جزوه‌ی آموزشیِ الگوریتم برای آمادگی در المپیاد کامپیوتر

تهیه و تنظیم:

عرفان عبدی جواد عابدی ایمان جامی

فهرست

پیچیدگی الگوریتم‌ها (Order):

یادگیری طراحیِ الگوریتم بدون اطلاعات درباره‌ی پیچیدگیِ الگوریتم‌ها تقریبا ناممکن است. معمولا یادگیری برنامه‌نویسی با مجموعه مسائلی آغاز می‌شود که تنها پیاده‌سازیِ آنها مهم هستند و در واقع الگوریتمِ بهینه‌ای برای حل آنها ارائه نمی‌شود. ولی کم‌کم با ورود به مسائل پیچیده‌تر متوجه خواهید شد که لزوما هر روشی که بصورت تئوری بتواند پاسخ مسئله را تولید کند نمی‌تواند جواب اصلی مسئله باشد.

مشکل اصلی در پیاده‌سازیِ چنین مسائلی محدودیت‌هایی است که ما با آنها مواجه هستیم. محدودیت‌هایی همچون:

- محدودیت در زمان اجرای الگوریتم: با افزایش ورودی‌های برنامه، تعداد دستوراتی که باید انجام شود تا پاسخ مسئله بدست آید نیز بیشتر می‌شود. یک کامپیوترِ معمولی کمتر از 10^9 دستور در یک ثانیه انجام می‌دهد. می‌توانید برنامه‌هایی که در گذشته نوشته‌اید را بررسی کنید و تعداد دستوراتی که برای هر بار اجرا انجام می‌شود را محاسبه کنید.
- محدودیت در ذخیره‌ی اطلاعات: در هر برنامه، متغیرهایی وجود دارد که برای ذخیره‌ی اطلاعات استفاده می‌شوند. این اطلاعات روی حافظه‌ی (RAM) کامپیوتر ذخیره می‌شوند. در اجرای برنامه‌های مختلف محدودیت‌هایی روی حافظه نیز داریم که باید در نظر گرفته شوند.
- محدودیت در حجم برنامه: هر برنامه‌ای که نوشته می‌شود نهایتا به یک فایل قابل اجرا تبدیل می‌شود که خود حجم مشخصی دارد. این محدودیت در برنامه‌هایی که روی کامپیوتر نوشته می‌شوند بی‌اثر هستند و معمولا در میکروکنترلرها و میکروپروسسورها اهمیت می‌یابد.

تمامی محدودیت‌های ذکر شده سبب می‌شوند تا به ازای مسائل مختلف بهینه‌ترین الگوریتم ممکن را ارائه کنیم. در حل مسائل پیشِ رو به محدودیت در حجم برنامه نمی‌پردازیم. ولی در مسائل مختلف امکان دارد محدودیت بین زمان اجرا و یا ذخیره‌ی اطلاعات را مورد توجه قرار دهیم.

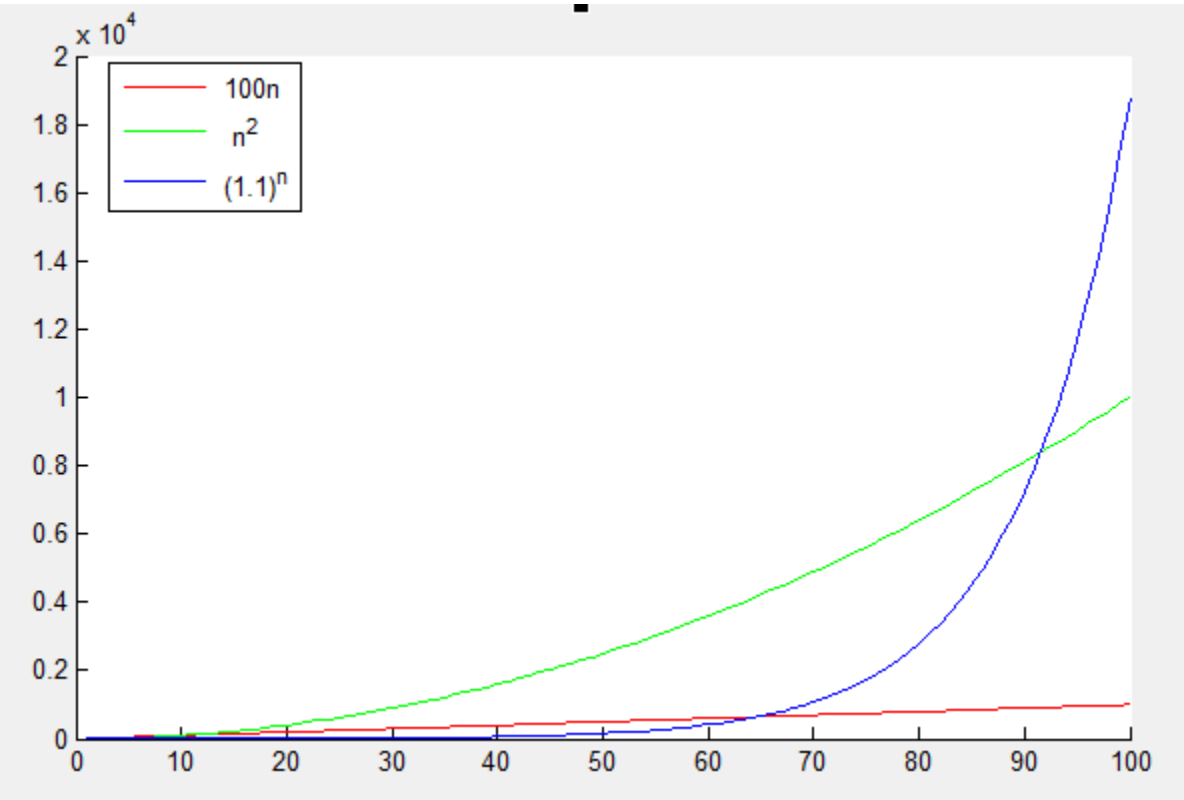
با مقدمات بالا به اهمیت بحث درباره‌ی پیچیدگی الگوریتم پی بردید. حال می‌توانیم مباحث اولیه را بیان کنیم:

شروع با یک مثال:

فرض کنید که مسئله‌ای مطرح شده و روش‌های مختلفی برای حل آن پیشنهاد شده است که همگی به پاسخ درست می‌رسند ولی زمان اجرای متفاوتی دارند. در صورتی که ورودیِ برنامه عددِ طبیعیِ n باشد:

- روش اول پس از $100n$ عملیات به جواب می‌رسد.
- روش دوم پس از n^2 عملیات به جواب خواهد رسید.
- و در نهایت روش سوم پس از 1.5^n عملیات جواب را بدست می‌آورد.

برای درک بهتر توانایی الگوریتم‌های بالا به ازای ورودی‌های مختلف زمان اجرای هرکدام را محاسبه کرده و نمودار زیر بدست آمده است:



نمودار ۱. تحلیل مجانبی پیچیدگی الگوریتم‌ها

همانطور که مشاهده می‌کنید هرچند که به ازای ورودی‌های بسیار کم ضریب پشت اعداد تاثیرگذار است، ولی درصورتی که ورودی افزایش یابد، جملاتی که برحسب ورودی (n) هستند رشد تابع را رقم خواهند زد.

قبل از اینکه بتوانیم درباره‌ی الگوریتم‌های مختلف مقایسه‌ای صورت دهیم باید زبان واحدی برای بیان پیچیدگی هر الگوریتم بیابیم. در مثال بالا دیدید که ضرایب در تحلیل پیچیدگی تاثیر شگرفی ندارند. پس در پیچیدگی از نوشتن آنها صرف‌نظر می‌کنیم. در اینصورت به عنوان مثال پیچیدگیِ دو الگوریتم که اولی $10n^3$ و دیگری $\frac{1}{2}n^3$ عملیات خواهند داشت یکسان است.

مهمتر از هر بحثی باید بتوانیم بین الگوریتم‌های مختلف مقایسه کنیم و الگوریتمی که پیچیدگیِ کمتری دارد را انتخاب کنیم. در نتیجه باید ابزاری برای مقایسه‌ی بین دو الگوریتم ارائه کنیم:

همانطور که برای مقایسه‌ی اعداد از سه نمادِ " $< = >$ " استفاده می‌کنیم برای مقایسه‌ی پیچیدگیِ الگوریتم‌ها نیز سه نماد داریم که هریک را معرفی خواهیم کرد. در تعریف‌های زیر فرض کنید که دو تابع $T(n)$ و $F(n)$ داده شده است و می‌خواهیم بین این دو مقایسه انجام دهیم.

تعریف ۱. می‌گوییم تابع $T(n) = O(F(n))$، اگر و تنها اگر اعداد ثابت c و n_0 وجود داشته باشند که به ازای هر $n > n_0$ داشته باشیم: $T(n) \leq cF(n)$.

در واقع تعریف بالا زمانی درست است که ضریبی از $F(n)$ از جایی به بعد بزرگتر مساوی $T(n)$ باشد.

تعریف ۲. می‌گوییم تابع $T(n) = \Omega(F(n))$، اگر و تنها اگر اعداد ثابت c و n_0 وجود داشته باشند که به ازای هر $n > n_0$ داشته باشیم: $T(n) \geq cF(n)$.

در واقع تعریف بالا زمانی درست است که ضریبی از $F(n)$ از جایی به بعد کوچکتر مساوی $T(n)$ باشد.

مثال ۱. اگر $T(n) = a_k n^k + a_{k-1}n^{k-1} + \cdots + a_0$ باشد، آنگاه $T(n) = O(n^k)$ خواهد بود. در واقع جملات با توان کمتر در محاسبه‌ی پیچیدگی تاثیرگذار نیستند و این نکته برای مقایسه‌ی آنها کار ما را آسان‌تر می‌کند. ولی $T(n) \neq O(n^{k-1})$ چون $\frac{T(n)}{n^{k-1}} \to \infty$ و در نتیجه هیچ ضریبی همانند c با شرایط مسئله نمی‌توان یافت.

مثال ۲. همچنین با تعاریف بالا داریم:

$$n^k = \Omega(n^{k-1})$$

$$2^{10n} = \Omega(2^n)$$

تعریف ۳. تابع $T(n) = \theta(F(n))$ است، اگر و تنها اگر $T(n) = O(F(n))$ و $T(n) = \Omega(F(n))$.

هر چند که تعاریفِ فوق کمی خشک و ریاضی‌وار به نظر می‌آیند، ولی پس از حل چند مثال و بررسی پیچیدگی الگوریتم‌های مختلف به اهمیت غیرقابل انکار آنها و سادگیِ کار با آنها پی خواهید برد.

در الگوریتم‌های مختلفی که ارائه می‌شود معمولا با چند حالت مواجه خواهید شد:

مهمترین پیچیدگی‌هایی که با آنها سر و کار دارید، پیچیدگی‌هایی به صورت چندجمله‌ای $\theta(n^k)$، نمایی $\theta(a^n)$، لگاریتمی $\theta(\log n)$ و یا ترکیبی از آنها خواهد بود که باید کم‌کم به آنها تسلط یابید.

جستجوی دودویی (Binary Search):

جستجوی دودویی ابزاری ساده اما بسیار مهم در حل مسائل الگوریتمی است که در بسیاری از مسائل به طور مستقیم و یا به عنوان ابزاری کمکی مورد استفاده قرار می‌گیرد.

ابتدا مسئله‌ی جستجوی دودویی را بررسی می‌کنیم و سپس کاربردهایی از این روش در طراحی الگوریتم ها را معرفی خواهیم کرد.

مسئله‌ی جستجوی عدد: n عدد متمایزِ $a_1, a_2, \dots, a_n$ از کوچک به بزرگ مرتب شده‌اند و یک عدد x نیز داده شده است. می‌خواهیم بدانیم آیا x در بین n عددِ داده شده وجود دارد یا نه؟ و در صورت وجود اندیس عدد یافت‌شده را اعلام کند.

در مواجهه با این مسئله ساده‌ترین ایده‌ای که به ذهن می‌رسد، شروع از ابتدای دنباله و مقایسه‌ی هر کدام از n عدد با عدد x است تا اگر عددی مثل a_i با x مساوی بود اندیس عدد (i) را به عنوان عدد مورد نظر اعلام کنیم و در صورتی که برابر با هیچ‌کدام از آنها نبود اعلام کنیم که چنین عددی وجود ندارد.

این روش یک مشکل بزرگ دارد و آن این است که در بعضی حالات (کدام حالات؟) ما مجبوریم تمام اعداد را بررسی کنیم تا متوجه بشویم که عدد مورد نظر وجود دارد یا نه.

اما با تعداد بسیار کمتری جستجو هم می‌توان فهمید که آیا x در بین اعداد وجود دارد یا نه!!!

نکته‌ای که در مسئله وجود دارد و ما در راه‌حل بالا از آن غافل شده‌ایم مرتب بودن دنباله است. ما از این فرض استفاده نکرده‌ایم و اگر دنباله مرتب هم نبود روش بالا پاسخ درست را اعلام می‌کرد.

حال ببینیم مرتب بودنِ دنباله چه کمکی در حل مسئله می‌کند: فرض کنید x را با عنصر a_i مقایسه کردیم، اگر این دو عدد با هم برابر بودند عدد و اندیس موردنظر را یافته‌ایم اما درغیر این صورت a_i یا از x بزرگتر است و یا کوچکتر. اگر a_i از x بزرگتر بود دیگر لازم نیست به ازای عددهای با اندیس بزرگتر از i دنبال x بگردیم، چون همه‌ی این اعضا از a_i بزرگتر هستند اما x از a_i کوچکتر است. در حالت دیگر یعنی اگر a_i از x کوچکتر باشد به ازای عددهای با اندیس کوچکتر از i دیگر لازم نیست دنبال x بگردیم (چرا؟). بنابراین توانستیم فضایی که در آن دنبال x می‌گردیم را کوچک کنیم!

اما با استفاده از این روش و اطلاعات اضافه‌ای که به ما می‌دهد اگر همچنان از ابتدای اعداد شروع کنیم و به جلو برویم باز هم در حالاتی مجبوریم همه‌ی n عدد را بررسی کنیم تا مطمئن شویم x وجود دارد یا نه. پس استفاده از این اطلاعات اضافی هم به تنهایی برای رسیدن به جواب مسئله کافی نیست.

ایده‌ی اصلی این است که ما عنصری را برای مقایسه با x انتخاب کنیم که پس از مقایسه، فضای مسئله به مقدار قابل توجهی کوچک شده باشد. این عنصر، $a_{n/2}$ یعنی عنصر با اندیس $n/2$ است. در مقایسه‌ی این عنصر با x اگر برابر بودند اندیس آن عدد مورد نظر خروجی است اما اگر x از $a_{n/2}$ کوچکتر باشد دیگر کافی است که در اعداد a_1 تا $a_{\frac{n}{2}-1}$ به دنبال x بگردیم و اگر هم x بزرگتر از $a_{n/2}$ باشد کافی است بین اعداد $a_{\frac{n}{2}+1}$ تا a_n دنبالِ x بگردیم که در هر کدام از این دو صورت فضای مسئله نصف می‌شود.

در ادامه با داشتن فضای جدید که $n/2$ عضو دارد باید به دنبال x باشیم. آن قدر عملیات بالا را ادامه می‌دهیم تا یا در حین انجام مراحل به عنصر x برسیم و اندیس آن را به عنوان خروجی بدهیم یا فضایی که در آن دنبال x می‌گردیم به یک عضو برسد و آن عضو هم برابر x نباشد که در این صورت x در بین این n عدد نبوده است.

به این روشِ جستجوی عدد در یک دنباله مرتب، "جستجوی دودویی" می‌گویند که دلیل این نام‌گذاری از روش انجام آن که در هر مرحله بازه‌ی اعداد جستجو را نصف می‌کنیم نتیجه شده است.

شرایط مهمی مسئله که باعث شد بتوانیم از جستجوی دودویی استفاده کنیم، مرتب بودن اعداد بود و اگر این شرایط وجود نداشت ما دیگر نمی توانستیم از این روش استفاده کنیم چرا که دیگر مقایسه‌ها هیچ اطلاعات اضافه‌ای در مورد بقیه اعداد نمی‌داد.

حال ببینیم در این روش ما با چند مرحله مقایسه به جواب رسیده‌ایم. در هر مرحله با مقایسه عنصر وسط دنباله تعداد اعدادی که بین آن‌ها دنبال x می‌گردیم نصف می‌شود و در انتها نیز وقتی به یک عنصر در فضای جستجوی‌مان رسیدیم کار تمام شده است. بنابراین تعداد مراحل (مقایسه‌ها) برابر است با تعداد دفعاتی که باید n را نصف کنیم تا به یک برسیم که برابر است با $\log(n)$. یعنی ما با تعداد $\log(n)$ مرحله (مقایسه) توانستیم مسئله را حل کنیم که این تعداد بسیار بهینه‌تر از حل اولیه‌ی مسئله یعنی n مقایسه است. برای داشتن دیدی از تفاوت تعداد این مقایسه‌ها فرض کنید که n برابر ۱۰۰۰۰۰۰ باشد یعنی ما باید بین ۱۰۰۰۰۰۰ عدد x را جستجو کنیم در این صورت با راه‌حل اول ۱۰۰۰۰۰۰ مقایسه باید انجام می‌دادیم اما با روش جستجوی دودویی تعداد مقایسه‌ها حدودا ۲۰ مقایسه خواهد بود که اختلاف بسیار زیادی با هم دارند.

کاربردها:

- یکی از کاربردهای مهم و رایج جستجوی دودویی، جستجو روی جواب مسئله است.

یک مثال از اینگونه مسائل به شکل زیر است:

مثال ۳. n عدد طبیعی و نامرتب و یک عدد k که $1 < k < n$ داده شده است. کوچکترین عدد طبیعی را بدون مرتب کردن دنباله پیدا کنید که تعداد اعداد بزرگتر از آن در بین این اعداد حداقل kتا باشد.

حل: می‌دانیم که جواب مسئله بین ماکسیمم این n عدد (که max می‌نامیم) و مینیمم این اعداد (که min می‌نامیم) است. برای حل مسئله در بازه‌ی (min, max) جستجوی دودویی را اعمال می‌کنیم. اما دیگر در مقایسه برای پیداکردن فضای جدیدِ جستجو، تنها یک مقایسه کافی نیست. عضو وسط بازه‌ی (min, max) را در نظر می‌گیریم و a می‌نامیم. حال با یک دور چک کردن کل اعداد و مقایسه با a متوجه می‌شویم چند عدد بین این n عدد وجود دارد که از a بزرگتر است. اگر این تعداد از k بیشتر بود ما ادامه‌ی جستجو را باید در بازه‌ی $(a + 1, max)$ پی بگیریم چون باید عددی بزرگتر پیدا کنیم که تعداد اعداد بزرگتر از آن بین n عدد کمتر شود. و در غیر این صورت باید در بازه‌ی (min, a) دنبال جواب مسئله بگردیم. بنابراین توانستیم فضای جستجو را نصف کنیم. حال آنقدر این روش را روی فضای جدید اعمال می‌کنیم تا عدد مورد نظر را پیدا کنیم.

- کاربرد دیگری از این روش استفاده به عنوان ابزاری برای طراحی الگوریتم‌های پیچیده است. به طور فرض کنید در حال طراحی الگوریتمی هستیم و نیاز داریم در بین یک دنباله مرتب از اعداد عددی را جستجو کنیم و یا نزدیکترین عدد به یک عدد در بین آنها را پیدا کنیم که همه اینها با استفاده از جستجوی دودویی به سادگی و کارآمدی بالا امکان پذیر است.

برنامه‌سازی پویا (Dynamic Programming):

برنامه‌سازی پویا ابزاری بسیار قدرتمند در حل مسائل الگوریتمی در سطوح مختلف است که در بسیاری از مسائل (که حتی گاهی فکر کردن به آنها هم سخت است) راه‌گشاست.

ایده اصلی این روش همان حل مسئله با استفاده از استقرا است.

مسئله: یک دنباله از اعداد حقیقی a_1 تا a_n داریم. می‌خواهیم حاصل جمع اعضای زیردنباله‌ای مانند a_i تا a_j (پشت سر هم) را پیدا کنیم که بیشترین جمع را در بین همه‌ی زیردنباله‌های این n عدد داشته باشد!

در اولین نگاه ممکن است ادعا کنیم انتخابِ همه‌ی اعداد دنباله بیشترین جمع ممکن را خواهد داشت اما نکته اینجاست که در سوال گفته نشده که همه‌ی اعداد مثبت هستند بنابراین ممکن است اعداد منفی نیز داشته باشیم که دیگر لزوما کل دنباله بیشترین جمع ممکن را ندارد (اینگونه اشتباهات رایج در نگاه اول به خاطر پیش‌فرض ما از سوالاتی است که تا بحال دیدیم چون در بسیاری از آنها اعداد دنباله مثبت هستند).

برای روشن تر شدن سوال یک مثال می‌آوریم. دنباله‌ی $1, -3, 3.2, -1, 2, -3, 1$ را در نظر بگیرید. زیردنباله‌ی مدنظر مسئله زیردنباله‌ی $3.2, -1, 2$ است که بیشترین جمع یعنی 4.2 را در بین همه زیردنباله‌های متوالی دارد.

در صورتی که همه اعداد دنباله منفی باشند دنباله‌ی تهی بیشترین جمع را دارد چرا که مجموع هر زیردنباله غیرتهی منفی خواهد شد!

راه‌حل ساده‌ی این مسئله استفاده از حلقه‌ی تودرتو است که به ازای همه‌ی زیردنباله‌های متوالی مجموع آنها را بدست آورده و بین‌شان ماکسیمم را پیدا می‌کنیم. اما این الگوریتم خیلی کارآمد نیست چرا که یک دنباله با n عضو $n(n-1)/2+1$ زیردنباله دارد (چرا؟) و ما به ازای هرکدام از آنها باید این مقدار جمع را بدست آوریم. بنابراین بطور شهودی حداقل همین تعداد عملیات برای بدست آوردن جواب نیاز داریم که در اصلاح می‌گوییم از $O(n^2)$ هزینه‌ی زمانی نیاز داریم. اما می‌خواهیم الگوریتمی ارائه دهیم که هر یک از n عضو را یکبار بررسی کند و در نهایت وقتی یک دور همه‌ی اعداد بررسی شدند حاصل‌جمع را بدست آورده باشد.

در ادامه با دید استقرایی به مسئله نگاه می‌کنیم! یعنی فرض می‌کنیم برای مقادیر کوچکتر از n مسئله را حل کرده‌ایم و سپس با استفاده از جواب بدست آمده برای مقادیر کوچکتر از n جواب مسئله برای n را بدست می‌آوریم.

به عنوان پایه‌ی استقرا فرض کنید که $n = 1$ باشد اگر a_1 عددی منفی باشد جواب موردنظر زیردنباله تهی خواهد بود که مجموع اعضایش 0 است اما اگر a_1 مثبت باشد جواب همان a_1 خواهد بود!

فرض استقرا: فرض می‌کنیم که به ازای دنباله‌های به طول کمتر از n بتوانیم زیردنباله‌ی متوالی با بیشترین جمع اعضا را پیدا کنیم.

با این فرض یک دنباله از n عدد حقیقی را در نظر می‌گیریم و عضو آخر را حذف می‌کنیم. طبق فرض استقرا زیردنباله با بزرگترین مجموع را پیدا می‌کنیم. اگر مجموعی که پیدا کردیم 0 بود یعنی تمام a_1 تا a_{n-1} منفی یا صفر بوده‌اند، در اینصورت اگر a_n مثبت بود خود a_n همان مجموع بیشینه خواهد بود وگرنه مجموع همان 0 می‌ماند. اما فرض کنید جوابی که با حذف a_n پیدا کرده‌ایم منفی نباشد و زیردنباله متوالی پیدا شده که بیشترین جمع را دارد a_i تا a_j باشد که بزرگترین جمع را می‌دهد. اگر j برابر با $n - 1$ باشد گسترش جواب با اضافه‌کردن عضو آخر به راحتی امکان‌پذیر است به این شکل که اگر a_n مثبت بود می‌توانیم آن را به زیردنباله‌ی پیدا شده اضافه کنیم و زیردنباله قطعا باز هم بزرگترین جمع را خواهد داشت اما در صورتی که j کوچکتر از $n - 1$ باشد دیگر کار به این راحتی نیست! چون زیردنباله‌ای که ماکسیمم جمع را دارد ممکن است همان زیردنباله‌ی a_i تا a_j بماند و یا زیردنباله‌ای باشد که a_n عضو آخر آن است! بنابراین فرض استقرا به میزان کافی قوی نیست که بتوانیم زیردنباله با ماکسیمم جمع را تشخیص بدهیم!

باید جوری فرضمان را قوی کنیم که با اضافه کردن a_n بتوانیم بفهمیم زیردنباله‌ی با مجموع ماکسیمم که a_n عضو آخر آن است چه مقداری است.

فرض استقرا را به این شکل تقویت می‌کنیم: فرض می‌کنیم به ازای هر دنباله با طول کمتر از n حاصل جمع زیردنباله با ماکسیمم مجموع (که با $maxsum$ نشانش می‌دهیم) و هم چنین به ازای هر i حاصل‌جمع زیردنباله با ماکسیمم مجموع که a_i عضو آخر آن زیر دنباله باشد (که با d_i نشان می‌دهیم) را داریم.

در استقرا به عنوان پایه مقدار $maxsum$ را برابر ماکسیمم a_1 و 0 و مقدار d_1 را برابر a_1 قرار می‌دهیم. چون در ابتدا تنها دنباله‌ای که a_1 درون آن است خودش است و مقدار جمع بزرگترین زیردنباله ($maxsum$) هم با داشتن تنها یک عدد در دنباله درصورتی که آن عضو مثبت باشد خودش است و در غیر این صورت 0 است.

دنباله اعداد a_1 تا a_n را در نظر می‌گیریم و a_n را حذف می‌کنیم. برای بقیه‌ی دنباله طبق فرض استقرا $maxsum$ و همچنین به ازای هر $(i < n)$ ، d_i (حاصل جمع زیردنباله با بیشترین جمع که a_i حتما عضو آخر آن باشد) را داریم. حالا باید با اضافه کردن a_n اطلاعاتمان در مورد مسئله را در این وضعیت بروز کنیم.

اطلاعاتی که باید بروز شوند حاصل maxsum و d_n هستند (d_iها که $i < n$، با اضافه شدن a_n تغییری نمی‌کنند). برای این بروزرسانی ابتدا باید d_n را محاسبه کنیم و اگر این مقدار از maxsum محاسبه شده در مرحله قبل بیشتر بود به این معنی است که یک زیر دنباله که a_nعضو آخر آن است پیدا کرده‌ایم که جمعی بزرگتر از زیردنباله‌هایی که قبلا پیدا کرده بودیم دارد بنابراین maxsum را باید برابر این مقدار جدید قرار دهیم.

d_n از دو طریق ممکن است بدست بیاید. یکی این که a_n به زیردنباله‌ای که ماکسیمم جمع را دارد و عضو آخر آن a_{n-1} است اضافه شود و یا این که a_n خودش به تنهایی یک زیردنباله را تشکیل دهد. بنابراین d_n برابر ماکسیمم این دو مقدار خواهد بود یعنی a_n و $d_{n-1} + a_n$.

همان طور که در بالا ذکر شد بعد از محاسبه‌ی d_n اگر d_n از maxsum بزرگتر شده بود مقدار maxsum برابر d_n قرار خواهد گرفت وگرنه همان مقدار قبلی خود را حفظ خواهد کرد.

در واقع اگر بخواهیم برای بروزرسانی یک شبه‌کد بنویسیم به این شکل خواهد بود.

$d_n = \max(d_{n-1} + a_n,\ a_n);$

$maxsum = \max(maxsum, d_n);$

با توجه به موارد گفته شده اگر از فرض استقرا شروع کنیم و با یک حلقه به ازای تمام اعداد 2 تا n عملیات update را انجام دهیم در نهایت مقدار درون maxsum مقدار موردنظر یعنی مجموع بزرگترین زیردنباله خواهد بود!

برای دنباله‌ی نمونه که در بالا هم بررسی شد جدول مقادیر d_i یا به اصطلاح جدول مقادیر داینامیک در هر مرحله را نشان می‌دهیم تا روند آن را دنبال کنید.

دنباله:

$$1, -3, 3.2, -1, 2, -3, 1$$

	d_1	d_2	d_3	d_4	d_5	d_6	d_7	**maxsum**
n=1	1	-	-	-	-	-	-	**1**
n=2	1	-3	-	-	-	-	-	**1**
n=3	1	-2	3.2	-	-	-	-	**3.2**
n=4	1	-2	3.2	2.2	-	-	-	**3.2**
n=5	1	-2	3.2	2.2	4.2	-	-	**4.2**
n=6	1	-2	3.2	2.2	4.2	1.2	-	**4.2**
n=7	1	-2	3.2	2.2	4.2	1.2	**2.2**	**4.2**

این دید در حل مسائل با استفاده از تکنیک استقرا و یافتن جواب از روی حالات قبلی را برنامه‌سازی به روش پویا یا dynamic programming می‌گویند که در آینده با دیدن مسائل بیشتر به اهمیت و قدرت بسیار زیاد این روش پی خواهید برد!

پسگرد(Back Tracking):

برای حل بسیاری از مسائل نیاز به استفاده از الگوریتم پیچیده‌ای وجود ندارد و استفاده از ساده‌ترین روش ممکن جواب را به ما می‌دهد. ساده‌ترین روش، بررسی تمام حالات ممکن در مساله است. با توجه به اینکه مساله از شما چه خواسته است می‌توانید انتخاب کنید که از این روش برای حل مسئله استفاده کنید. سوال‌هایی همانند مثال‌های زیر می‌توانند اینگونه حل شوند:

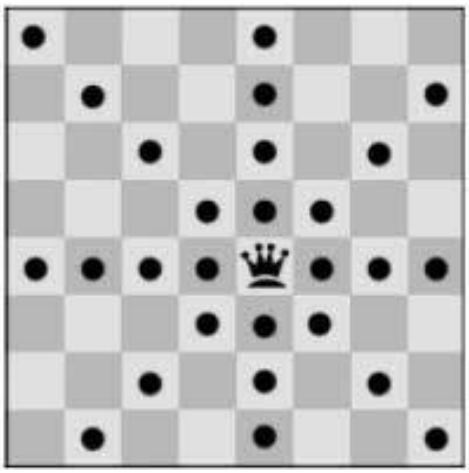
۱. ... بین حالات ممکن، حالتی را بیابید که هزینه مینیمم باشد (مسائل بهینه‌سازی).

۲. یافتن حالت خاصی از مسئله با ویژگیِ مشخص.

مهمترین ایده‌ای که در حل اینگونه مسائل بدان توجه می‌شود توانایی حذف شاخه‌ی بزرگی از حالات مورد بررسی است تا به کمک آن سرعت اجرای برنامه را افزایش دهیم. در حالت کلی زمان بررسی حالات مختلف بسیار زمان‌بر خواهد بود و برای رسیدن به جواب بهینه باید راهکاری برای کاهش این حالات ارائه دهیم.

با ذکر یک مثال شروع می‌کنیم:

مثال ۴. (مسئله‌ی n وزیر) جدولی $n \times n$ داده شده است. می‌خواهیم n وزیر با قوانین بازیِ شطرنج طوری در خانه‌های آن قرار دهیم بطوری که هیچ‌کدام از آنها یکدیگر را تهدید نکنند. درصورتی که روشی برای اینکار وجود دارد، مکان هر وزیر را در خروجی چاپ کنید. خانه‌هایی که وزیر می‌تواند تهدید کند در شکل زیر نشان داده شده‌اند:



شکل ۱. حرکت وزیر در شطرنج

جواب. برای حل این سوال، می‌خواهیم تمام حالات ممکن برای قرارگیری n وزیر روی صفحه‌ی شطرنج را بررسی کنیم و در صورتی که یکدیگر را تهدید نمی‌کردند به عنوان جواب چاپ کنیم.

همانطور که بالاتر گفتیم مهمترین نکته در استفاده از Back Tracking حذف حالات ناخواسته است. در واقع باید تلاش کنیم با کمترین بررسی ممکن حالت مناسب را بیابیم. برای پیمایش این حالات روش‌های مختلفی را می‌توانیم پیاده‌سازی کنیم که به ترتیب درباره‌ی آنها صحبت می‌کنیم:

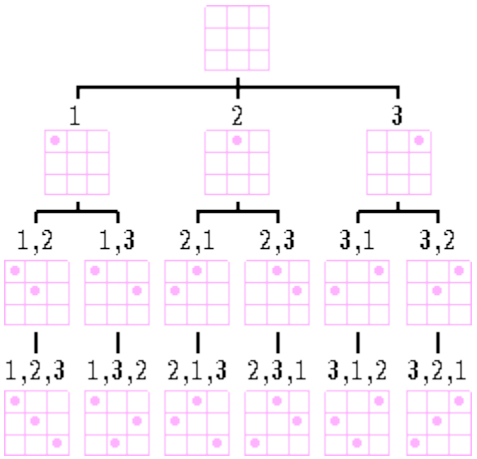
۱) در ساده‌ترین روش، باید n وزیر را در n خانه‌ی مختلف از جدول قرار دهیم و در نهایت بررسی کنیم که هیچ دو وزیری یکدیگر را تهدید نمی‌کنند. احتمالا به نظر شما هم آمده که حالات بسیاری زیادی از موارد مورد بررسی خیلی ساده حذف می‌شوند چرا که دو وزیر در یک سطر و یا یک ستون قرار خواهند گرفت. پس روش بهتری ارائه می‌دهیم.

۲) در این حالت از یک تابع بازگشتی استفاده می‌کنیم. در هر مرحله، در هر سطر یک وزیر قرار می‌گیرد و به ازای هر قرارگیری، همان تابع به ازای سطر بعد صدا زده می‌شود. در نهایت اگر روشی پیدا شد، خروجی تابع true خواهد بود و در غیر اینصورت false برگردانده می‌شود. هرچند که این روش بهینه به نظر می‌آید ولی بررسی این موضوع که وزیری که هم‌اکنون می‌خواهیم قرار دهیم در چه خانه‌هایی می‌تواند باشد نیز باید به خوبی پیاده‌سازی شود. روش اولیه این است که آرایه‌ای دوبعدی تعریف می‌کنیم و در خانه‌ای از آن true قرار دهیم اگر وزیری آن را تهدید می‌کند. در صورتی‌که وزیری در یک خانه قرار می‌گیرد باید خانه‌های آرایه بروزرسانی شوند که زمان‌بر خواهد بود.

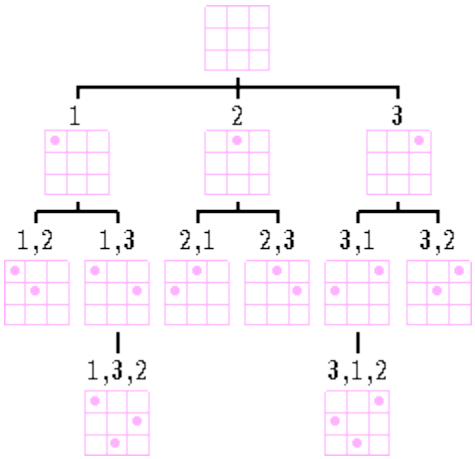
۳) روش نهایی شبیه به روش دوم است با این تفاوت که به جای نگهداری آرایه‌ی دوبعدی، سه آرایه‌ی یک بعدی تعریف می‌کنیم. هر وزیر سطر، ستون و دو خط موازی قطرهای جدول را تهدید می‌کند. هیچ دو وزیری نباید وجود داشته باشند که سطر و یا ستون مشترکی داشته باشند و همچنین اختلاف و یا مجموع سطر و ستون آنها با یکدیگر برابر باشد (چرا؟). تابعی که ارائه کردیم هیچ دو وزیری را در یک سطر قرار نمی‌دهد. در نتیجه تنها سه آرایه برای ذخیره‌سازی وضعیت خانه‌های ممکن تعریف می‌کنیم و در هر مرحله نیاز به بروزرسانیِ تنها سه خانه خواهیم داشت.

اکنون شما با فضای اصلیِ حاکم بر این الگوریتم آشنا شدید. همانطور که در مثال فوق دیده شد برای رسیدن به حالتِ مورد نظرِ حکم، ما تمام حالات موجود برای قرارگیری وزیرها را بررسی نکردیم بلکه در طول الگوریتم حالات بسیار زیادی را از همان ابتدا حذف کردیم.

شکل زیر را ببینید. در هر مسئله از Back Tracking شما برای رسیدن به حالات مورد نظر مسئله از ساختار درخت ریشه‌دار استفاده می‌کنید که برگ‌های این درخت، حالات مختلف مسئله هستند. هر چه زودتر متوجه شویم که در زیردرخت مورد بررسی حالت بهینه‌ای وجود ندارد، تعداد حالات بیشتری را حذف کرده‌ایم و درنتیجه سرعت اجرای الگوریتم را افزایش داده‌ایم. پس تلاش ما این است که زیردرختی بزرگتر از حالات را حذف کنیم. به مثال بعدی دقت کنید:



شکل ۲. بررسی تمام حالات



شکل ۳. حذف حالات غیر بهینه

مثال ۵. جهانگردی به کشوری سفر کرده است که n شهر دارد و بین تمام شهرها جاده وجود دارد. ولی زمان سفر از هر شهر به شهر دیگر می‌تواند متفاوت از یکدیگر باشد. می‌خواهیم کمترین مدت زمانی را بیابیم که این جهانگرد از یک شهر شروع به سفر کند، به هر شهر یک بار سفر کند و در نهایت به شهر اولیه بازگردد.

جواب. این سوال، یکی از معروف‌ترین مسائل حوزه‌ی الگوریتم است که TSP نامیده می‌شود. برای حل مسئله تمام دورهای ممکن را می‌پیماییم و مجموع زمان را محاسبه کرده و اگر بهینه بود انتخاب می‌کنیم. با توجه به اینکه شما می‌خواهید یک بار کشور را دور بزنید فرقی نمی‌کند که از چه شهری شروع کنید.

در هر مرحله شهرهایی که تاکنون به آنها سفر نکرده‌ایم را لیست کرده و به همه‌ی آنها به نوبت سفر می‌کنیم تا بالاخره به حالت شهر اولیه بازگردیم.

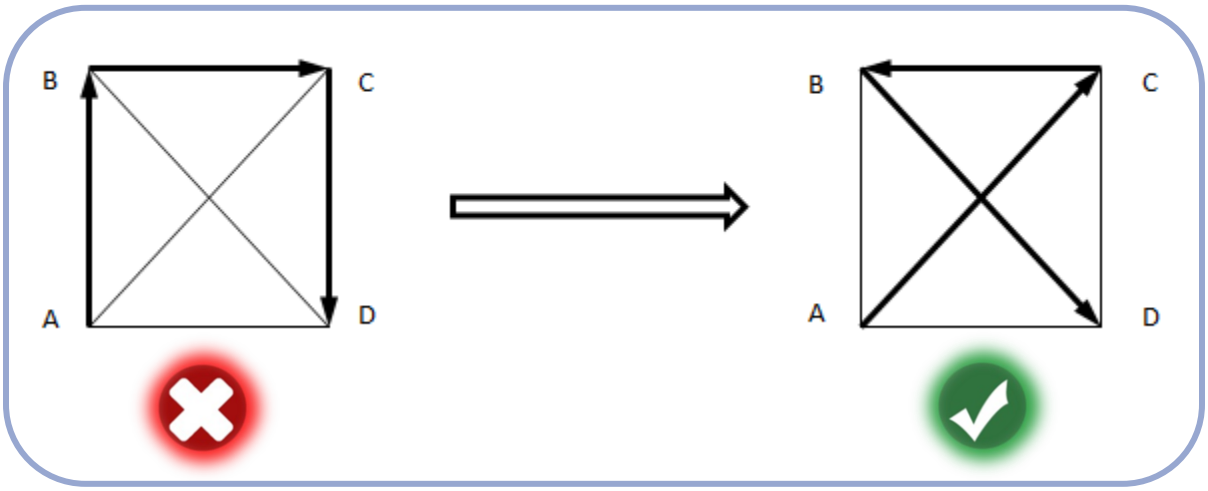
باز هم تلاش خود را می‌کنیم تا حالات غیربهینه را حذف کنیم. برای اینکار راهکارهای زیر را پی می‌گیریم:

۱) در هر مرحله بین شهرهایی که می‌توانیم سفر کنیم، شهرها را به ترتیب فاصله انتخاب می‌کنیم. چون احتمال دست یافتن به مسیر بهینه در شهرهای نزدیکتر بیشتر است. (شاید این سوال پیش آید که مگر قرار نیست که ما همه‌ی حالات را بپیماییم، پس چه فرقی می‌کند که با چه ترتیبی مسیرمان را مشخص کنیم. در پاراگراف زیر کاربرد این ایده مشخص می‌شود.)

۲) فرض کنید مدت زمان بهترین سفرمان، Answer باشد. اگر در مرحله‌ای از سفر، مدت زمانمان بیش از Answer شد این مسیر بهینه نخواهد بود. در نتیجه همیشه این شرط را در طول حرکت بررسی می‌کنیم تا حالات غیربهینه حذف گردند.

۳) فرض کنید که آخرین چهار شهری که ما تا به حال پیموده‌ایم به ترتیب A و B و C و D باشند. در صورتی که:

$$e_{AB} + e_{BC} + e_{CD} > e_{AC} + e_{CB} + e_{CD} \implies e_{AB} + e_{CD} > e_{AC} + e_{CD}$$



شکل ۴. جستجوی مسیر بهینه‌تر

ترتیب سفر ما به این چهار شهر می‌توانست بهتر باشد (با ترتیب A و C و B و D). در نتیجه در این حالت نیز سفرمان را متوقف می‌کنیم.

در هر مسئله‌ی Back Tracking می‌توان ایده‌های بسیاری را بررسی کرد و لزوما به ایده‌هایی که ما مطرح کردیم ختم نمی‌شود. شرط‌هایی که گاها باید پیاده‌سازی و بررسی شوند تا متوجه شویم به زمان اجرای برنامه کمک می‌کنند یا خیر.

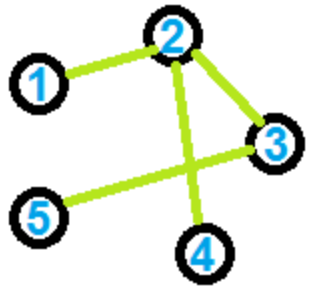
گراف:

در بسیاری از مسائل، می‌توان مساله را به گراف مدل کرد تا با استفاده از الگوریتم‌هایی که در گراف بلدیم سوال را حل کنیم. به عنوان مثال در سوالی گفته می‌شود کشوری داریم شامل چندین شهر که بین بعضی از شهر ها جاده وجود دارد. می‌خواهیم با عبور از کمترین جاده از شهر ۱ به شهر n برسیم. در این صورت می‌توان شهرها را با راس در گراف و جاده ها را با یال‌های بین آنها مدل کرد و سپس با استفاده از الگوریتم‌هایی که جلوتر به آنها می‌پردازیم فاصله‌ی بین شهرها را بدست می‌آوریم.

قبل از اینکه وارد الگوریتم‌های اولیه‌ی گراف شویم ابتدا باید ببینیم هنگامی که می‌خواهیم یک گراف را ذخیره کنیم چه روش‌هایی پیشِ رو داریم.

روش‌های ذخیره‌ی گراف در برنامه‌نویسی:

هر گرافی از تعدادی راس و یال تشکیل شده است، یکی از ساده‌ترین روش‌ها برای نگهداری یک گراف استفاده از ماتریس مجاورت است. برای مثال فرض کنید گراف زیر را داشته باشیم:



شکل ۵. گراف مثال

که ماتریس مجاورت آن می‌شود:

شماره راس	۱	۲	۳	۴	۵
۱	۰	۱	۰	۰	۰
۲	۱	۰	۱	۱	۰
۳	۰	۱	۰	۰	۱
۴	۰	۱	۰	۰	۰
۵	۰	۰	۱	۰	۰

همانطور که می‌بینید یک جدول ۵ در ۵ داریم و اگر بین دو راس یال باشد در جدول در درایه‌ی متناظر با آن دو، یک و در غیر اینصورت صفر قرار داده‌ایم.

در بسیاری از مسائل گراف موردنظر به صورت ورودی به ما داده می‌شود، در اغلب اوقات در ورودی به ما یک n (تعداد راس‌های گراف) یک m (تعداد یال‌های گراف) و در m خط بعد به ما m یال به صورت a b داده می‌شود که نشانگر وجود یال بین a و b می‌باشد. برای مثال گراف شکل بالا به فرمت گفته شده، به صورت زیر خواهد بود:

۵ ۴

۲ ۱

۳ ۲

۴ ۲

۳ ۵

برای پیاده‌سازی آن از آرایه دوبعدی استفاده می‌کنیم.

در زیر برنامه‌ی خواندن یک گراف و ریختن آن در ماتریس مجاورت نوشته شده است:

```
1  #include <iostream>
2  #include <vector>
3
4  using namespace std;
5
6  int g[100][100];
7  int n, m;
8
9  int main(){
10     cin >> n >> m;
11     for(int i = 0; i < m; i++){
12        int a, b;
13        cin >> a >> b;
14        g[a][b] = g[b][a] = 1;
15     }
16     system("pause");
17 }
18
```

از مزایای ماتریس مجاورت سادگیِ نوشتن آن است اما از معایب آن می‌توان به افزایش زمان اجرا اشاره کرد.

اما روش دیگر، نگهداری لیست همسایه‌های هر راس است. در این روش برای هر راس یک وکتور تعریف می‌کنیم و هنگامی که همسایه‌های رئوس مختلف را می‌خوانیم آنها را به ته لیست هرکس اضافی می‌کنیم برای مثال لیست مجاورت رئوس بالا می‌شود:

۲:۱

۳:۲،۱،۴

۲:۳،۵

۲:۴

۳:۵

کد زیر روش خواندن ورودی و ریختن آن در لیست مجاورت را نشان می‌دهد:

```
#include <iostream>
#include <vector>

using namespace std;

vector <int> [N] adj;

int main(){
    int n, m;
    cin >> n >> m;
    for(int i=0; i < m; i++){
       int a, b;
       cin >> a >> b;
       adj[a].push_back(b);
       adj[b].push_back(a);
    }
    system("pause");
}
```

الگوریتم‌های اولیه‌ی گراف:

حال که با نحوه‌ی نگهداری گراف‌ها آشنا شده‌ایم، می‌توانیم الگوریتم‌های اولیه‌ی گراف را یاد بگیریم. از مهمترین نیازهای ما در گراف می‌توان به:

۱- تشخیص همبندی

۲- تشخیص وجود مسیر بین دو راس

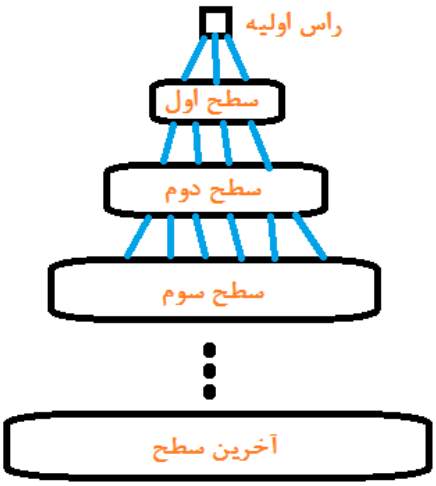
۳- یافتن کوتاه‌ترین فاصله بین دو راس

۴- پیمایش گراف‌ها به منظورهای مختلف

اشاره کرد. در زیر دو الگوریتم اولیه‌ی پیمایش گراف را به اختصار معرفی می‌کنیم:

جستجوی اول سطح (BFS)

در این روشِ پیمایش گراف، یک راس گراف را در نظر می‌گیریم و این مجموعه‌ی تک عضوی را A_0 می‌نامیم. سپس تمام همسایه‌های آن را در نظر گرفته و مجموعه‌ی آنها را A_1 می‌نامیم، سپس تمام همسایه‌های رئوسی که در A_1 آمده است و تا به حال در هیچ مجموعه‌ای نیامده است را درنظر می‌گیریم و آنها را A_2 می‌نامیم، آنقدر ادامه می‌دهیم تا به مجموعه‌ی A_iای برسیم که دیگر تمام همسایه‌های رئوس آن قبلا دیده شده باشد. حال اگر راسی در مجموعه‌ی A_j آمده باشد فاصله‌اش تا راس اولیه j است.



شکل ۶. الگوریتمِ BFS

از جمله اطلاعاتی که از این روش پیمایش بدست می‌آید می‌توان به فاصله‌ی رئوس از یک راس خاص، تشخیص همبندی گراف و تشخیص وجود مسیر بین دو راس مشخص اشاره کرد. هرچند که کاربردهای دیگر نیز برای این الگوریتم وجود دارد که از حوصله‌ی این درس خارج است.

برای پیاده سازی BFS باید یک آرایه‌ی mark بگیریم که در آن مشخص می‌کنیم هر راسی دیده شده است یا خیر، همچنین یک لیست می‌گیریم و در ابتدا راس اولیه را در آن قرار می‌دهیم و هر راس جدیدی که می‌بینیم به انتهای لیست برای پردازش اضافه می‌کنیم و از ابتدای رئوس روی لیست حرکت می‌کنیم تا هنگامی که به انتهای لیست برسیم.

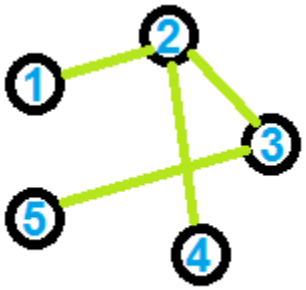
برای درک بهترِ پیاده‌سازی به مثال زیر توجه کنید:

مثال ۶. برنامه‌ای بنویسید که n و m را بگیرد و سپس m یال بگیرد و در خروجی فاصله‌ی دو راسِ 1 و n را چاپ کند.

```cpp
#include <iostream>
#include <vector>

using namespace std;

const int N = 100;

vector <int>adj[N];
bool mark[N];
int depth[N];

int main(){
    int n, m;
    cin >> n >> m;
    for(int i = 0; i < m; i++){
       int a, b;
       cin >> a >> b;
       adj[a].push_back(b);
       adj[b].push_back(a);
    }
    int que[N], h = 0, t = 1;
    que[0] = 1;//gharar dadane rase aval dar list
    mark[1] = true;
    for(int h = 0;h < t; h++){
        int v = que[h];
        for(int i = 0; i < adj[v].size(); i++){
            int u = adj[v][i];
            if(!mark[u]){
              depth[u] = depth[v] + 1;
              mark[u] = true;
              que[t++] = u;
            }
        }
    }
    cout << depth[n] << endl;
    system("pause");
}
```

جستجوی اول عمق (DFS)

در این روش از یک راس شروع به پیمایش گراف می‌کنیم و در هر مرحله در هر راسی که هستیم به یکی از همسایه‌های آن که تا بحال ندیده‌ایم می‌رویم واگر تمام همسایه‌های آن را قبلا دیده باشیم به راسی باز می‌گردیم که به وسیله‌ی آن به این راس آمده بودیم. برای مثال گراف زیر را در نظر بگیرید:



شکل ۷. گراف مثال

فرض کنید DFS خود را از راس ۳ شروع کرده باشیم، ابتدا به یکی از همسایه‌های آن که تا بحال دیده نشده می‌رویم، مثلا راس ۲. سپس به راس ۱ می‌رویم، حال تنها همسایه‌ی ۱ راسِ ۲ است که قبلا دیده شده. پس کارِ DFS در راس ۱ تمام شده و به همان راسی که آن را صدا زده یعنی راس دو بازمی‌گردیم. حال از همسایه‌های ۲ رئوس ۱ و ۳ دیده شده‌اند، پس به راس ۴ می‌رویم و سپس به همان راس ۲ بازمی‌گردیم و کار راس ۲ نیز تمام شده است و به راس ۳ بازمی‌گردیم. سپس از همسایه‌های آن راس ۵ مانده که به آن می‌رویم و سپس به ۳ بازگشته و DFS کلی تمام شده است، از جمله ویژگی‌های DFS می‌توان به پیاده‌سازی ساده‌ی آن اشاره کرد. همچنین اطلاعات اولیه‌ای که به ما می‌دهد همبندی گراف و یا وجود مسیر بین دو راس است. البته الگوریتم‌های بسیاری بر اساس DFS نوشته می‌شوند که در اینجا بحث نمی‌شوند.

برای پیاده‌سازیِ DFS از توابع بازگشتی استفاده می‌کنیم و مانند BFS یک آرایه‌ی مارک برای اینکه وارد راس تکراری نشویم نگه‌داری می‌کنیم.

مثال ۷. برنامه‌ای بنویسید که ابتدا یک گراف را از ورودی بگیرد و سپس مشخص کند آیا بین راس ۱ و n مسیر وجود دارد یا خیر.

```cpp
#include <iostream>
using namespace std;
const int N = 100;
int n, m;
bool mark[N];
bool g[N][N];
void dfs(int v){
     mark[v] = true;
     for(int i = 1; i <= n; i++)
             if(mark[i] == 0 && g[v][i] == 1)
                           dfs(i);
}
int main(){
    cin >> n >> m;
    for(int i = 0; i < m; i++){
            int a, b;
            cin >> a >> b;
            g[a][b] = g[b][a] = true;
    }
    dfs(1);
    if(mark[n] == 1)
                  cout << "Yes" << endl;
    else
                  cout << "No" << endl;
    system("pause");
}
```