)</code>	$\mathcal{O}(n)$	$\mathcal{O}(1)^\dagger$	$\mathcal{O}(\log n)$

* How to reach $\mathcal{O}(1)$ for an unsorted array and a balanced tree? ↘

† How to reach $\mathcal{O}(1)$ when deleting the min in a sorted array?

从大到小, 删最后一个、

加一个变量 ↘

Practical way to maintain a dictionary where:

- The data is stored in an array
- Each key is hashed and stored at index “the hash of the key”
- Keys with a similar hash are store in a linked list

Good hash function: **all indices occur with equiprobability**

Practical way to maintain a dictionary where:

- The data is stored in an array
- Each key is hashed and stored at index “the hash of the key”
- Keys with a similar hash are store in a linked list

Good hash function: all indices occur with equiprobability

Example. A common choice is $H(k) = k \bmod m$, with m a prime not too close from a power of 2.

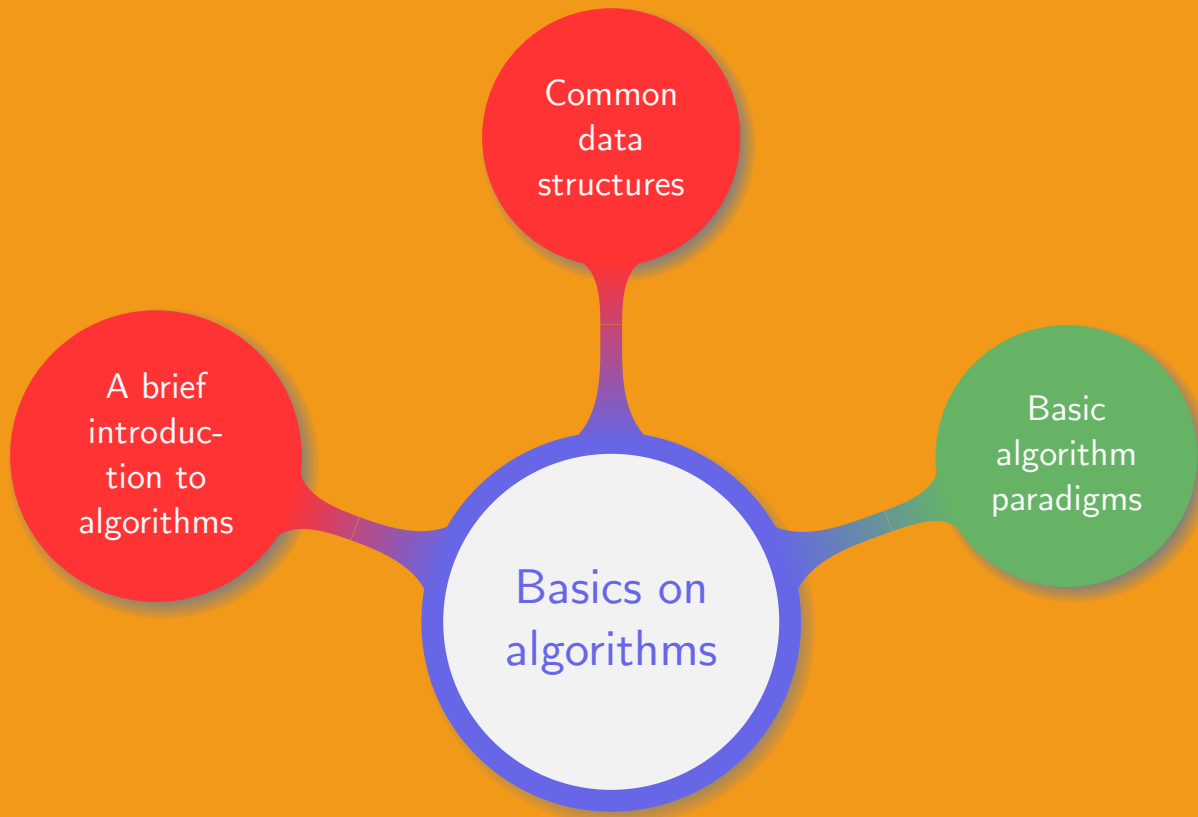
见作业

For $n = 2000$, 701 would be a good choice if one desires to have about three keys stored at each index.

- **Strings:** array of characters; use **suffix trees**/arrays for pattern matching
- **Geometric** element: define regions as polygons using segments and points in an array or a tree
- Graphs: consider the adjacency matrix or an adjacency list; **graph algorithms vary depending on the structure**
- **Sets:** bit vector where the element in the set is the index and the value store is 1 or 0 depending whether the element is in the set; dictionaries can be used for fast membership queries

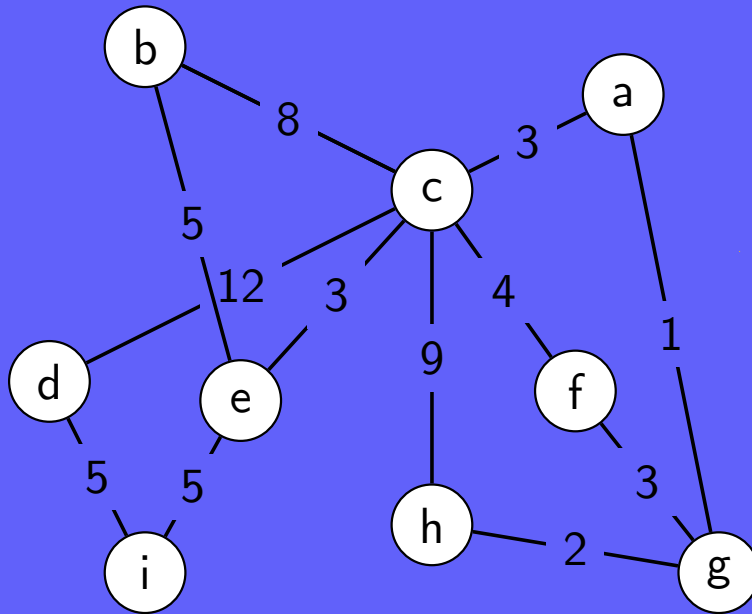
A few points to remember when selecting a data structure:

- Data can be represented in many ways
- No data structure is fast in all aspects
- Choosing the wrong data structure can be disastrous in terms of performance
- Several choices are often possible
- Identifying the best data structure is **often not critical**
- Always aim for clear, simple, and efficient data structures



A first graph problem

MST



Simple problem:

- We have n computers connected by wires
- Using different wires implies different costs

We expect to:

- Connect all the computers to the network
- Minimize the cost

Problem (Minimum Spanning Tree (MST))

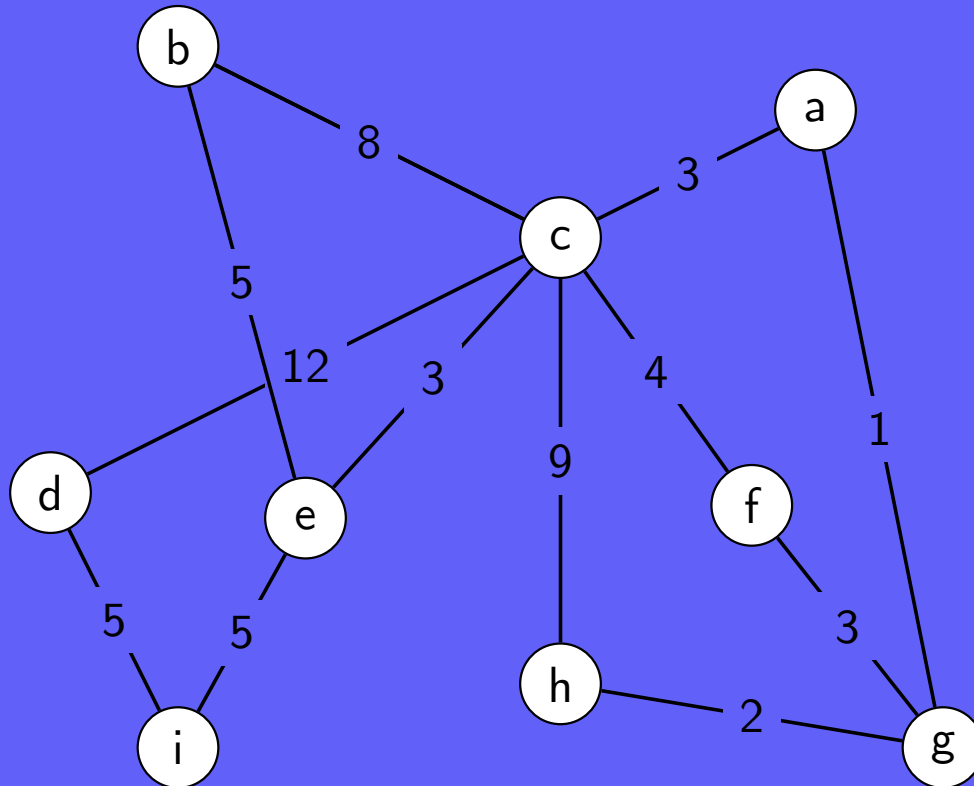
Given a weighted graph G , find a subgraph T such that:

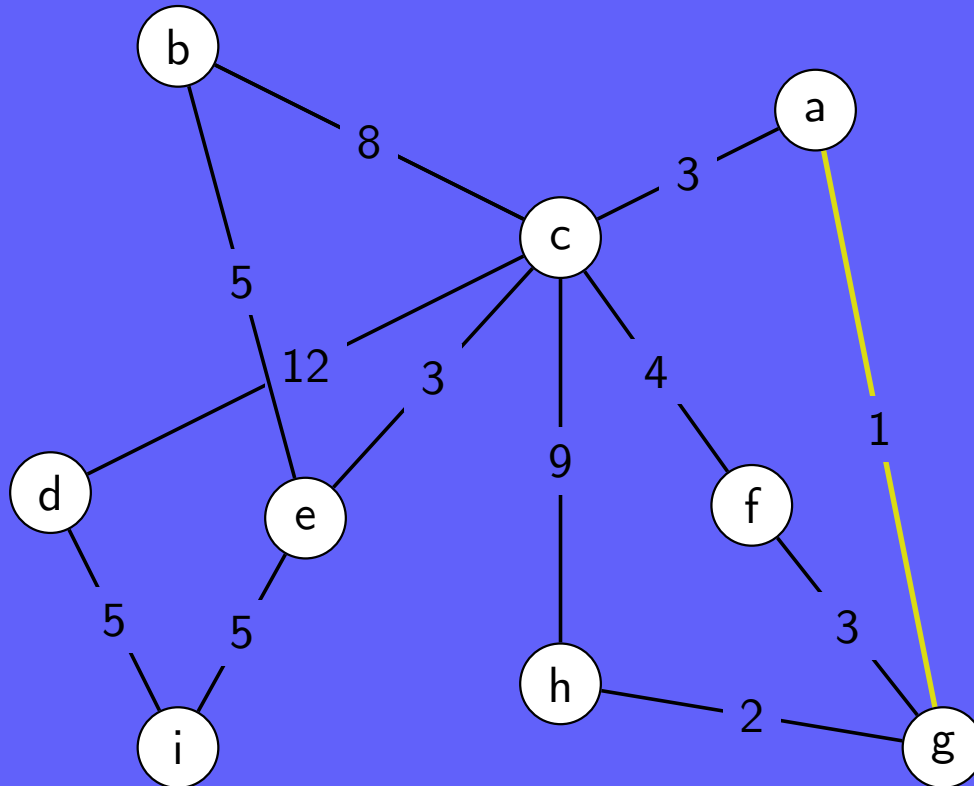
- 1 All the vertices on G are connected on T ,
- 2 The total weight, defined as the sum of the weight of all the edges in T , is minimized.

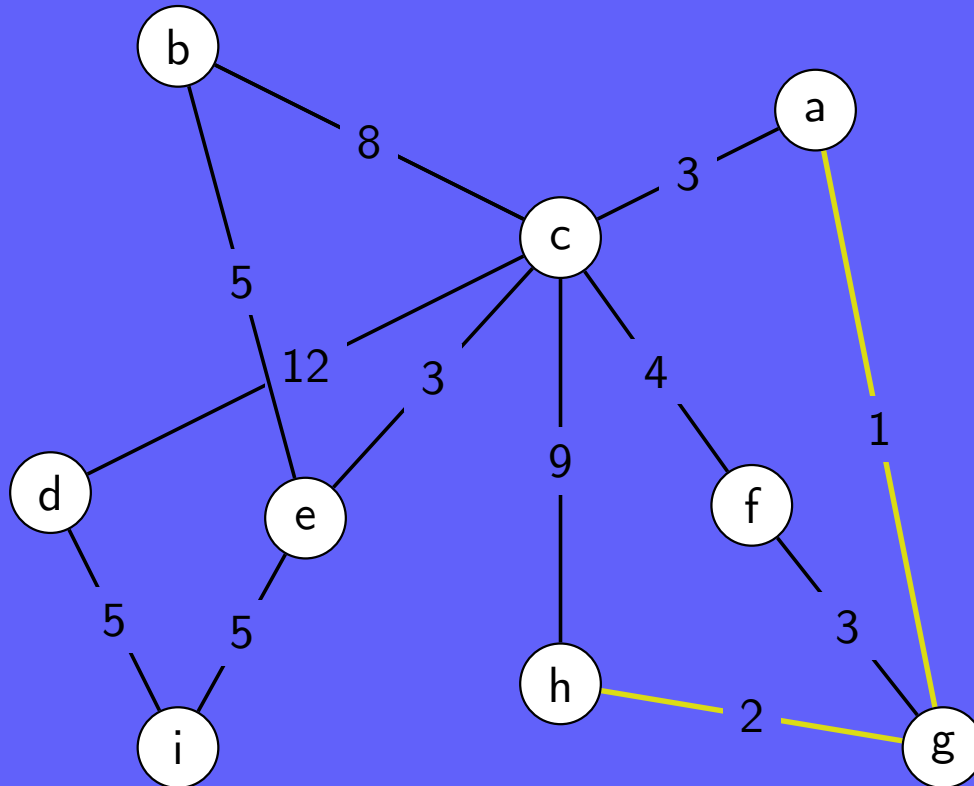
The graph T is a *minimum spanning tree* for G .

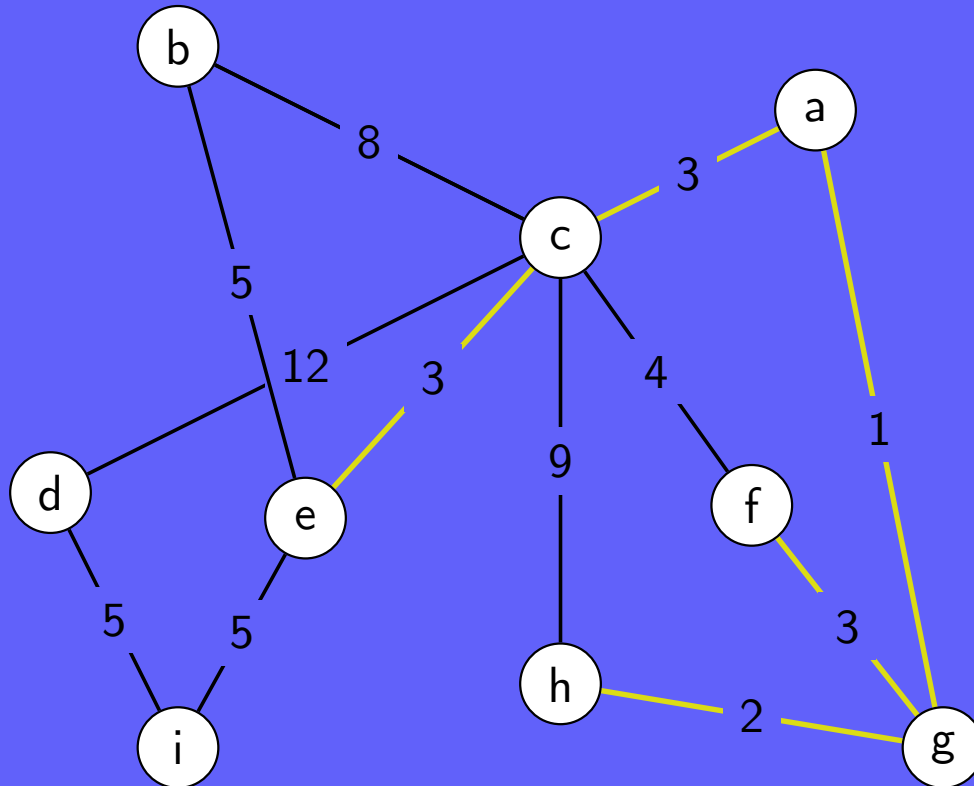
Remark. Note that T is a tree: if it contained a cycle, at least one edge could be removed, allowing a lower weight while preserving the connected property of T .

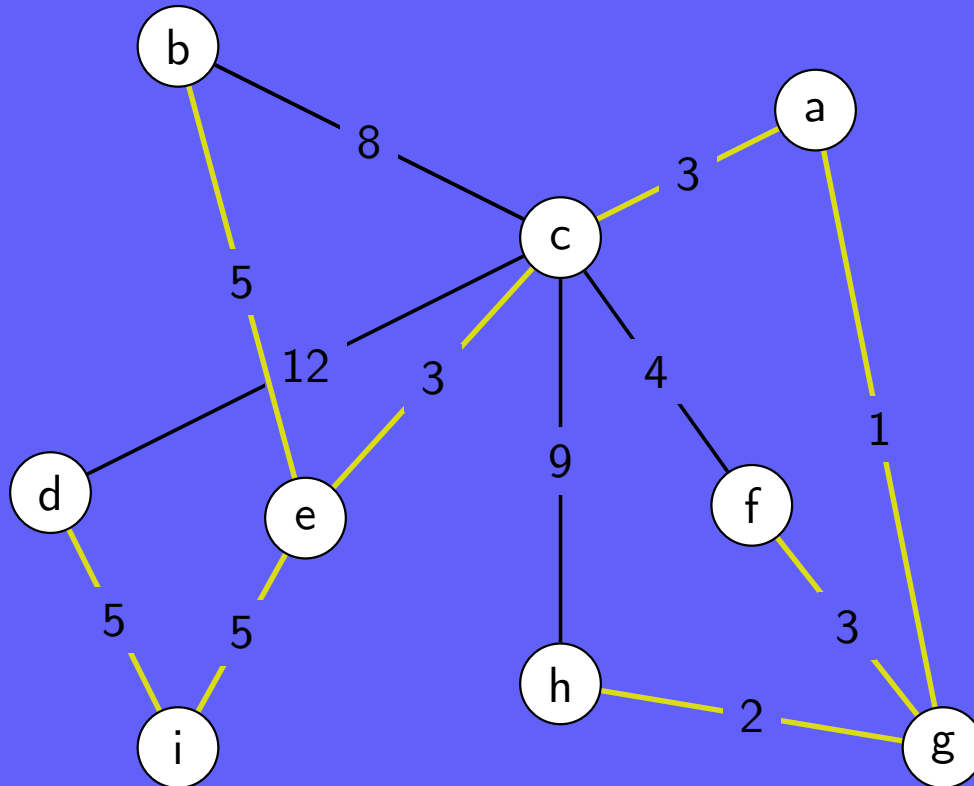
no cycle !











Algorithm. (*Kruskal*)

Input : A graph $G = \langle V, E \rangle$

Output: A minimum spanning tree T for G

```
1 Sort the edges  $G.E$  by weight;
2  $T \leftarrow \emptyset$ ;
3 for edges  $(u, v)$  in  $G.E$ , in non-decreasing order do
4   | if adding  $(u, v)$  does not create a cycle then
5   |   | add edge  $(u, v)$  to  $T$ 
6   | end if
7 end for
8 return  $T$ 
```

Algorithm. (*Kruskal*)

Input : A graph $G = \langle V, E \rangle$

Output: A minimum spanning tree T for G

```
1 Sort the edges  $G.E$  by weight;  
2  $T \leftarrow \emptyset$ ;  
3 for edges  $(u, v)$  in  $G.E$ , in non-decreasing order do  
4   | if adding  $(u, v)$  does not create a cycle then  
5   |   | add edge  $(u, v)$  to  $T$   
6   | end if  
7 end for  
8 return  $T$ 
```

union-find.

What needs to be specified?

Correctness of Kruskal's algorithm

Theorem

Assuming the previous notations, Kruskal's algorithm produces a minimum spanning tree for G .

Proof. Let $G = \langle V, E \rangle$ be a graph and let v and w be two vertices connected by an edge. If S is the set of all the vertices with a path to v before e is added, then $w \notin S$, otherwise this would define a cycle. Moreover if there was an edge with smaller weight than e , connecting S and $V - S$, then it would have already been added. Therefore e is the cheapest edge connecting $V - S$ to S , and as such belongs to a minimum spanning tree of G .

Clearly by design the algorithm will not generate any cycle. Moreover as G is connected and all the edges are explored, $V - S$ and S will be linked at some stage. Hence T is connected. \square

很不严谨的证明 233

Issue: how to represent the data such that whether or not adding an edge creates a cycle can be **efficiently tested**?

For each edge joining two vertices v and w :

- Identify all the connected components of v and w
- If the edge is to be included, merge the two components

Issue: how to represent the data such that whether or not adding an edge creates a cycle can be efficiently tested?

For each edge joining two vertices v and w :

- Identify all the connected components of v and w
- If the edge is to be included, merge the two components

Extra notes:

- No edge needs to be removed
- No component needs to be split
- Everything must be done efficiently

477也就图一乐, 真算法还得 google -

Representing data using:

- An array: testing can be done in constant time; merging requires linear time
- A graph: merging is only adding an edge; testing requires a full graph traversal

见 lab 2 讲解

Representing data using:

- An array: testing can be done in constant time; merging requires linear time
- A graph: merging is only adding an edge; testing requires a full graph traversal

Implement a new data structure containing:

- A pointer to the parent
- The *rank*, or depth, of the sub-tree

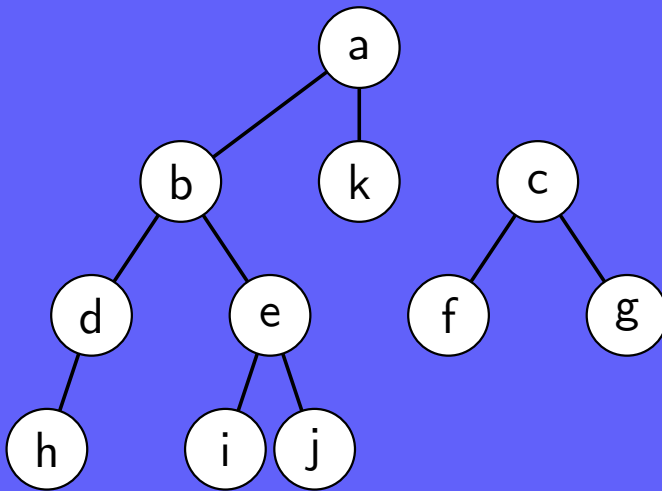
The union-find data structure

The two operations are:

- **Find**(v): find the root of the tree for vertex v
- **Union**(v, w): link the root of the tree containing v to the root of the tree containing w (or the other way around)

Process:

- We have two sub-trees



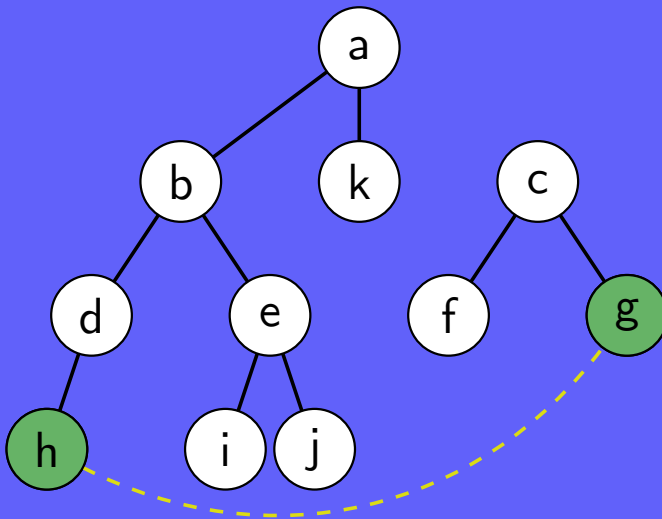
The union-find data structure

The two operations are:

- $\text{Find}(v)$: find the root of the tree for vertex v
- $\text{Union}(v, w)$: link the root of the tree containing v to the root of the tree containing w (or the other way around)

Process:

- We have two sub-trees
- On the graph an edge joins the vertices h and g



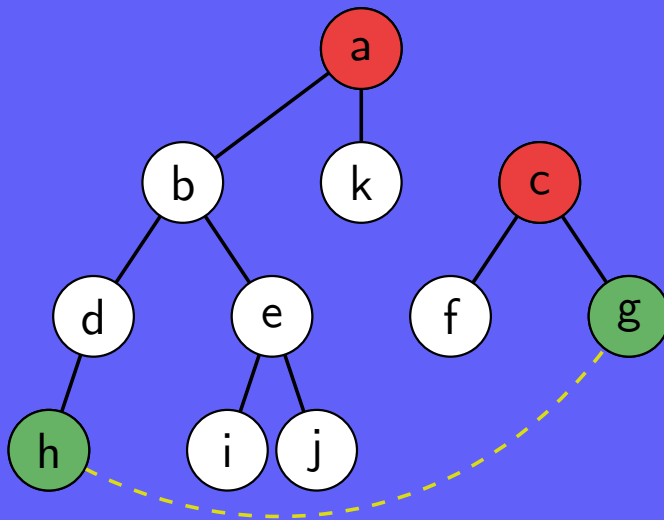
The union-find data structure

The two operations are:

- $\text{Find}(v)$: find the root of the tree for vertex v
- $\text{Union}(v, w)$: link the root of the tree containing v to the root of the tree containing w (or the other way around)

Process:

- We have two sub-trees
- On the graph an edge joins the vertices h and g
- Find on h and g returns a and c , respectively



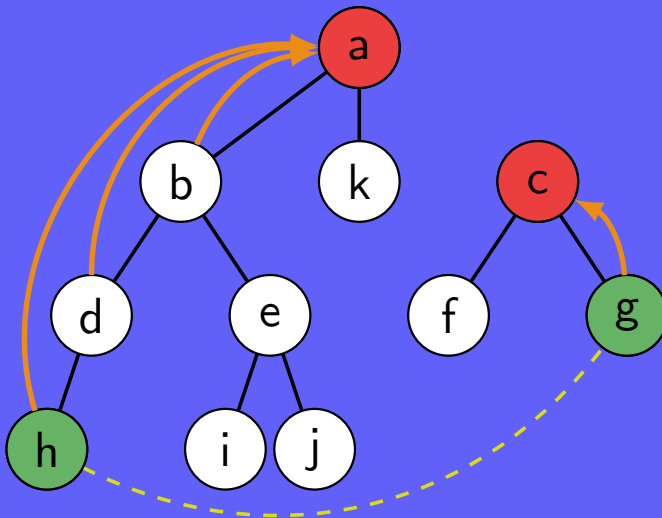
The union-find data structure

The two operations are:

- $\text{Find}(v)$: find the root of the tree for vertex v
- $\text{Union}(v, w)$: link the root of the tree containing v to the root of the tree containing w (or the other way around)

Process:

- We have two sub-trees
- On the graph an edge joins the vertices h and g
- Find on h and g returns a and c , respectively
- Update parents for h, d, b and g



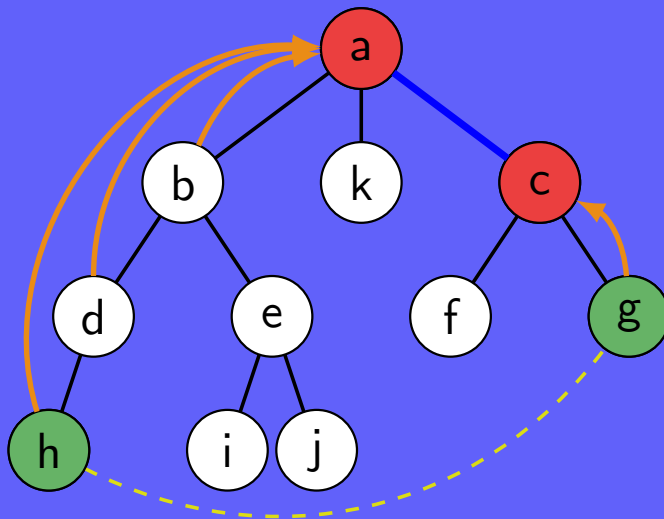
The union-find data structure

The two operations are:

- $\text{Find}(v)$: find the root of the tree for vertex v
- $\text{Union}(v, w)$: link the root of the tree containing v to the root of the tree containing w (or the other way around)

Process:

- We have two sub-trees
- On the graph an edge joins the vertices h and g
- Find on h and g returns a and c , respectively
- Update parents for h, d, b and g
- Union connects c to a



Algorithm.

```
1 Function GenSet( $x$ ):  
2   |  $x.parent \leftarrow x; x.rank \leftarrow 0;$   
3 end  
4 Function Find( $x$ ):  
5   | if  $x.parent \neq x$  then  $x.parent \leftarrow \text{Find}(x.parent)$  ;  
6   | return  $x.parent$   
7 end  
8 Function Union( $x, y$ ):  
9   |  $X \leftarrow \text{Find}(x); Y \leftarrow \text{Find}(y);$   
10  | if  $X.rank > Y.rank$  then  $Y.parent \leftarrow X;$   
11  | else  $X.parent \leftarrow Y;$   
12  | if  $X.rank = Y.rank$  then  $Y.rank++;$   
13 end
```

深度
+1

Algorithm. (*Kruskal with find-union*)

Input : A graph $G = \langle V, E \rangle$

Output: A minimum spanning tree T

```
1 Sort the edges  $G.E$  by weight;
2  $T \leftarrow \emptyset$ ;
3 for edges  $(u, v)$  in  $G.E$ , in non-decreasing order do
4   | if Find( $u$ )  $\neq$  Find( $v$ ) then
5   |   | add edge  $(u, v)$  to  $T$ ;
6   |   | Union( $u, v$ )
7   | end if
8 end for
9 return  $T$ 
```

meta search engine (best)

Problem (Counting inversions)

Given a list of n elements a_0, \dots, a_{n-1} , determine how many pairs $(a_i, a_j)_{\substack{0 \leq i, j \leq n \\ i \neq j}}$ are not in ascending order. Such pairs are called *inversions*.

Remark. This problem has numerous applications such as

- Voting theory
- Analysis of search engines ranking
- Collaborative filtering

Given 6 movies compare the ranking of two users:

Movie	A	B	C	D	E	F
First user	1	2	3	4	5	6
Second user	1	3	5	2	4	6

Given 6 movies compare the ranking of two users:

Movie	A	B	C	D	E	F
First user	1	2	3	4	5	6
Second user	1	3	5	2	4	6

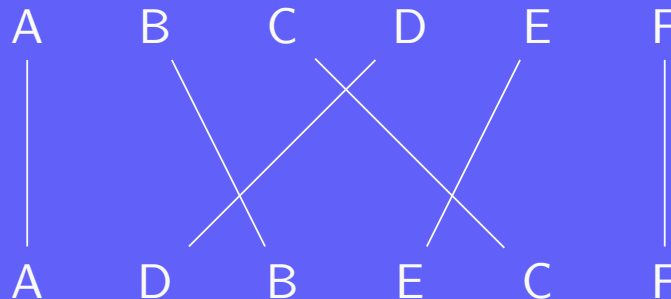
Inversions: (3, 2), (5, 2), (5, 4)

Given 6 movies compare the ranking of two users:

Movie	A	B	C	D	E	F
First user	1	2	3	4	5	6
Second user	1	3	5	2	4	6

Inversions: $(3, 2)$, $(5, 2)$, $(5, 4)$

A simple geometrical view:



Divide and conquer approach

Strategy for solving the counting inversions problem (1.40):

- 1 Divide: split the list L into two halves L_1 and L_2
- 2 Conquer: recursively count inversions in each list
- 3 Combine: count inversions for the pairs (l_i, l_j) with l_i and l_j belonging to L_1 and L_2 respectively

The sum of the three counts is the total number of inversion in L

Example.

1 5 4 8 10 2 6 9 3 7

1 5 4 8 10

2 6 9 3 7

Divide and conquer approach

Strategy for solving the counting inversions problem (1.40):

- ① Divide: split the list L into two halves L_1 and L_2
- ② Conquer: recursively count inversions in each list
- ③ Combine: count inversions for the pairs (l_i, l_j) with l_i and l_j belonging to L_1 and L_2 respectively

The sum of the three counts is the total number of inversion in L

Example.

1 5 4 8 10 2 6 9 3 7

1 5 4 8 10

(5,4)

2 6 9 3 7

(6,3), (9,3), (9,7)

1 4 5 8 10

2 3 6 7 9

Divide and conquer approach

Strategy for solving the counting inversions problem (1.40):

- ① Divide: split the list L into two halves L_1 and L_2
- ② Conquer: recursively count inversions in each list
- ③ Combine: count inversions for the pairs (l_i, l_j) with l_i and l_j belonging to L_1 and L_2 respectively

The sum of the three counts is the total number of inversion in L

Example.

1 5 4 8 10 2 6 9 3 7

1 5 4 8 10

(5,4)

2 6 9 3 7

(6,3),(9,3),(9,7)

1 4 5 8 10

2 3 6 7 9

(4,2),(4,3),(5,2),(5,3),(8,2),(8,3),(8,6),(8,7),(10,2),(10,3),(10,6),(10,7),(10,9)

Algorithm. (*Merge and count*)

Input : Two sorted lists: $L_1 = (l_{1,1}, \dots, l_{1,n_1})$, $L_2 = (l_{2,1}, \dots, l_{2,n_2})$

Output: Number of inversions *count*, and L_1 and L_2 merged into L

1 **Function** MergeCount(L_1, L_2):

2 $count \leftarrow 0$; $L \leftarrow \emptyset$; $i \leftarrow 1$; $j \leftarrow 1$;

3 **while** $i \leq n_1$ **and** $j \leq n_2$ **do**

4 **if** $l_{1,i} \leq l_{2,j}$ **then**

5 append $l_{1,i}$ to L ; $i++$;

6 **else**

7 append $l_{2,j}$ to L ; $count \leftarrow count + n_1 - i + 1$; $j++$;

8 **end if**

9 **end while**

10 **if** $i > n_1$ **then** append $l_{2,j}, \dots, l_{2,n_2}$ to L ;

11 **else** append $l_{1,i}, \dots, l_{1,n_1}$ to L ;

12 **return** *count* and L

13 **end**

trick part

和 merge sort 很像

Sort and count

Algorithm. (*Sort and count*)ocaml 的好例子
tail recursion**Input** : A list $L = (l_1, \dots, l_n)$ **Output:** The number of inversions *count* and L sorted

```

1 Function SortCount( $L$ ):
2   if  $n=1$  then return 0 and  $L$ ;
3   else
4     Split  $L$  into  $L_1 = (l_1, \dots, l_{\lceil n/2 \rceil})$  and  $L_2 = (l_{\lceil n/2 \rceil+1}, \dots, l_n)$ ;
5      $count_1, L_1 \leftarrow$  SortCount( $L_1$ );
6      $count_2, L_2 \leftarrow$  SortCount( $L_2$ );
7      $count, L \leftarrow$  MergeCount( $L_1, L_2$ );
8   end if
9    $count \leftarrow count_1 + count_2 + count$ ;
10  return  $count$  and  $L$ 
11 end

```

- How to present pseudocode?
- What are the two main categories of data structure?
- What is a greedy algorithm?
- Describe the divide and conquer strategy
- How is the Union-Find data structure working?

1001011111010100010101001111000111000100010111000000100110100001111101
1010010110011100111011101010001110000010001101110101011011111101010000
1101110011100011010011010110011011010101010100111101000010101010101000
0001100111100010000001101010110111101111001100000010101010101110100001
1100101110000100001001010110000011100010101000100110110010001101100100
0101011010111000010101100100101110001010011110000100111001110100
0110001111101000001011010011111000101101110011001010111101001101
111010010000010011101111010010000110011001001001111010001001001
1001111000111101001010111010011101000111100111101110100010
01111101111110011110010010010010010010011001100100100001111110000110
1110110110011100001001010101011001101011011111011111010
1101101000100110010100100111100001010011100001000101110010011011010011
0000110101010001100110000111000100101110011110000101001101100100000110
1101111010010101011101011111000010100010011101100000010110111100000011
0111101000100101011011001110011101011011110010110101011001100110101100
1011011011011111010000000011101110001111010111011001110011010011100000

Thank you!