جزوه طراحی و تحلیل الگوریتم ها (Algorithm Design)

نویسندگان: عباس عزیزمحمدی

مهدی دادنجانیان

نام استاد: قاسم مهدور

سال ۱۴۰۲-۱۴۰۳

عزیزمحمدی - دادنجانیان

فهرست مطالب

مقدمه ____ ۱
شبه کد(Pseudo code) ____ ۲
تحلیل الگوریتم ها ____ ۳
فصل ۱ . پیچیدگی زمانی(Time complexity) ____ ۴
مرتبه(order) ____ ۵
آشنایی با ۳ نماد جانبی ____ ۶
قانون های ساده برای بدست آوردن مرتبه زمانی ____ ۷
فصل ۲ . تقسیم و غلبه(Divide & Conquer) ____ ۸
جست و جوی دودویی(Binary Search) ____ ۸
محاسبه مرتبه زمانی ____ ۹
۱) روش استقرا ____ ۹
۲) روش قضیه اصلی ____ ۱۰
مرتب سازی ادغامی(Merge Sort) ____ ۱۱
فصل ۳: برنامه نویسی پویا (Dynamic Programing) ____ ۱۶
الگوریتم هایی بر اساس رویکرد برنامه نویسی پویا ____ ۱۶
الگوریتم یافتن !n با رویکرد پویا: ____ ۱۶
الگوریتم فیبوناچی با هر دو رویکرد: ____ ۱۷
مرتبه چند جمله ای ____ ۱۷
محاسبه $\binom{n}{k}$ با هر ۲ رویکرد : ____ ۱۷
روش بدست آوردن $D[i,j]$ از روی درایه های مشخص شده ____ ۲۱
فصل ۴ . رویکرد حریصانه (Greedy Approach) ____ ۲۴
الگوریتم تپه نورد ____ ۲۴
مسئله کوله پشتی 0-1(Knapsack Problem) ____ ۲۶
الگوریتم حریصانه کوله پشتی 0-1 ____ ۲۸

کوله پشتی کسری ۲۹
الگوریتم های حریصانه Prime و Kruskal ۳۰
درخت پوشا(Spanning Tree) ۳۰
درخت پوشای کمینه(Minimum Spanning Tree – MST) ۳۲
رمزنگاری هافمن (Huffman Encoding) ۳۹
الگوریتم دایکسترا (Dijkstras Algorithm) ۴۳
بازگشت به عقب (Back Tracking) ۴۶
۱. جست و جوی اول عمق (Depth Frirst Search) ۴۷
۲. جست و جوی اول سطح(Breath Frirst Search) ۴۸
فصل ۵. پیچیدگی محاسباتی (Computational Complexity) ۵۱
اندازه ورودی (input size) ۵۱
مسئله توقف (Halt) ۵۳
رده P ۵۳
رده NP ۵۴

ع.ر.محمدی - دارابخانیان

مقدمه

هدف این درس: *روش بدست آوردن کارایی الگوریتم ها *چندین روش پایه ای ساخت الگوریتم ها * توان یا کران الگوریتم ها

گام های یک الگورتیم باید یکی پس ازدیگری برداشته شود تا به پاسخ برسیم.

تعریف الگوریتم: به دنباله ای از دستورات که برای انجام کاری یا رسیدن به پاسخ باید اجرا شوند.(الگوریتم درست کردن یک غذا یا...)

الگوریتم ارائه شده برای یک مسئله همواره باید پاسخ درست را باز گرداند . برای نمونه الگو ریتم زیر بعضی مواقع درست کار نمی کند.

```
float abs(float x)
{
    return -x;
}
```

این تابع، همواره قدر مطلقX را برنمی گرداند پس این الگوریتم برای یافتن قدر مطلق Xدرست نیست. چون باید به ازای هر عضو در دامنه درست کار کند.

الگوریتم جستجو خطی: ورودی : عدد مثبت n و آرایه A و مقدار K خروجی : اندیس درایه K اُم یا ۱- اگر K درون A نباشد.

مسئله : درایه K را بیابد.

```
int Find(int A[], int n, int k)
{
    int i;
    for (i = 0; i < n; i++)
        if (A[i] == k)
            return i;
    return -1;
}
```

الگوریتم بدست آوردن میانگین یک آرایه: ورودی: آرایه A وعدد n خروجی: میانگین درایه های A

مسئله: حاصل میانگین درایه های A را بیابید.

۱

```
float avg(float A[], int n)
{
    float S = 0;
    for (int i = 0; i < n; i++)
        S += A[i];
    return S / n;
}
```

۲

```
float avg(float A[], int n)
{
    float S = 0;
    for (int i = 0; i < n; i += 2)
        S += A[i];
    for (int j = 1; j < n; j += 2)
        S += A[j];
    return S / n;
}
```

اکنون برای یک مثله ۲ الگوریتم نوشتیم و باید یکی از این ۲ را بر اساس بهتر بودن زمانی یا حافظه ای انتخاب کنیم.

الگوریتم ۲ سخت تر خواهد بود زیرا دستورهای بیشتری دارد. الگوریتم ها را بر اساس چند معیار می توان با هم مقایسه کرد، اما در بیشتر مواقع الگوریتمی که زمان کمتری سپری می کند بهتر خواهد بود.

شبه کد(Pseudo code)

در این درس برای نمایش الگوریتم ها از زبان شبه کدها با این ویژگی ها کمک می گیریم:

* از تو رفتگی (indentation) برای نمایش دستورهای درون یک دستور استفاده می کنیم.

```
for i=1 to 20
    j = i*j
```

← این دستور درون حلقه *for* قرار دارد.

* از دستور های for، while ، if برای ساختار دادن به الگوریتم ها استفاده می کنیم.

برای مثال شبه کد زیر از ۱ تا ۱۰ را نمایش می دهد.

```
for i=1 to 10
    print(i);
```

* **برای دسترسی به یک درایه از آرایه از [ ] استفاده می کنیم.**

* **اولین اندیس در آرایه ها از اندیس ۱ شروع می شود.**

* **برای دسترسی به بخش ها یا ویژگی های یک ساختار از . استفاده می کنیم.**

`N.name`

* **توابع با مقدار فراخوانی می شوند. و برای دسترسی به بخشی از یک آرایه می توان از [ ] براکت استفاده کرد. برای مثال `A[2:5]` درایه های آرایه A را از ۲ تا ۵ نشان می دهد.**

*** دستور return بی درنگ تابع را به پایان می رساند و دستور error الگوریتم را به پایان می رساند و یک خطا را نشان می دهد.**

الگوریتم مرتب سازی درجی

```
Sort (A, n)
    for i=2 to n
        k=A[i]
        j=i-1
        While(j>0 and A[j]>k)
            A[j+1]= A[j]
            j= j-1
        A[j+1]=k
```

نکته اگر در عبارت منطقی A or B ، درست باشد به طور کلی وضعیت B بررسی نمی شود و همچنین در عبارت A and B، اگر A نادرست باشد به طور کلی وضعیت B بررسی نخواهد شد.

تحلیل الگوریتم ها

هدف از تحلیل (analysis) الگورتیم ها ، یافتن زمان یا حافظه مورد نیاز برای اجرای آنها می باشد. الگوریتم ها را می توان بر اساس تعداد دستورها ، سختی پیاده سازی، داشتن پیش نیاز و ... بررسی کرد ، اما تحلیل زمانی ارزش بیشتری دارد. زیرا پس از پیاده سازی یک الگوریتم با کمک یک زبان سطح بالا، تنها زمان اجرای آن اهمیت خواهد داشت. از تحلیل فضایی هم چشم پوشی می کنیم. زیرا امروزه اثر دستگاه ها حافظه نسبتا زیادی دارند (هرچند الگوریتمی که حافظه بسیار زیادی بگیرد، اصلا مناسب نخواهد بود).

چرا تحلیل (زمانی) الگوریتم ها اهمیت دارد؟

مثال زیر نشان می دهد که با به کار بردن یک الگوریتم بدون تحلیل آن می تواند فقط هدر دادن زمان باشد!

الگوریتم محاسبه جمله nاُم دنباله فیبوناچی

```
f1(n)
    A[1]= 1
    A[2]= 1
    for i=3 to n
        A[i] = A[i-1] + A[i-2]
    return A[n]
```

```
f2(n)
    if n<2
        return 1
    else
        return f2(n-1) + f2(n-2)
```

که اگر هر دو الگوریتم را روی یک دستگاه امروزی پیاده کنیم زمان اجرای آن ها اینگونه خواهد بود.

n	f1	f2
40	41 ns	1048 µs
80	81 ns	13 min
160	160 ns	36 years
200	210 ns	4*10^13 years

فصل ۱. پیچیدگی زمانی(Time complexity)

برای بدست آوردن زمان اجرا باید الگوریتم را به یک زبان مثلا ++C پیاده سازی کرد و آن را اجرا کرد. و چون زمان اجرا به دستگاه و.... وابسته است از این رو نمی توان زمان ها را مقایسه کرد برای همین پیچیدگی زمانی الگوریتم ها را محاسبه می کنیم. این مفهوم به معنای تعداد محاسباتی است که الگوریتم انجام می دهد.

```
Average(A, n)
    S = 0            1
    for i=1 to n          n (تکرار i)
        S = S + A[i]  2*n(تکرار حلقه)
    S = S / n    2
    return S
```

$$T(n) = 1 + n + 2n + 2 = 3n + 3$$

```
Sum(A, B, n)
    for i=1 to n          n
        for j=1 to n          n
            C[i,j] = A[i,j] + B[i,j]   2*n
    return C
```

$$(2 * n + n) * n = 3n^2$$

$$T(n) = 3n^2 + n$$

مرتبه (order)

یافتن پیچیدگی زمانی کمی دشوار می باشد. مرتبه اجرا (order) هم اندازه پیچیدی زمانی در خود دانش دارد اما بدست آوردن آن بسیار ساده می باشد. به سادگی با چشم پوشی از ضرایب و جمله های کم اهمیت درون پیچدگی زمانی، به مرتبه اجرا می رسیم. برای مثال پیچیدگی آخرین الگوریتی که بررسی کردیم 3n^2 + n بود.

پس آن الگوریتم مرتبه n^2 دارد.

زمانی که اندازه ورودی افزایش می یابد ، جمله های کم اهمیت در برابر جمله بابیشترین درجه یا پراهمیت ترین دیگر تاثیری ندارد. به همین سادگی اگر اندازه ورودی افزایش یابد، از ضرایب نیز می توان چشم پوشی کرد.

مرتبه اجرا	پیچیدگی زمانی
N^5	N^5 + 1000n^4 – 3
$\sqrt{n}$	$\sqrt{\boldsymbol{n}}$+ 999
2^n	2^n + 1000 n^1000

مثال:

```
Sum(A, n)
    S = 0
    for i=1 to n
        S = S + A[i]
    return S
```

به سادگی دریافت می شود که پیچیدگی این الگوریتم $T(n) = 3n + 1$ می باشد. پس مرتبه آن n خواهد بود.

از سوی دیگر پیچیدگی زمانی الگوریتم زیر به آرایه و K بستگی خواهد داشت. برای مثال اگر $k = 4$ و

$A = [4, 9, 2, 3, 5]$آنگاه الگوریتم به سرعت خاتمه می یابد. ولی اگر $k = 8$ باشد، الگوریتم زمان بیشتری باید صرف کند.

```
Find (A, n, k)
    for i=1 to n
        if A[i] == k
```

```
            return i
    return -1
```

پس پیچیدگی زمانی این الگوریتم در بهترین حالت ۱ خواهد بود.(i=1) و در بدترین حالت n خواهد بود. زیرا حلقه تکرار n بار انجام می شود.

می توان پیچیدگی زمانی را در بدترین حالت، متوسط و یا بهترین حالت بدست آورد. با داشتن پیچیدگی زمانی در بهترین حالت، نمی توان درباره بکارگیری الگوریتم تصمیم گرفت. برای مثال اگر یک پیچیدگی زمانی برابر با $10n^3 + 1000$ باشد با فرض $n = 1000$، در بهترین حالت روی یک پردازنده که توان انجام یک میلیارد کار در ثانیه را دارد، برابر با مقدار زیر خواهد بود :

$$\frac{10 * 1000^3 + 1000}{1000000000} = 10S$$

به طور خلاصه همواره داشتن زمان اجرا در بدترین حالت بیشترین قدرت تصمیم گیری را به ما می دهد.

تحلیل یا بررسی یک الگوریتم زمانی که اندازه ورودی بسیار بزرگ باشد ،تحلیل یا بررسی مجانبی می گویند و حافظه مورد نیاز برای یک الگوریتم، زمانی که اندازه ورودی به سمت بی نهایت میل کند، تحلیل مجانبی فضایی می گوییم.

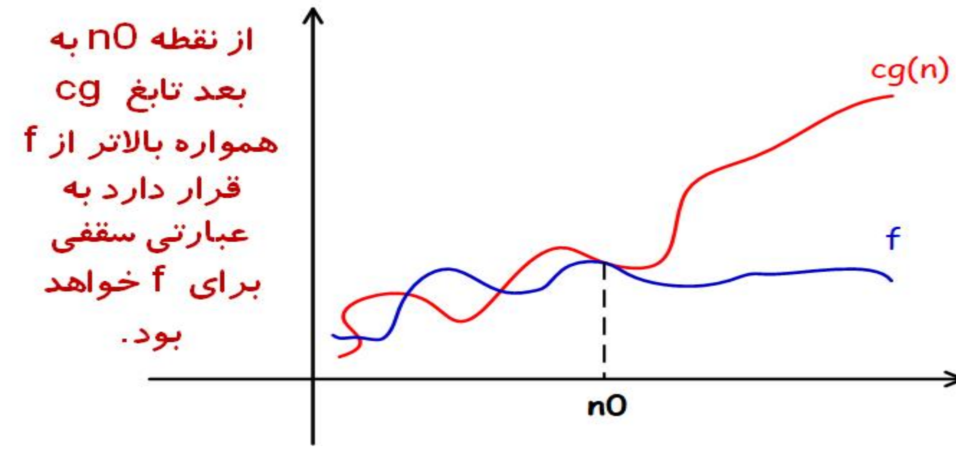
آشنایی با ۳ نماد جانبی

۱) O : سقفی برای یک تابع است.

۲) Ω :کمینه یا کفی برای یک تابع را نشان می دهد.

۳) θ: هم زمان هم سقف و هم کف را برای یک تابع مشخص می کند.

طبق تعریف f(n) ازمرتبه O(g(n)) است، اگر و تنها اگر $f_{(n)} \le cg(n)$ $\exists c > 0 \quad \forall n \ge n0$



به همین ترتیب اگر الگوریتمی از مرتبه $O(n^k)$ باشد، آنگاه با بیشتر شدن ورودی ها زمان اجرا با توان k رشد می کند. برای مثال اگر الگوریتمی از مرتبه $O(n^3)$ باشد و تعداد ورودی ها را دو برابر کنیم، آنگاه مرتبه آن(زمان اجرا) الگوریتم $2^3 = 8$ برابر می شود.

قانون های ساده برای بدست آوردن مرتبه زمانی

* **اگر ۲ الگوریتم پشت سر هم اجرا شوند، آنگاه مرتبه کل برابر است با ماکزیمم آن دو.** مثلا داریم:

$$f \in O(n), g \in O(n^2) \rightarrow \text{در کل} \in O(n^2)$$

* **اگر یک الگوریتم ، قطعه دیگر را فرا بخواند آنگاه با ضرب مرتبه آن دو می توان مرتبه زمانی را در کل بدست آورد.**

مثال) مرتبه زمانی الگوریتم زیر را بدست آورید.

```
for i=1 to n^2                 ---------->  O(n^2)
    for j=1 to n      O(n)   }
        S = S + i*j   O(1)   }  O(n * 1) = O(n)
    m = S/2           ---------->  O(1)
    for n=1 to n^3    O(n^3) }
        A[n] = m      O(1)   }  O(n^3 * 1) = O(n^3)
```

$\text{Max} = O(n^3)$

$O(n^3 * n^2) = O(n^5)$

راه دیگری برای بدست آوردن مرتبه زمانی وجود دارد که آن این است که پیچیدگی زمانی الگوریتم را حساب کرده (T_n)

سپس ضرایب و جملات کم اهمیت را حذف کنیم که باز هم به $O(n^5)$ می رسیم. پس با 2 برابر شدن اندازه ورودی ، $2^5 = 32$ زمان اجرا 32 برابر خواهد شد. یعنی اگر اندازه ورودی را 10 برابر کنیم، زمان اجرا تقریبا 100000 برابر خواهد شد.

الگوریتی که پیچیدگی زمانی آن چند جمله ای باشد آن را ، **کاربردی یا مهار شدنی** می گویند. از طرفی اگر پیچیدگی زمانی یک الگوریتم چند جمله ای نباشد برای نمونه نمایی باشد ، آن الگوریتم **مهار ناشدنی** است. از چنین الگوریتمی نمی توان استفاده کرد. برای مثال اگر یک الگوریتم مرتبه زمانی آن $O(2^n)$ باشد، اگر اندازه ورودی به 100 برسد، زمان اجرا کم و بیش بی نهایت خواهد شد.

فصل ۲. تقسیم و غلبه(Divide & Conquer)

در این رویکرد تلاش می کنیم تا یک نمونه (instance) از مسئله را به یک یا چند نمونه کوچکتر بخش یا تقسیم کنیم. پس از پایان یافتن پاسخ برای نمونه های کوچک یا همان غلبه بر آنها پاسخ مسئله را بدست می آوریم.

رویکرد تقسیم و غلبه بالا به پایین می باشد.

در اینجا یک مسئله با 100 نمونه داده داریم که آن ها را به ۳ بخش کوچک تقسیم می کنیم و سپس بخش های کوچکتر را به بخش های کوچکتر تبدیل می کنیم تاجایی این تقسیم کردن را ادامه می دهیم که غلبه به بخشهای کوچک یا یافتن پاسخ برای آن ها برایمان ساده باشد. پس از آن با کنار هم گذاشتن پاسخ بخش های کوچک، پاسخ خود نمونه را بدست می آوریم.

برای نمونه در جستجوی دودویی از این رویکرد کمک گرفته شده است.

جست و جوی دودویی (Binary Search)

یک الگوریتم برای یافتن دادهای دلخواه است که در آن از رویکرد تقسیم و غلبه کمک گرفته شده است. با بخش کردن یک آرایه از پیش مرتب شده ، می توان به تندی یک درایه دلخواه را یافت.

جست و جوی دودویی بازگشتی

```
BinarySearch(A,s,k,e)
    if e<s
        return 0
    m = ⌊(s+e) / 2⌋
    if A[m] == k
        return m
    if A[m] < k
        return BinarySearch(A,m+1,k,e)
    return BinarySearch(A,s,k,m-1)
```

جست و جوی دودویی غیر بازگشتی

```
BinaraySearch(A, n, k)
    s = 1
    e = n
    while s<e
        m = ⌊(s+e) / 2⌋
        if A[m] == k
            return m
        if A[m] < k
            s = m+1
        if A[m] > k
            e = m-1
    return 0
```

عزیز معدنی - دانشجو

پیچیدگی زمانی الگوریتم اینگونه خواهد بود.

T(n)= 4+ T(n/2)

T(1) = 1

بدترین حالت زمانی رخ می دهد که عددی که دنبالش هستیم در آرایه نباشد.

حال میخواهیم ببینیم که این الگوریتم مرتبه چند است؟ در این بخش دو روش برای محاسبه مرتبه زمانی الگوریتم های بازگشتی ارائه می کنیم:

محاسبه مرتبه زمانی

۱) روش استقرا

در این روش ما اعداد مختلف را امتحان کرده تا به الگویی برسیم؛ سپس مرتبه آن را از روی رابطه بدست آمده تشخیص می دهیم. برای مثال قبل در بدترین حالت داریم:

$T(1) = 1 \rightarrow T(2^0) = 4 * (0) + 1 = 1$

$T(2) = 5 \rightarrow T(2^1) = 4 * (1) + 1 = 5$

$T(4) = 9 \rightarrow T(2^2) = 4 * (2) + 1 = 9$

$T(8) = 13 \rightarrow T(2^3) = 4 * (3) + 1 = 13$

$T(16) = 17 \rightarrow T(2^4) = 4 * (4) + 1 = 17$

$\vdots$

$$\left.\begin{array}{l} T(2^k) = 4k + 1 \\ 2^k = n \rightarrow k = log_2^n \end{array}\right\} \longrightarrow T(n) = 4log_2^n + 1 \longrightarrow T(n) \in O(logn)$$

البته ما در این روش می توانیم از اعداد نیز صرف نظر کنیم. یعنی به جای محاسبه $T(n) = 4 + T(\frac{n}{2})$ بیاییم و $T(n) = T(\frac{n}{2})$ را محاسبه کنیم پس داریم :

$T(1) = 1 \rightarrow T(2^0) = (0) + 1 = 1$

$T(2) = 2 \rightarrow T(2^1) = (1) + 1 = 2$

$T(4) = 3 \rightarrow T(2^2) = (2) + 1 = 3$

$T(8) = 4 \rightarrow T(2^3) = (3) + 1 = 4$

$T(16) = 5 \rightarrow T(2^4) = (4) + 1 = 5$

⋮

$$\left.\begin{array}{l} T(2^k) = k + 1 \\ 2^k = n \rightarrow k = log_2^n \end{array}\right\} \longrightarrow T(n) = log_2^n + 1 \longrightarrow T(n) \in O(logn)$$

مرتبه های زمانی از بهترین تا بدترین:

$$1, log\ n, n, nlog\ n, n^2, \dots, n^k, k^n, n^n, n!$$

$\longrightarrow$

۲) روش قضیه اصلی

به طور کلی اگر پیچیدگی زمانی الگوریتم های بازگشتی به فرم $T(n) = a\,T(\frac{n}{b}) + cn^k$ باشد، می توان از **قضیه اصلی** کمک گرفت.

قضیه اصلی : پس اگر فرم کلی معادله بازگشتی به فرم $T(n) = a\,T(\frac{n}{b}) + cn^k$باشد آنگاه داریم:

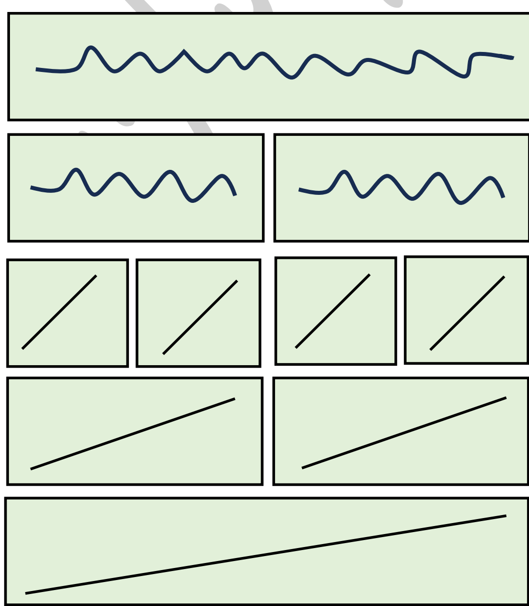
I. اگر $a < b^k \leftarrow T(n) \in O(n^k)$

II. اگر $a > b^k \leftarrow T(n) \in O\left(n^{log_b^a}\right)$

III. اگر $a = b^k \leftarrow T(n) \in O(n^k * log_2^n)$

در مثال قبل داشتیم $T(n) = 4 + T(\frac{n}{2})$ که a = 1 و b = 2 و c = 4 وk = 0 و چون حالت $a = b^k$ برقرار است، پس الگوریتم از مرتبه O(logn) می باشد.

در روش تقسیم و غلبه این گونه عمل می کنیم:



مرتب سازی ادغامی(Merge Sort)

در این الگوریتم که از رویکرد تقسیم و غلبه استفاده می کند برای مرتب سازی آرایه را به بخشی های کوچیک تقسیم می کنیم و آنقدر ادامه می دهیم که بتوانیم به بخش های کوچک به سادگی غلبه کنیم.

پس از مرتب کردن بخش های کوچک با ادغام یا merge کردن آن ها می توان به آرایه مرتب شده رسید.

تقسیم

4	9	2	7	8	1	3	5

4	9	2	7	8	1	3	5

4 9 | 2 7 | 8 1 | 3 5

4 | 9 | 2 | 7 | 8 | 1 | 3 | 5

7	6	11	21	7	9	12

7 6 11 | 21 7 9 12

7 | 6 11 | 21 7 | 9 12

7 | 6 | 11 | 21 | 7 | 9 | 12

(ادغام)غلبه

4 9 | 2 7 | 1 8 | 3 5

2 4 7 9 | 1 3 5 8

1 2 3 4 5 7 8 9

7 | 6 11 | 7 21 | 9 12

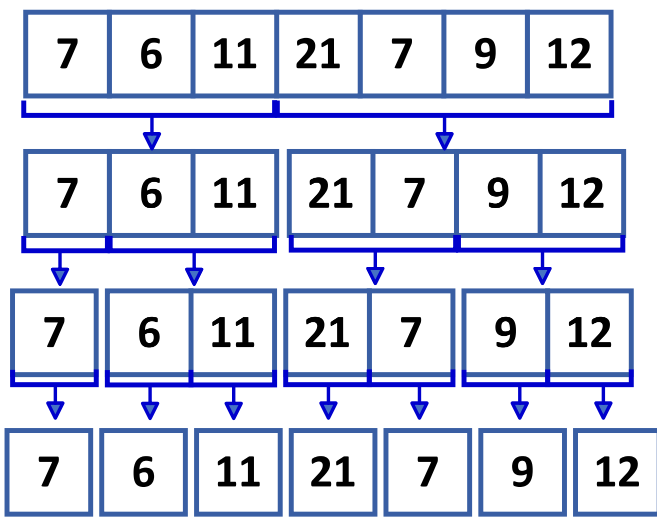
6 7 11 | 7 9 12 21

6 7 7 9 11 12 21

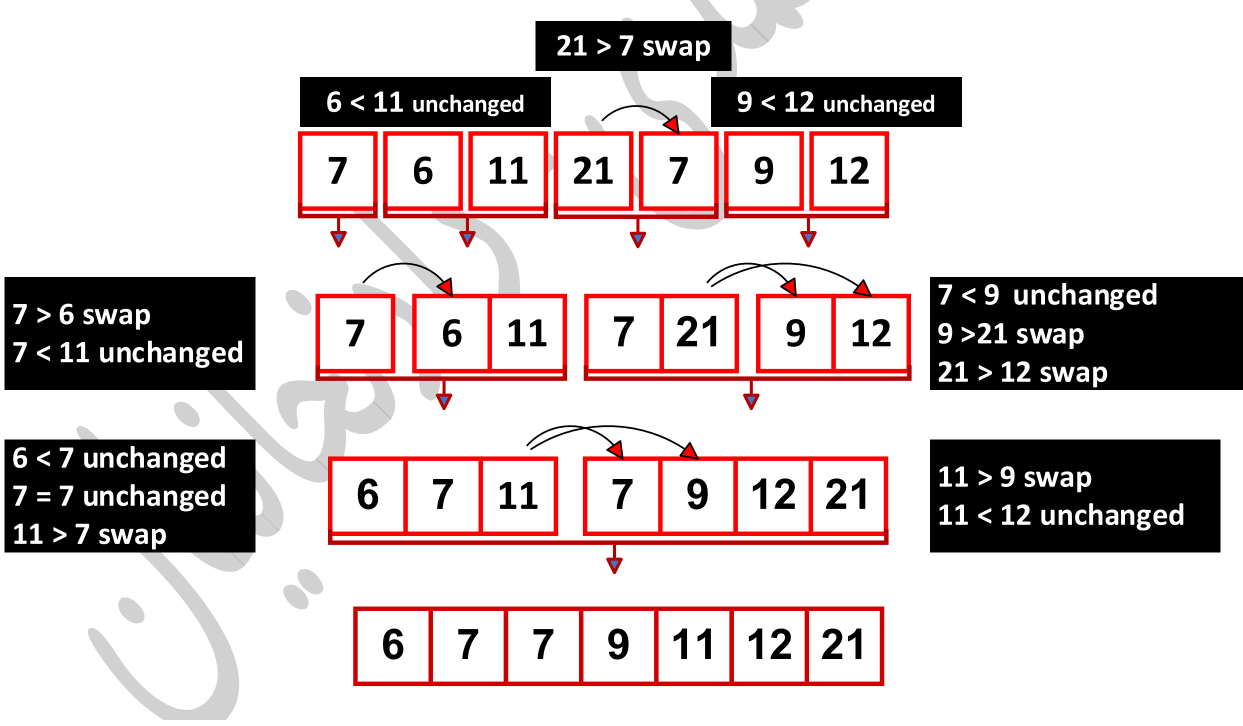
برای مرتب سازی به دو الگوریتم نیاز داریم

* الگوریتم خرد کردن آرایه به تکه ها و فراخوانی روی آن ها

مثلا داریم:



* الگوریتم ادغام دو تکه مرتب شده به یک تکه



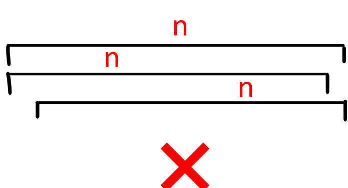
در زیر کد مراتب بالا را می بینیم

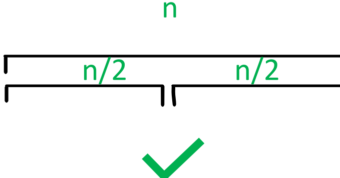
```
Sort (A,n)
    if n==1
        return A
    m =⌊n/2⌋
    L[1: m] = A[1:m]
    R[1: n-m] = A[m+1:n]
```

```
    L = Sort (L,n)
    R = Sort (R, n-m)
    return Merge (L,m,R,n-m)

Merge (A,a,B,b)
    i=1
    j=1
    k=1
    while (i<a and j<b)
        if A[i] < B[j]
            C[k] = A[i]
            i = i+1
        else
            C[k] = B[j]
             j = j+1
        k = k+1
    while i<a
        C[k] = A[i]
        i = i+1
        k = k+1
    while j<b
        C[k] = B[j]
        J = J+1
        k = k+1
```

الگوریتم Merge Sort با کمک رویکرد تقسیم و غلبه ساخته شده است. در این الگوریتم داده ها به دو نیمه بخش می شوند. برای همین الگوریتم بسیار کارا می باشد. این الگوریتم از مرتبه O(nlogn) است، که بهترین مرتبه برای مرتب سازی است. اگر n نمونه داده به دو یا چند بخش با اندازه نزدیک به n تقسیم شوند، آنگاه رویکرد تقسیم و غلبه کارایی نخواهد داشت.!

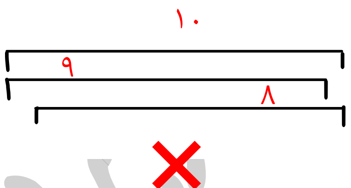




برای نمونه اگر بخواهیم جمله n اُم دنباله فیبوناچی را با این رویکرد بیابیم، آنگاه الگوریتم حاصل مرتبه خوبی نخواهد داشت.

این الگوریتم برای یافتن جمله n اُم، به دو جمله n-1 و n-2 نیاز دارد، پس کارا نخواهد بود.

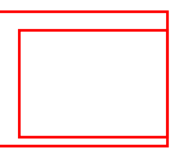
```
fibo(n)
    if n <= 1
        return 1
    else
        return fibo(n-1) + fibo(n-2)
```



اگر برای یافتن جمله n اُم به جمله های n/2 و n/3 نیاز داشتیم، آنگاه الگوریتم کارا می شد.

رویکرد تقسیم و غلبه برای یافتن $\binom{n}{k}$ هم کارایی ندارد. زیرا

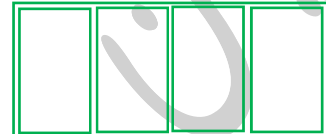
$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$



برای یافتن $\binom{n}{k}$ به $\binom{n-1}{k-1}$ و $\binom{n-1}{k}$ نیاز داریم. در اینجا یک مستطیل با اندازه n*k به ۲ بخش کوچک تقسیم شده است!

بخش های کوچک هم جوشانی زیادی دارند، پس الگوریتم کارا نخواهد بود.

اگر برای یافتن $\binom{n}{k}$ به $\binom{\frac{n}{2}}{k}$ و $\binom{\frac{n}{3}}{k}$ نیاز داشتیم، آنگاه الگوریتم کارا می شد.



در چه مواقعی نباید از روش تقسیم و غلبه کمک گرفت؟

۱) یک نمونه به اندازه n، به دو یا چند نمونه با اندازه تقریبا n تقسیم شود.

۲) یک نمونه به اندازه n به حدودا n نمونه با اندازه $\frac{n}{c}$ تقسیم شود.(c یک عدد ثابت است.)

یافتن !n با رویکرد تقسیم و غلبه پیشنهاد می شود.

زیرا برای یافتن !n به !(n-1) نیاز داریم.

```
fact(n)
    if n == 0
        return 1
    else
        return fact(n-1) * n
```

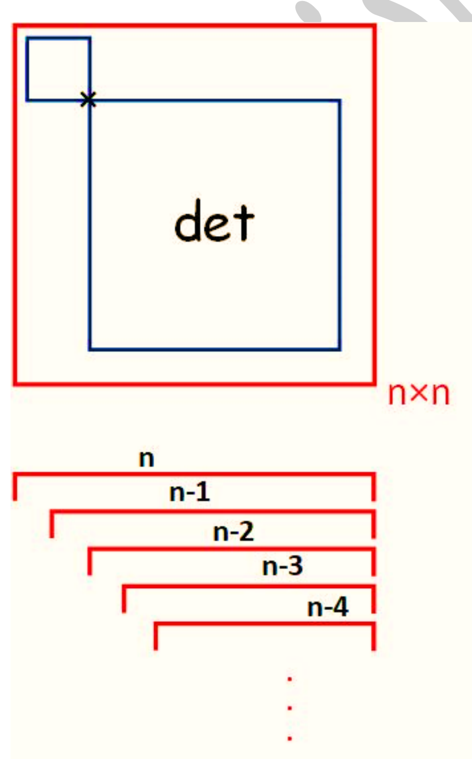
تکه های ساخته شده هم پوشانی ندارند، پس الگوریتم کارا خواهد بود.

یافتن دترمینان یک ماتریس با رویکرد تقسیم و غلبه پیشنهاد نمی شود.! زیرا برای یافتن دترمینان یک ماتریس $n * m$ به n تا دترمینان ماتریس های $(n-1) * (n-1)$ نیاز داریم.

حتی اگر بخواهیم یک درایه تکراری درون یک آرایه را بیابیم، باز هم رویکرد تقسیم و غلبه ناکارا خواهد بود.

در اینجا n داده به $(n-1) + (n-2) + (n-3) + \ldots + 1$ داده تقسیم می شود،

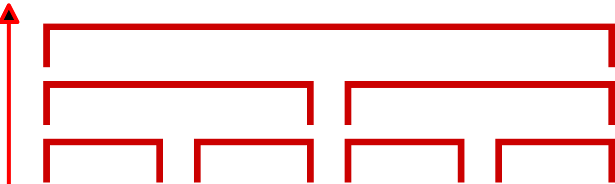
و باز هم این الگوریتم کارا نخواهد بود.!



(جست و جو هایی که باید انجام شود.)

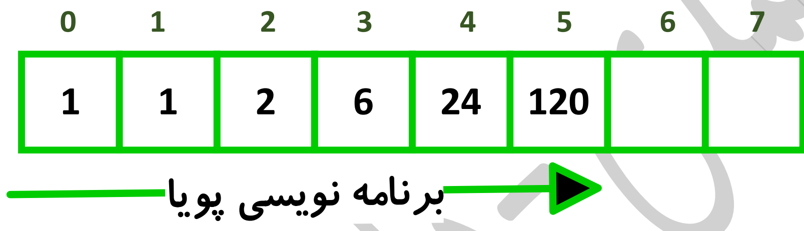
فصل ۳: برنامه نویسی پویا (Dynamic Programing)

این رویکرد یک رویکرد پایین به بالا (bottom - up) می باشد. در این رویکرد حل مسئله با یافتن پاسخ برای نمونه های کوچک مسئله و ذخیره آنها به پاسخ اصلی می رسیم.



الگوریتم هایی بر اساس رویکرد برنامه نویسی پویا

برای مثال برای فاکتوریل داریم:



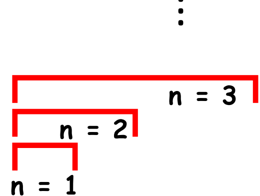
با این رویکرد می توان !5 را بدست آورد که باید از !0 شروع کرد و به !5 با ذخیره قبلی ها رسید.

الگوریتم یافتن !n با رویکرد پویا:

```
Factorial (n)
    for i=2 to n
        A[i] = A[i-1] * i
    return A[n]
```

مرتبه O(n)

⋮



الگوریتم فیبوناچی با هر دو رویکرد:

الگوریتم بر پایه رویکرد برنامه نویسی پویا

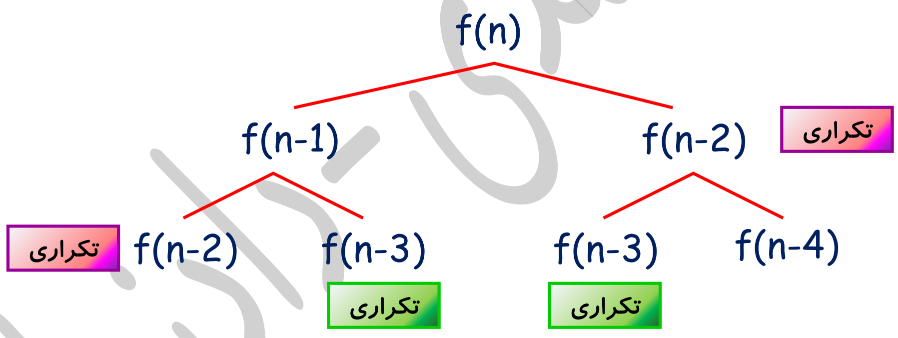
```
fibo(n)
    A[1] = 1 , A[2] = 1
    for i=2 to n
        A[i] = A[i-1] + A[i-2]
    return A[n]
```

الگوریتم بر پایه رویکرد برنامه نویسی تقسیم و غلبه

```
fibo(n)
    if n==1 or n==2
        return 1
    else
        return fibo(n-1) + fibo(n-2)
```

در این رویکرد با کنار هم گذاشتن پاسخ برای

نمونه های کوچک به پاسخ خود مسئله می رسیم.



مرتبه چند جمله ای

در برنامه نویسی پویا، پاسخ ها درون آرایه ذخیره میشوند و نیاز به محاسبه ۲ یا چند باره نداریم.

محاسبه $\binom{n}{k}$ با هر ۲ رویکرد :

$$\binom{n}{k} = \begin{cases} 1 & k = 0 \\ 1 & n = 0 \\ \binom{n-1}{k-1} + \binom{n-1}{k} \end{cases}$$

پویا

```
B(n, k)
    for i=0 to n
        for j=0 to k
            if(j==0 or i==j)
                A[i, j] = 1
            else
                A[i, j] = A[i-1, j-1] + A[i-1, j]
    return A[n, k]
```

تقسیم و غلبه

```
B(n, k)
    if k==0 or n=0
        return 1
    else
        return B(n-1, k-1) + B(n-1, k)
```

به کمک قضیه اصلی می توان دریافت که رویکرد تقسیم و غلبه پیچیدگی $2\binom{n}{k}$ دارد؛ در بهترین حالت $k=0$ یا $n=k$ و در بدترین حالت $k = \frac{n}{2}$ پس داریم:

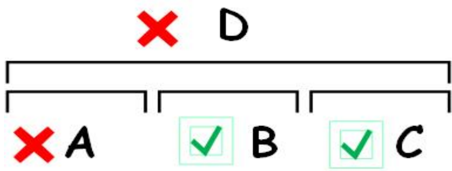
اگر $n \to \infty$ برود، آنگاه می توان از $\left(\frac{n}{2}\right)!$ در مقابل $n!$ چشم پوشی کرد پس الگوریتم از مرتبه $O(n!)$ می باشد. اما از طرفی در برنامه نویسی پویا برنامه باید نیمی از یک ماتریس $n * n$ را پرکند. پس از $O(n^2)$ می باشد.

عزیز محمد - دارابیان

ماتریس خیام – پاسکال

	0	1	2	3	4	5	6	7	8
0	1								
1	1	1							
2	1	2	1						
3	1	3	3	1					
4	1	4	6	4	1				
5	1	5	10	10	5	1			
6	1	6	15	20	15	6	1		
7	1	7	21	35	35	21	7	1	
8	1	8	28	56	70	56	28	8	1

نکته: **اگر قانون بهینگی برقرار باشد ، آنگاه می توان از برنامه نویسی پویا کمک گرفت. اگر جواب بهینه برای یک نمونه از مساله همواره شامل جواب های بهینه برای همه زیر مسئله ها باشد، آنگاه قانون بهینگی برقرار است.**



در اینجا توانسته ایم با پیدا کردن پاسخ برای زیر مسئله ها، پاسخی برای مسئله اصلی پیدا کنیم. اما این پاسخ بهترین نیست زیرا یکی از زیر مسئله ها بهترین پاسخ را دارانمی باشد.

انطباق رشته ها

با چندین روش میتوان دورشته را به هم منطبق کرد در اینجا ما از تعریف **لون اشتاین**، استفاده می کنیم.

فاصله لون اشتاین بین دو رشته، مهم ترین تعداد درج، حذف، انطباق یا جایگزینی مورد نیاز برای تبدیل یک رشته به رشته دیگر است. *** جابه جایی نیست!**

برای نمونه فاصله ali از amir برابر است با ۳، زیرا با ۳ تغییر می توان یکی از این رشته ها را به دیگری تبدیل کرد.

با حرف (L) و افزودن m و افزودن r، می توان از رشته ali به amir رسد.

$i\ a - - -$
$- a - - -$
→ **با فاصله ۴**

الگوریتم برپایه ی پویا مینویسیم که بهترین انطباق را پیشنهاد دهد.

انطباق زیر هم وجود دارد ولی فاصله بهینه و یا **لون اشتاین** را نمی دهد.

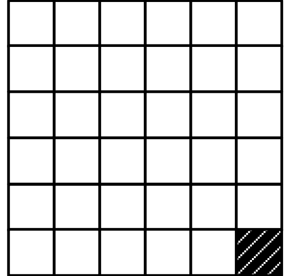
در اینجا فاصله دو رشته، ۶ است در حالی که کمترین فاصله این دو رشته از هم که در قبل اشاره شد ۴ است.

$i\ a - - - -$
$- - a\ m\ i\ r$
→ **با فاصله ۶**

ماتریس D که $m+1$ و $n+1$ سطر وستون دارد، تعداد ویرایش های مورد نظر (نیاز) برای تبدیل رشته ی S به T را نشان می دهد. $D[i,j]$ تعداد و یرایش های لازم برای تبدیل i حرف ابتدایی S به j حرف ابتدایی T می باشد

برای نمونه اگر ۳ = D[۴,۶] باشد آنگاه برای تبدیل ۴ حرف ابتدایی رشته S به ۶ حرف ابتدایی رشته T باید ۳ ویرایش انجام دهیم.

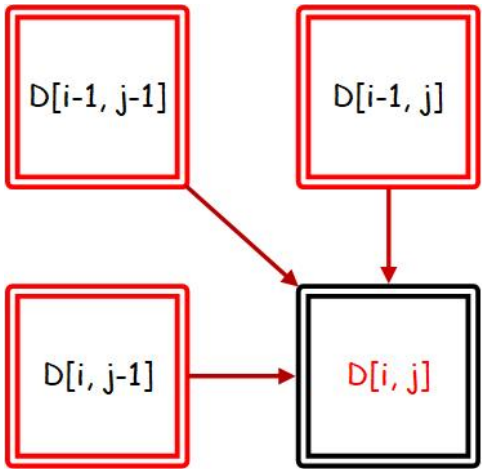
الگوریتم زیر که برپایه رویکرد برنامه نویسی پویا است تعداد تغییرات لازم، یعنی $D[m+1,n+1]$ را می یابد. درایه $D[m+1,n+1]$ نشان می دهد برای تبدیل کل رشته ی S به رشته T چند تغییر یا ویرایش لازم است؟



هدف یافتن این درایه می باشد

درایه $D[i,j]$ از روی درایه های $D[i-1,\ j]$ و $D[i,\ j-1]$ و $D[i-1,\ j-1]$ بدست می آید.

از روی این ۳ درایه می توان به $D[i,j]$ رسید



روش بدست آوردن $D[i,j]$ از روی درایه های مشخص شده

*** اگر حروف iاُم رشته S و jاُم رشته T برابر باشد،آنگاه $D[i,j]$ باید برابر با $D[i-1,\ j-1]$ می شود.**

*** اگر حروف iاُم رشته S و jاُم رشته T برابر نباشد، آنگاه $D[i,j]$ باید برابر با**

$min\ (D[i-1,\ j], D[i,\ j-1]) + 1$ **شود.**

رشته S طول m و رشته T طول n دارد. این الگوریتم تعداد تغییرات لازم برای تبدیل S به T را باز می گرداند.

```
L(S, T)
    for i=0 to m
        D[i, 0] = i
```

ستون 0اُم ماتریس را پر میکند

```
    for j=0 to n
        D[0, j] = j
```

سطر 0اُم ماتریس را پر میکند

	0	1	2	3
0	0	1	2	3
1	1			
2	2			
3	3			

$D[i,0]$ باید برابر i شود؛ زیرا برای تبدیل یک رشته با طول i به یک رشته با طول 0 باید i تا ویرایش انجام دهیم.

```
for i=1 to m
    for j=1 to n
        if S[i] = T[j]
            D[i, j] = D[i-1, j-1]
        else
            D[i, j] = min(D[i-1, j], D[i, j-1]) + 1
return D[m, n]
```

(به طور کلی یعنی i تا حرف از آن حذف کنیم)(برای نمونه اگر بخواهیم `return D[m, n]` رشته ab را با رشته تهی منطبق کنیم باید ۲ حرف از آن حذف کنیم). همچنین برای D[0, j] باید همین کار را انجام دهیم.

مثال۱: فاصله لون اشتاین دو رشته hassan و hossein را می یابیم.

به یک ماتریس 1+6 در 1+7 نیاز داریم...

		h	o	s	s	e	i	n
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
h	1	1	0	1	2	3	4	5	6
a	2	2	1	2	3	4	5	6	7
s	3	3	2	3	2	3	4	5	6
s	4	4	3	4	3	2	3	4	5
a	5	5	4	5	4	3	4	5	6
n	6	6	5	6	5	4	5	6	5

$$hassan \rightarrow h - ss - \ - n$$

$$hossein \rightarrow h - ss - \ \ i\, n$$

مثال۲۱: فاصله لون اشتاین دو رشته amir و ali را می یابیم.

			a	l	i
		0	1	2	3
	0	0	1	2	3
a	1	1	0	1	2
m	2	2	1	2	3
i	3	3	2	3	2
r	4	4	3	4	3

$amir \rightarrow a - i$

$ali \rightarrow a - i -$

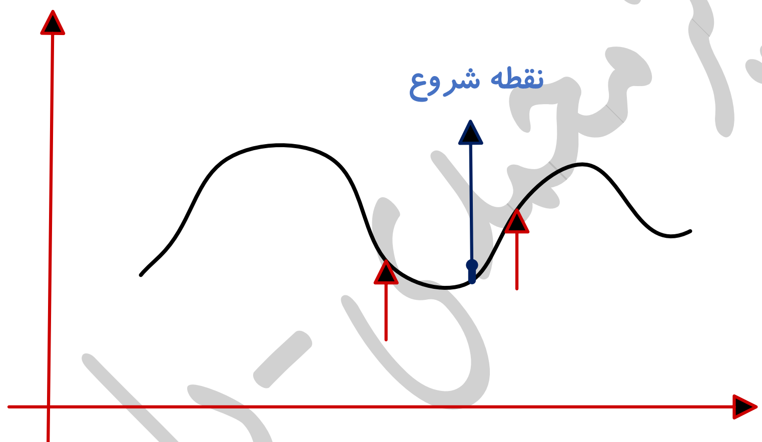
فصل ۴. رویکرد حریصانه (Greedy Approach)

الگوریتم هایی که بر پایه این رویکرد ساخته شده اند، بدون توجه به گذشته و آینده، در لحظه بهترین کار شدنی را انجام می دهند.

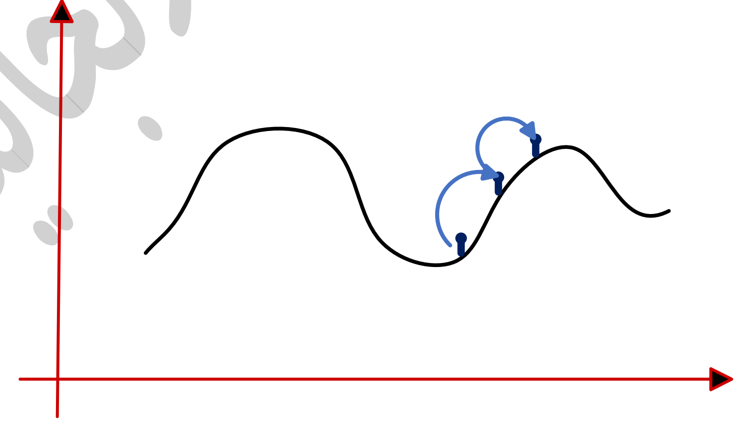
الگوریتم تپه نورد

یک تپه نورد که با رویکرد حریصانه می خواهد به قله برسد، بدون توجه به راهی که آماده و راهی که در پیش دارد، یک گام به قله نزدیک شود؛ **لذا شاید به نوک تپه نرسد و در یک قله کوتاه تر گیر کند!**

الگوریتم زیر بیشینه یا ماکسیمم یک تابع را می یابد. این الگوریتم حریصانه است.



بی آنکه به نقطه های قبلی یا بعدی توجه کنیم، یک واحد به سمت بیشینه حرکت می کنیم.



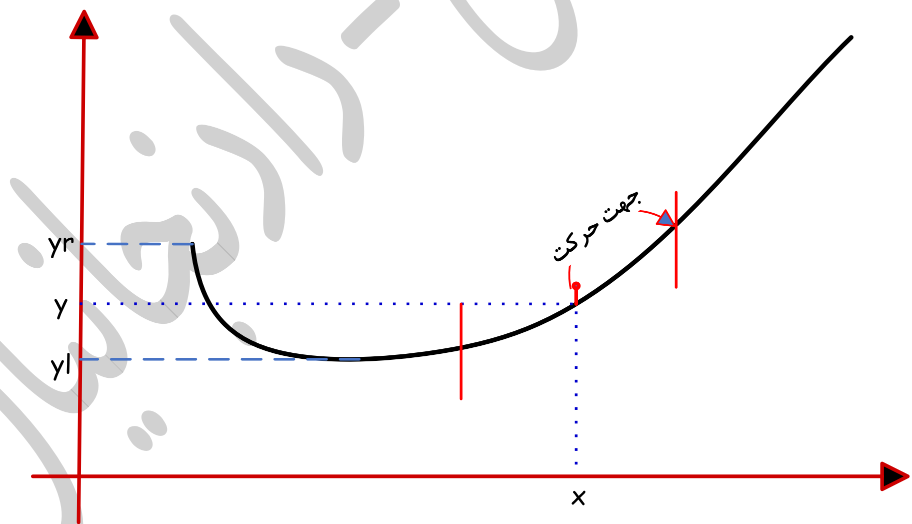
با کمک این الگوریتم ماکسیمم یا بیشینه حرکت یافت می شود؛ پس بهترین حرکت که از نقطه شروع می توان داشت، یک واحد به سمت راست رفتن است. (**در این مثال الگوریتم ما را به یک بیشینه محلی(local maximum) می رساند و بهترین پاسخ را نمی یابد**).

عزیز محبی - دارنگیان

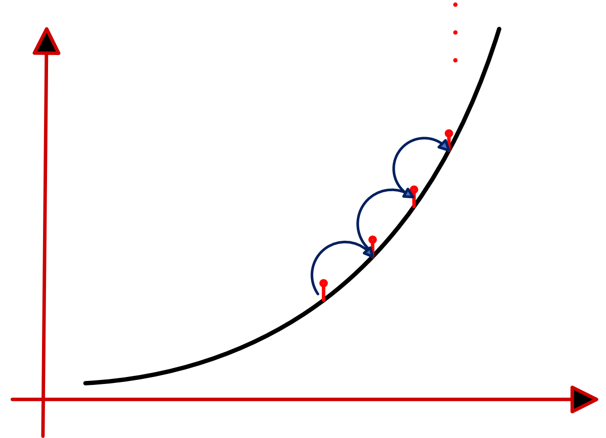
```
Max(f, x)
    y = f(x)
    yr = f(x + 1)
    yl = f(x - 1)
    while yr > y or yl > y
        if yr > yl
            x = x+1
        else
            x = x-1
        y = f(x)
        yr = f(x+1)
        yl = f(x-1)
    return x
```

در کد بالا f تابع و x نقطه شروع حرکت است. در هر لحظه بدون توجه به اینکه y قبلاً چه مقداری داشته، یک واحد به سمت ماکسیمم حرکت می کنیم.

نکته: ممکن است این تابع ماکسیمم مطلق را نیافته و یک ماکسیمم محلی را باز گرداند.



تذکر: اگر تابع P به شکل زیر باشد، آنگاه الگوریتم پایان نمی یابد



اگر ماکسیمم با شروع از نقطه x برابر بی نهایت باشد آنگاه این الگوریتم پایان نمی یابد!

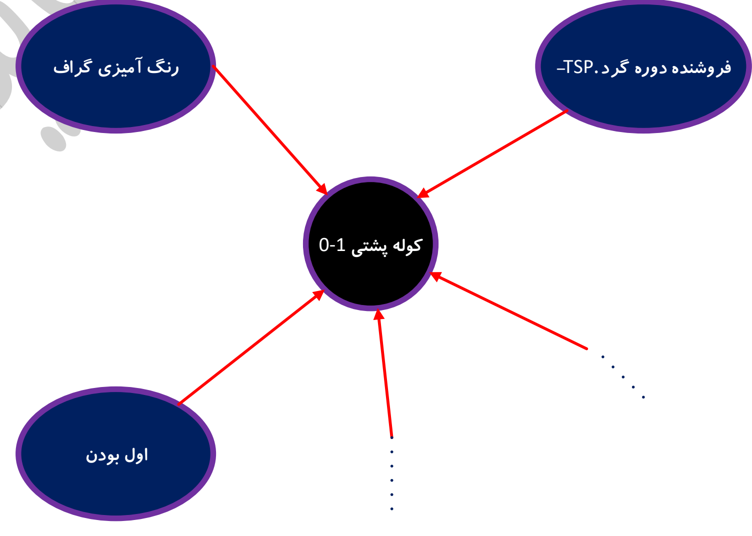
برای برخی مسائل الگوریتم حریصانه بهترین پاسخ را می یابد و در برخی مسائل نیز یافتن بهترین پاسخ تضمین نشده است.

مثلا یکی از الگوریتم های حریصانه که برای مسائل سخت طراحی شده است، **مسئله کوله پشتی 0-1** است که برخی مواقع بهترین پاسخ را می یابد.

مسئله کوله پشتی 0-1(Knapsack Problem)

هدف یافتن ترکیبی از کالاها که مجموع ارزش آنها بیشینه شود اما مجموع وزن آنها از ظرفیت کوله پشتی بیشتر نشود.

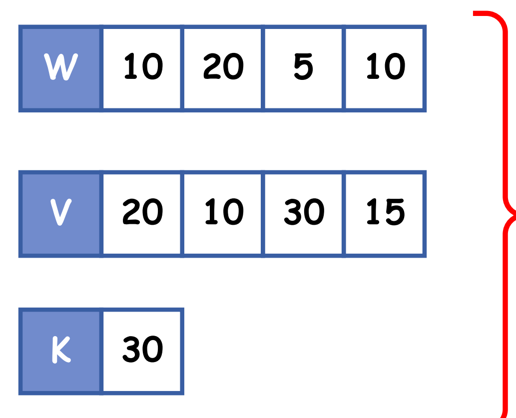
مسئله کوله پشتی 0-1 یکی از مهمترین مسئله ها در علم کامپیوتر است. با رویکرد های گوناگون می توان آن را حل کرد اما تا کنون کسی نتوانسته الگوریتمی برای آن بنویسد که در زمان چند جمله ای پاسخ را بیابد.



اهمیت مسئله کوله پشتی ۰-۱ ۱

عزیز محمدی - ترادیغبان

مثال: بهترین پاسخ نمونه مسئله زیر را بیابید.



W	10	20	5	10
V	20	10	30	15
K	30			

W = weight وزن
V = value ارزش
K = capacity ظرفیت کوله پشتی

در این نمونه (instance) چهار کالا داریم. پس با بررسی 2^n ترکیب می توان بهترین پاسخ را یافت.

وزن	ارزش	کالا های انتخاب شده
0	0	[ ][ ][ ][ ]
10	20	[✓][ ][ ][ ]
30	30	[✓][✓][ ][ ]
15	50	[✓][ ][✓][ ]
20	35	[✓][ ][ ][✓]
25	40	[ ][✓][✓][ ]
		⋮
35	-1	[✓][✓][✓][ ]
		⋮
25	65	[✓][ ][✓][✓] بهترین پاسخ

در این نمونه از مسئله باید $2^4 = 16$ ترکیب را بیازماییم.

برخی از آنها نیز قبول نیستند؛ زیرا مجموع وزن انتخاب شده از ظرفیت کوله پشتی بیشتر می شود.

لذا بهترین پاسخ با ارزش 65 و وزن 25 می باشد.

در اینجا ما یک الگوریتم بر پایه رویکرد حریصانه برای این مسئله می سازیم که پاسخ بسیار خوبی برای هر نمونه از مسئله می یابد.

الگوریتم حریصانه کوله پشتی 0-1

***کالا ها را بر اساس ارزش هر کیلو به صورت نزولی مرتب می کنیم.**

*** تا زمانی که مجموع وزن کالاها بیشتر از ظرفیت کوله پشتی نشده است، کالاها را از روی فهرست انتخاب می کنیم.**

یا به طور خلاصه :

کالا ها را بر اساس ارزش هر کیلو مرتب می کنیم. در ادامه هر کالا را یا باید انتخاب کرد یا اگر انتخاب آن باعث اضافه وزن شود، از انتخاب آن چشم پوشی می کنیم.

مثال: کالا هایی که می توان از یک میوه فروشی برداشت.

	گوجه	پیاز	ترازو	انبه
W	10	3	20	5
V	50	60	1000	150
K	24			

W = weight وزن
V = value ارزش
K = capacity ظرفیت کوله پشتی

با رویکرد حریصانه پاسخ این نمونه مسئله را میابیم. شاید الگوریتم بهترین پاسخ را نیابد. در برخی مسائل رویکرد حریصانه بهترین پاسخ یا یک پاسخ خوب می یابد. برای مسئله کوله پشتی 0-1 رویکرد حریصانه یک پاسخ خوب می یابد.

عزیز محمدی - آذر ریحانیان

در آغاز کالا ها را بر اساس هر کیلو یشان مرتب می کنیم و پس از آن به ترتیب کالا ها را یا انتخاب کرده یا از انتخاب آنها صرف نظر می کنیم...

کالا	ارزش هر کیلو	وزن	انتخاب یا عدم انتخاب
ترازو	50	20	✔
انبه	30	5	✘
پیاز	20	3	✔
گوجه	5	10	✘

در اینجا پاسخ پیدا شده **ترازو و پیاز** که با ارزش **۱۰۶۰** و وزن **۲۳** می باشد.

نکته: در کوله پشتی 0-1 نمی توان بخشی یا کسری از یک کالا را برداشته برای نمونه نمی توانیم ۴ کیلو انبه برداریم! یک کالا یا انتخاب می شود(1) یا انتخاب نمی شود(0).

کوله پشتی کسری

در مسئله کوله پشتی کسری می توان بخشی یا کسری از یک کالا را برداشت برای نمونه می توانیم کسری از ترازو یا انبه را برداریم!

نکته: رویکرد حریصانه بهترین پاسخ برای مسئله کوله پشتی کسری را می یابد. اما اگر مسئله کوله پشتی 0-1 باشد، یافتن پاسخ بهینه تضمین نشده است.

اگر نمونه قبلی را کسری در نظر بگیریم، آنگاه رویکرد حریصانه حتما بهترین پاسخ را می یابد.

در اینجا نیز کالا ها را بر اساس ارزش هر کیلویشان مرتب می کند. در ادامه تا پرشدن کوله پشتی از با ارزش ترین کالای باقی مانده بر میداریم...

کالا	ارزش هر کیلو	وزن برداشته شده
ترازو	50	20
انبه	30	4
پیاز	20	0
گوجه	5	0

این بار مسئله کسری در نظر گرفته شده است. الگوریتم حریصانه پاسخ را یافته:

باید همه ترازو 4 کیلو انبه برداریم.

همیشه وزن کالا های برداشته شده برابر با ظرفیت کوله پشتی خواهد بود. در اینجا ارزش آنها **۱۱۲۰** می شود.

در برخی مسائل رویکرد حریصانه همواره بهترین پاسخ را می یابد.

الگوریتم های حریصانه Prime و Kruskal

دو الگوریتم حریصانه Prime و Kruskal هر دو بهترین درخت پوشای کمینه را می یابند. (**الگوریتم های حریصانه پیچیدگی کمی دارند و هم تندی یا یک پاسخ خوب یا بهترین پاسخ را می یابند. اینکه پاسخ یافت شده همواره بهترین است یا نه بستگی به مسئله دارد**).

در ادامه راجع به الگوریتم های Digkstra و Prime و Kruskal خواهیم گفت. این الگوریتم ها روی گراف ها تعریف شده است.

پیش از پرداختن به این الگوریتم ها باید چند تعریف را دانست...

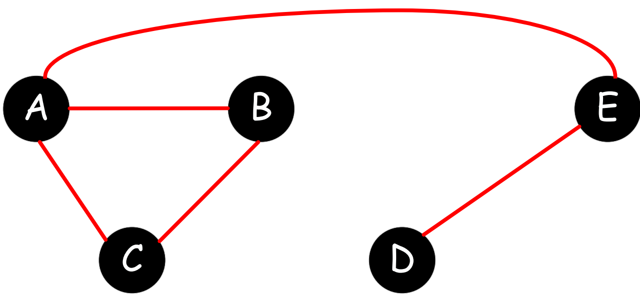
درخت پوشا(Spanning Tree)

درختی است که از روی یال ها و راس های یک گراف ساخته شده است.

تذکر: در یک درخت همواره تعداد یال ها یکی کمتر از تعداد راس ها است.

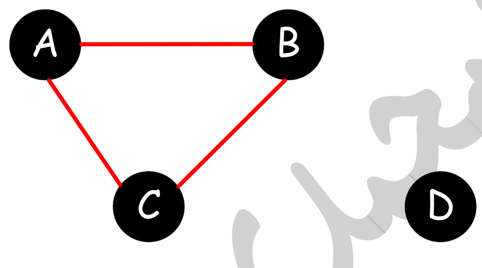
برای نمونه اگر G یک گراف با رئوس $\{A, B, C, D, E\}$:

عزیز محمدی - داریخانیان



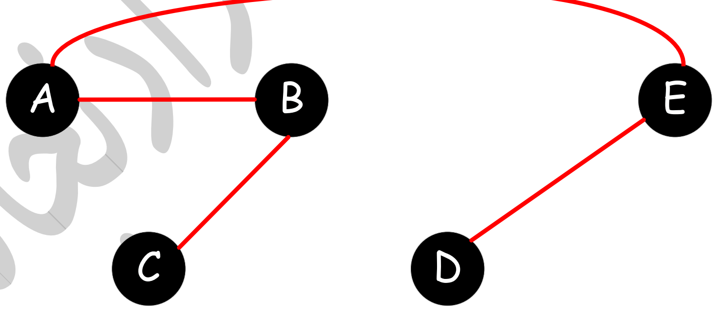
پوشا است چرا که همه رئوس G را دارد **درخت نیست چرا که دور دارد.**

مثال۲: اگر G یک گراف با رئوس $\{A, B, C, D, E\}$ باشد آنگاه



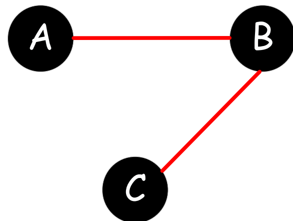
نه پوشا است **نه درخت است**

مثال۳: اگر G یک گراف با رئوس $\{A, B, C, D, E\}$ باشد آنگاه



هم پوشاست؛ چرا که همه راس هایG را در خود دارد **هم درخت است چرا که دور ندارد.**

مثال۴: اگر G یک گراف با رئوس $\{A, B, C, D, E\}$ باشد آنگاه



پوشا نیست؛ چرا که راس های E,D را ندارد. **درخت است**

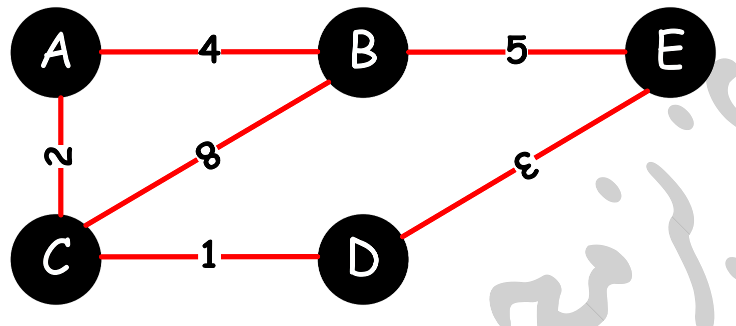
عزیز مجیدی - دارالعابدان

اگر گراف وزن دار باشد آنگاه می توان برای آن درخت پوشای کمینه هم یافت.

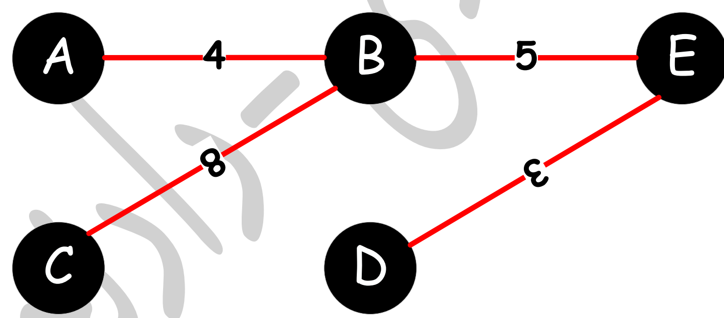
درخت پوشای کمینه(Minimum Spanning Tree - MST)

درخت پوشایی که مجموع وزن یال های آن کمینه یا مینیمم باشد را درخت پوشای کمینه می گویند.

برا ی نمونه اگر G یک گراف با رئوس $\{A, B, C, D, E\}$ باشد:

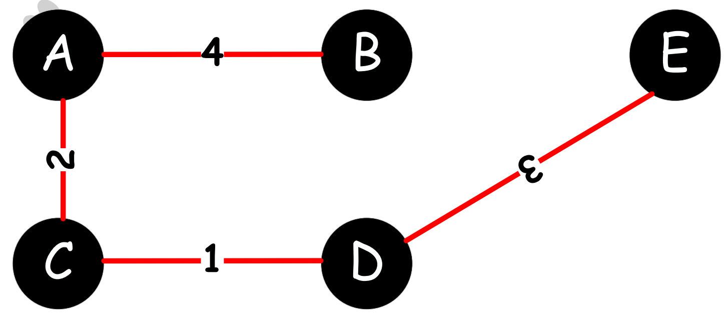


آنگاه



یک درخت پوشا با وزن ۲۰ می باشد

با بررسی درخت های پوشای مختلف می توان دریافت که



در خت پوشا با وزن ۱۰ می باشد و این درخت کمترین وزن را دارد پس MST برای گراف G است.

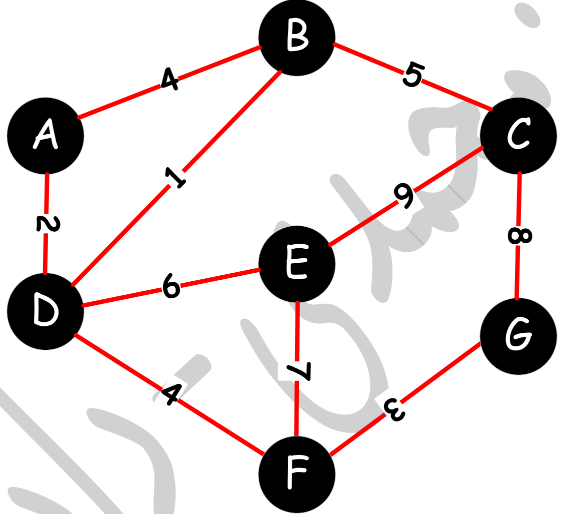
یافتن درخت پوشای کمینه

الگوریتم Prime

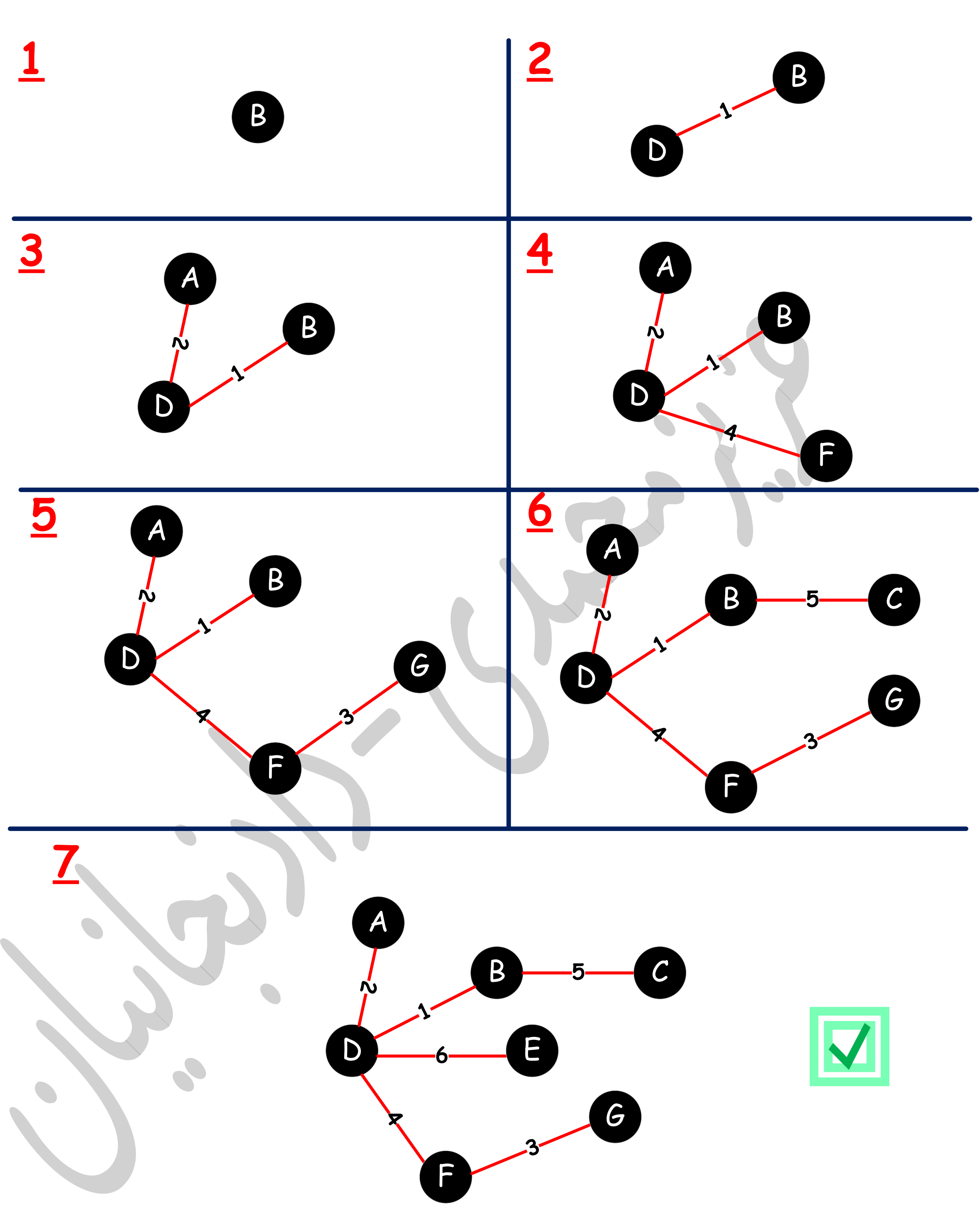
این الگوریتم حریصانه MST را می یابد. درخت پوشای یک گراف با n راس دقیقا n-1 یال خواهد داشت.

در الگوریتم Prim از یک راس دلخواه شروع کرده و در ادامه به صورت حریصانه، نزدیک ترین راس از مجموعه راس های انتخاب نشده را انتخاب کرده و به مجموعه راس های خود می افزاییم.

برای نمونه اگر این گراف را داشته باشیم:



و الگوریتم Prim را از راس B شروع کنیم، آنگاه به این پاسخ می دهیم....



با شروع از راس B به این MST می رسیم. **یک گراف می تواند چندین MST داشته باشیم.**

شبه کد الگوریتم Prim

$F = \emptyset$ یال های انتخاب شده

$Y = \{V1\}$ راس شروع خواهد بود

$while\ Y! = V$ راس های گراف است در مجوعه V
$Select\ a\ vertex\ in\ V - Y\ that\ is\ nearest\ to\ Y;$
$add\ selected\ vertex\ to\ Y;$
$add\ the\ edge\ connecting\ selected\ vertex\ to\ Y\ to\ F;$

در الگوریتم پریم:

- **ابتدا یک راس را انتخاب می کنیم.**
- **تا رسیدن به MST**

*** نزدیک ترین راس انتخاب نشده به یکی از راس های انتخاب شده را به مجموعه خود می افزاییم.**

الگوریتم Kruskal

این الگوریتم نیز حریصانه می باشد.

در این الگوریتم حریصانه یال ها را به صورت صعودی مرتب کرده و از فهرست یال ها n-1 یال که ایجاد دور نکنند را انتخاب می کنیم.

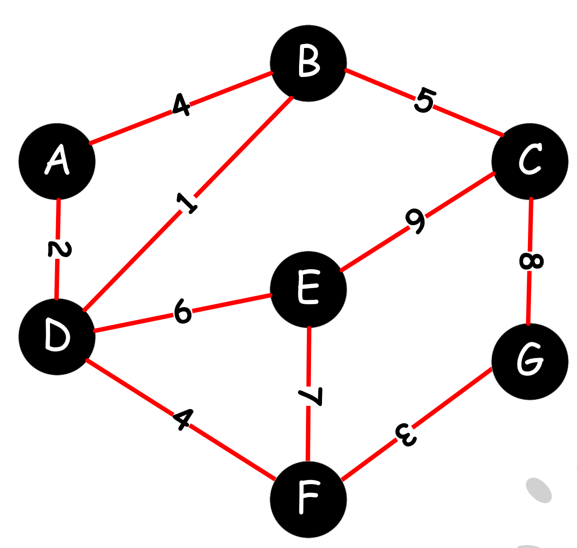
به بیان دیگر:

- **در آغاز، هر راس یک مجموعه، با یک درایه مشخص می شود.**
- **تا رسیدن به MST**

***از فهرست مرتب شده یال ها، یالی که دو مجموعه جدا گانه را به هم متصل می کند را انتخاب می کنیم (اگر در ابتدا انتهای یال درون یک مجموعه باشد از آن صرف نظر می کنیم).**

برای نمونه اگر این گراف را داشته باشیم.

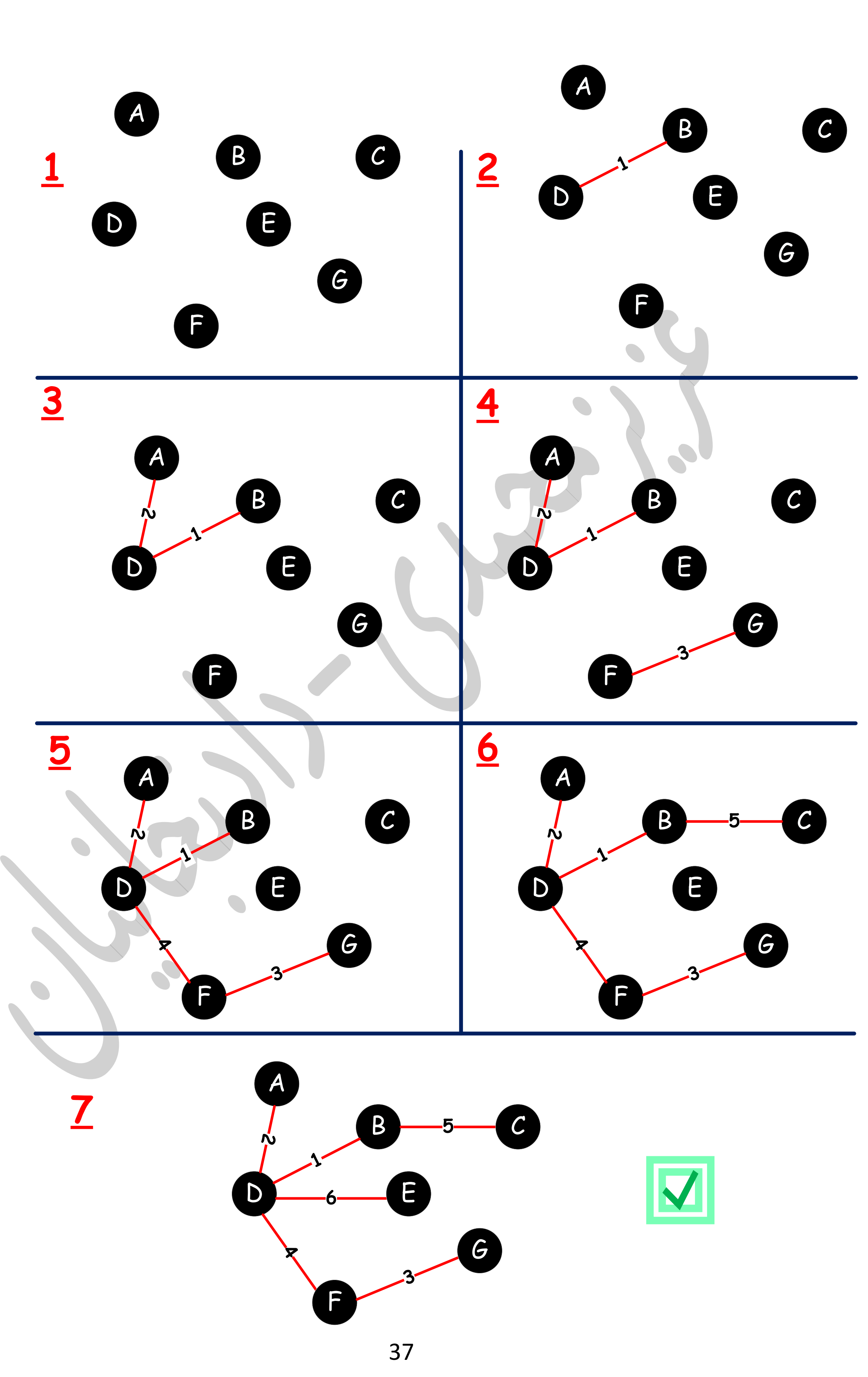
ا.م.محمدی - دارابخانیان



و الگوریتم Kruskal را روی آن اجرا کنیم داریم:

فهرست یال ها:

یال	وزن	
B_D	1	✓
A_D	2	✓
F_G	3	✓
A_B	4	✗
D_F	4	✓
B_C	5	✓
D_E	6	✓
E_F	7	✗
C_G	8	✗
C_E	9	✗



عزم محمدي - دار الريحان

شبه کد الگوریتم Kruskal

$F = \emptyset$ یال های انتخاب شده

$$Create\ disjoint\ subsets\ of\ V\ one\ for\ each\ vertex.$$

$$Sort\ the\ edges\ in\ E\ in\ nondecreasing\ order.$$

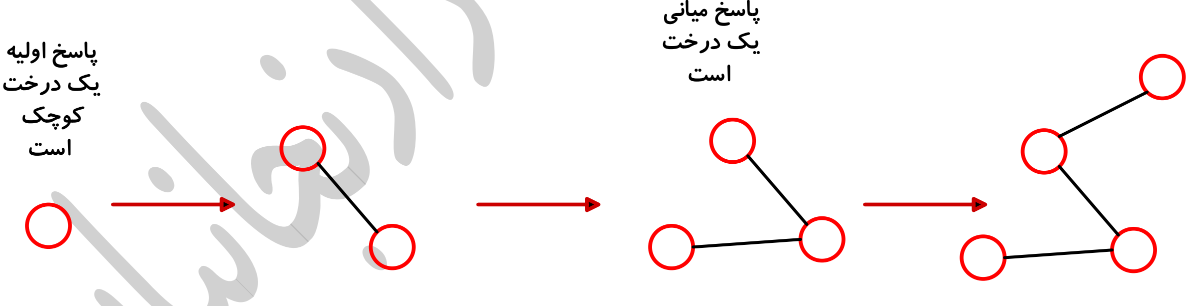
$$While\ all\ the\ subsetes\ are\ not\ merged:$$

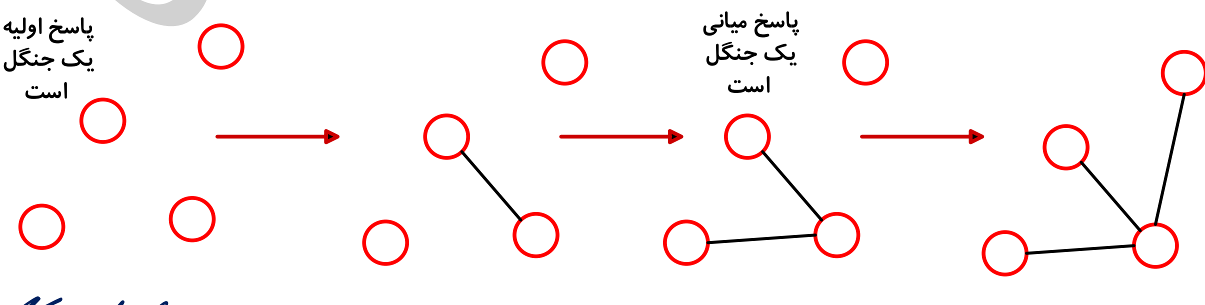
- $Select\ next\ edge$
- $if\ the\ edge\ connects\ two\ vertices\ disjoint\ subsets\ then$
  $*\ merge\ the\ subsets$
- $add\ the\ adge\ to\ F$

تفاوت روش Prim و Kruskal

۱) در Prim با یک درخت کوچک شروع کرده و هر بار راس به آن می افزاییم. برای همین پاسخ های میانی همواره درخت خواهند بود.

۲) در Kruskal با یک جنگل (jungle) شروع کرده و هر بار دو درخت آن جنگل را به هم متصل می کنیم. پس پاسخ های میانی همواره جنگل خواهند بود.





رمزنگاری هافمن (Huffman Encoding)

یک الگوریتم حریصانه است که همزمان یک نوشته را **فشرده** و **رمزگذاری** می کند.

برخی حروف کاربرد بیشتری در یک نوشته دارند با دادن فضای کوچک تر به آنها میتوان یک نوشته را فشرده و هم رمزگذاری کرد.

برای نمونه حرف s کاربرد بیشتری در برابر حرف q دارد.اگر هنگام ذخیره سازی به s فضای کوچک تر و به q فضای بزرگ تری بدهیم آنگاه نوشته جای کوچک تری می گیرد.

(**در حالت عادی هر حرف درون یک بایت که هشت بیت دارد ذخیره میشود**)

در الگوریتم فشرده سازی هافمن یک حرف هر چقدر پرکاربرد تر باشد جای کمتری میگیرد.

برای نمونه شاید S در سه بیت ذخیره شود نه در هشت بیت!

در حالت عادی رشته(i_see_tennis_sets) درون ۱۷ بایت ذخیره میشود به عبارتی ۱۳۶=۸*۱۷ بیت نیاز است

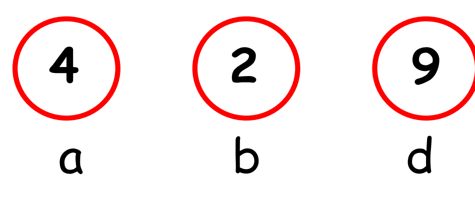
می توان دید در این رشته حروف s وe تواتر(تکرار) هستند؛ پس با دادن کد کوتاه تر از8 بیت به آنها رشته را فشرده می کنیم.

الگوریتم هافمن

(**باید فراوانی هر کدام از حروف یا نشانه ها را بدست آورد**)

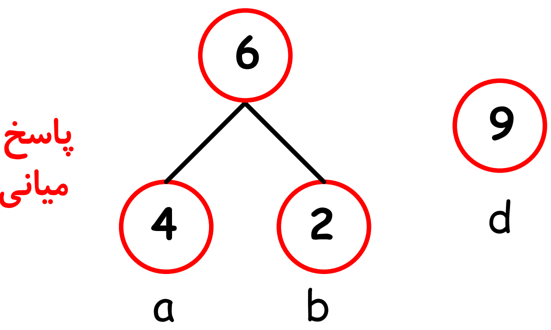
- **در آغاز هر حرف یک درخت یا یک گره می باشد.**
- **تا رسیدن به یک درخت**

* **باید دو درخت که کمترین فراوانی را دارند باهم ترکیب کنیم. این درخت مجموع وزن دو درخت را خواهد داشت.**



جنگل اولیه

برای نمونه اگر فراوانی حروف a، b و d برابر با ۴و۲و۹ باشد، آنگاه در آغاز هر حروف یک درخت کوچک خواهد بود



در ادامه تا رسیدن به یک درخت، باید **دو درخت با وزن کمتر** را به هم متصل کنیم. ریشه این درخت تازه ساخته شده گره ای باوزن مجموع دو درخت خواهد بود.

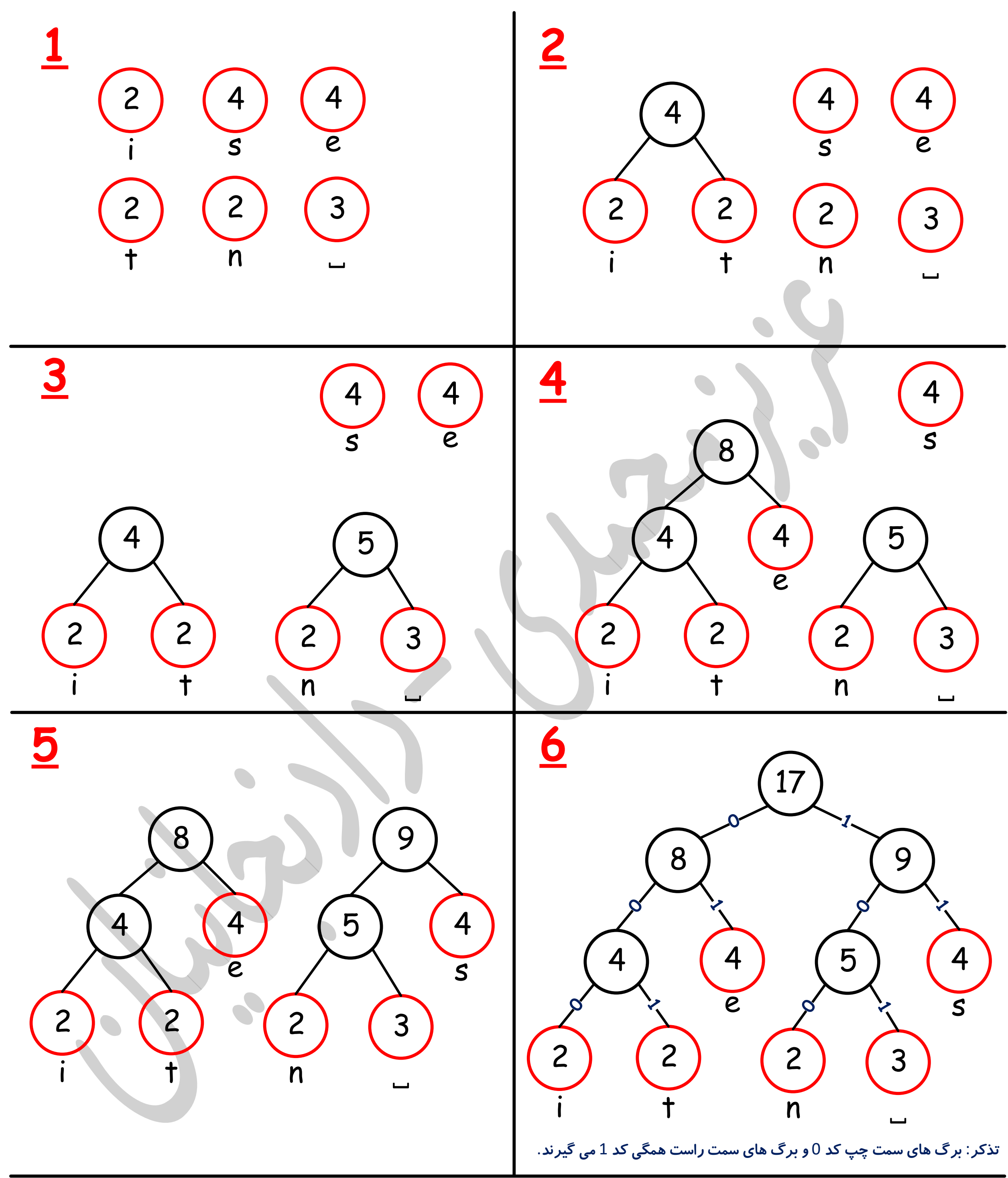
در اینجا درخت های (2 b) , (4 a) باید باهم ترکیب شوند.

پس از ترکیب این دو به پاسخ میانی می رسیم.

مثال: درخت هافمن رشته(i_see_tennis_sets) را بیابید.

فراوانی	حروف
۲	i
۴	s
۴	e
۲	t
۲	n
۳	␣

درخت هافمن رشته نوشته شده به صورت زیر است...



از روی درخت هافمن میتوان کد ذخیره سازی کوتاه ورمزی هر حرف را بدست آورد که هر حرف مسیر رسیدن از ریشه به آن حرف است

(**با حرکت به سمت راست ۱ و با حرکت به سمت چپ ۰ به کد حرف افزوده میشود**)(قسمت 6 شکل)

در اینجا به sکد ۱۱ داده میشود یعنی به جای ذخیره s درون ۸ بیت آن را درون دو بیت ذخیره می کنیم 😉

یا حرف t کد ۰۰۱ را بدست آورده است. پس هنگام ذخیره سازی به جای t، مقدار ۰۰۱ را ذخیره می کنیم.

کد بدست آمده برای حروف رشته نوشته شده:

حرف	کد	فراوانی
i	000	2
s	11	4
e	01	4
t	001	2
n	100	2
␣	101	3

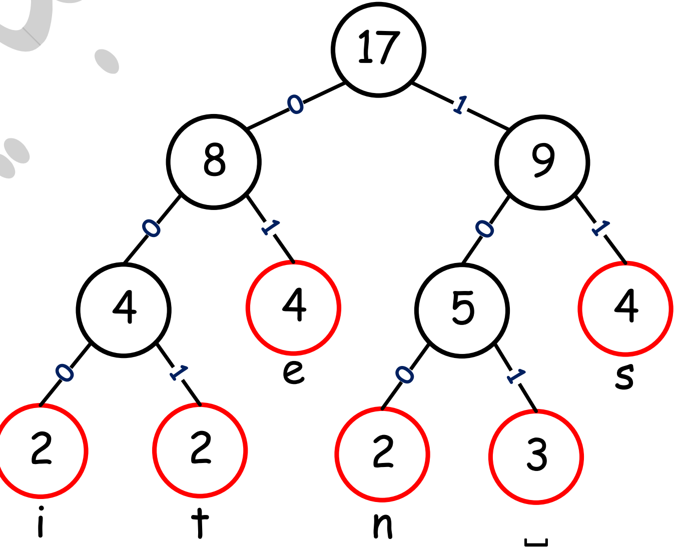
میتوان دید در اینجا میتوان فشردگی $0.31 = \frac{43}{136} = \frac{2*3+4*2+4*2+2*3+3*3}{17*8}$ در اینجا رشته 0.31فضای اولیه را اشغال می کند!

با داشتن درخت یا جدول کد ها رشته فشرده ورمزی شده را به حالت اولیه درآورد.

کدهافمن رشته: 000101110101101001010010000011101110100111

تذکر: بدون درخت یا جدول نمیتوان به رشته اولیه پی برد!

اگر درخت یا جدول را داشته باشیم، می توان رشته را رمزگشایی کرد:



رشته اولیه: (i␣see␣tennis␣sets)

روش بدست آوردن رشته اولیه: از ریشه شروع به حرکت می کنیم تا به یک برگ برسیم. حرف روی آن برگ را به رشته خود میافزاییم هر حرکت به راست یا چپ یک ۱ یا ۰ معرفی میکند.

الگوریتم دایکسترا (Dijkstras Algorithm)

این الگوریتم حریصانه کوتاه ترین فاصله همه راس ها از یک راس دلخواه را می یابد.

هنگام اجرای این الگوریتم باید

- **مجموعه S**: مجموعه راس هایی که کوتاه ترین فاصله میان آنها و راس دلخواه یافت شده
- **آرایه d**: درایه های این آریه کوتاه ترین فاصله تا راس دلخواه را نشان می دهند. (**d[i] نشان می دهد کوتاه ترین فاصله میان راس دلخواه و i چقدر است**).

آرایه d کوتاه ترین فاصله هر راس تا راس دلخواه را بیان می کند. یک نسخه از الگوریتم آرایه p هم دارد که درایه های آن درخت کوتاه ترین مسیر را نشان می دهد

G: گراف, S: راس دلخواه

p[i] نشان می دهد در مسیر کوتاه ترین فاصله پیش از راس i کدام راس بوده است. برای نمونه اگر p[4] برابر ۲ باشد آنگاه پیش از راس ۴ام راس دوم در مسیر کوتاه ترین فاصله بوده است.

(راس هایی که فاصله آنها تا راس S را یافته ایم علامت گذاری می کنیم یا به S می افزییم. برای سادگی می توان از آرایه V استفاده کرد. اگر V[i] = 1 باشد، راس iام به S افزوده شده است یا راس iام علامت گذاری mark است).

شبه کد:

$Dijkstra(G,S)$:

  $for\ each\ vertex\ i\ in\ G$:

    $d[i] = \infty$

    $p[i] = -$

    $V[i] = 0$

  $d[S] = 0$

  $while\ there\ exists\ an\ unmarked\ vertex$:

    $let\ i\ ba\ an\ unmarked\ vertex\ suchthat\ d[i]\ is\ minimum$

    $V[i] = 1$

    $for\ each\ edge\ (i,j)\ in\ G$:

عزیز محمدی – دارابیجانیان

$if\ V[j]\ =\ 0\ and\ d[i]\ +\ M[i,j]\ <\ d[j]$:

$d[j] = d[i] + M[i,j]$

$p[i] = j$

توضیحات کد بالا:

1)تابع Dijkstra با دو ورودی گراف G و راس مبدا S تعریف می‌شود.

2) در یک حلقهfor ، برای تمام رئوس i در گراف:G

- **فاصله i تا S را با d[i] به صورت بی‌نهایت (infinite) مقداردهی اولیه می‌کنیم** .دلیل این کار این است که در ابتدا هیچ مسیری از S به i شناخته شده نیست.
- **p[i]را که نشان‌دهنده راس قبلی i در کوتاه‌ترین مسیر تا S است، با null مقداردهی اولیه می‌کنیم** .در ابتدا هیچ راس قبلی برای i در کوتاه‌ترین مسیر تا S وجود ندارد.
- **وضعیت بررسی i را با V[i] به 0 (بررسی نشده) تنظیم می‌کنیم** . این نشان می‌دهد که i هنوز در الگوریتم دایکسترا بررسی نشده است.

3) فاصله S تا خودش را با d[S] به 0 تنظیم می‌کنیم .این بدیهی است، زیرا هر راسی به طور مستقیم فاصله ۰ تا خودش دارد.

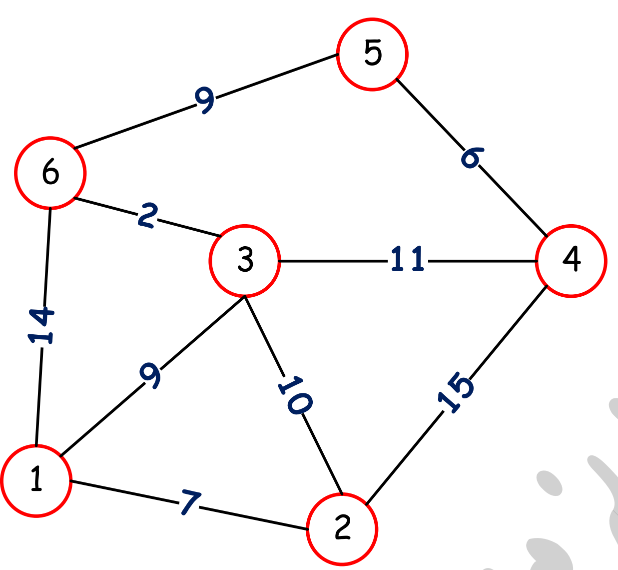
4) در یک حلقه:while

- **راس i را با کمترین d[i] از بین رئوس بررسی نشده پیدا می‌کنیم** .این راس i رئوس با کوتاه‌ترین مسیر شناخته شده تا آن لحظه از S خواهد بود.
- **وضعیت بررسی i را با V[i] به ۱ (بررسی شده) تغییر می‌دهیم** .این نشان می‌دهد که i در الگوریتم دایکسترا بررسی شده است.

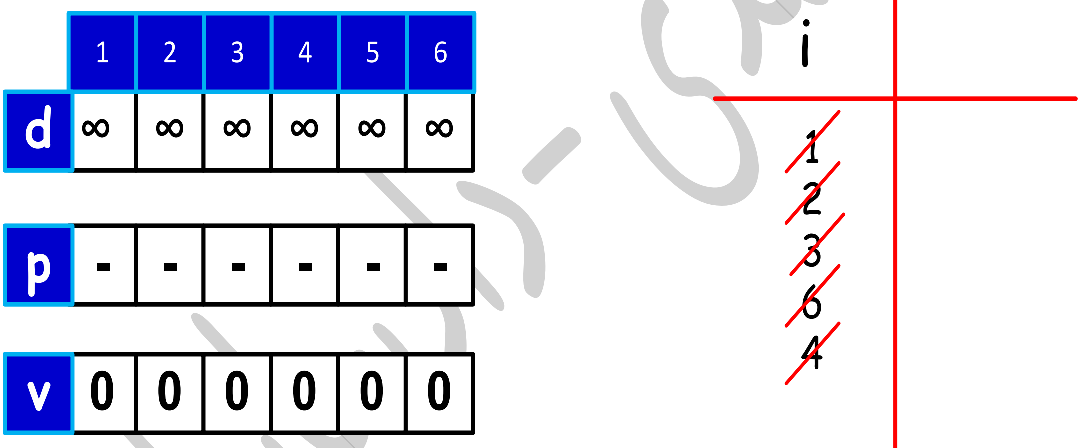
5) برای هر یال (i, j) در گراف:G

- **اگر وضعیت بررسی j با V[j] برابر 0 (بررسی نشده) باشد و d[i] + M[i, j] < d[j]:**
  - **فاصله j تا S را با d[j] به d[i] + M[i, j] به‌روزرسانی می‌کنیم** .M[i, j] وزن یال (i, j) است. این به معنای یافتن مسیری کوتاه‌تر از S به j از طریق i است.

برای مثال گراف زیر را در نظر بگیرید...



الگوریتم برای یافتن کوتاه ترین فاصله همه راس ها از راس (1) اجرا میکنیم.

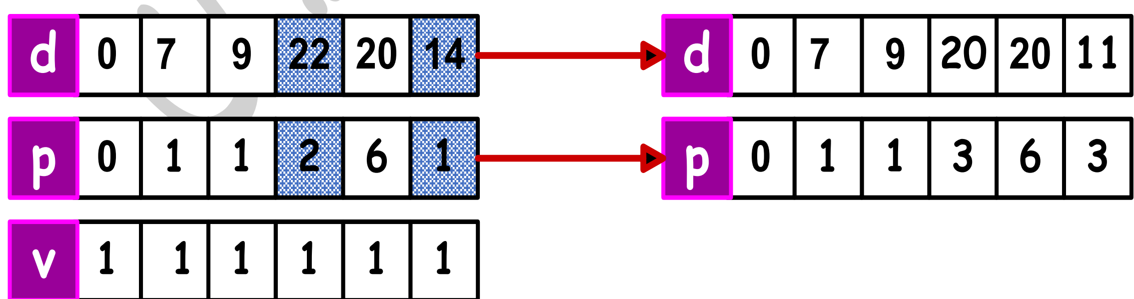


	1	2	3	4	5	6
d	∞	∞	∞	∞	∞	∞

p	-	-	-	-	-	-

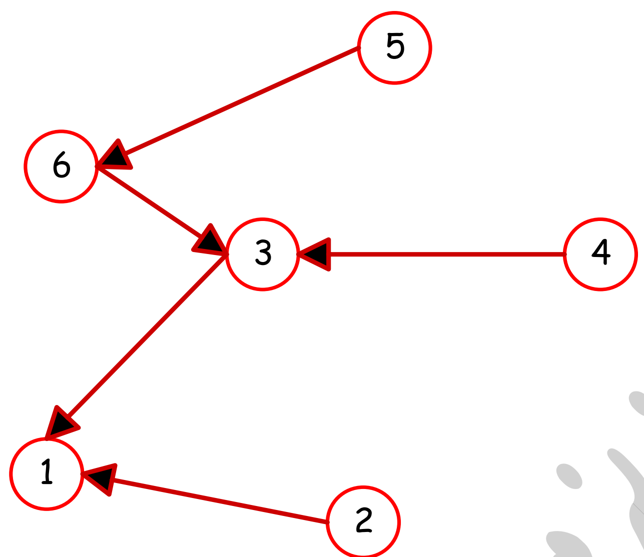
v	0	0	0	0	0	0

i
~~1~~
~~2~~
~~3~~
~~6~~
~~4~~



d	0	7	9	22	20	14
p	0	1	1	2	6	1
v	1	1	1	1	1	1

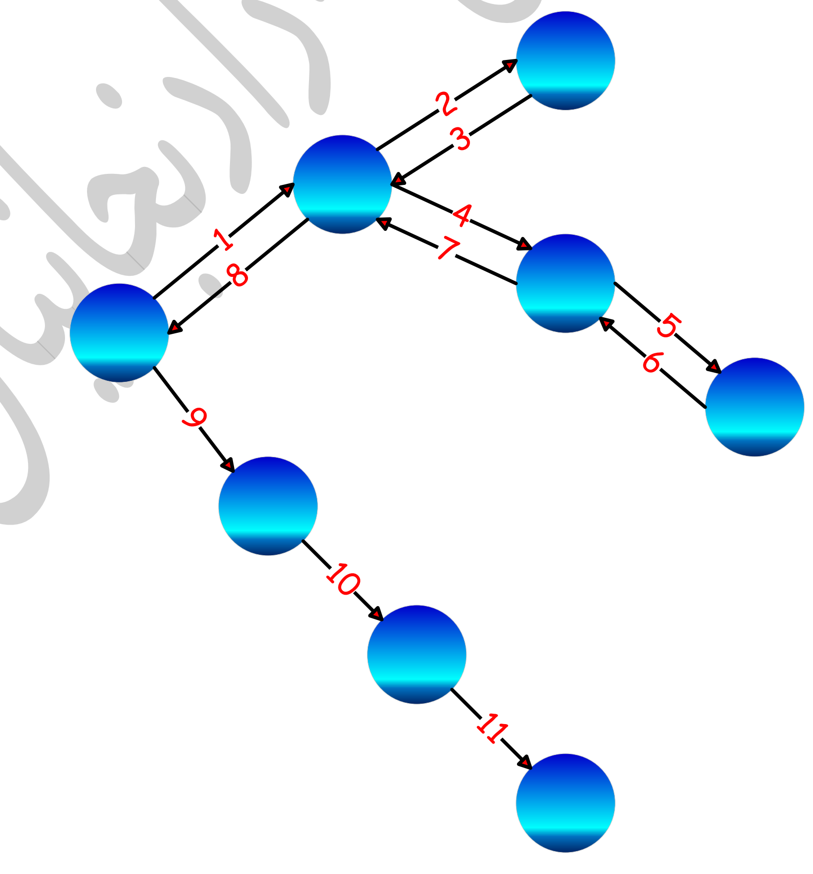
d	0	7	9	20	20	11
p	0	1	1	3	6	3



درخت کوتاه ترین فاصله ها که از روی آرایه p بدست آمده است.برای نمونه p[6] برابر3 است.پس پیش از راس 6ام باید راس 3 ام بوده باشیم تا کوتاه ترین مسیر را بپیماییم

نکته: اینکه گراف یال با وزن منفی داشته باشد آنگاه این الگوریتم درست کار نمی کند.

بازگشت به عقب (Back Tracking)



عزیز محمدی - دارابخانیان

در این رویکرد با رسیدن به بن بست، راه دیگری را طی می کنیم.

برای نمونه اگر بخواهیم از یک ماز(maze) با یک مسیر پر پیچ و خم بیرون برویم، باید از این رویکرد کمک گرفت.

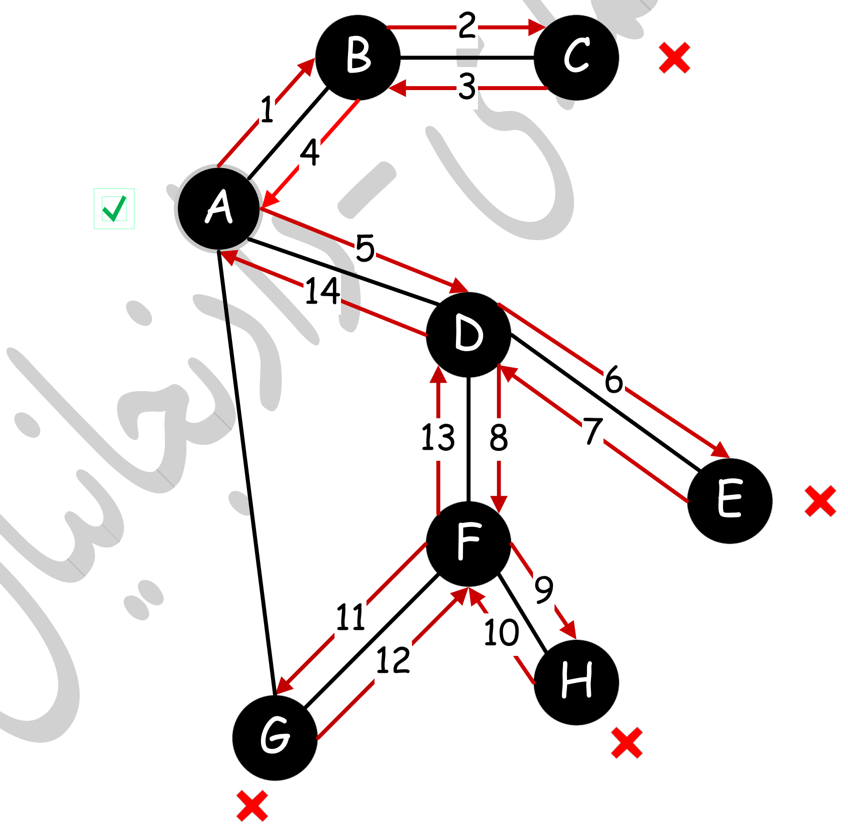
دو روش کلی برای جست و جوی گراف ها قابل تصور است:

۱. جست و جوی اول عمق (Depth Frirst Search)

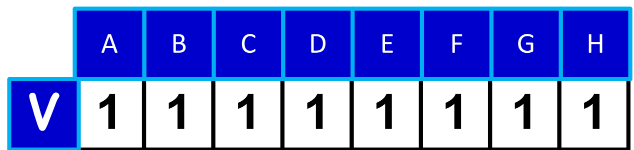
در این روش جست و جوی گراف ها از رویکرد بازگشت به عقب کمک گرفته شده است.

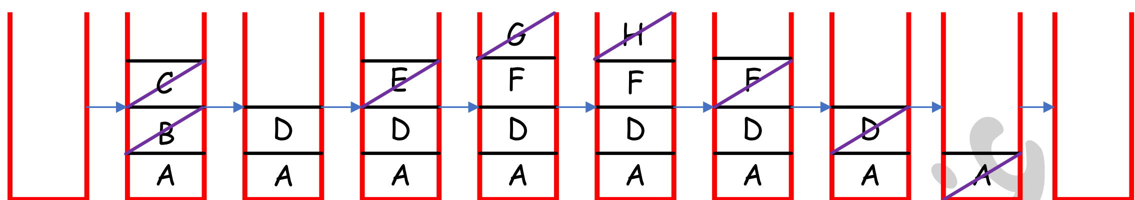
در DFS از یک راس دلخواه شروع کرده و مسیری را طی می کنیم. با رسیدن به بن بست (**یعنی راس برای ادامه دادن وجود نداشته باشد یا همسایه ملاقات**(visit) **نشده نداشته باشیم**) به سمت عقب باز می گردیم.

اعمال DFS با شروع از راس A



زهرا محمدی - آذربایجان





برای انجام DFS یا هر الگریتم دیگری که بر پایه رویکرد بازگشت به عقب باشد به یک پشته (stack) برای نگهداری مسیر طی شده نیاز داریم.

می توان الگوریتم را به صورت بازگشتی نوشت و مدیریت پشته را بر عهده سیستم عامل گذاشت.

الگوریتم DFS

- **راس دلخواه v را وارد پشته کن و آن را برچسب بزن.**
- **تا زمانی که پشته خالی نیست:**
  - **راس v را برابر راس برداشته شده از بالای پشته قرار بده.**
  - **برای همه راس‌های w که از راس دلخواه v به آن‌ها یال هست و برچسب نخورده‌اند:**
    - **راس w را وارد پشته کن و آن را برچسب بزن.**

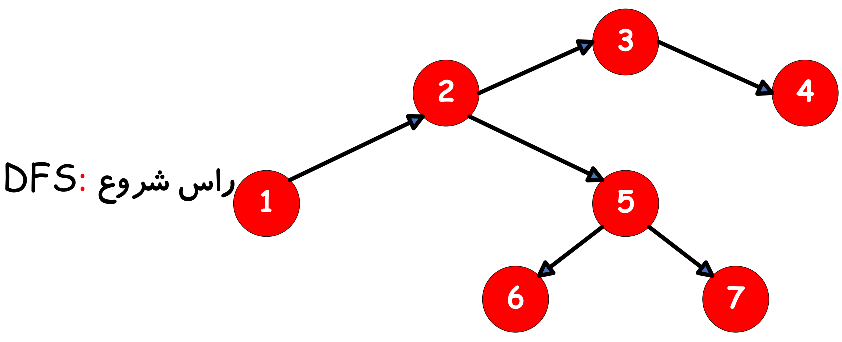
هر الگورتیم بر پایه رویکرد بازگشت به عقب را می توان با یک پشته یا به صورت بازگشتی نوشت.

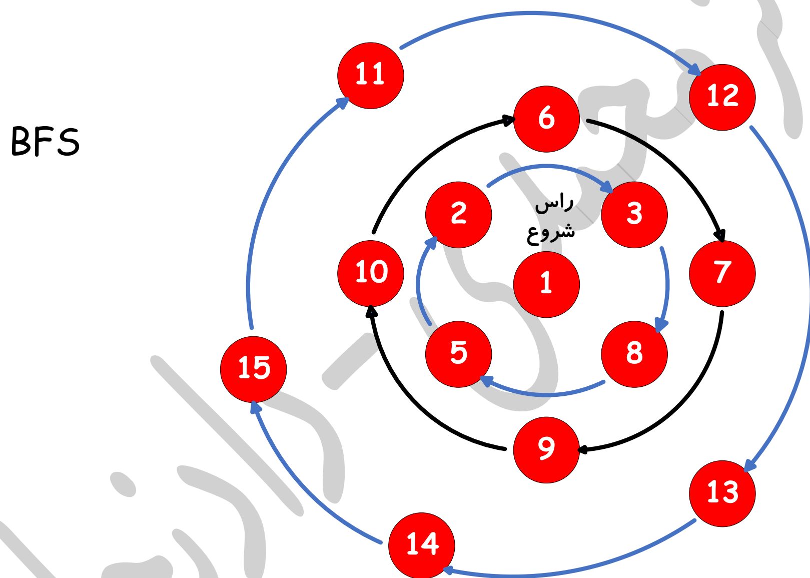
برای نمونه الگورتیم DFS را میتوان به شکل بازگشتی هم نوشت (**در حالت کلی استفاده از پشته کاراتر از یک الگورتیم بازگشتی است**).

۲. جست و جوی اول سطح(Breath Frirst Search)

الگوریتم BFS

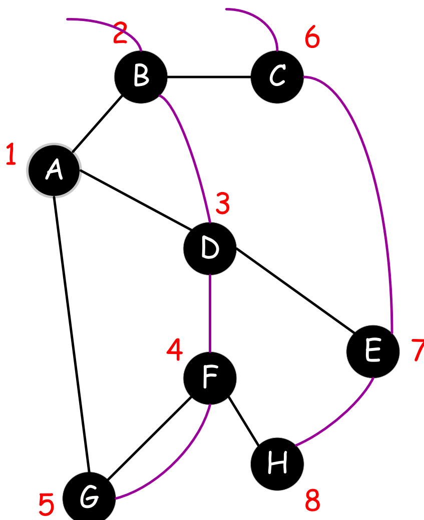
در این جست و جو ابتدا همه همسایه های یک راس ملاقات شده و سپس نوبت به همسایه ها می رسد.





اعمال BFS با شروع از راس A

عزیزمحمدی – داریخانیان



- **راس دلخواه را درون یک صف خالی قرار می دهیم.**
- **تا زمانی که صف خالی نشده است:**
  - **یک را س را از صف برداشته،**
  - **اگر ملاقات نشده است آن را ملاقات کرده و همسایه های (ملاقات نشده اش) را به صف می افزاییم.**

عزیز محمدی - ادریجانیان

فصل ۵. پیچیدگی محاسباتی (Computational Complexity)

هدف از این فصل دسته بندی مسائل بر اساس دشواری ذاتی آنها می باشد. برخی از مسائل ذاتا دشوار تر از دیگر مسائل هستند. برای نمونه محاسبه $a * b$ ذاتا دشوار تر از $a + b$ است.

پیش از دسته بندی مسائل بر اساس دشواری یا پیچیدگی که دارند باید بهتر با "اندازه ورودی*(input size)* " آشنا شد.

اندازه ورودی (input size)

فضای مورد نیاز برای نوشتن ورودی در مبنای به جز 1 را اندازه ورودی می گوییم.

برای نمونه اگر ورودی یک مسئله را در مبنای 10 بنویسیم، آنگاه اندازه ورودی همان تعداد رقم ها می باشد. اگر تابعی روی یک رشته کاری را انجام دهد، آنگاه اندازه ورودی I نیست؛ بلکه اندازه ورودی تعداد حروف آن رشته می باشد.

برای مثال تابع زیر برای تشخیص اول بودن یا نبودن نوشته شده است.

```
Prime(m)
    for i=2 to m-1
        if m % i = 0
            return 0
    return 1
```

در اینجا اندازه ورودی فضای مورد نیاز برای نوشتن m در یک مبنای دلخواه غیر 1 می باشد (**به اشتباه ممکن است تصور شود که در اینجا اندازه ورودی یک است؛ زیرا یک عدد به تابع داده می شود**).

برای نمونه اگر از مبنای 10 استفاده کنیم، آنگاه هنگام فراخوانی $i = Prime(957)$ اندازه ورودی 3 می باشد؛ زیرا ورودی را با همه رقم نشان داده ایم.

به سادگی می توان دریافت که تعداد محاسبه های انجام شده توسط این تابع (**در بدترین حالت**) بسیار بیشتر از اندازه ورودی است.

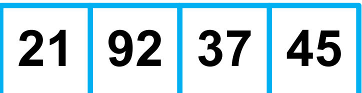
اگر همه محاسبه ها در همان مبنای 10 باشد، آنگاه یک عدد n رقمی به این تابع بدهیم(**در بدترین حالت**) پیچیدگی زمانی از مرتبه $O(10^n)$ خواهد بود!

برای مثال اگر $Prime(9043257)$ را فرا بخوانیم آنگاه (در بدترین حالت) تعداد محاسبه ها کم و بیش 10^7 خواهد بود. اندازه ورودی در اینجا *7* است؛ زیرا ورودی (***در مبنای 10***) هفت رقم دارد. پس تابع از $O(10^n)$ می باشد. اگر محاسبه ها در مبنای *10* باشد.

حال اگر همه محاسبه ها در مبنای 2 باشد، باز هم این تابع نمایی خواهد بود که از مرتبه $O(2^n)$ می شود که در آن n اندازه ورودی در مبنای 2 است.

اگر ورودی یک الگوریتم چندین عدد یا یک آرایه باشد، آنگاه به طور معمول اندازه ورودی را برابر با تعداد آن عدد ها یا طول آرایه در نظر می گیریم (**مقدار دقیق اندازه ورودی باز هم در اینجا برابر است با فضای مورد نیاز برای نوشتن ورودی. یعنی تعداد عدد ها ضرب در فضای مورد نیاز برای نوشتن هر عدد که از فضای مورد نیاز برای نوشتن هر عدد صرف نظر می شود؛ زیرا یک ضریب است و هنگام محاسبه مرتبه زمانی از ضرائب چشم پوشی می کنیم**).

برای مثال اگر ورودی تابع آرایه



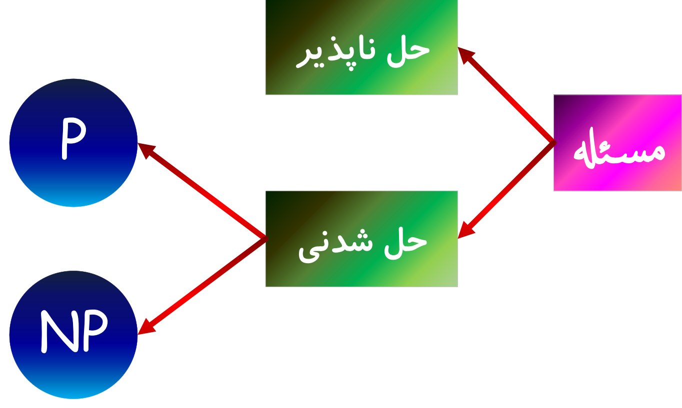
باشد، آنگاه اندازه ورودی را 4 در نظر می گیریم. زیرا ورودی 4 عدد دارد (**اما اندازه واقعی 2*4 است چرا که 4 عدد 2 رقمی داریم اما از 2 چشم پوشی کرده و اندازه ورودی را 4 در نظر می گیریم. این چشم پوشی کردن باز هم در مرتبه تاثیری ندارد**)!

به صورت خلاصه

- **اگر الگوریتم یک ورودی داشته باشد، اندازه مورد نیاز برای نوشتن آن در یک مبنای دلخواه بزرگتر از 1 می باشد.**
- **اگر الگوریتم چندین ورودی داشته باشد، باز هم اندازه ورودی درست و واقعی فصای مورد نیاز برای نوشتن همه آنها است. اما برای سادگی تعداد ورودی ها را به عنوان اندازه ورودی در نظر می گیریم و از اندازه تک تک ورودی ها صرف نظر می کنیم.**

برای نمونه انداز ورودی الگوریتم مرتب سازی تعداد ورودی ها است. همچنین این مقدار برای تشخیص زوج یا فرد بودن فضای مورد نیاز برای نوشتن عدد است و نه یک!

در ادامه مسائل را بر اساس پیچیدگی محاسباتی که دارند بخش بندی می کنیم...



برخی از **مسائل نا شدنی** هستند! برای نمونه نشان داده شده است که مسئله توقف (Halt) حل ناپذیر است.
یعنی هیچگاه نمی توان برای آن یک الگوریتم نوشت!

مسئله توقف (Halt)

در این مسئله هیچگاه نمی توان الگوریتمی ساخت که نشان دهد یک الگوریتم مانند A به ازای یک ورودی مانند d متوقف می شود یا نه!

الگوریتم H مشخص می کند که الگوریتم A به ازای ورودی d متوقف می شود یا نه. به عبارتی دیگر

$H(A,d) \rightarrow$ بله یا خیر

- اگر $H(A,d)$ را اجرا کنیم آنگاه پاسخ بله است.
- اگر *A* به ازای *d* متوقف شود آنگاه خیر است.
- اگر *A* به ازای *d* متوقف نشود، مسئله توقف بیان می کند که هیچگاه نمی توان الگوریتم *H* را ساخت.

می توان مشخص کرد که یک الگوریتم خاص به ازای یک ورودی خاص متوقف می شود یا نه.

2	4	1	3

برای نمونه اگر *A* الگوریتم مرتب سازی ادغامی باشد و *d* هم یک آرایه

باشد آنگاه می توان گفت الگوریتم متوقف می شود؛ اما نمی توان به ازای هر الگوریتم و هر ورودی گفت که آن الگوریتم متوقف می شود یا نه!

مسائل حل شدنی را می توان به دو رده کلی تقسیم کرد:

۱. رده *P*
۲. رده *NP*

رده P

P حرف ابتدایی Plynomial به معنی چند جمله ای می باشد.

یک مسئله در رده یاد شده P قرار می گیرد اگر پیچیدگی زمانی آن یک چند جمله ای باشد.

برای نمونه مسئله "مرتب سازی" به رده P تعلق دارد. زیرا الگوریتم "MergeSort" را برای حل آن داریم. این الگوریتم از $O(nlogn)$ می باشد. یعنی مرتبه آن یک چند جمله ای است.

الگوریتم زیر برای مسئله "یافتن یک عدد" نوشته شده است:

```
Find(A, n, k)
    for i=1 to n
        if A[i] = k
            return i
    return -1
```

این الگوریتم اندیس عدد k را باز می گرداند. اگر k درون آرایه n عنصری A باشد، الگوریتم 1- را باز می گرداند.

اگر مرتبه های زمانی الگوریتمی به یکی از صورت های $1, n, n^2, n^4, n^6, n^{10}, n^{1000}, logn, nlogn, n^2logn, \ldots$ باشد به رده *P* تعلق دارد.

اگر مرتبه های زمانی الگوریتمی به یکی از صورت های $2^n, 3^n, 10^n, k^n, n!,\ n^n, n^{n^n}, \ldots$ باشد دیگر به رده *P* تعلق ندارد.

مثلا در مسئله کوله پشتی *0-1* باید ترکیبی از *n* کار را بیابیم که بیشترین ارزش را داشته باشد اما از ظرفیت کوله پشتی بیشتر نشود. برای یافتن پاسخ باید 2^n ترکیب را بررسی کنیم. پس الگوریتم دست کم باید 2^n محاسبه انجام دهد و به همین ترتیب از مرتبه $O(2^n)$ این خواهد بود. به همین دلیل "مسئله کوله پشتی *0-1*" به رده *P* تعلق ندارد چرا که 2^n یک چند جمله ای نیست.

رده NP

NP حرف ابتدایی Non deterministic Plynomial به معنی چند جمله ای غیر قطعی است.

یک مسئله به رده NP تعلق دارد اگر بتوان درستی پاسخ ارائه شده برای آن را در زمان چند جمله ای بررسی کرد.

برای نمونه مسئله "مرتب سازی" به NP تعلق دارد.

اگر پاسخی برای این مسئله ارائه شود آنگاه در زمان چند جمله ای می توانیم درستی آن پاسخ را بررسی کنیم.

برای نمونه اگر این آرایه با اندازه *n = 4* را داشته باشیم

3	9	1	6

و شخص ادعا کند که پاسخ مسئله به ازای آرایه نوشته شده

1	3	6	9

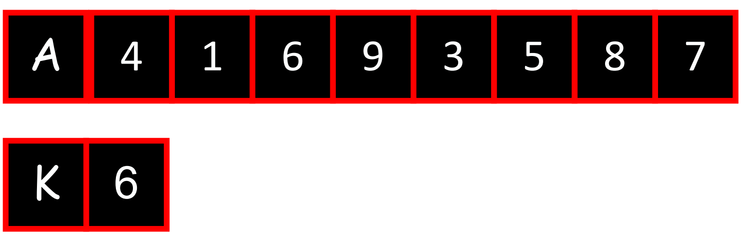
می باشد، آنگاه در زمان چند جمله ای نسبت به اندازه ورودی که 4 باشد، می توانیم درستی آن ادعا را بررسی کنیم. کافی است از روی آرایه گذر کنیم، اگر درایه ای کوچک تر از قبلی بود، ادعا آن شخص نادرست است.

مسئله "یافتن یک عدد" هم به NP تعلق دارد.

اگر شخصی ادعا کند که پاسخ را یافته و اندیس عدد مطلوب را بدهد، آنگاه در $O(1)$ می توانیم درستی ادعای آن شخص را بررسی کنیم.

ایرج محمدی دارابی

برای نمونه اگر



و شخص ادعا کند که پاسخ 5 می باشد، آنگاه به راحتی و در زمان $O(1)$ می توان دریافت که ادعای آن شخص نادرست است.

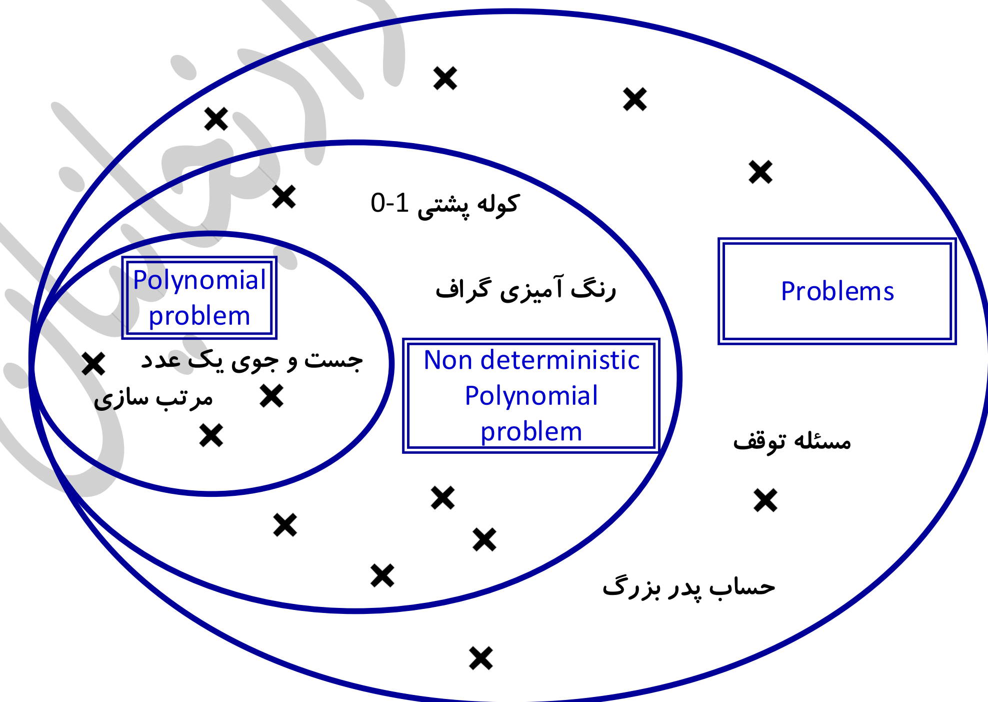
نکته: بسیاری از مسائل مطرح به NP تعلق دارند.

ثابت کردیم مسئله "مرتب سازی" هم به P تعلق دارد چرا که:

***به P تعلق دارد زیرا الگوریتم MergeSort را برای آن نوشته ایم که از مرتبه $O(nlogn)$ پاسخ را می یابد.**

***به NP تعلق دارد؛ زیرا کمی پیشتر ثابت کردیم اگر شخصی ادعا کند پاسخ را یافته در زمان $O(n)$ می توانیم ادعای آن را بررسی کنیم.**

در حالت کلی هر مسئله به P تعلق داشته باشد به NP هم تعلق دارد.



به نظر می آید مسائل درون P ذاتا ساده تر از مسائل بیرون از آن می باشد.