نمونه سوالات مربوط به بخش ۷

۱،۷ یکی از راه های پیمایش، پیمایش میان وند است(in order). در این پیمایش ابتدا فرزند چپ سپس پدر و بعد از آن فرزند راست خوانده میشود. رایج ترین نوع پیاده سازی این الگوریتم، استفاده از توابع بازگشتی است. حال شما سعی کنید این پیمایش را بدون استفاده از هیچ تابع بازگشتی پیاده سازی کنید و پیچیدگی زمانی آنرا نیز بدست آورید.

راه حل:

پیاده سازی الگوریتم به صورت زیر است:

۱) ابتدا یک پشته خالی تعریف میکنیم.

۲) سپس ریشه گراف را به عنوان نود فعلی در نظر میگیریم.

۳) نود فعلی را در پشته push میکنیم و نود فعلی را برابر با بچه چپ نود فعلی در نظر میگیریم.
(current = current->left) تا زمانی که نود فعلی برابر NULL شود.

۴) اگر نود فعلی NULL بود و پسته ما خالی نبود،

الف) عنصر بالای پسته را pop میکنیم.

ب) عنصر pop شده زا چاپ میکنیم و نود فعلی را برابر با بچه راست عنصر pop شده قرار میدهیم.
(current = popped_item->right)

ج) برو به مرحله سه.

۵) اگر نود فعلی NULL بود و پشته نیز خالی بود الگوریتم به پایان رسیده است.

راه حل بهتر دیگری نیز وجود دارد به دلیل بهینه نبودن استفاده از پشته از نظر حافظه برای تبدیل الگوریتم های بازگشتی به غیر بازگشتی که به این صورت است:

۱) نود فعلی را ریشه درخت در نظر میگیریم

۲) تا زمانی که نود فعلی NULL نشده است:

اگر نود فعلی بچه چپ نداشت:

الف) داده موجود در نود فعلی را چاپ میکنیم

ب) نود فعلی را برابر بچه راست خود میگذاریم (current = current->right)

در غیر این صورت:

الف) زیر درخت سمت چپ که نود فعلی ریشه آن است را در نظر میگیریم و نود فعلی را برابر بچه راست راسترین نود این زیر درخت میگذاریم

ب) به بچه چپ این نود میرویم (current = current->left)

پیچیدگی زمانی نیز واضح است در هر دو حالت برابر O(n) است.

۲,۷ الگوریتمی غیر بازگشتی برای پیمایش پیشوندی یک درخت بنویسید.

راه حل:

الگوریتم را در مراحل زیر پیاده سازی میکنیم.

۱) ابتدا یک پشته خالی در نظر میگیریم و ریشه را در آن push میکنیم.

۲) تا زمانی که پشته خالی نبود مراحل زیر را انجام میدهیم.

الف) یک عنصر را از پشته pop کن و آنرا چاپ کن.

ب) بچه سمت راست عنصر چاپ شده را در پشته push کن.

ج) بچه سمت چپ عنصر چاپ شده را در پشته push کن.

۳) با خالی شدن پشته الگوریتم به پایان میرسد.

راه حل بدون استفاده از پشته:

```
void BinarySearchTree::preorderNonRecursive(BinarySearchTreeNode* root)
{
   BinarySearchTreeNode* current = root;
   while (current) {
      if (!current->visited) {
         current->print();
         current->visited = true;
      }

      if (current->left && !current->left->visited) {
         current = current->left;
         continue;
      } else if (current->right && !current->right->visited) {
         current = current->right;
```

```
            continue;
        }

        if (current->left)
            current->left->visited = false;
        if (current->right)
            current->right->visited = false;

        current = current->parent;
    };
    if (root)
        root->visited = false;
}
```

۳،۷ پیمایش پسوندی به این صورت است که ابتدا فرزند چپ و راست و سپس پدر فرزندان مشاهده میشود. (Post Order)

پیاده سازی این الگوریتم پیمایش به صورت بازگشتی روش متداول آن است. حال سعی کنید الگوریتمی به صورت غیر بازگشتی برای پیاده سازی این پیمایش ارایه دهید. در مرحله بعد سعی کنید با یک ساختمان داده نیز راه حلی را ارایه دهید.

راه حل:

راه حل اول: پیاده سازی با دو پشته:

۱) ابتدا ریشه درخت را در پشته اول push میکنیم.

۲) تا زمانی که پشته اول خالی نیست:

الف) یک عنصر را از پشته اول pop و در پشته دوم push میکنیم

ب) به ترتیب بچه چپ و راست عنصر pop شده را در پشته اول push میکنیم

۳) محتوای پشته دوم را به ترتیب pop و چاپ میکنیم

راه حل دوم: پیاده سازی با یک پشته:

۱) ابتدا یک پشته خالی تعریف میکنیم.

۲) تا زمانی که نود فعلی NULL نشده:

الف) ابتدا بچه راست سپس بچه چپ را در پشته push میکنیم

ب) نود فعلی را برابر بچه چپ نود فعلی قرار میدهیم (current = current->left)

۳) یک عنصر را از پشته pop میکنیم و نود فعلی را برابر آن قرار میدهیم (current = popped)

الف) اگر عنصر pop شده بچه راست داره و آن بچه راست عنصر بالای پشته است، بچه راست را از پشته pop میکنیم و نود فعلی را در پشته push میکنیم و سپس نود فعلی را برابر بچه راست نود فعلی قرار میدهیم.

ب) در غیر این صورت، نود فعلی را چاپ کرده و سپس نود فعلی را NULL میگذاریم.

۴) مراحل ۲ و ۳ را تا زمانی که پشته خالی نیست تکرار میکنیم.

راه حل بدون استفاده از پشته:

```
void BinarySearchTree::postorderNonRecursive(BinarySearchTreeNode* root) {
  BinarySearchTreeNode* current = root;
  while (current) {
    if (current->left && !current->left->visited) {
      current = current->left;
      continue;
    } else if (current->right && !current->right->visited) {
      current = current->right;
      continue;
    }

    if (!current->visited) {
      current->print();
      current->visited = true;
    }

    if (current->left)
      current->left->visited = false;
    if (current->right)
      current->right->visited = false;

    current = current->parent;
  };
  if (root)
    root->visited = false;
}
```

۴,۷ الگوریتمی غیر بازگشتی ارایه دهید که با دریافت دو گراف به ما بگوید آیا این دو گراف مشابه هستند یا خیر.

راه حل:

ابتدا دو صف را تعریف میکنیم و به نام های q1 و q2 مینامیم. سپس ریشه های دو گراف را به تابع الگوریتم خود میدهیم که نام های root1 و root2 دارند. ابتدا حالت های خاص را چک میکنیم به این صورت که اگر دو گراف داده شده تهی بودند true بازگردانده شود. اگر تنها یکی از آنها تهی بود false برگردانده شود.

سپس اگر دو گراف از این شرایط عبور کردند یعنی هیچ کدام تهی نیستند. حال root1 و root2 را به ترتیب در q1 و push q2 میکنیم. سپس تا زمانی که q1 و q2 هر دو غیر خالی هستند:

عنصر جلو دو صف را مقایسه میکنیم و اگر برابر نبودند false برگردانده میشود. عناصر جلو صف را از صف خارج میکنیم. حال چک میکنیم اگر هر دو عنصر بچه چپ داشتند در صف ها وارد میکنیم در غیر این صورت اگر یکی داشت و دیگری نداشت false برگردانده میشود. همین کار را با بچه های چپ انجام میدهیم.

اگر الگوریتم به پایان خود رسید و مقداری برگردانده نشده بود true برمیگردانیم.

```
// Iterative method to find height of Bianry Tree
bool areIdentical(Node *root1, Node *root2)
{
   // Return true if both trees are empty
   if (!root1  && !root2) return true;

   // Return false if one is empty and other is not
   if (!root1 || !root2)  return false;

   // Create an empty queues for simultaneous traversals
   queue<Node *> q1, q2;

   // Enqueue Roots of trees in respective queues
   q1.push(root1);
   q2.push(root2);

   while (!q1.empty() && !q2.empty())
   {
      // Get front nodes and compare them
      Node *n1 = q1.front();
      Node *n2 = q2.front();

      if (n1->data != n2->data)
        return false;

      // Remove front nodes from queues
      q1.pop(), q2.pop();

      /* Enqueue left children of both nodes */
      if (n1->left && n2->left)
      {
         q1.push(n1->left);
```

```
            q2.push(n2->left);
        }

        // If one left child is empty and other is not
        else if (n1->left || n2->left)
            return false;

        // Right child code (Similar to left child code)
        if (n1->right && n2->right)
        {
            q1.push(n1->right);
            q2.push(n2->right);
        }
        else if (n1->right || n2->right)
            return false;
    }

    return true;
}
```

۵,۷ الگوریتمی غیر بازگشتی و بهینه برای بدست آوردن عدد x به توان y با پیچیدگی زمانی O(log y) ارایه دهید.

راه حل:

اگر از حلقه به صورت معمول استفاده کنیم به پیچیدگی O(n) میرسیم برای همین راه حل را با استفاده از داده های به دست آمده در مراحل اجرا بهینه تر میکنیم.

```
/* Iterative Function to calculate (x^y) in O(logy) */
int power(int x, unsigned int y)
{
    int res = 1;     // Initialize result

    while (y > 0)
    {
        // If y is odd, multiply x with result
        if (y & 1)
            res = res*x;

        // n must be even now
        y = y>>1; // y = y/2
        x = x*x;  // Change x to x^2
    }
    return res;
}
```

۶,۷ الگوریتم جستجو دودویی را بصورت غیر بازگشتی بنویسید و سعی کنید راه شما از نظر حافظه نیز بهینه تر از الگوریتم معمول جستجو دودویی باشد و پیچیدگی زمانی آنرا نیز به دست آورید.

راه حل:

تکه کد زیر مراحل الگوریتم را به سادگی شرح میدهد.

```
// A iterative binary search function. It returns location of x in
// given array arr[l..r] if present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
  while (l <= r)
  {
    int m = l + (r-l)/2;

    // Check if x is present at mid
    if (arr[m] == x)
        return m;

    // If x greater, ignore left half
    if (arr[m] < x)
        l = m + 1;

    // If x is smaller, ignore right half
    else
         r = m - 1;
  }

  // if we reach here, then element was not present
  return -1;
}
```

۷,۷ الگوریتمی غیر بازگشتی و بهینه از نظر حافظه برای پیدا کردن عمق یک درخت ارایه دهید.

راه حل:

در هر مرحله مل پیمایش Level Oreder میزنیم و راس ها را در صف اضافه میکنیم. تکه کد زیر به خوبی این مراحل را شرح میدهد.

```
// Iterative method to find height of Bianry Tree
int treeHeight(node *root)
{
   // Base Case
   if (root == NULL)
      return 0;

   // Create an empty queue for level order tarversal
   queue<node *> q;

   // Enqueue Root and initialize height
   q.push(root);
   int height = 0;

   while (1)
   {
      // nodeCount (queue size) indicates number of nodes
      // at current lelvel.
      int nodeCount = q.size();
      if (nodeCount == 0)
         return height;

      height++;

      // Dequeue all nodes of current level and Enqueue all
      // nodes of next level
      while (nodeCount > 0)
      {
         node *node = q.front();
         q.pop();
         if (node->left != NULL)
            q.push(node->left);
         if (node->right != NULL)
            q.push(node->right);
         nodeCount--;
      }
   }
}
```

۸,۷ مساله برج هانوی به این صورت است که سه میله و تعدادی دیسک با اندازه های متفاوت و به ترتیب در یکی از میله ها گذاشته شده است. هدف بازی جا به جایی همه دیسک ها به همان ترتیب به یک میله دیگر است. به صورتی که در هر مرحله فقط حق جا به جا کردن یک دیسک وجود دارد و نمیتوان دیسک بزرگتر را بالای دیسک کوچکتر از خود قرار داد. این الگوریتم را به ازای تعداد دلخواه دیسک بصورت غیر بازگشتی پیاده سازی کنید.

راه حل:

۱) ابتدا تعداد حرکات لازم را حساب میکنیم به این صورت که، تعداد حرکات برابر دو به توان تعداد دیسک ها منهای یک است

۲) اگر تعداد دیسک ها زوج بود میله مقصد و کمکی را در مرحله ۳ جا به جا میکنیم (یعنی هرجا مقصد بود مینویسیم کمکی و بالعکس)

۲) از i = 1 تا تعداد حرکات لازم و کامل شدن برج:

الف) اگر i % 3 == 1 دیسکی که مجاز است را بین مبدا و مقصد جا به جا میکنیم

ب) اگر i % 3 == 2 دیسکی که مجاز است را بین مبدا و کمکی (میله وسط) جا به جا میکنیم

ج) اگر i % 3 == 0 دیسکی که مجاز است را بین کمکی و مقصد جا به جا میکنیم

منظور از حرکت مجاز به هم نخوردن شرط اینکه دیسک بزرگتر نباید روی دیسک کوچکتر باشد است.

و به دلیل شرط اولیه که با تعداد دیسک ها تعین کردیم پیچیدگی زمانی مساله O(2^n) است.

نمونه سوالات بخش ۸,۲,۱

۱,۸,۲,۱ الگوریتمی ارایه دهید که یال های برشی یک گراف را به دست آورد و مرتبه زمانی الگوریتم خود را نیز به دست آورید.

راه حل:

برای پیدا کردن یال های برشی ابتدا سعی میکنیم راس های برشی را پیدا کنیم. از یکی از راس ها شروع کرده و DFS میزنیم. دو مقدار زیر را نگه داری میکنیم، اولی عدد راس دیده شده هنگامی که توسط DFS دیده میشود (num(v))و دومین عدد هم کمترین مقدار اولین عدد است که با زیر درخت راس دیده شده توسط DFS است (low(v)). برای اولین با که یک راس دیده میشود. مقدار low(v) برابر num(v) خواهد بود و واضح است که مقدار low(v) کم نمیشود چون تمام رئوس زیر درخت v دارای num بیشتر از num(v) است فقط در حالتی که یکی از فرزندان v به یکی از پدرانش متصل باشد. حالا فرض کنید راس u دارای فرزند v و وقتی الگوریتم DFS بررسی راس v را تمام میکتد، آنگاه low(v) >= num(v) باشد. این یعنی آن حالت خاص نیز پیش نمیاید. اگر این حال برای راس v پیش بیاید یعنی u یک راس برشی است. اگر حذف شود فرزندش که v میباشد دیگر هیچ مسیری به پدران u ندارد و گراف ناهمبند میشود مگر حالتی که آن ریشه درخت باشد. شرط اینکه راس برشی باشد این است که حداقل دو فرزند باشد. فقط مساله ای که میماند این است که low(v) را تعیین کنیم. این عدد را باید در بین DFS زدن تعیین کرد. به این صورت که اگر در راس u باشیم و DFS بررسی یک فرزند آن مانند v را تمام کند انگاه low(u) = min[low(u), low (v)] میشود و حالت دیگر نیز این است که راس v را قبلا دیده باشیم، در اینصورت low(u) = min[low(u), num(v)] میشود. چک کردن برشی بودن یک راسُ بعد از تمام شدن DFS روی یک فرزند و قبل از بروز رسانی low پدر صورت میگیرد. در صورتی که به ازای فرزندی مانند v فهمیدیم low(v) >= num(v) است؛ نتیجه میگیریم که راس u برشی است و یال u,v یال برشی. هزینه زمانی این الگوریتم هم O(E + V) است.

۲,۸,۲,۱ الگوریتمی با استفاده از DFS ارایه دهید که حداقل یال هایی که نیاز است به گراف اضافه شود که قویا همبند شود را پیدا کند و هزینه زمانی را نیز بدست آورید.

راه حل:

**ابتدا دو بار روی گراف و گراف transpose آن DFS میزنیم تا مولفه های قویا همبند را پیدا کنیم. حال یک گراف جدید ایجد میکنیم که رئوس آن مولفه های قویا همبند هستند و اگر بین دو مولفه قویا همبند یال وجود داشته باشد، در گراف جدید بین آن دو، یال جهت دار میگذاریم. به وضوح گراف جدید یک DAG است چرا که اگر دور داشت به این معنی بود که رئوس روی دور همگی به یکدیگر راه داشتند و از ابتدا مولفه های قویا همبند جداگانه نبودند و باید همگی یک مولفه میشدند و نه چند مولفه.
در این DAG تعداد رئوسی که یال ورودی ندارند را میشماریم و آنرا c1 مینامیم. سپس تعداد رئوسی که یال خروجی ندارند را**

میشماریم و آنرا c2 مینامیم. اگر DAG فقط یک راس داشته باشد، لازم نیست یالی اضافه کنیم در غیر این صورت باید به تعداد max(c1, c2) یال اضافه شود تا گراف قویا همبند شود.

۳،۸،۲،۱ مجموعه ای از عبارات منطقی داریم که هر عبارت منطقی از or دو متغیر میباشد که هرکدام از این دو متغیر میتواند به شکل نقیض شده باشد. الگوریتم بهینه ای ارایه دهید که مشخص کند آیا میتوان متغیر های یک مجموعه عبارات را به گونه ای مقدار دهی کرد که تمام عبارات مقدار true پیدا کنند. برای مثال:

E1 = A + B

E2 = ~A + C

E3 = ~C + ~B

اگر مقدار A برابر false و مقدار B برابر true و مقدار C برابر false باشد آنگاه تمامی عبارت های E1, E2, E3 مقدار true خواهند داشت.

راه حل:

مساله را با گراف به این صورت که به ازای هر متغیر x در مساله دو راس x, ~x اضافه میکنیم و به ازای هر عبارت مانند x + y یال های (~x, y) و (y, ~x) را نیز به گراف اضافه میکنیم، مدل سازی میکنیم. وجود یال (u, v) در گراف بیان میکند که اگر U درست باشد آنگاه V هم درست است. حال روی این گراف دو بار DFS میزنیم و مولفه های قویا همبند را پیدا میکنیم. سپس به ازای هر مولفه قویا همبند یک راس میگذاریم و اگر بین دو راس از دو مولفه یال وجود دارد با جهت درست دو راس را با یال به هم وصل میکنیم اما نمیتوان اطمینان حاصل کرد که دور بوجود نمیاید چون اگر دور داشته باشیم، اجتماع این دو مولفه همبند خود که مولفه همبند بوجود میاورد. سپس به ازای هر متغیر شماره مولفه آن را نیز ذخیره میکنیم. در آخر هم به ازای هر متغیر x چک میکنیم شماره مولفه X با ~X مساوی نباشد. اگر به ازای همه رئوس این قائده برقرار بود میتوان گفت امکان حل این مساله وجود دارد. حالا برای اینکه بفهمیم مقدار هر متغیر چقدر باشد به این شکل عمل میکنیم که آخرین گرافی را که از مولفه های قویا هم بند ساختیم یک DAG است. حالا روی آن Topological Sort میزنیم. اگر مولفه متغیر x قبل از متغیر مولفه ~x بیاید، آنگاه x را false میگذاریم و در غیر این صورت true. البته توجه کنید که اگر گراف ناهمبند باشد و مسیری از مولفه شامل x به مولفه شامل ~x نباشد یا بالعکس، در این صورت متغیر x میتواند هر مقداری داشته باشد.

۴،۸،۲،۱ الگوریتمی برای پیدا کردن مولفه های قویاً همبند گراف های جهت دار ارائه دهید.

راه حل:

یک گراف جهت دار قویاً همبند است اگر بین هر جفت از راس های آن مسیری وجود داشته باشد. به بزرگترین زیر گراف ممکن که قویاً همیند باشد مولفه ی قوی همبندی SCC آن گراف می گوییم.با استفاده از الگوریتمی موسوم به kosaraju که از DFS استفاده می کند می توانیم تمام مولفه های همبندی یک گراف را با هزینه ی زمانی O(v+e) بدست آوریم.

الگوریتم به این ترتیب اجرا می شود:

۱) پشته ی خالی ای به نام S می سازیم و پیمایش DFS را روی گراف اجرا می کنیم. در پیمایش DFS پس از صدا کردن بازگشتی DFS بر روی راس های مجاور یک راس، آن راس را در پشته push می کنیم

۲) تمام یال ها را برعکس می کنیم تا گراف ترانهاده بدست آید.

۳) تا وقتی که پشته خالی نشده است تک تک راس ها را از پشته pop می کنیم، راس pop شده را v می نامیم، و بر روی v الگوریتم DFS را اجرا می کنیم. DFS حاصل راس های مولفه ی همبندی راس v را می دهد.

بررسی پیچیدگی این الگوریتم: الگوریتم DFS را صدا می زند، سپس گراف را ترانهاده می کند و دوباره DFS را صدا می زند، DFS هزینه ی O(v+e) را دارد، همچنین ترانهاده کردن گراف نیز همین هزینه را دارد، برای ترانهاده کردن گراف کافی است تمامی لیست های مجاورت را بپیماییم. از نظر order هزینه ی زمانی این الگوریتم بهینه ترین است ولی الگویرتم های دیگری هم با همین order وجود دارند ولی تنها از یک DFS استفاده می کنند، مانند الگوریتم tarjan.

۵,۸,۲,۱ فرض کنید T درختی است که حداقل دو راس دارد. کمترین و بیشترین تعداد مفصل هایی که T دارد را بدست آورید و بگویید در هر حالت چند مولفه دوسو همبند دارد.

راه حل:

الف) T میتواند حداقل یک و حداکثر n-1 مفصل داشته باشد. اگر T شامل شامل یک راس با درجه n-1 باشد این راس تنها مفصل است. اگر T مسیری با n راس و n-1 یال باشد، آنگاه n-2 راس درجه ۲ همگی مفصل اند.

ب) در همه حالت ها درختی با n راس دارای n-1 مولفه دوسو همبند است. هر یال یک مولفه دو سو همبند است.

۶,۸,۲,۱ یک ماتریس ۲ بعدی داریم، تعداد مولفه های همبندی گراف متناظر با آن را بدست آورید.

راه حل:

به مجموعه ۱ هایی که کنار هم قرار گرفته اند یک مولفه ی همبندی می گوییم. در نظر گرفتن این نکته ضروری است که هر خانه ی ماتریس می تواند با ۸ همسایه اش همبند باشد. با استفاده از ساختمان داده ی مجموعه های مجزا به حل سوال می پردازیم.

الگوریتم ما به این شیوه عمل می کند:

۱) متغییر result که تعداد مولفه های همبندی است را برابر ۰ در نظر بگیر

۲) تمام درایه های ماتریس را پیمایش کن

۳) اگر مقدار هر درایه ۱ بود، هر ۸ همسایه اش را برسی کن، اگر همسایه اش نیز یک بود آن ها را متحد کن و همسایه های آن ها را نیز برسی کن

۴) حال آرایه ای به سایز (طول * عرض) ماتریس بساز که فراوانی هر مجموعه را ذخیره کند

۵) حال دوباره درایه های ماتریس را پیمایش کن

۶) اگر مقدا درایه ۱ بود، مجموعه ی متناظرش را بیاب

۷) اگر فراوانی مجموعه در آرایه ی بالا ۰ بود، result را یک واحد افزایش بده

۷,۸,۲,۱ الگوریتمی برای پیدا کردن مولفه های قویاً همبند گراف های جهت دار ارائه دهید. به طوری که تنها مجاز به یک بار استفاده کردن از DFS باشید.

راه حل:

یک گراف جهت دار قویاً همبند است اگر بین هر جفت از راس های آن مسیری وجود داشته باشد. به بزرگترین زیر گراف ممکن که قویاً همبند باشد مولفه ی قوی همبندی SCC آن گراف می گوییم.با استفاده از الگوریتمی موسوم به tarjan که از DFS استفاده می کند می توانیم تمام مولفه های همبندی یک گراف را با هزینه ی زمانی O(v+e) بدست آوریم.

الگوریتم به این ترتیب اجرا می شود:

۱) جست و جوی DFS یک درخت یا جنگل DFS نتیجه می دهد.

۲) مولفه های قویاً همبند زیر درخت های درخت DFS را تشکیل می دهند.

۳) اگر ما بتوانیم head این زیر درخت را بیابیم، می توانیم تمام راس های این زیر درخت به علاوه ی خود head را ذخیره کنیم که در واقع یک مولفه ی قویاً همبند را به ما می دهد.

۴) هیچ یال بازگشتی ای از یک مولفه به مولفه ی دیگر وجود ندارد.

برای پیدا کردن head مولفه ها به محاسبه کردن آرایه های desc و low می پردازیم. low[u] راسی را نمایش میدهد که زود تر از بقیه ی راس ها مشاهده شده است که توسط زیر درختی با ریشه ی u قابل دسترسی است.یک راس u همان head است اگر disc[u] = low[u] باشد.

مقدار disc در واقع نشان دهنده ی زمانی است که برای اولین بار یک راس را در حین پیمایش DFS مشاهده می کنیم. مقدار low برای هر راس نشان دهنده ی بزرگترین جد قابل دسترس در زیر درخت آن راس است. برای هر راس u وقتی DFS شروع می شود، low همان مقدار اولین disc اش است. بعداً در حین اجرای DFS بر روی تک تک فرزندان مقدار low راس u در دو حالت تغییر می کند.

۱) اگر یال درختی داشته باشیم: low[u] = min(low[u], low[v])

۲) اگر یال بازگشتی داشته باشیم: low[u] = min(low[u], disc[v])

پیچیدگی این الگوریتم همان پیچیدگی DFS است.

تکه کد زیر الگوریتم را پیاده سازی کرده است.

```
// A C++ program to find strongly connected components in a given
// directed graph using Tarjan's algorithm (single DFS)
#include<iostream>
#include <list>
#include <stack>
#define NIL -1
using namespace std;

// A class that represents an directed graph
class Graph
{
   int V;    // No. of vertices
   list<int> *adj;    // A dynamic array of adjacency lists

   // A Recursive DFS based function used by SCC()
   void SCCUtil(int u, int disc[], int low[],
             stack<int> *st, bool stackMember[]);
public:
   Graph(int V);   // Constructor
   void addEdge(int v, int w);   // function to add an edge to graph
   void SCC();    // prints strongly connected components
};

Graph::Graph(int V)
{
   this->V = V;
   adj = new list<int>[V];
}
```

```
void Graph::addEdge(int v, int w)
{
   adj[v].push_back(w);
}

// A recursive function that finds and prints strongly connected
// components using DFS traversal
// u --> The vertex to be visited next
// disc[] --> Stores discovery times of visited vertices
// low[] -- >> earliest visited vertex (the vertex with minimum
//             discovery time) that can be reached from subtree
//             rooted with current vertex
// *st -- >> To store all the connected ancestors (could be part
//           of SCC)
// stackMember[] --> bit/index array for faster check whether
//                  a node is in stack
void Graph::SCCUtil(int u, int disc[], int low[], stack<int> *st,
                    bool stackMember[])
{
   // A static variable is used for simplicity, we can avoid use
   // of static variable by passing a pointer.
   static int time = 0;

   // Initialize discovery time and low value
   disc[u] = low[u] = ++time;
   st->push(u);
   stackMember[u] = true;

   // Go through all vertices adjacent to this
   list<int>::iterator i;
   for (i = adj[u].begin(); i != adj[u].end(); ++i)
   {
      int v = *i;  // v is current adjacent of 'u'

      // If v is not visited yet, then recur for it
      if (disc[v] == -1)
      {
         SCCUtil(v, disc, low, st, stackMember);

         // Check if the subtree rooted with 'v' has a
         // connection to one of the ancestors of 'u'
         // Case 1 (per above discussion on Disc and Low value)
         low[u]  = min(low[u], low[v]);
      }

      // Update low value of 'u' only of 'v' is still in stack
      // (i.e. it's a back edge, not cross edge).
      // Case 2 (per above discussion on Disc and Low value)
```

```
        else if (stackMember[v] == true)
            low[u]  = min(low[u], disc[v]);
    }

    // head node found, pop the stack and print an SCC
    int w = 0;  // To store stack extracted vertices
    if (low[u] == disc[u])
    {
        while (st->top() != u)
        {
            w = (int) st->top();
            cout << w << " ";
            stackMember[w] = false;
            st->pop();
        }
        w = (int) st->top();
        cout << w << "\n";
        stackMember[w] = false;
        st->pop();
    }
}

// The function to do DFS traversal. It uses SCCUtil()
void Graph::SCC()
{
    int *disc = new int[V];
    int *low = new int[V];
    bool *stackMember = new bool[V];
    stack<int> *st = new stack<int>();

    // Initialize disc and low, and stackMember arrays
    for (int i = 0; i < V; i++)
    {
        disc[i] = NIL;
        low[i] = NIL;
        stackMember[i] = false;
    }

    // Call the recursive helper function to find strongly
    // connected components in DFS tree with vertex 'i'
    for (int i = 0; i < V; i++)
        if (disc[i] == NIL)
            SCCUtil(i, disc, low, st, stackMember);
}
```

۸,۸,۲,۱ فرض کنید c, C' مولفه قویا همبند متمایز گراف جهت دار G باشند. فرض کنید یال (u, v) وجود دارد به طوری که u عضو c و v عضو C' باشد. آنگاه $f(c) > f(C')$.

راه حل:

با توجه بهاینکه مولفه همبند قوی، c یا C'، در جریان DFS راس پیدا شده اول داشتند دو حالت را در نظر میگیریم.

اگر $d(C') > d(c)$ ، x اولین راس پیدا شده در c باشد در زمان $x.d$ تمام راس ها در c و C' سفید هستند. در آن زمان G حاوی یک مسیر از x به هر راس در c است که فقط متشکل از راس های سفید هستند. از آنجا که یال (u, v) برای هر راس w عضو C' یک مسیر در G در زمان $x.d$ از x به w فقط متشکل از راس های سفید وجود دارد. بنا به قضیه مسیر سفید تمام راس ها در c, C' بچه های x در درخت DFS میشوند از آنجایی که x دیرترین زمان خانمه هر یک از بجه هایش را دارد از این رو $x.f = f(c) > f(C')$.

اگر به جای آن داشته باشیم $d(c) > d(C')$، فرض کنید y اولین راس پیدا شده در C' باشد. در زمان $y.d$ تمام راس ها در C' سفید هستند و G حاوی یک مسیر از y به هر راس در C' است که فقط از راس های سفید تشکیل شده است. با به قضیه مسیر سفید تمام راس ها در C' بچه های y در درخت DFS میشوند بنابراین $y.f = f(C')$. در زمان $y.d$ تمام راس ها در c سفید هستند. از آنجایی که یال (u, v) از c به C' وجود دارد نتیجه میگیریم که یک مسیر از C' به c نمیتواند وجود داشته باشد. در اینجا هیچ راس در c از y قابل دسترسی نیست بنابراین در زمان $y.f$ تمام راس ها در c همچنان سفید هستند پس برای هر راس w عضو c داریم $w.f > y.f$که نتیجه میدهد $f(c) > f(C')$