** به نام آنچه باید آنچه دیگر دیگر نوشت **

جلسه اول 16 . 12 . 86

۱- مقدمه‌ای بر پیچیدگی الگوریتم‌ها
۲- الگوریتم‌های تقسیم و حل
۳- الگوریتم‌های بازگشت به عقب
۴- الگوریتم‌های حریصانه
۵- الگوریتم‌های به روش برنامه‌ریزی پویا
۶- مقدمه‌ای بر الگوریتم‌های انشعاب و محدودسازی Branch & Bound

Big O : تعداد کارهایی که در یک الگوریتم انجام می‌شود را به کمک توابعی می‌شماریم و برای بررسی پیچیدگی الگوریتم‌ها از Big O ($O(n)$) که تابعی از n است استفاده می‌شود. (گاهی با Ω و Θ هم نمایش داده می‌شود)

مثال: پیچیدگی الگوریتم‌های زیر را بیابید:

```
{ sum := 0 ;                    مقدار sum را صفر قرار می‌دهد ← ① کار انجام می‌شود
  readln (n) ;                  مقدار n را از ورودی می‌خواند ← ① کار انجام می‌شود
  for i:=1 to n do              { دفعه اول: i ← 1  i≤n? مقایسه → ② ×1
                                  دفعه بعدی: i ← i+1  i≤n? " → ③ ×n
    sum := sum + i ;            ② ×n   این سه عمل را n بار انجام می‌دهد
                                  حاصل sum و sum+i جمع
  writeln (sum) ; ①
}
```

پیچیدگی : $1+1+2+3n+2n+1 = \boxed{5+5n}$

* چون تابع حاصل یک تابع خطی است بنابراین کافی است پیچیدگی الگوریتم را یک تابع خطی فرض کنیم زیرا رشد تابع برای n مطرح است بنابراین داریم

$O(f(n)) = O(n)$

Subject:
Year: Month: Date:

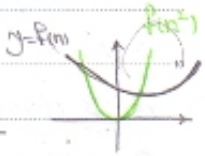
```
n:=1;                                  مقدار یک را در n قرار می‌دهد ← ① کار
while n<>0 do                          n را با صفر مقایسه می‌کند (n<>0 ?) ← ① کار
{ n:=n*2 mod 38;                       فرض می‌کنیم این 3 کار m بار انجام می‌شود  ③ × m →
  for i:=1 to n do
  sum:= sum+i;                         ② × 1 + ③ × n + ② × n = 2 + 5n
}                                      دنبالدل  دنبالدل  سطر دوم
```

* با توجه به اینکه حلقه for داخل حلقه while قرار دارد کارهای داخل for هم به تعداد m بار (فرض) انجام می‌شود یعنی $(2+5n)\times m$

پیچیدگی : $1+1+3m+(2+5n)m$

$= 2+3m+2m+5nm = \underline{2+5m+5nm}$

درجه 0 درجه 1 درجه 2



باز هم باید رشد تابع را در نظر بگیریم ← $O(5mn+5m+2) = O(n^2)$

نکته: پیچیدگی دو حلقه که بهم وابسته نیستند برابر است با پیچیدگی max دو حلقه یعنی چنانچه پیچیدگی یک حلقه n و دیگری n^2 باشد ← پیچیدگی کل الگوریتم n^2 خواهد بود

مثال: پیچیدگی bulde sort ؟

```
a) for pass:=1 to |A|-1 do        (|A| : تعداد اعضای A)
b)   for i:=1 to |A|-pass do
c)     if A[i] > A[i+1] then  ④
d)       Replace (i, i+1);    ③ (کمترین مقدار)
```

$\boxed{4+C}$

با توجه به مثال ۱: $2+3(x-1)$

فرض می‌کنیم تعداد اعضای A برابر x است یعنی $\boxed{|A| = x}$

* از آنجا که در این الگوریتم دو حلقه بهم وابسته اند نمی‌توانیم بگوییم چون دو حلقه داریم پیچیدگی n^2 است. بنابراین باید پیچیدگی را حساب کنیم.

سطر a: دفعه اول for ، ② کار و دفعات بعدی ③×n کار انجام می‌دهد که $|A|-1 = x-1$ طبق فرض

PAPCO

سطر b: بار اول pass=1 ← x-1 بار اجرا می‌شود. دفعه دوم pass=2 ← x-2 بار

$$\Rightarrow \underbrace{3(x-1)}_{\text{دفعه اول}} + \underbrace{3(x-2)}_{\text{دفعه دوم}} + 3(x-3) + \dots + 3(1)$$

سطر c: در این سطر ④ کار انجام می‌شود که عبارتند از:

1. پیدا کردن $A[i]$ 2. اضافه کردن یک واحد به i: $(i+1)$

3. پیدا کردن $A[i+1]$ 4. مقایسه $A[i] \overset{?}{>} A[i+1]$

سطر d: با در نظر گرفتن بهترین حالت در این سطر الگوریتم ③ کار انجام می‌شود:

1. پیدا کردن i 2. پیدا کردن $i+1$ 3. replace کردن

اما از آنجایی که ممکن است برای replace کردن بیش از یک کار انجام می‌شود ⇐ ⓒ فرض می‌کنیم

* تعداد دفعاتی که دو سطر آخر انجام می‌شوند:

$$\underbrace{(4+C)(x-1)}_{\text{دفعه اول}} + \underbrace{(4+C)(x-2)}_{\text{دفعه دوم}} + (4+C)(x-3) + \dots$$

* تعداد کل دفعاتی که Ⓔ انجام می‌شود:

$$Ⓔ: (3+4+C)\sum_{i=1}^{x}(x-i)$$

⇐ با بنابراین تعداد کل دفعاتی که الگوریتم انجام می‌شود عبارتست از:

$$f: 2+3(x-1)+(3+4+C)\underbrace{\sum_{i=1}^{x}(x-i)}_{*}$$

$$* \sum_{i=1}^{x}(x-i) \equiv \sum_{i=1}^{x-1} i = \frac{(x-1)(x-1+1)}{2} = \frac{x(x-1)}{2}$$

$$\Rightarrow = \underbrace{2}_{\text{درجه ۰}} + \underbrace{3(x-1)}_{\text{درجه (1)}} + \underbrace{\frac{(7+C)x(x-1)}{2}}_{\text{درجه (2)}} = f(x)$$

$$\boxed{O(f(x)) = O(n^2)}$$

در اینجا به طور اتفاقی پیچیدگی الگوریتم همان n^2 شده است.
یعنی با وجود وابستگی دو حلقه باز هم $O(n^2)$ داریم.

Subject:
Year. Month. Date. ( )

مثال: در اینجا هم دو حلقه بهم وابسته اند.

```
n := 1
for i:=1 to n do
  for j:=1 to [log_10 i] + 1 do
```

$\rightarrow n$

$\rightarrow \log_{10} n$

i	1	10	100	1000	10^4	10^{10}	10^{20}
j	$0+1$	$1+1$	$2+1$	$3+1$	$4+1$	$10+1$	$20+1$
	1	2	3	4	5	11	21

حلقه بیرونی n کار انجام می دهد و حلقه داخلی را به اندازه کاربرش اجرا می کند بنابراین داریم:

$\log_{10} n * n \rightarrow O(f(x)) = O(n \log n)$

نکته: همیشه بدترین حالت را برای $O(f(x))$ در نظر می گیریم.

مثال: فرض می کنیم داخل حلقه c کار انجام می شود.

```
i := 1
while i<n do
{ ≡
  i := i*2 ;
}
```

2^0 (مقدار i)	$i \rightarrow 1 \rightarrow c$ کار
2^1	$i \rightarrow 2 \rightarrow c$
2^2	$i \rightarrow 4 \rightarrow c$
2^3	$i \rightarrow 8 \rightarrow c$
2^4	$i \rightarrow 16 \rightarrow c$
⋮	⋮
	$i \rightarrow 2^m \rightarrow c$
	$i \rightarrow 2^{m+1} \rightarrow X$ وارد حلقه نمی شود

$m=0$ وقتی $\rightarrow$ (1) یک c انجام می شود

$m=1$ " $\rightarrow$ " " c دو (2)

$m=2$ " $\rightarrow$ " " c سه (3)

⋮

$m=m$ " $\rightarrow (m+1)c$ *

$$\begin{cases} 2^m < n \\ 2^{m+1} \geq n \end{cases} \rightarrow 2^m < n \leq 2^{m+1} \rightarrow m < \log_2 n \leq m+1$$

$$\boxed{m = \log_2 n}$$

$* \rightarrow c(\log_2 n + 1) \Rightarrow O(f(x)) = O(\log n)$

حلقه ها به هم وابسته اند $O(n) \neq n^2$

PAPCO

مثال: پیچیدگی الگوریتم تولید سری فیبوناچی؟ (الگوریتمی که عضو nام سری فیبوناچی را برگرداند)

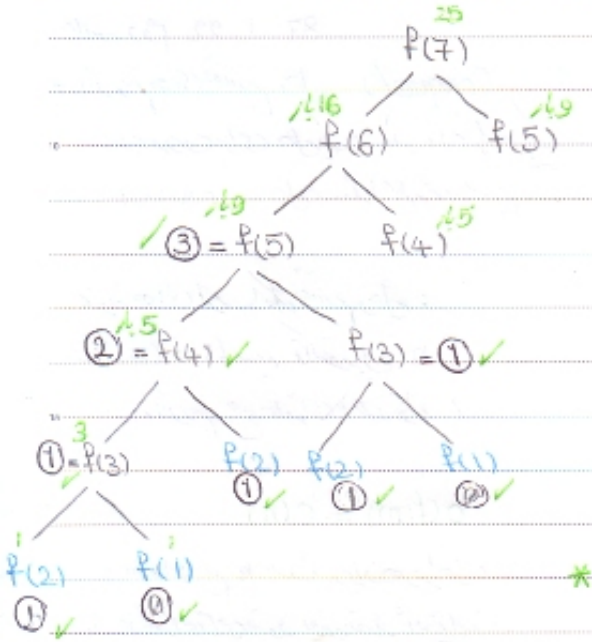
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

مثلاً n=5 عضو پنجم سری = 3

یعنی: 3 = 2 + 1

n=5 n=4 n=3

```
fibo (n)
{ if n=1   return 0 ;
  if n=2   return 1 ;
  return (fibo(n-1) + fibo(n-2))
```



چطور می‌بینیم برای f(5) تابع 9 بار اجرا می‌شود

f(1) ← 1 بار

f(4) ← 5 بار

f(6) ← 16 بار

⋮

* می‌توانیم فرض کنیم تعداد اجرای هر f(n) برابر است با: تعداد اجرای عضو دست چپی * 2

یعنی: $f(n) \approx 2 * f(n-1)$

$* \rightarrow T(n) = 2^k T(n-k)$

زمان صرف شده $T(n) \approx 2T(n-1)$

$= 2(2T(n-2)) = 4T(n-2)$

$= 2(2(2T(n-3))) = 8T(n-3)$

$= 16T(n-4) \ldots \rightarrow *$

اگر n بار انجام شود ← n-k = 0 می‌شود

$O(n) = 2^n$

اما دقیق‌تر آن است که n-1 را در نظر بگیریم

$2^{n-1} T(n-(n-1)) = 2^{n-1}$

Subject:
Year. Month. Date.

n	$\log n$	n	$n\log n$	n^2	n^3	2^n
1	0	1	0	1	1	2
10	1	10	10	10^2	10^3	$10^3 \approx 2^{10}$
10^2	2	10^2	2×10^2	10^4	10^6	$10^{30} \approx 2^{100}$
10^3	3	10^3	3×10^3	10^6	10^9	$10^{300} \approx 2^{1000}$
10^6	6	10^6	6×10^6	10^{12}	10^{18}	$10^{300,000}$

جلسه دوم 87.1.22

* الگوریتم های تقسیم و حل (Divide And Conquer):

" در این روش برای حل یک مسئله آن را به یک یا چند مسئله ساده تر (کوچکتر) تقسیم کرده و با حل مسائل ساده تر به حل مسئله اصلی می رسیم.

نمونه هایی از مسائل تقسیم و حل:

مثال (1): محاسبه فاکتوریل n ؟

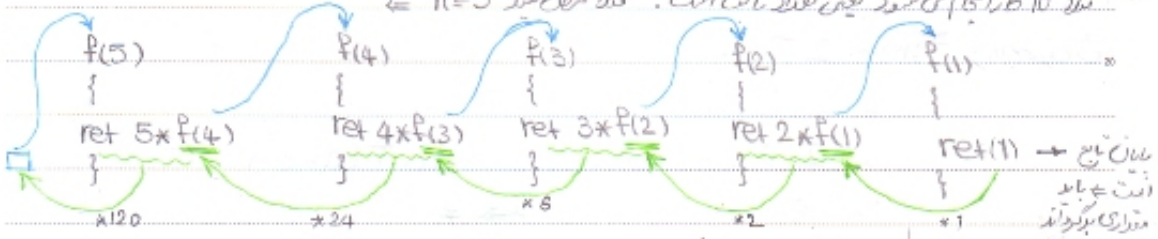
" (در این الگوریتم هیچ حلقه ای وجود ندارد)

```
fact(n)
{
  if n=1 then return 1 ;
  return n * fact(n-1) ;
}
```

عملاً به معنی پایان تابع است

$O(f(n)) = O(n)$

پیچیدگی این الگوریتم n است زیرا در هر فراخوانی مثلاً 10 کار انجام می شود یعنی تعداد ثابت است. مثلاً فرض کنید n=5 ⇐



* هر مقدار باید به فراخواننده خودش برگردانده شود

نکته: هرگاه تابعی فراخوانده شود آن تابع انجام شده و جواب آن برمی گردد (قانون فراخوانی توابع)

PAPCO

الگوریتم Fact یک الگوریتم تقسیم و حل است زیرا مثلاً در محاسبه $f(5)$ آن را به $f(4), \ldots, f(1)$ شکستیم. در این مثال مسئله بزرگ ما به یک مسئله کوچکتر تقسیم شده است. یعنی $f(5)$ به $f(4)$ شکسته شده و برای محاسبه $f(4)$ احتیاج است که دوباره شکسته شود اما در حالت کلی $f(5)$ فقط به یک مسئله کوچکتر که همان $f(4)$ است شکسته شده است.

- مثال (۲): الگوریتم مرتب کردن n عدد به صورت صعودی؟ (Mergesort)

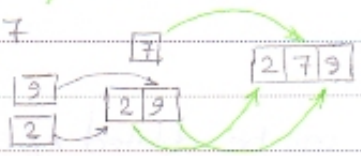
برای مرتب کردن اعداد از روش تقسیم و حل، وسط آرایه A[n] را پیدا کرده دو قسمت آرایه را sort کردن و در آخر این دو قسمت را با هم Merg (ترکیب) می‌کنیم.

مثلاً:

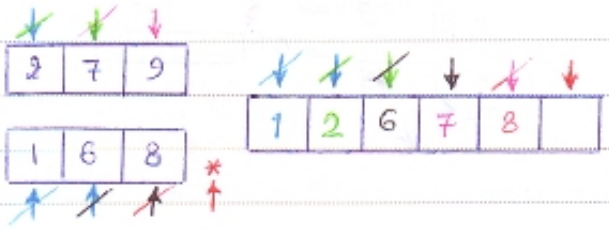
9 2 7 { 1 6 8

" 9 2 { 7

9 { 2



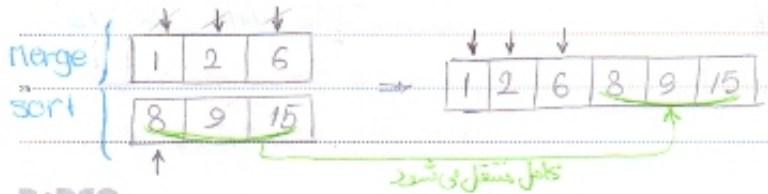
کل Mergsort را برای نیمه راست اعداد می‌بینیم به همین صورت نیمه چپ هم sort شده و در نهایت به صورت زیر با هم ادغام می‌شوند



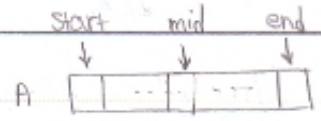
Merge: سه اشاره‌گر در نظر می‌گیریم اعداد دو اشاره‌گر آرایه A را با هم مقایسه کرده عدد کوچکتر به آرایه B منتقل می‌کنیم و یک اشاره‌گر آرایه B و آرایه‌ای که انتقال از آن صورت گرفته یک واحد اضافه می‌کنیم تا جایی که در مرحله * دیگر نتوانیم مقایسه کنیم (چون اشاره‌گر آرایه دوم دیگر به جایی اشاره نمی‌کند و از محدوده خارج شده) ⇐ هرگاه چنین اتفاقی رخ داد آرایه‌ای که هنوز اعداد آن به آرایه B منتقل نشده‌اند را به طور کلی به آرایه B منتقل می‌کنیم.

مثال عددی:

1 2 6 { 9 8 15



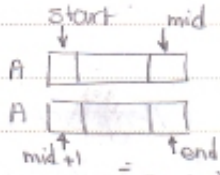
Subject:
Year. Month. Date. / /




```
Merge sort (A, start, end)
{
  if start = end then return(1);
  mid := [(start + end) / 2] ;
  Merge Sort (A, start, mid);
  Merge Sort (A, mid+1, end);
  Merge (A, start, mid, end)
}
```

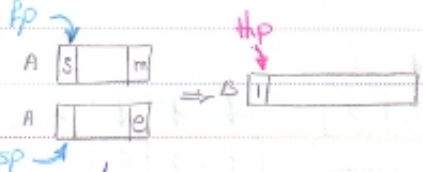
شرط پایان بازگشتی

یعنی آرایه تنها یک عضو دارد



ادغام دو قسمت Sort شده

```
Merge (A, S, m, e)
{
  if S = e then return; /* Can be deleted */
  fp := S ;          first pointer
  Sp := m+1 ;        scond pointer
  thp := 1 ;         (یک آرایه B تعریف می‌کنیم)
* while fp ≤ m and sp ≤ e do
    { if A[fp] < A[sp] then
        { B[thp] := A[fp] ;
          fp := fp + 1 ; }
      else { B[th] := A[sp] ;
             sp := sp + 1 ; }
      thp := thp + 1 ;
    } // end while
* while fp ≤ m do
    { B[thp] := A[fp] ;
      fp := fp + 1 ;
      thp := thp + 1 ; }
```



* این حلقه while: فاز اول Merge کردن را انجام می‌دهد یعنی با مقایسه دو قسمت آرایه A، Sort شده اعداد را به B منتقل می‌کند

* این حلقه while وقتی استفاده می‌شود که قسمت دوم آرایه A کاملاً منتقل شده و در قسمت اول آرایه A هنوز عدد یا اعداد باقی مانده

```
* while sp<e do
  { B[th] := A[sp] ;
    sp := sp+1 ;
    thp := thp+1 ; }
```

* این حلقه while درست برعکس حلقه قبلی است
یعنی مربوط به وقتی است که در قسمت دوم آرایه A عدد
یا اعداد باقی مانده

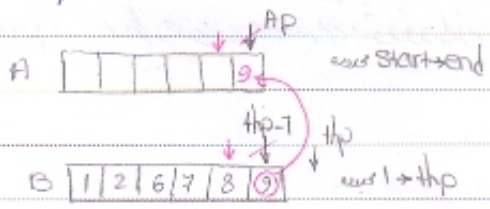
توجه: در merge (ادغام کردن) دو فاز داریم:

1. sort کردن آرایه اصلی (A) در دو قسمت ← (while اول)
(یعنی مقایسه دو قسمت تا حد امکان و انتقال به آرایه سوم (B))
2. انتقال اعداد باقیمانده آرایه A به آرایه B (while دوم و سوم)

ادامه الگوریتم

```
Ap := e ;
thp := thp-1;
while thp≥1 do
  { A[Ap] := B[thp]
    thp := thp-1;
    Ap := Ap-1 ;
  }
```

- در این بخش الگوریتم آرایه sort شده که
در B قرار دارد را به A برمی‌گردانیم.



مثال (۳): الگوریتم محاسبه max یک آرایه (روش تقسیم و حل)

آرایه را به دو قسمت تقسیم کرده، max
هر بخش را به دست می‌آوریم و در نهایت
max این دو قسمت را با هم مقایسه کرده
و max کل را به عنوان خروجی می‌دهیم

```
find_max (A, start, end)
{
  if start=end ther return A[start];
  mid := (start + end / 2 );
  max1 := find_max (A, start, mid);
  max2 := find_max (A, mid+1, end);
  if max1 > max2 then return max1;
  else return max2 ;
}
```

PAPCO

Subject:
Year. Month. Date.

روش دوم پیدا کردن max :

```
Find-max(A,n)
{
  if n=1 then return A[1];
  max := Find-max(A,n-1);
  if A[n] > max then return A[n];
    else return max;
}
```

مسئله n تایی را به یک آرایه n-1 تایی تبدیل کردیم تا ساده تر شود.

مزایای الگوریتم های تقسیم و حل :

- این نوع الگوریتم ها یک راه عمومی پیش روی ما می گذارند که وابسته مسئله خاصی نیست اگرچه ممکن است کار را کند کند.
- مزیت دیگر این الگوریتم ها مربوط به زمانی است که بیش از یک cpu در کامپیوتر وجود دارد که در این صورت هر بخش تقسیم شده توسط یک cpu ، به طور همزمان محاسبه می شود و در این صورت سرعت افزایش خواهد یافت.

جلسه سوم 29.1.87.

مثال (1) الگوریتم محاسبه $\sum_{i=1}^{n} i^6$ (به روش تقسیم و حل)

برای حل این مسئله دو راه حل وجود دارد :

1- مسئله را از n مرحله به n-1 مرحله می شکنیم

2- بازه را نصف کرده و هر کدام را جداگانه محاسبه می کنیم

روش اول : (1)

```
f(n)
{
  if n=1 return 1;
  returnt n*n*n*n*n*n + f(n-1);
}
```

PAPCO

روش دوم: (۲)

```
f(start, end)
{ if start = end then return end*...*end ;   (6 بار)
  mid := [(start + end)/2] ;
  return f(start, mid) + f(mid+1, end);
}
```

توجه: در سطر اول اگر به جای اینکه end را ۶ بار در خودش ضرب کنیم، یک می‌گذاشتیم، الگوریتم غلط بود زیرا هیچ جا توان 6 حساب نمی‌شد؛ مثلاً فرض کنید f(1,2) را بخواهیم محاسبه کنیم و ret (1) در سطر اول داشته باشیم:

$f(1,2) = f(1,1) + f(2,2) = 1+1 = 2$ ✗

$f(1,2) = f(1,1) + f(2,2) = (1*\cdots*1) + (2*\cdots*2) = 65$ ✓

حال به محاسبه پیچیدگی هر یک از این دو الگوریتم می‌پردازیم:

if → 1+7 = 5×3+2 =

پیچیدگی روش (۱):

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

$\Rightarrow \boxed{T(n) = ak + T(n-a)}$ فرمول کلی

از آنجا که مقدار T(1) برای ما مشخص است ← $\boxed{n-a=1}$ عدد ثابت

$$\Rightarrow T(n) = (n-1)k + T(1)$$
$$= (n-1)k + c \rightarrow O(f(n)) = O(n)$$

پیچیدگی روش (۲):

$$T(n) = k + 2T(n/2)$$
$$= k + 2(k + 2T(n/4))$$
$$= 3k + 4T(n/4)$$
$$= 3k + 4(k + 2T(n/8))$$
$$= 7k + 8T(n/8)$$
$$= 15k + 16T(n/16)$$

$\Rightarrow \boxed{T(n) = (2^a - 1)k + 2^a T(n/2^a)}$ فرمول کلی

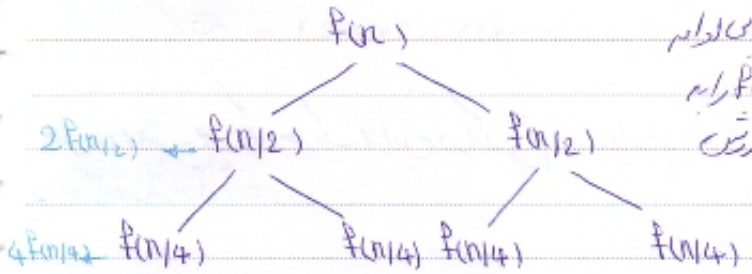
Subject:
Year. Month. Date.

باید کار را آنقدر ادامه دهیم تا $\frac{n}{2^a} = 1$ شود زیرا $T(1)$ یک عدد ثابت و مشخص است، اما چون n همواره توانی از 2 نیست ← در نظر می‌گیریم

$$\frac{n}{2^a} \leq 1 \rightarrow n \leq 2^a \rightarrow \log_2^n \leq a$$

$$T(n) = (2^{\log_2^n} - 1)k + nT(1)$$
$$= (n-1)k + n$$
$$= nk - k + n$$
$$= (k+1)n - k' \Rightarrow O(f(n)) = O(n)$$

توجه: در روش درختی کار را تا جایی ادامه می‌دهیم که برگ‌های درخت $f(1)$ را به ما بدهد در این صورت پیچیدگی از روش درختی هم بدست می‌آید.



نکته: در روش تقسیم و حل شکستن باید روی بازه انجام شود، مثلاً در مثال $\sum i^6$ اگر به جای بازه $[1, n]$، n^6 را بشکنیم مسأله بزرگتر را به مسأله کوچکتری تقسیم نکردیم و در آن صورت الگوریتم حاصل تقسیم و حل نخواهد بود.

مثال: پیچیدگی الگوریتم Merg Sort را بدست آورید.

$$T(n) = k + 2T(n/2) + \underbrace{k'n}^{*} + \underbrace{k''n}^{*}$$

* در سه while اول k' تا کار برای n صفر انجام می‌شود
* در while آخر برای جابجایی آرایه B به آرایه A، k'' تا کار برای n صفر انجام می‌دهد

$$= k + 2T(n/2) + pn$$
$$= k + 2(k + 2T(n/4) + pn/2) + pn$$
$$\checkmark * = 3k + 4T(n/4) + 2pn$$
$$= 3k + 4(k + 2T(n/8) + pn/4) + 2pn$$

$\checkmark\!\!\!\!\times = 7k + 8T(n/8) + 3pn$

$\checkmark\!\!\!\!\times = 15k + 16T(n/16) + 4pn$

$\Rightarrow \boxed{T(n) = (2^a - 1)k + 2^a T(n/2^a) + apn}$

$n = 2^a \rightarrow a = \log_2^n$

$\rightarrow T(n) \simeq (n-1)k + nT(1) + pn\log_2^n$

$= qn + pn\log_2^n \rightarrow$ این معادله از مرتبه $n\log n$ است

$\rightarrow \Theta(f(n)) = O(n\log n)$

مثال: محاسبه بزرگترین مقسوم علیه دو عدد را به روش تقسیم و حل (غیربازگشتی)

(36, 20)

$\frac{36}{16} \frac{20}{1}$ $\frac{20}{4} \frac{16}{1}$ $\frac{16}{0} \frac{4}{4}$ $\rightarrow$ bmm(36,20)=4

غیربازگشتی

```
Bmm (a,b)
{while a mod b ≠ 0 do
   { R:= a mod b;
     a:=b;
     b:=R;
   }
 return b;
}
```

بازگشتی

```
Bmm (a,b)
{R:=a mod b;
 if R=0 Return b;
 return Bmm(b,R);
```

برج های هانوی:

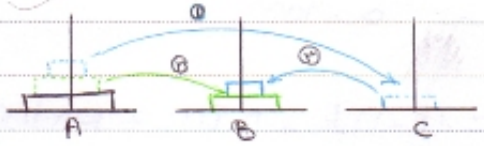
در مسئله برج های هانوی (Hanoi_Towers) سه میله و تعدادی دیسک داریم که شعاع های متفاوتی دارند. باید دیسک ها را طوری به میله C منتقل کنیم که ۱- در هر مرحله فقط یک دیسک جابجا شود ۲- همواره در تمام مدت جابجایی ها دیسک کوچکتر باید روی دیسک بزرگتر قرار گیرد. تا در نهایت دیسک به صورت نزولی (شعاع) در میله C مرتب شوند.

Subject:
Year. Month. Date. ( )

① $A \xrightarrow{2} B$
② $A \rightarrow C$
③ $B \xrightarrow{2} C$

مراحل کلی

① $\begin{cases} A \rightarrow C \\ A \rightarrow B \\ C \rightarrow B \end{cases}$



$A \xrightarrow{2} B$

② $\{ A \rightarrow C$

③ $\begin{cases} B \rightarrow A \\ B \rightarrow C \\ A \rightarrow C \end{cases}$

$B \xrightarrow{2} C$

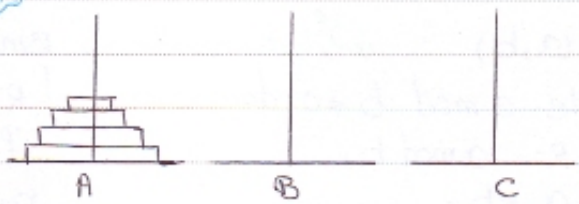
حال فرض کنید با 4 دیسک بخواهیم این کار را انجام دهیم باز هم یک روالی به صورت بالا خواهیم داشت:

① $A \xrightarrow{3} B$ $\begin{cases} A \xrightarrow{2} C \\ A \rightarrow B \\ C \xrightarrow{2} B \end{cases}$

مراحل کلی

② $A \rightarrow C$

③ $B \xrightarrow{3} A$

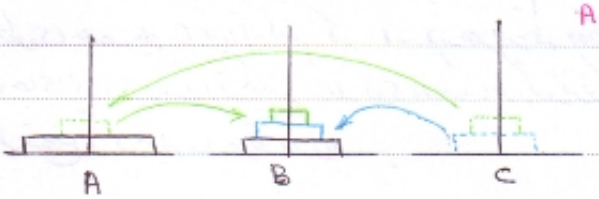


$\begin{cases} A \rightarrow B \\ A \rightarrow C \\ B \rightarrow C \end{cases}$ $A \xrightarrow{2} C$

$A \rightarrow B$

$\begin{cases} C \rightarrow A \\ C \rightarrow B \\ A \rightarrow B \end{cases}$ $C \xrightarrow{2} B$

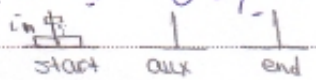
$A \xrightarrow{3} B$



و به همین ترتیب سه دیسک را از B به C منتقل می‌کنیم (بعد از اینکه A را به C منتقل کردیم)
- پس می‌توانیم به یک قانون کلی برسیم:

برای 2 دیسک:	برای 3 دیسک:	برای 4 دیسک:	برای 5 دیسک: ←	برای n دیسک
$A \xrightarrow{2} B$	$A \xrightarrow{3} B$	$A \xrightarrow{4} B$	$A \xrightarrow{4} B$	$A \xrightarrow{n-1} B$
$A \rightarrow C$	$A \rightarrow C$	$A \rightarrow C$	$A \rightarrow C$	$A \rightarrow C$
$B \xrightarrow{2} C$	$B \xrightarrow{3} C$	$B \xrightarrow{4} C$	$B \xrightarrow{4} C$	$B \xrightarrow{n-1} C$

حالا با این قانون می‌توان به راحتی الگوریتم برج‌های هانوی را به روش تقسیم و حل نوشت:
(تقسیم و حل است زیرا مثلاً برای انتقال 5 مهره ابتدا 4 مهره را منتقل می‌کنیم.)




```
Hanoi_Towers(start, end, aux, n)
{ if n=1 then
      {write(start, '→', end);
       return;
      }
  Hanoi_Towers(start, aux, end, n-1);   → n-1 دیسک را به میله کمکی منتقل می‌کند
  write(start, '→', end);               → بزرگترین دیسک را به end می‌برد
  Hanoi_Towers(aux, end, start, n-1);   → n-1 دیسک در میله کمکی را به میله نهایی منتقل می‌کند
}
```

- پیچیدگی: (خودم حل کردم)

$$T(n) = k + 2T(n-1)$$
$$= k + 2(k + 2T(n-2)) = 3k + 4T(n-2)$$
$$= 3k + 4(k + 2T(n-3)) = 7k + 8T(n-3)$$
$$= 7k + 8(k + 2T(n-4)) = 15k + 16T(n-4)$$

$$T(n) = (2^a - 1)k + 2^a T(n-a) \rightsquigarrow \boxed{n-a=1}$$

Subject:
Year. Month. Date. ( )

$$= (2^{n-1} - 1)k + 2^{n-1} \overset{\text{ثابت}}{T(1)}$$
$$= 2^{n-1}k - k + 2^{n-1}$$
$$= 2^{n-1}q - k = \frac{2^n}{2}q - k \rightarrow O(f(n)) = O(2^n)$$

جلسه چهارم 87.2.5

* الگوریتم های برگشت به عقب (Back Tracking):

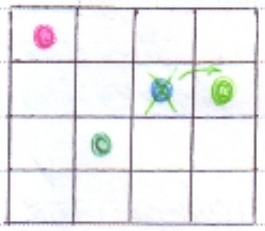
در این روش برای حل یک مسئله که راه حل های مختلفی دارد، یک راه را انتخاب کرده و جلو می رویم و هر جا به دوراهی رسیدیم یکی را انتخاب می کنیم و تا جایی که به شکل این مسیر نرسیدیم این کار را ادامه می دهیم، هر گاه به بن بست رسیدیم به آخرین دوراهی برمی گردیم و راه دوم را انتخاب می کنیم.

- نمونه هایی از مسائل back tracking:

(1) الگوریتم 8 وزیر:

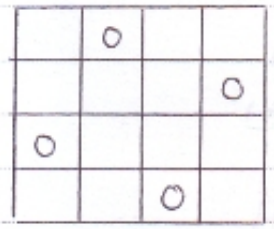
باید 8 وزیر را در یک مربع 8x8 طوری قرار دهیم که در راستای افق، عمودی و مورب هر وزیر، وزیر دیگری قرار نگرفته باشد یعنی در هر سطر، هر ستون و هر قطر مربع باید تنها یک وزیر وجود داشته باشد. برای روشن شدن مطلب، 4 وزیر را در نظر می گیریم (4x4)

- در هر سطر فقط یک وزیر می تواند قرار بگیرد.



در سطر اول: back tracking می گوید که 4 انتخاب وجود دارد (از چپ به راست حرکت می کنیم و اولین گزینه را انتخاب می کنیم.)

- در سطر دوم: ۲ انتخاب وجود دارد زیرا خانه اول و دوم سطر دوم نمی تواند وزیری قرار بگیرد ← اولین جا مناسب انتخاب می شود



- در سطر سوم: هیچ انتخابی وجود ندارد بنابراین به سطر دوم برمی گردیم و راه دوم را انتخاب می کنیم و به سطر سوم می رویم...

- این سوال را آنقدر ادامه می دهیم تا به پاسخ مورد نظر که مطابق شکل روبرو است برسیم.

الگوریتم 8 وزیر:

```
nQueen (n)
{
  row := 1;
  can Be solved := true;          → فرض می‌کنیم جایی جواب نداریم
  Back Track := false;            → جهت حرکت را مشخص می‌کند (در شروع کار برگشت نداشتیم)
* while (row ≤ n and can Be solved) do
    { if not back track then col := 1;   → اگر برگشت نبودی از ستون اول شروع کن
      else { col := find Queen (row);    ← وزیر را در خانه قبل بررسی داریم و از ستون بعدی کار را ادامه می‌دهیم
             Squere [row, col] := 0;     → 0: وزیر در خانه (r,c) نمی‌تواند
             col := col + 1;             1: وزیر در خانه (r,c) قرار دارد
             }
    while can Not place (row, col) and col ≤ n do
        col := col + 1;
    if col ≤ n then
      { squere [row, col] := 1;
        row := row + 1;
        back track := false;
      }
    else { row := row - 1;            * یعنی مسئله حل نمی‌شود زیرا در سطر (r) از آرایه سطر هیچ ستون مناسبی پیدا نشد.
           back track := true;
         * if row = 0 then can be solved := false; }
      }
  } // while //
} // function //
```

* اگر بتوانیم وزیر را در (row, col) قرار دهیم و از while بیرون می‌آییم ← یعنی تا وقتی وزیر را قرار نداده، هر بار یکی به ستون اضافه می‌کند تا جای مناسب را پیدا کند سپس از حلقه بیرون می‌آید.

Subject:
Year. Month. Date. ( )

اصل الگوریتم تا همین جا تمام می‌شود اما برای واضح شدن الگوریتم، توابع فراخوانی شده را نیز می‌نویسیم:

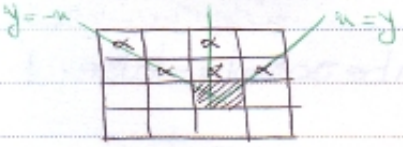
```
find Queen (row)* →
{ for c:=1 to n do
      if squere[row,c]=1      return c;
}
```

(خانه‌ای از سطر row که وزیر در آن قرار دارد را پیدا می‌کند)

```
* Can Be Solved (row, col) →
{ for r:=1 to row-1 do
      if squere[r, col]=1  return true;
  for r:=1 to row-1 do
      for c:=1 to n do
          if |r-row| = |c-col| and squere[r,c]=1
              return true;
  return;
}
```

(خانه‌هایی از جدول که وزیر نمی‌تواند قرار بگیرد را برمی‌گرداند)

* برای بررسی هر خانه جدول فقط سطرهای بالایی آن خانه را بررسی می‌کنیم؛
خانه‌هایی که شماره ستونشان با شماره ستون خانه مورد بررسی یکی است ← وزیر نمی‌تواند باشد



→ |r-row| = |c-col| →
وزیر نمی‌تواند قرار بگیرد.

توجه: این الگوریتم (8 وزیر) فقط یک جواب را می‌دهد اگر بخواهیم همه جواب‌ها را بدهد باید در هر مرحله جواب را چاپ کند، back track را true کند و دوباره به ابتدای الگوریتم برگردد.

پیچیدگی این الگوریتم n^n است زیرا در هر سطر، n ستون را چک می‌کند؛ البته باید توجه داشته باشیم

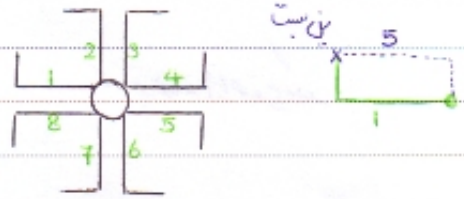
PAPCO

که پیچیدگی این الگوریتم قابل محاسبه نیست، اما ممکن است پیچیدگی $n!$ هم بتوان به این الگوریتم نسبت داد.

(۲) الگوریتم حرکت اسب (L) در صفحه شطرنج:

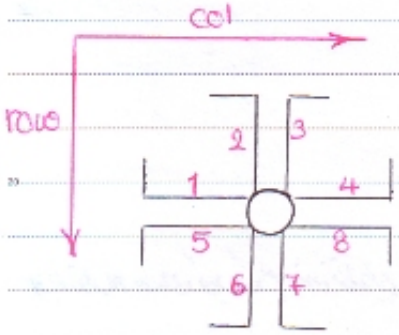
یک اسب در خانه مشخصی از شطرنج قرار دارد می‌خواهیم اسب را طوری حرکت دهیم که از تمام خانه‌های شطرنج فقط **یکبار** عبور کند.

- در خانه شروع، مطابق شکل،



8 حالت وجود دارد، که دو به دو عکس هم هستند. یعنی اگر با حرکت (1) به جلو برویم و به پی بست برسیم بلد با حالت (5) به نقطه اول برگردیم.

- برای راحتی کار حرکت‌ها را طوری شماره‌گذاری می‌کنیم که جمع هر دو حالت قرینه برابر 9 باشد. در این صورت وقتی با حالت x حرکت کرده و به پی بست رسیم آنگاه با حالت $9-x$ به نقطه قبل برمی‌گردیم و آنجا حالت $x+1$ را انتخاب می‌کنیم. (مثلاً با حرکت (1) می‌رویم، به پی بست می‌رسیم با (8) برمی‌گردیم و حرکت (2) را انتخاب می‌کنیم. برای حفظ حالات به یک آرایه دوبعدی که در بین $[-2,2]$ دارد احتیاج داریم. (برای حذف آرایه یک مقدار ثابت (مثلاً 3) را به همه اضافه می‌کنیم تا نمایه‌ها حذف شوند)



	row	col
1	-1	-2
2	-2	-1
3	-2	+1
4	-1	+2
5	+1	-2
6	+2	-1
7	+2	+1
8	+1	+2

$\xrightarrow{+3}$

21
12
14
25
41
52
54
45

66 66 66 66

← شماره حرکت‌ها را به این شکل تغییر دادیم تا آرایه نیاز نداشته باشیم

Subject:
Year. Month. Date. ( )

جلسه پنجم 87.2.12

با توجه به اینکه (1-) خانه شروع بوده و خانه‌های صفر نشان دهنده این است که اسب از آن خانه عبور نکرده و اعداد 1 تا 8 هم سایر خانه را نشان می‌دهد، الگوریتم حرکت اسب را به صورت زیر می‌نویسیم

الگوریتم حرکت اسب:

```
                          شروع   ایجاد صفحه شطرنج
Knight movement (n, r1, c1)
{ for i=1 to n do
    for j=1 to n do
        square [i, j] := 0;      →  همه خانه‌ها را صفر می‌کند
  square [r1, c1] := -1;
  CanBeSolved := true;
  row := r1;   col := c1;
* Counter := 1;                 →  برای شمردن تعداد پایان الگوریتم یک متغیر Counter تعریف می‌کنیم
  move := 1;                    →  متغیر move شماره حرکت را مشخص می‌کند
  while CanBeSolved and Counter < n*n do
   { * while move ≤ 8 and CanNotMove (n, row, col, move) do
          move := move + 1;
     if move ≤ 8                                                    *
      { MoveTo (n, row, col, move);
        square [row, col] := move;
        Counter := Counter + 1;
  ←   * Move := 1;
      }                                                             * متغیر m به ما می‌گوید در کدام خانه قرار داریم
     else { m := square [row, col];
            if m = -1  canBeSolved := false;
            else { square [row, col] := 0;
                   moveto (n, row, 9-m);
```