the lecture (week 1). Consider the following stream of characters as input:

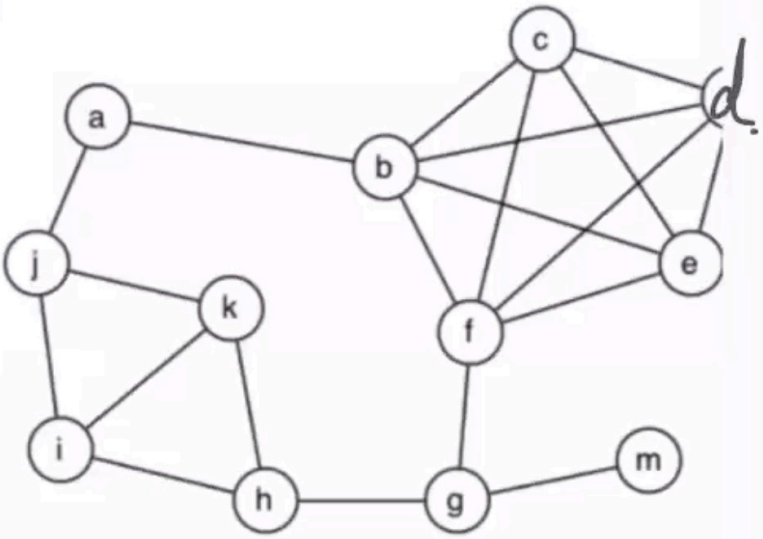
```
x = [ lim_x->0 { log_2 (
[y^3 / sqrt{z^2}] + xi * 4
} ]
```

What is the state of the stack at the point when the algorithm terminates (either with success or with failure)?
(Show the elements of the stack from bottom to top. For example, if the bottom element is **(** *, the next element is* **[** *and the top element is* **}** *then your input should be:* **([}** *. If the stack is empty, your input should be:* **empty** *)*

Answer:

[{()]

Consider the following graph:



Apply the breadth-first search algorithm from the lecture (week 5) to find a path from **j** to **h** in this graph. Consider neighbours in alphabetical order and show the order in which vertices are enqueued. *(For example, if **b** is enqueued first, followed by **g**, then **j** and no other node is enqueued before the algorithm terminates, then enter: **b,g,j**)*

Recall the Bracket Matching Algorithm from the lecture (week 1). Consider the following stream of characters as input:

```
while (func(i) > 0) { if (a[
a[i + 1] - func(a[i])] > 2)
{ i++; } } }
```

What is the state of the stack at the point when the algorithm terminates (either with success or with failure)?
*(Show the elements of the stack from bottom to top. For example, if the bottom element is **(** , the next element is **[** and the top element is **}** then your input should be: **([}** . If the stack is empty, your input should be: **empty**)*

回顾一下讲座（第1周）中的括号匹配算法。考虑以下字符流作为输入：

```
int vectorsum (int * vec [],
int n) {for (n = 1; n <f (vec
[n]; n ++) ;}
```

当算法终止时（成功或失败）堆栈的状态是什么？

（从底部到顶部显示堆栈的元素。例如，如果底部元素是 **(**，*下一个元素是* **[** *并且顶部元素是* **}**，*则您的输入应为：* **([}**。*如果堆栈为空，则您的输入应为：* **empty**）

What is the time complexity of the following algorithm, assuming that each stack operation takes O(1) time?

O(n)

```
Input integer array A of size n
create empty stack S
push A[0] onto S
for i=1...n-1 do
|  if A[i] < A[i-1] then
|      push A[i] onto S
|  else
|  |  t=0
|  |  while S is not empty do
|  |     t = t + pop(S)
|  |  end while
|  |  push t onto S
|  end if
```

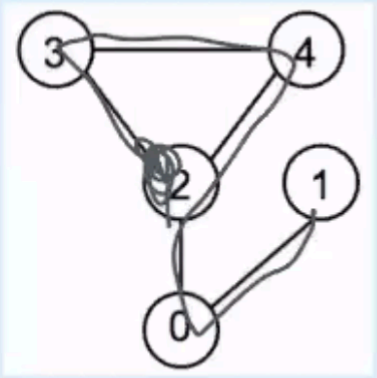
What is the time complexity of the following algorithm, assuming that each stack operation takes O(1) time?

```
Input integer array A of size n
create empty stack S
push A[0] onto S
for i=1...n-1 do
|  if A[i] < A[i-1] then
|      push A[i] onto S
|  else
|  |  t=0
|  |  while S is not empty do
|  |    t = t + pop(S)
|  |  end while
|  |  push t onto S
|  end if
end for
```

Select one:

- $O(n \cdot \log n)$
- $O(2^n)$
- $O(n^3)$
- $O(n)$
- $O(n^2)$

Consider the following graph:

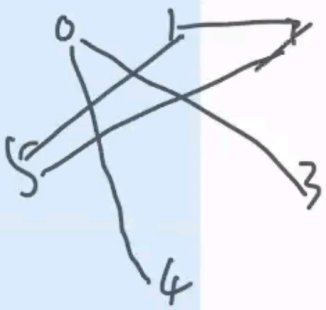


1 - 0 - 2 - 4 - 3 - 2

Give an Euler path for this graph if one exists. Hint: there may be more than one correct solution.
(Enter your path in this format: 1-2-3 dash separated no spaces. If no Euler path exists, enter: ***none****)*

Consider the following adjacency matrix of an undirected graph G:

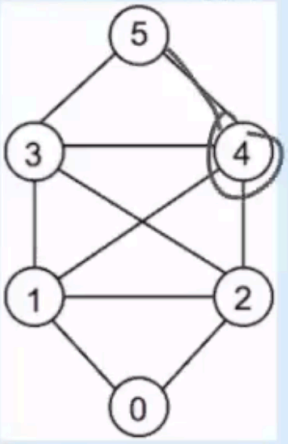
	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	0	0	1	1	0
[1]	0	0	1	0	0	1
[2]	0	1	0	0	0	1
[3]	1	0	0	0	0	0
[4]	1	0	0	0	0	0
[5]	0	1	1	0	0	0



How many connected components does this graph have?

Answer:

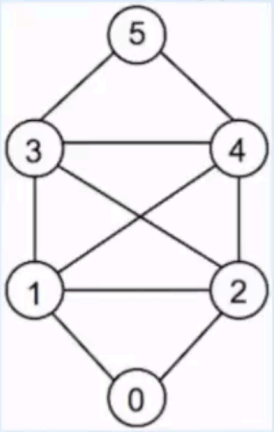
Consider the following graph:



Apply the Hamiltonian path finding algorithm from the lecture (week 5) to find a Hamiltonian path from 4 to 0 in this graph. Consider neighbours give the path that the algorithm has found when it returns with success.
*(Enter the path in the following format: **1-2-3** dash separated no spaces.)*

Answer:

Consider the following graph:



Apply the Hamiltonian path finding algorithm from the lecture (week 5) to find a Hamiltonian path from 3 to 5 in this graph. Consider neighbours in ascending order and give the path that the algorithm has found when it returns with success.
*(Enter the path in the following format: **1-2-3** dash separated no spaces.)*

```
void freeList(NodeT *head) {
   if (head == NULL)
      return;
   else {
      NodeT *p = head->next;
      free(head);
      freeList(p->next);
   }
}
```

```
void freeList(NodeT *head) {
   if (head->next != NULL) {
      freeList(head->next);
      free(head);
   } else {
      free(head);
   }
}
```

```
void freeList(NodeT *head) {
   if (head == NULL)
      return;
   else {
      freeList(head->next);
      free(head);
   }
}
```

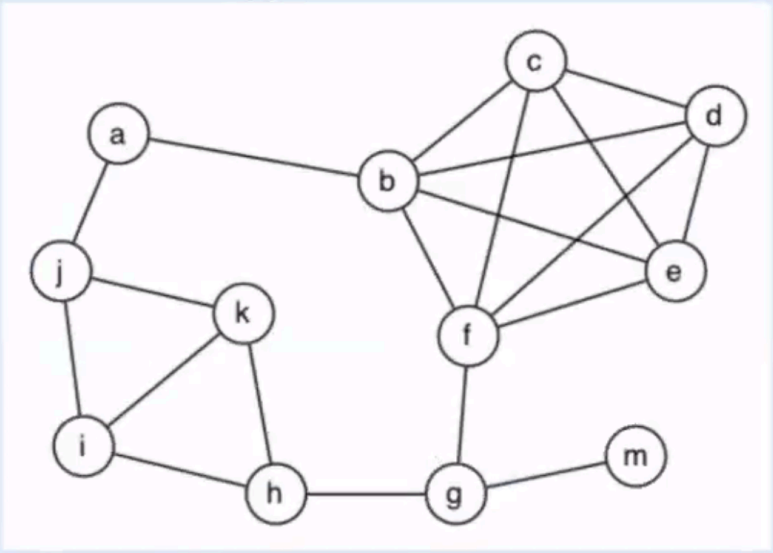
```
void freeList(NodeT *head) {
   if (head != NULL)
      freeList(head->next);
   if (head != NULL)
      free(head);
}
```



```
void freeList(NodeT *head) {
   if (head->next != NULL) {
      freeList(head->next);
      free(head);
   } else {
      free(head);
   }
}
```

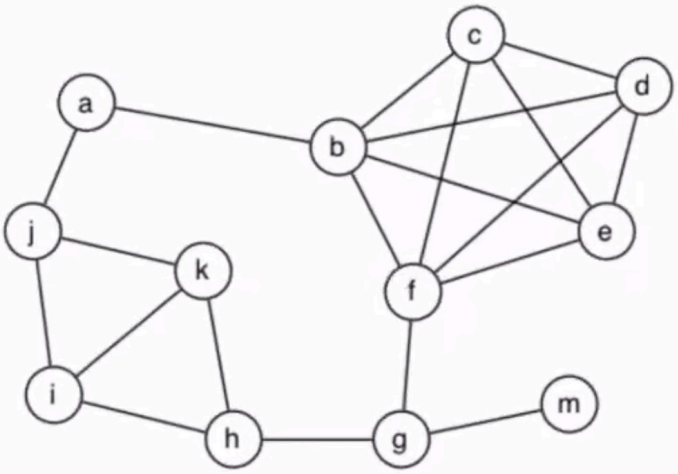
```
void freeList(NodeT *head) {
   if (head != NULL)
      freeList(head->next);
   if (head != NULL)
      free(head);
}
```

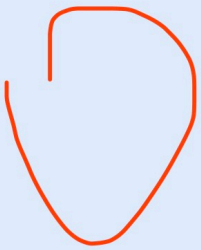
Consider the following graph:



Apply the breadth-first search algorithm from the lecture (week 5) to find a path from **f** to **h** in this graph. C
neighbours in alphabetical order and show the order in which vertices are enqueued.
*(For example, if **b** is enqueued first, followed by **g**, then **j** and no other node is enqueued before the algorit
then enter: **b,g,j**)*

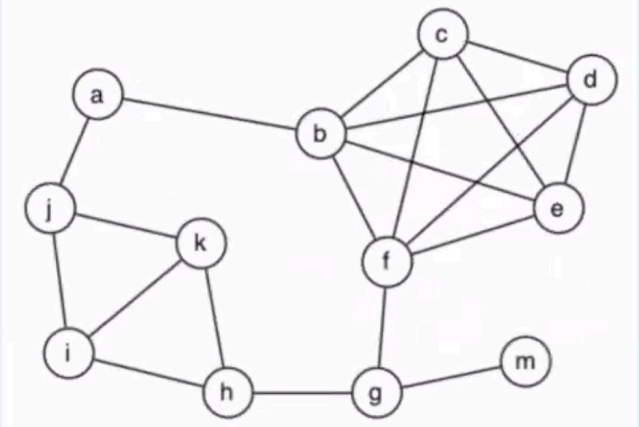
Consider the following graph:





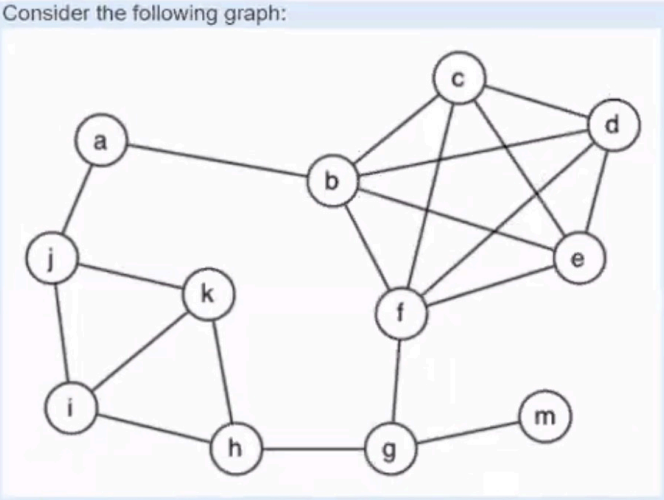
Apply the breadth-first search algorithm from the lecture (week 5) to find a path from **f** to **h** in this graph. C
neighbours in alphabetical order and show the order in which vertices are enqueued.
*(For example, if **b** is enqueued first, followed by **g**, then **j** and no other node is enqueued before the algorit
then enter: **b,g,j**)*

Consider the following graph:



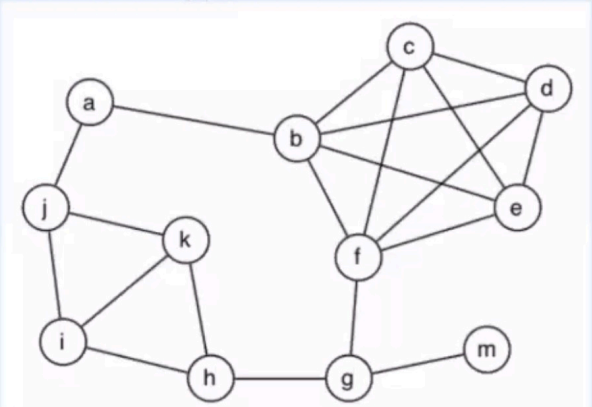
Apply the breadth-first search algorithm from the lecture (week 5) to find a path from **j** to **h** in this graph. Consider neighbours
alphabetical order and show the order in which vertices are enqueued.
*(For example, if **b** is enqueued first, followed by **g**, then **j** and no other node is enqueued before the algorithm terminates, ther*

Consider the following graph:



Apply the breadth-first search algorithm from the lecture (week 5) to find a path from **g** to **c** in this graph. Consider neighbours in alphabetical order and show the order in which vertices are enqueued.
*(For example, if **b** is enqueued first, followed by **g**, then **j** and no other node is enqueued before the algorithm terminates, then enter: **b,g,j**)*

Consider the following graph:



Apply the breadth-first search algorithm from the lecture (week 5) to find a path from **g** to **c** in this graph. Consider neighbours in alphabetical order and show the order in which vertices are enqueued.
*(For example, if **b** is enqueued first, followed by **g**, then **j** and no other node is enqueued before the algorithm terminates, then enter: **b,g,j**)*

18 / 33

Recall the Bracket Matching Algorithm from the lecture (week 1). Consider the following stream of characters as input:

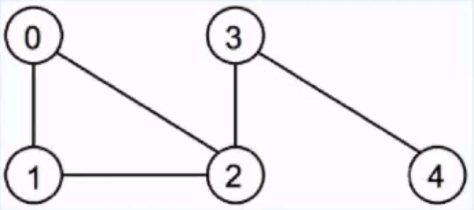
```
while (func(i) > 0) { if (a[a[i + 1] - func(a[i])] > 2) { i++; } } }
```

What is the state of the stack at the point when the algorithm terminates (either with success or with failure)?
*(Show the elements of the stack from bottom to top. For example, if the bottom element is **(** , the next element is **[** and th element is **}** then your input should be: **([}** . If the stack is empty, your input should be: **empty**)*

Answer: }

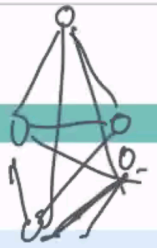
18 / 33

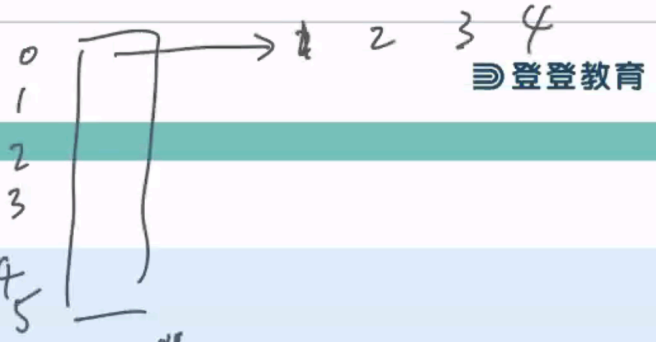
Consider the following graph:



Give an Euler path for this graph if one exists. Hint: there may be more than one correct solution.
*(Enter your path in this format: **1-2-3** dash separated no spaces. If no Euler path exists, enter: **none***

21 / 33





登登教育

Tick **all** statements about graph representations that are correct.

Select one or more:

- [ ] It is always the case that if a node v belongs to a clique of 5 nodes, then there are at least 5 elements in the adjacency list for v given an adjacency-list representation of the graph.
- [x] Deleting all the edges of a graph can always be achieved in $O(V)$ time in the array-of-edges representation, where V is the number of vertices.
- [x] Deleting an edge can always be achieved in $O(V)$ time in the adjacency-list representation, where V is the number of vertices.
- [ ] Given an unsorted array-of-edges representation of a graph, checking whether two vertices are adjacent can always be achieved in $O(V)$ time, where V is the number of vertices.

22 / 33

O(1)

O(1)

O(E)

Tick **all** statements about graph representations that are correct.

Select one or more:

- [ ] It is always the case that if a node v belongs to a clique of 5 nodes, then there are at least 5 elements in the adjacency list for v given an adjacency-list representation of the graph.
- [x] Deleting all the edges of a graph can always be achieved in $O(V)$ time in the array-of-edges representation, where V is the number of vertices.
- [x] Deleting an edge can always be achieved in $O(V)$ time in the adjacency-list representation, where V is the number of vertices.
- [ ] Given an unsorted array-of-edges representation of a graph, checking whether two vertices are adjacent can always be achieved in $O(V)$ time, where V is the number of vertices.

22 / 33

O(1)

Tick **all** statements about graph representations that are correct.

O(1) O(E)

Select one or more:

- [ ] It is always the case that if a node v belongs to a clique of 5 nodes, then there are at least 5 elements in the adjacency list for v given an adjacency-list representation of the graph.
- [x] Deleting all the edges of a graph can always be achieved in $O(V)$ time in the array-of-edges representation, where V is the number of vertices.
- [x] Deleting an edge can always be achieved in $O(V)$ time in the adjacency-list representation, where V is the number of vertices.
- [ ] Given an unsorted array-of-edges representation of a graph, checking whether two vertices are adjacent can always be achieved in $O(V)$ time, where V is the number of vertices.

O(E)


```
typedef struct {
   char  name[32];   32
   int   status;     4
   float salary;     4
} workerT;                  = 40

typedef struct node {       160
   workerT team[4];
   int     department, floor;   4  4
   struct node *prev, *next;
} NodeT;                    8   8
```

192

```
typedef struct {
   int   **edges;        8
   int   nV, nE;         4  4
   float density[2];     8
} graphT;                          24

typedef struct node {
   graphT graphs[5];     x5 = 120
   double averageGraphSize;  = 8
   struct node *next;    => 8
} NodeT;                           = 136
```

Consider the following structure:

```
typedef struct {
   int   ID, birthday[3];
   float salary[2];
} workerT;

typedef struct node {
   workerT team[5];
   int     shift;
   char    teamID[4];
   struct node *prev, *next;
} NodeT;
```

On a CSE computer, how many bytes does one variable of type **NodeT** take?

Answer:

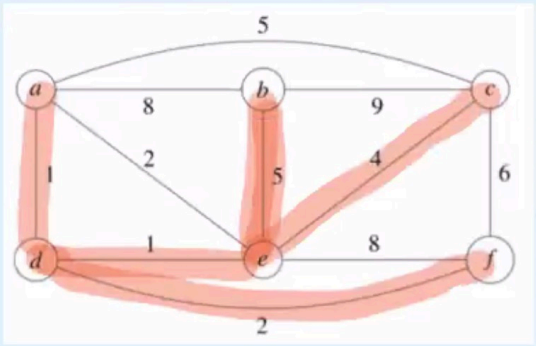
An algorithm for solving a problem runs in exponential time $O(10^n)$. If your computer can solve the problem in one day for any input of size 1,000 using an implementation of this algorithm, what is the maximum problem size that can be solved in one day with a computer that is 1,000 times faster?

Select one:

- a. 3000
- b. 1003
- c. 1001000
- d. 1000000

28 / 33

Consider the following weighted undirected graph:



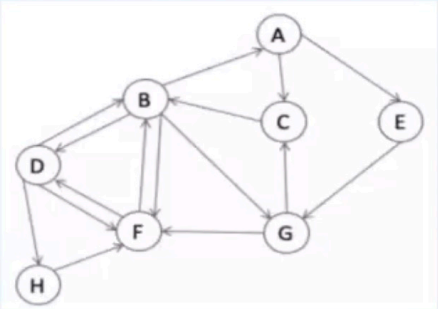
1 + 1 + 2 + 4 + 4

What is the sum of the weights of all of the edges in the minimum spanning tree (MST) of this graph?

Select one:

- a. 15
- b. 11
- c. 13
- d. 9

Run the Breadth First Search algorithm, as discussed in the lectures, on the following directed graph with vertex F as the source and note the order in which nodes are visited. If a vertex has multiple neighbours, consider them in ascending order. Select **all** options below that are true. For this question, you can select multiple options.



Select one or more:

- 1. None of the other choices are correct.
- 2. D will be visited after A.
- 3. H will be visited before E.
- 4. G will be visited before H.
- 5. E will be visited before G.

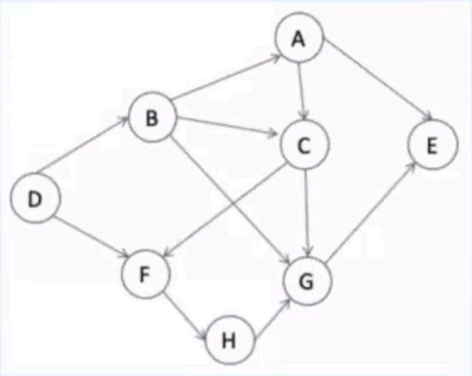
Which of the following statements about graph traversals is *not* correct?

Select one:

- a. Both depth-first search and breadth-first search can be used to traverse a graph.
- b. Depth-first search can be used to find a shortest path from a source vertex to a destination vertex in O(V + E) time, where V is the number of vertices and E the number of edges of a graph.
- c. Both depth-first search and breadth-first search can be used to detect a cycle in a graph.
- d. Breadth-first search can be used to find a shortest path from a source vertex to a destination vertex in O(V + E) time, where V is the num of vertices and E the number of edges of a graph

30 / 33

Run the Depth First Search algorithm, as discussed in the lectures, on the following directed graph with vertex D as the source and find paths to all possible nodes. If a vertex has multiple neighbours, consider them in ascending order. For the path from D to G, select the statement that is true.



Select one:

- a. H is the predecessor of G and F is the predecessor of D.
- b. H is the predecessor of G and C is the predecessor of F.
- c. None of the other choices are true.
- d. B is the predecessor of G and D is the predecessor of B.
- e. C is the predecessor of G and A is the predecessor of C.