برنامه ریزی پویا

(Dynamic Programming)

(برای دانشجویان علوم کامپیوتر)

محمّد فرشی

دانشکدهٔ ریاضی

دانشگاه یزد

تاریخ آخرین ویرایش: ۴ بهمن ۱۳۹۱

فهرست مطالب

فصل ۱

مقدمه

۱.۱ برنامه‌ریزی پویا چیست؟

در علوم کامپیوتر، برنامه‌ریزیِ پویا [۱] یک روش بهینه سازی است که برای دسته‌ای از الگوریتم‌های پس گرد[۲] زمانی که زیرمسأله‌ها بطور مکرر فراخوانی می‌شوند استفاده می‌شود. این روش در سال ۱۹۵۳ توسط ریاضی‌دانی به نام ریچارد بلمن[۳] معرفی شد. برنامه‌ریزیِ پویا در ریاضی و علوم کامپیوتر روشی شناخته شده است که از آن در نوشتن الگوریتم‌های بهینه با استفاده از حذف اجرای چند بارهٔ یک زیر مسأله یکسان استفاده می‌شود. تعریف برنامه‌ریزیِ پویا در ریاضی و علوم کامپیوتر متفاوت است. نشان داده شده است که روش علوم کامپیوتری برای برنامه‌ریزیِ پویا کارآیی بالاتری دارد زیرا محاسبات تکراری را حذف می‌کند در حالی که در روش ریاضیِ برنامه‌ریزیِ پویا امکان کاهش فضای حافظه بیشتر است.

دقت کنید که کلمه پویا[۴] را نباید با زبانهای برنامه‌نویسی پویا نظیر LISP و Scheme اشتباه گرفت. همچنین کلمه برنامه‌ریزی[۵] ربطی به مبحث برنامه‌نویسی کامپیوتری ندارد. در مبحث الگوریتم، برنامه‌ریزیِ پویا به تکنیکی اشاره می‌کند که یک جدول[۶] را با استفاده از مقادیر موجود در سایر جداول تکمیل می‌کند. دلیل استفاده از کلمه پویا آن است که مقادیر یک جدول با استفاده از مقادیر موجود در سایر جداول محاسبه می‌شوند. همچنین به دلیل استفاده از جداول به آن برنامه‌ریزی می‌گویند؛ مشابه جدول برنامه‌ها که توسط شبکه‌های تلویزیونی ارائه می‌شود.

برنامه‌ریزیِ پویا شبیه روش تقسیم و حل [۷] مسائل را با استفاده از ترکیب کردن جواب زیرمسأله‌ها حل می‌کند. الگوریتم‌های تقسیم و حل ، مسأله را به زیر مسأله‌های مستقل تقسیم می‌کند و پس از حل زیر مسأله‌ها به صورت بازگشتی، نتایج را با هم ترکیب کرده و جواب مسأله اصلی را بدست می‌آورد. به عبارت دقیق‌تر، برنامه‌ریزیِ پویا در مواردی قابل استفاده است که زیرمسأله‌ها مستقل نیستند؛ یعنی زمانی که زیرمسأله‌ها دارای زیر-زیر مسأله‌های یکسان هستند. در این حالت روش تقسیم و حل با اجرای مکرر زیرمسأله‌های یکسان، بیشتر از میزان لازم محاسبات انجام می‌دهد. یک الگوریتم برنامه‌ریزیِ پویا زیرمسأله‌ها را یک بار حل و جواب آنها را در یک جدول ذخیره می‌کند و با این کار از تکرار اجرای زیرمسأله‌ها در زمانی که مجدداً به جواب آنها نیاز است جلوگیری می‌کند.

---

[۱] dynamic programming

[۲] backtraking algorithms

[۳] Richard Bellman

[۴] dynamic

[۵] programming

[۶] منظور از جدول یک آرایه یک یا چند بعدی است.

[۷] divide-and-conquer

۱.۱.۱ یک مثال: دنباله فیبوناچی

قبل از شروع بحث برنامه‌ریزیِ پویا به تکنیکی به نام تکنیک به خاطر سپاری[۸] می‌پردازیم. بنابر تعریف، nامین عدد فیبوناچی به صورت زیر تعریف می‌شود:

$$F(n) = \begin{cases} n & \text{اگر } n = ۰, ۱ \\ F(n-۱) + F(n-۲) & \text{در غیر اینصورت} \end{cases}.$$

الگوریتم ۱.۱.۱ nامین عدد فیبوناچی را محاسبه می‌کند.

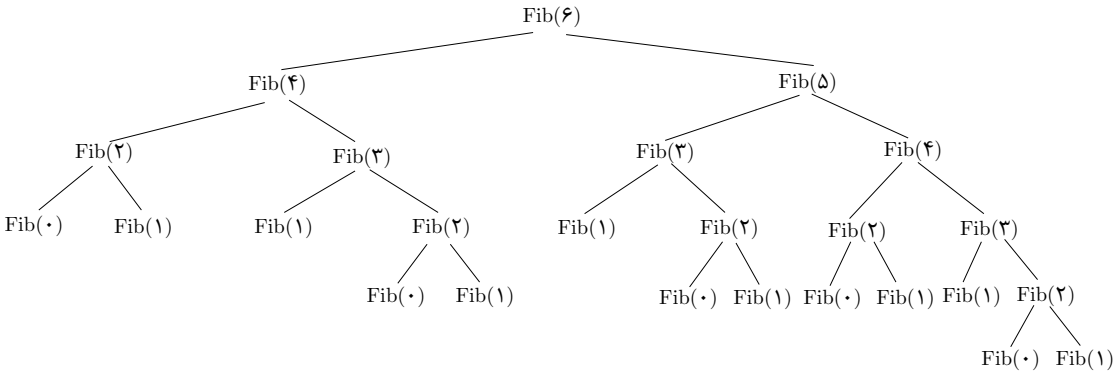
Algorithm 1.1.1: Fib(n)

Input: n.

Output: $F(n)$.

```
1 if n == 0 یا 1 then
2 |   return n;
3 else
4 |   return Fib(n − 1)+Fib(n − 2);
5 end
```

برای بررسی زمان اجرای این الگوریتم، نگاهی به درخت اجرای آن برای محاسبه ششمین عدد فیبوناچی در شکل ۱.۱ می‌اندازیم: تعداد برگ‌های درخت اجرای الگوریتم و در نتیجه زمان اجرای آن نمایی است.



شکل ۱.۱: درخت اجرای الگوریتم ۱.۱.۱ به ازای $n = ۶$.

همانطور که در درخت اجرای الگوریتم می‌توان دید، تابع Fib به طور مکرر به ازای $n = ۰$ و $n = ۱$ و همچنین به تعداد کمتری به ازای $n = ۳$ و $n = ۴$ فراخوانی شده است. یک راه برای بهبود زمان اجرای الگوریتم، ذخیره کردن مقادیر محاسبه شده در یک آرایه در اولین برخورد و استفاده از این مقادیر در زمان نیازهای بعدی است. این روش را به خاطر سپاری می‌نامند. به طور خلاصه این روش به صورت زیر است:

۱. با یک الگوریتم پس گرد شروع کن.

۲. برای حل زیر مساله ابتدا جدول را برای وجود جواب برای زیر مساله بررسی کن و در صورت موجود بودن جواب آن را برگردان.

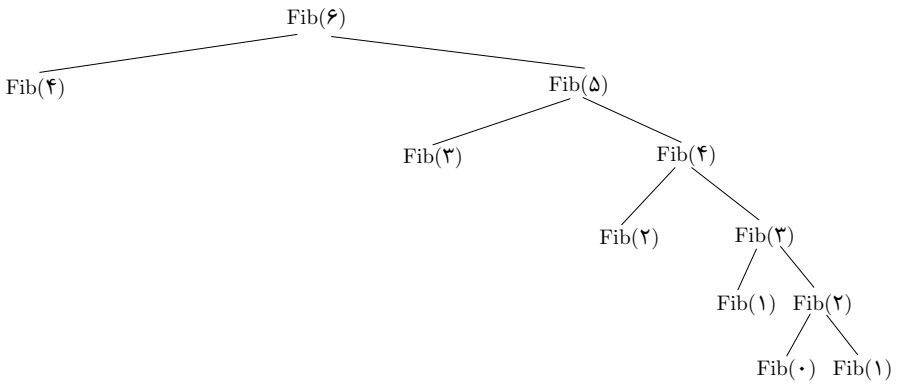
[۸] memoization

۳. در غیر اینصورت، زیر مساله را حل کن و جواب را در جدول ذخیره کرده و سپس مقدار را برگردان.

با استفاده از تکنیک به خاطر سپاری، الگوریتم فیبوناچی به صورت الگوریتم ۱.۱.۲ در می‌آید. در این الگوریتم آرایه f مقادیر اعداد فیبوناچی را نگهداری می‌کند و تمام درایه‌های آن در ابتدا ی اجرای الگوریتم با مقدار -۱ پر شده است. روش کار این الگوریتم تقریبا واضح است. اگر مقدار $\text{Fib}(n)$ قبلاً محاسبه شده باشد، به جای فراخوانی مجدد تابع، مقدار آن که در آرایه موجود است بازگردانده می‌شود. در غیراینصورت، مقدار آن محاسبه و برای استفاده‌های بعدی در آرایه ذخیره می‌شود. درخت اجرای این الگوریتم در شکل ۲.۱ آمده است.

Algorithm 1.1.2: Fib(n)

```
Input: n.
Output: F(n).
if n == 0 یا 1 then
    return n;
else
    if f[n] ≠ −1 then
        return f[n]
    else
        f[n] =Fib(n − 1)+Fib(n − 2);
        return f[n];
    end
end
```



شکل ۲.۱: درخت اجرای الگوریتم ۱.۱.۲ به ازای $n = ۶$.

بوضوح می‌توان دید که زمان لازم برای اجرای الگوریتم ۱.۱.۲ کمتر از الگوریتم ۱.۱.۱ است زیرا برای مقادیری که قبلاً محاسبه شده‌اند تابع فراخوانی نمی‌شود. چون به ازای هر مقدار کوچکتر از n، مقدار $F(n)$ فقط و فقط یک بار محاسبه می‌شود، زمان اجرای الگوریتم ۱.۱.۲ متناسب با $O(n)$ است. با توجه به اینکه برای محاسبه $F(n)$، تمام مقادیر $F(۰)$ تا $F(n)$ در حافظه اصلی ذخیره می‌شوند، حافظه زیادی برای الگوریتم نیاز است. با توجه به اینکه مقدار $F(n)$ فقط به $F(n-۱)$ و $F(n-۲)$ وابسته است، اگر مقدار $F(n)$ از پایین به بالا محاسبه شود، نیازی به نگهداری سایر مقادیر در حافظه نیست. در این روش، برای محاسبه $F(n)$، ابتدا $F(۲)$ را محاسبه می‌کنیم. سپس با جمع کردن $F(۲)$ و $F(۱)$ مقدار $F(۳)$ را محاسبه می‌کنیم. از این مرحله، نیازی به $F(۰)$ و $F(۱)$ نیست و می‌توان آنها را از حافظه حذف کرد. با استفاده از $F(۲)$ و $F(۳)$ می‌توان $F(۴)$ را محاسبه کرد و مقدار $F(۲)$ را از حافظه حذف کرد. با ادامه این روند

می‌توان $F(n)$ را محاسبه کرد بدون آنکه به ذخیره کردن تمام مقادیر در حافظه نیاز باشد (ر. ک. الگوریتم ۱.۱.۳). توجه کنید که در الگوریتم قبلی، پس از محاسبه $F(n)$ مقدار عدد فیبوناچی برای تمام مقادیر کوچکتر از n نیز در دسترس است اما این خاصیت در الگوریتم اخیر به دلیل ذخیره نکردن تمام مقادیر میانی وجود ندارد.

Algorithm 1.1.3: Fib(n)

Input: n.
Output: $F(n)$.

```
if n == 0 یا 1 then
    return n؛
end
u = 0؛
v = 1؛
for i = 2 to n do
    t = u + v؛
    u = v؛
    v = t؛
end
return v؛
```

همانطور که در بالا مشاهده شد، با نگاه به یک الگوریتم به خاطر سپاری بازگشتی به صورت پایین به بالا، می‌توان آن را به یک الگوریتم تکراری[۹] که جدول جواب‌ها را پر می‌کند تبدیل کرد. در این حالت، برخی از زیر مساله‌های حل شده ممکن است در نهایت مورد استفاده قرار نگیرند. در حقیقت این تفاوت تکنیک به خاطر سپاری و برنامه‌ریزیِ پویا است. از لحاظ تئوری پیچیدگی، روش به خاطر سپاری با برنامه‌ریزیِ پویا یکسان است. در برنامه‌ریزیِ پویا ابتدا زیر مسأله‌هایی که برای حل مسأله نیاز است را مشخص می‌کنیم و سپس آنها را از پایین به بالا بوسیله یک روش تکراری حل می‌کنیم.

نکته: مقدار عدد nام فیبوناچی دارای فرمول ریاضی زیر است:

$$F(n) = \frac{\phi^n - (۱ - \phi)^n}{\sqrt{۵}}$$

که در آن $\phi = \frac{۱+\sqrt{۵}}{۲}$ که به نسبت طلایی[۱۰] مشهور است. در نتیجه براحتی می‌توان $F(n)$ را به ازای مقادیر بزرگ n نیز در زمان کوتاهی محاسبه کرد و استفاده از الگوریتم‌های فوق لازم نیست. در اینجا فقط برای توضیح برنامه‌ریزیِ پویا از این مثال استفاده شده است.

بازگشت به بحث برنامه‌ریزیِ پویا ، برنامه‌ریزیِ پویا معمولاً برای حل مسائل بهینه‌سازی استفاده می‌شود. در این دسته از مسائل، جواب‌های متعددی برای مسأله موجود است و هدف یافتن جوابی است که بهینه (ماکزیمم یا مینیمم) باشد. لازم به ذکر است که جواب بهینه لزوماً منحصربفرد نیست و روش‌های مختلف ممکن است به یک مقدار بهینه منجر شود. روند طراحی یک الگوریتم برنامه‌ریزیِ پویا را می‌توان به مراحل زیر تقسیم کرد:

۱. تشخیص ساختارِ یک جواب بهینه،

۲. تعریف کردن یک مقدار بهینه به طور بازگشتی،

۳. محاسبه یک مقدار بهینه به روش پایین به بالا،

[۹] iterative

[۱۰] golden ratio

۴. ساختن یک جواب بهینه با استفاده از اطلاعات محاسبه شده.

مراحل ۱ تا ۳، مقدار بهینه را برای مسأله ارائه می‌کند. در صورتی که فقط مقدار بهینه مورد نیاز باشد، مرحله ۴ را می‌توان حذف کرد. در مرحله ۴ از اطلاعات اضافیِ ذخیره شده در مراحل قبلی برای ساختن یک جواب بهینه استفاده می‌کند. به عنوان مثال، مسأله یافتن کوتاهترین مسیر بین دو رأس یک گراف را در نظر بگیرید. اگر فقط طول کوتاهترین مسیر بین دو رأس مورد نیاز باشد، اجرای مراحل ۱ تا ۳ کافی است. اما برای یافتن مسیری با کوتاهترین طول، اجرای مرحله ۴ نیز مورد نیاز است. همانطور که قبلاً اشاره شد، طول کوتاهترین مسیر منحصر بفرد است اما ممکن است مسیرهای متعددی در گراف با این طول بین دو رأس مورد نظر موجود باشد.

در فصل‌های آتی روش برنامه‌ریزیِ پویا برای حل برخی مسائل بهینه‌سازی ارائه می‌شود.

فصل ۲

برنامه‌ریزی خط تولید

۱.۲ مقدمه

یک کارخانه خودروسازی دارای دو خط تولید خودرو است (ر. ک. شکل ؟؟). یک شاسی خودرو[۱] وارد هر کدام از خطوط تولید می‌شود و قطعات آن در ایستگاه‌های مختلف به آن وصل می‌شود و در پایان خط، خودروِ تکمیل شده خارج می‌شود. هر خط تولید دارای n ایستگاه است

[۱] chassis