دانشکده مهندسی کامپیوتر

جزوه درس
ساختمان‌های داده

استاد درس: سید صالح اعتمادی

پاییز ۱۳۹۸

جلسه ۸

Tail Recursion، برنامه نویسی پویا

سهراب نمازی نیا - ۱۳۹۸/۷/۲۲

جزوه جلسه ۸ام مورخ ۱۳۹۸/۷/۲۲ درس ساختمان‌های داده تهیه شده توسط سهراب نمازی نیا. در جهت مستند کردن مطالب درس ساختمان های داده، بر آن شدیم که از دانشجویان جهت مکتوب کردن مطالب کمک بگیریم. هر دانشجو می‌تواند برای مکتوب کردن یک جلسه داوطلب شده و با توجه به کیفیت جزوه از لحاظ کامل بودن مطالب، کیفیت نوشتار و استفاده از اشکال و منابع کمک آموزشی، حداکثر یک نمره مثبت از بیست نمره دریافت کند. خواهش‌مند است نام و نام خانوادگی خود، عنوان درس، شماره و تاریخ جلسه در ابتدای این فایل را با دقت پر کنید.

۱.۸ Tail Recursion

طبق آنچه که جلسه گذشته در مورد الگوریتم Quick Sort گفته شد، در هر مرحله از این الگوریتم، با انتخاب Pivot مناسب، عناصر کوچکتر از Pivot را در سمت چپ آن و عناصر بزرگتر را نیز در سمت راست آن نگهداری میکردیم [۱]. اکنون میخواهیم مدل بهینه تری از Quick Sort را نسبت به جلسه گذشته بررسی کنیم. می خواهیم الگوریتم Quick Sort را به گونه ای پیاده سازی کنیم که تضمین شود در بدترین حالت ممکن Space Complexity آن برابر با log n است. در این روش ابتدا سمت چپ عنصر Pivot را به همان روش بازگشتی مرتب میکنیم. اما در سمت راست، به جای مجددا فراخوانی کردن تابع Quick Sort، اندیس شروع لیست را به اندیس میانی منتقل میکنیم و این عمل تا زمانی که اندیس اشاره گر به اول لیست از اندیس اشاره گر به آخر لیست کوچکتر است، ادامه می یابد. پس در واقع تا زمانی که شرط مورد نظر برقرار است، قسمت سمت چپ Pivot همانند حالت عادی الگوریتم Quick Sort به صورت بازگشتی انجام میشود. اما به جای فراخوانی مجدد تابع به صورت بازگشتی برای سمت راست، اندیس اشاره گر به ابتدای لیست را تغییر میدهیم. پس در واقع تا اینجا ما توانستیم یکی از روابط بازگشتی را از الگوریتم خود حذف کنیم. در زیر شبه کد مربوط به این الگوریتم را مشاهده میکنید:

```
long[] A;
while l < r do
    m ← partition(A, l, r);
    QuickSort(A, l, m - 1);
    l ← m + 1;
end
```

Algorithm 1: Tail Recursion

حال میخواهیم عمل بازگشتی را آگاهانه تر انجام دهیم. به این معنی که لزوما سمت چپ Pivot برای انجام رابطه بازگشتی انتخاب نشود. این بار باید از بین سمت راست و سمت چپ، بازه ای را که طول کوتاه تری دارد برای این کار انتخاب کنیم. برای درک بهتر به شبه کد زیر توجه فرمایید :

```
long[] A;
while l < r do
    m = partition(A, l, r);
    if (m - l) < (r - m) then
        QuickSort(A, l, m - 1);
        l ← m + 1;
    else
        QuickSort(A, m + 1, r);
        l ← m - 1;
    end
end
```

Algorithm 2: Tail Recursion

۱.۸

۲.۸ Intro Sort

در الگوریتم Quick Sort انتخاب Pivot مناسب نیز مهم است. بهترین Pivot عنصری است که بعد از Partition بندی، آرایه را به دو زیر مجموعه متعادل تبدیل کند. متعادل به این معنا است که هر دو زیرمجموعه تعداد اعضای نزدیک به هم داشته باشند. به همین دلیل برای اینکه Pivot از کوچکترین عنصر بودن و یا بزرگترین عنصر بودن فاصله بگیرد، آن را به این روش انتخاب میکنیم: سه عضو دلخواه آرایه را انتخاب میکنیم. عموما این سه عضو را عضو ابتدایی، میانی و انتهایی آرایه در نظر میگیرند. سپس عنصر میانی از لحاظ مقدار را از بین این سه عنصر به عنوان Pivot انتخاب میکنیم.

این الگوریتم که نمونه بهینه سازی شده از الگوریتم Quick Sort است را الگوریتم Intro Sort میگویند.
یک بهینه سازی ممکن دیگر در این الگوریتم این است که اگر عمق درخت بازگشتی از مقدار خاصی بزرگتر شد، الگوریتم را تغییر داده و برای مرتب کردن داده ها از الگوریتم Heap Sort استفاده شود. [۲]

۳.۸ برنامه نویسی پویا

در روش تقسیم و حل دیدیم که میتوان یک مساله را به دو یا چند مسئله کوچکتر تقسیم کرد و با حل زیر مسئله ها، جواب مسئله نهایی را بدست آورد. اما مشکلی که وجود داشت این بود که گاهی مجبور بودیم که یک زیر

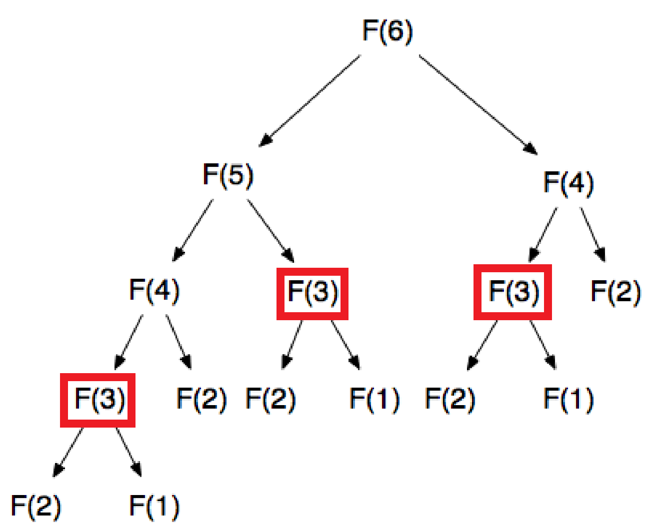
مسئله را که ممکن بود زیر مسئله تعداد زیادی از زیر مسئله های دیگر نیز باشد، چندین بار محاسبه کنیم که این موضوع از لحاظ Time Complexity به عنوان یک مشکل اساسی محسوب میشود. اما اگر ما مسئله ها را به ترتیبی حل کنیم که در هر مسئله تمام زیر مسئله های لازم برای آن از قبل حل و ذخیره شده باشند، دیگر به مشکل اشاره شده برنخواهیم خورد [۳]. این روش حل مسائل را برنامه نویسی پویا میگویند. به دو مثال زیر در مورد برنامه نویسی پویا توجه فرمایید :

۱.۳.۸ دنباله فیبوناچی

دنباله فیبوناچی را در نظر بگیرید:

```
Fib[n] = Fib[n - 1] + Fib[n - 2]    Fib(0) = 0, Fib(1) = 1
```

اگر بخواهیم برای محاسبه جمله ای دلخواه از دنباله فیبوناچی به روش بازگشتی عمل کنیم، برای مقادیر بزرگ زمان بسیار زیادی برای محاسبه حاصل طول خواهد کشید. دلیل این موضوع این است که در درخت بازگشتی مربوطه، خیلی از مقادیر فیبوناچی بار ها محاسبه میشوند. به شکل زیر دقت کنید:
۱.۸



شکل ۱.۸: درخت بازگشتی فیبوناچی عدد ۶

برای رفع این مشکل باید جملات این دنباله را به ترتیبی بدست بیاوریم که در هر مرحله، جملاتی که برای بدست آوردن عدد فیبوناچی در آن مرحله نیاز داریم از قبل محاسبه و ذخیره شده باشند. بدست آوردن این ترتیب در برخی مسائل کمی دشوار می باشد. اما در مورد دنباله فیبوناچی به راحتی میتوان به کمک برنامه نویسی پویا این مسئله را حل کرد. یک آرایه را در نظر بگیرید. اگر جملات دنباله فیبوناچی را به ترتیب در این آرایه ذخیره کنیم، همواره مسئله های پیش نیاز برای ما از پیش حل شده خواهند بود. به شبه کد زیر دقت کنید:

```
long[] Fibs;
Fibs[0] = 0;
Fibs[1] = 1;
long i = 2;
while i <= n do
  | Fibs[i] = Fibs[i - 1] + Fibs[i - 2];
  | i++;
end
```

Algorithm 3: Fibonacci, dynamic programming

۲.۳.۸ مسئله خرد کردن پول

حال میخواهیم به بررسی مثالی دیگر از کاربرد برنامه نویسی پویا بپردازیم. فرض کنید انواع معینی سکه داریم و قرار است پول دلخواهی را با کمترین تعداد سکه های لازم خرد کنیم. برای حل این مسئله الگوریتم حریصانه لزوما به جواب درست منتهی نمیشود. به عنوان مثال فرض کنید سکه های ۵، ۱۰، ۲۰ و ۲۵ تومانی موجود است و قرار است که اسکناسی چهل تومانی را به کمک کمترین تعداد آن ها خرد کنیم. به شبه کد الگوریتم حریصانه برای حل این مسئله دقت کنید:

```
Change ← empty collection of coins
while money > 0 do
  | coin ← largest denomination that does not exceed money
  |  add coin to Change
  |  money ← money − coin
end
return change;
```

Algorithm 4: greedy way to solve "Change Problem"

اگر بخواهیم به روش حریصانه این مسئله را حل کنیم، نهایتا به این جواب خواهیم رسید که به یک سکه ۲۵ تومنی، یک سکه ۱۰ تومانی و یک سکه ۵ تومانی برای خرد کردن این اسکناس لازم است. در حالی که همین مسئله را فقط با ۲ سکه ۲۰ تومانی میتوان حل کرد. پس الگوریتم حریصانه نمیتواند پاسخگوی این مسئله باشد.

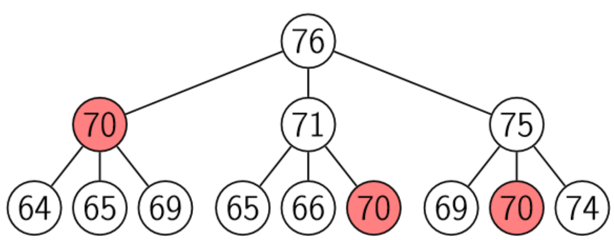
حال میخواهیم این مسئله را به روش تقسیم و حل بررسی کنیم. به کمک این روش میتوانیم بگوییم که در هر مرحله همه حالت های ممکن را محاسبه کرده و آن که به کمترین تعداد سکه لازم منجر میشود، همان جواب مسئله باشد. به عنوان مثال برای همان مسئله مطرح شده، به جای بدست آوردن جواب برای اسکناس ۴۰ تومانی، ابتدا فرض میکنیم یک بار از اسکناس ۲۵ تومانی استفاده کرده ایم و مسئله را برای پول باقیمانده حل میکنیم. این به این معنی است که جواب برای اسکناس چهل تومانی یکی بیشتر از جواب برای آن پول باقی مانده خواهد شد. اما این تنها یک حالت ممکن است. باید همین کار را برای سکه های دیگر هم به جای سکه ۲۵ تومانی انجام دهیم و نهایتا کمترین حاصل از میان جواب های بازگشتی بدست آمده، جواب نهایی مسئله برای خرد کردن اسکناس ۴۰ تومانی خواهد بود.

به شبه کد این مسئله به روش تقسیم و حل دقت فرمایید:

```
MinNumCoins ← ∞;
if money = 0 then
  | return 0
end
while i <= number of Coins do
  | if money >= Coins[i] then
  |   | NumCoins ← RecursiveChange(money − Coins[i], coins)
  |   | if NumCoins +1 < MinNumCoins then
  |   |   | MinNumCoins ← NumCoins +1;
  |   | end
  | end
  | i ← i + 1;
end
return MinNumCoins;
```

Algorithm 5: Change problem, divide and conquer

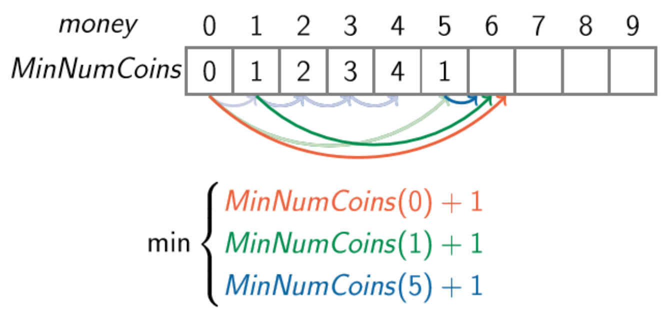
اگر اسکناس ورودی مسئله مقدار خیلی بزرگی داشته باشد، الگوریتم تقسیم و حل مشابه آنچه در مسئله فیبوناچی اتفاق افتاد به دلیل حل کردن مجدد مسئله های تکراری قادر نخواهد بود تا در زمان کوتاه به جواب نهایی برسد. به عنوان مثال فرض کنید سکه های ۱، ۵ و ۶ تومانی در اختیار داریم و میخواهیم یک اسکناس ۷۶ تومانی را با کمترین تعداد سکه ممکن به روش تقسیم و حل خرد کنیم. همانطور که در شکل میبینید، ما چندین بار این مسئله را برای یک اسکناس ۷۰ تومانی حل میکنیم که این کار از لحاظ زمانی برای مقادیر بزرگ، الگوریتم ما را با مشکل مواجه خواهد کرد.
۲.۸



شکل ۲.۸: درخت بازگشتی برای مسئله خرد کردن پول (اسکناس ۷۶ تومانی)

به همین دلیل به سراغ روش برنامه نویسی پویا برای حل این مسئله خواهیم رفت. در این روش، ترتیب پر کردن خانه های آرایه همانند مسئله فیبوناچی است. مطابق شکل زیر، میخواهیم جواب مسئله را برای یک اسکناس ۷ تومانی بدست آوریم و سکه های موجود ۱، ۵ و ۶ تومانی میباشند. فرض کنید ما تمام زیر مسئله های لازم برای پاسخ دادن به این مسئله را حل نموده ایم و جدول تا خانه شماره ۶ کامل شده است. حال بسته به این که کدام یک از سکه ها به عنوان اولین سکه انتخاب شود، مسئله به سه زیر مسئله ی از قبل حل شده تقسیم میشود. کمترین جواب از میان این سه مسئله را انتخاب کرده و با یک جمع میکنیم. حاصل جواب خانه شماره ۷ خواهد بود. این عمل را تا آخرین خانه ادامه میدهیم تا جواب نهایی مسئله بدست آید.

۳.۸



شکل ۳.۸: حل مسئله خرد کردن پول به روش برنامه نویسی پویا

به شبه کد مربوط به حل پویای این مسئه دقت فرمایید:

```
MinNumCoins(0) ← 0;
for m from 1 to money do
    MinNumCoins(m) ← ∞;
    for i from 1 to NumberOfCoins do
        if m > Coins[i] then
            NumCoins ← MinNumCoins(m − Coins[i]) + 1;
            if NumCoins < MinNumCoins[m] then
                MinNumCoins[m] ← NumCoins
            end
        end
    end
end
return MinNumCoins[money];
```

Algorithm 6: Change problem, dynamic programming

۴.۸ خلاصه

به طور خلاصه برای جمع بندی مطالب جلسه هشتم میتوان به موارد زیر اشاره کرد:

- یکی از روش های بهینه سازی الگوریتم Quick Sort، Tail Recursion نام دارد که در آن با کاهش یکی از دو رابطه بازگشتی، تضمین میشود که عمق درخت بازگشتی مربوطه از log n بیشتر نخواهد شد. در این روش به جای یکی از روابط بازگشتی، اندیس اشاره گر به ابتدای آرایه را در حلقه برنامه آپدیت میکنیم.
- یکی دیگر از روش های بهینه سازی الگوریتم Quick Sort، Intro Sort نام دارد که در آن با انتخاب تقریبی میانه به عنوان Pivot از غیر متعادل شدن تفاوت تعداد اعضای سمت چپ و راست آن جلوگیری میکند.

- برنامه نویسی پویا در واقع به روش تقسیم و حل کمک میکند که یک زیر مسئله را برای بار های متوالی حل نکند. در این روش باید به ترتیبی مسائل را حل کنیم که زیر مسئله های ما همیشه از پیش حل شده باشند. مسئله فیبوناچی و خرد کردن اسکناس دو نمونه از کاربرد های برنامه نویسی پویا محسوب میشوند.

در جلسه آینده مسائل بیشتر و پیچیده تری از برنامه نویسی پویا بررسی خواهند شد.

Bibliography

[1] https://www.toptal.com/developers/sorting-algorithms.

[2] https://www.geeksforgeeks.org/heap-sort/.

[3] https://www.javatpoint.com/divide-and-conquer-method-vs-dynamic-programming.