به نام خدا

داک آموزشی رویداد گُلابی

مرحله اول

سوال اول

دانشکده مهندسی کامپیوتر

دانشگاه صنعتی شریف

نیم سال اول ۰۱ـ۰۰

دبیر رویداد:

محمدطه جهانی نژاد

مسئول مرحله اول:

ایمان محمدی

طراحان داک آموزشی سوال اول:

ایمان محمدی

نگار باباشاه

بهار دیبایی‌نیا

محمد صادقی

ویراستاران داک آموزشی سوال اول:

ایمان محمدی

نگار باباشاه

شایان صالحی

مسئول لتک داک:

حسین علی‌حسینی

فهرست

مطالب

بخش ۱. الگوریتم‌های حریصانه

الگوریتم‌های مسائل بهینه سازی معمولاً مراحل مختلفی را طی می‌کنند، و در طی این مراحل انتخاب‌هایی را انجام می‌دهند. یک رویکرد این است که در هر لحظه با انتخاب گزینه‌ای که به نظر بهترین می‌آید، می‌توان به جواب بهینه رسید. به الگوریتم هایی با این رویکرد **حریصانه** گفته می‌شود.
در روش حریصانه، رسیدن به هدف در هر گام مستقل از گام قبلی و بعدی است. یعنی در هر مرحله برای رسیدن به هدف نهایی، مستقل از این که در مراحل قبلی چه انتخاب‌هایی صورت گرفته است و انتخاب فعلی ممکن است چه انتخاب‌هایی در پی داشته باشد، انتخابی که در ظاهر بهترین انتخاب ممکن است صورت می‌پذیرد. به همین دلیل است که به این روش، روش حریصانه گفته می‌شود.
نمونه‌ای مشهور از به کار بردن روش حریصانه، مسئله‌ی «خرد کردن پول» است. فرض کنید می‌خواهیم مبلغ amount واحد پول را با n سکه $a_1 < a_2 < \ldots < a_n$ خرد کنیم و از هر سکه به هر تعدادی در اختیار داریم. حال سوال این است که چگونه می‌توانیم با استفاده از تعدادی دلخواه از سکه‌های $a_1 \ldots, a_n$ مبلغ amount را تشکیل دهیم، به صورتی که تعداد سکه‌های استفاده شده تا حد امکان کم باشد؟ (فرض کنید $a_1 = 1$)
روشی برای رسیدن به یک جواب منطقی، الگوریتم حریصانه‌ای است که در هر مرحله بزرگترین سکه‌ی موجود که از مقدار مبلغ باقی‌مانده کوچک‌تر است را انتخاب می‌کند و به مجموعه‌ی جواب اضافه کرده و سپس از مبلغ باقی مانده کم می‌کند (دقت کنید از آن جا که کوچک‌ترین سکه همواره برابر با ۱ است این الگوریتم پایان‌پذیر است). به دلیل بزرگ بودن سکه‌ها تا حد امکان تعداد سکه‌های به دست آمده در این روش، به مقدار بهینه نزدیک است. اما جواب این الگوریتم در حالت‌های زیادی از جواب بهینه‌ی مسئله بزرگ‌تر است.
برای مثال روش حریصانه برای مجموعه‌ی سکه‌های {۱،۵،۶} و ۱۰ واحد پول جواب غیر بهینه زیر را می‌دهد

- **10=6+1+1+1+1 (Greedy=5)**
- **10=5+5 (Optimal=2)**

یک پیاده‌سازی برای الگوریتم ذکر شده به این صورت است:

```
int ChangeCoin(int Coins[], int Amount, int Result[]){
    int i = 0, Count = 0, j = 0;
    while(Coins[i] != -1 && Amount > 0){
        if(Amount >= Coins[i]){
            Amount -= Coins[i];
            Result[j++] = Coins[i];
            Count++;
        }
        else
            i++;
    }
    if(Amount != 0)
        return -1;
    return Count;
}
```

در این تابع، Coins ارزش سکه‌ها و اسکناس‌های موجود و Amount مقدار مورد نیاز برای تولید را مشخص می‌کنند. پاسخ نهایی در آرایه Result قرار گرفته و تعداد آن‌ها با متغیر Count به محل فراخوانی بازگشت داده می‌شود. فرض شده است انتهای لیست سکه‌ها و اسکناس‌ها با عدد ۱ـ مشخص شده و به ترتیب نزولی مرتب هستند. یعنی عنصر اول بزرگترین اسکناس موجود است.

ساختار الگوریتم حریصانه

کلیت روش حریصانه در هر مرحله، انتخاب یک عنصر از عناصر موجود است. این عنصر قسمتی از جواب مسئله است که به مجموعه عناصر نهایی اضافه می‌شود. در طی این مسیر گام‌های زیر اتفاق می‌افتد:

۱. روال انتخاب حریصانه: در این گام یک عنصر برای اضافه شدن به مجموعه جواب انتخاب می‌شود. معیار یا روال انتخاب عنصر برای اضافه شدن، ارزش آن عنصر است. بستگی به نوع مسئله هر عنصر ارزشی دارد که با ارزش‌ترین آن‌ها انتخاب می‌شود.

۲. امکان‌سنجی و افزودن: پس از انتخاب یک عنصر به صورت حریصانه، باید بررسی شود که آیا امکان اضافه کردن آن به مجموعه جواب‌های قبلی وجود دارد یا نه. گاهی اضافه شدن عنصر یکی از شرایط اولیه مسئله را نقض می‌کند که باید به آن توجه نمود.

اگر اضافه کردن این عنصر هیچ شرطی را نقض نکند، عنصر اضافه خواهد شد؛ وگرنه کنار گذاشته شده و بر اساس گام اول عنصر دیگری برای اضافه شدن انتخاب می‌شود. اگر گزینه دیگری برای انتخاب وجود نداشته باشد، اجرای الگوریتم به اتمام می‌رسد.

۳. بررسی اتمام الگوریتم: در هر مرحله پس از اتمام گام ۲ و اضافه شدن یک عنصر جدید به مجموعه جواب، باید بررسی کنیم که آیا به یک جواب مطلوب رسیده‌ایم یا نه؟ اگر نرسیده باشیم به گام اول رفته و چرخه را در مراحل بعدی ادامه می‌دهیم.

به زبان ساده، در روش حریصانه طی هر مرحله یک عنصر به روش حریصانه به مجموعه جواب اضافه شده، شرط محدودیت‌ها بررسی شده و در صورت نبود مشکل، عنصر و عناصر بعدی به همین ترتیب به مجموعه جواب اضافه می‌شوند. در طی این گام‌ها اگر به یک شرط نهایی خاص برسیم، یا امکان انتخاب عنصر دیگری برای اضافه کردن به مجموعه جواب وجود نداشته باشد، الگوریتم پایان یافته و مجموعه جواب به دست آمده به عنوان جواب بهینه ارائه می‌شود. توجه داشته باشید که ممکن است بر اساس نوع مسئله، ترتیب اضافه شدن عناصر به مجموعه جواب اهمیت داشته باشد.
برای آشنایی بیشتر با الگوریتم‌های حریصانه، می‌توانید **این‌جا** را مطالعه کنید.

بخش ۲. جست‌وجوی دودویی

جست‌وجوی دودویی یا Binary Search الگوریتمی سریع برای پیدا کردن یک مقدار در یک آرایه مرتب است. اساس کار این الگوریتم، تقسیم و حل می‌باشد. در باینری سرچ اگر بخواهیم مقدار x را در آرایه‌ای مرتب پیدا کنیم، ابتدا x را با مقدار وسط آرایه مقایسه می‌کنیم. اگر برابر باشند که x پیدا شده است. اگر x بزرگ‌تر باشد، در نیمه سمت راست آرایه و در غیر این صورت در نیمه سمت چپ آرایه به دنبال x می‌گردیم.
فرض کنید می‌خواهیم ۲۳ را در آرایه زیر پیدا کنیم:

index	0	1	2	3	**4**	5	6	7	8	0
value	2	5	8	12	**16**	23	38	56	72	91

در مرحله اول ۲۳ با مقدار وسطی آرایه یعنی ۱۶ مقایسه می‌شود و چون ۲۳ بزرگ‌تر است، در نیمه سمت راست آرایه به دنبال ۲۳ می‌گردیم.

index	5	6	**7**	8	9
value	23	38	**56**	72	91

در مرحله دوم ۲۳ با ۵۶ مقایسه می‌شود و چون ۲۳ کوچک‌تر است، در نیمه سمت چپ به دنبال ۲۳ می‌گردیم.

index	**5**	6
value	**23**	38

در مرحله سوم ۲۳ با ۲۳ مقایسه می‌شود و چون برابر است مقدار ۲۳ که در ایندکس ۵ می‌باشد، پاسخ مسئله است. فرض کنید می‌خواهیم بسته به شرایط اولین یا آخرین محل حضور عدد x در آرایه‌ای مرتب را پیدا کنیم. کد تابعی که این کار را انجام می‌دهد در صفحه بعد آمده است:

```c
// Function to find the first or last occurrence of a given number in
// a sorted integer array. If `searchFirst` is true, return the
// first occurrence of the number; otherwise, return its last occurrence.
int binarySearch(int nums[], int n, int target, int searchFirst)
{
    // search space is nums[…lowhigh]
    int low = 0, high = n - 1;

    // initialize the result by -1
    int result = -1;

    // loop till the search space is exhausted
    while (low <= high)
    {
        // find the mid-value in the search space and compares it with the target
        int mid = (low + high)/2;

        // if the target is found, update the result
        if (target == nums[mid])
        {
            result = mid;

            // go on searching towards the left (lower indices)
            if (searchFirst) {
                high = mid - 1;
            }

            // go on searching towards the right (higher indices)
            else {
                low = mid + 1;
            }
        }

        // if the target is less than the middle element, discard the right half
        else if (target < nums[mid]) {
            high = mid - 1;
        }

        // if the target is more than the middle element, discard the left half
        else {
            low = mid + 1;
```

```
        }
    }

    // return the found index or -1 if the element is not found
    return result;
}
```