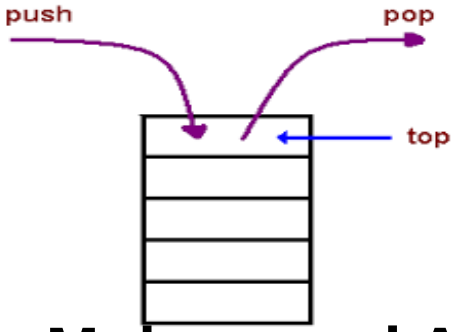
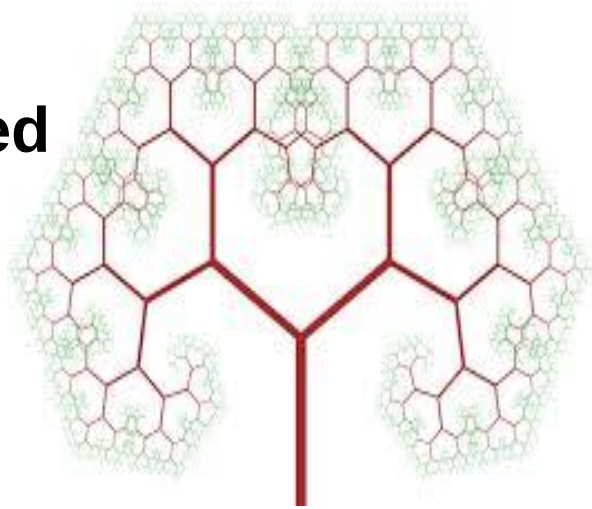
# جامعة حماة
# الكلية التطبيقية
# قسم تقنيات الحاسوب
# الفصل الثاني


- المقرر: الخوارزميات وبنى المعطيات
- المحاضرة :الخامسة

**Dr. Mohammed AL-Mohammed**

# Chapter 5

## الخوارزميات العودية

## Recursion Algorithms

# العودية Recursion

العودية: هي طريقة لحل المشاكل التي تنطوي على كسر المشكلة(المسألة ) إلى مسائل فرعية أصغر وأصغر حتى تحصل على مسألة صغيرة بما فيه الكفاية بحيث أنه يمكن حلها بشكل بسيط .

وعادة ما تتضمن العودية تابع يستدعي نفسها. في حين أنه قد لا يبدو مثل ماهو ظاهرعلى الواجهة.

العودية تسمح لنا بكتابة حلول أنيقة للمسائل التي قد يكون من الصعب جدا برمجتها.

-أي خوارزمية عودية يجبُ أنْ يكون عنْدها حالة أساسيةbase case.

-أي خوارزمية عودية يجبُ أنْ تُغيّر حالتها وتتحرّك نحو الحالة الأساسية.

-أي خوارزمية عودية يجبُ أنْ تستدعي نفسها، بشكل تكراري (عودي).

# العودية Recursion

**تعريف العودية:** هي تقنية لحل المشكلة وذلك بحل نفس المشكلة ولكن بحجم مُدخلات أقل (أصغر) وفق نفس النوع.

**بتعريف آخر:** هو استدعاء نفس خوارزمية حل المشكلة ولكن بحجم مدخلات أقل حتى نحصل على مشكلة صغيرة نستطيع حلها بشكل مباشر وهذه المشكلة الصغيرة تدعى بـ الحالة القاعدية *Base case*.

**بتعريف آخر:** العودية هي خاصية جديدة تصاغ لها الخوارزميات ويمكن أن تؤديها حلقات التكرار العادية حيث تقوم العودية بإيجاد حلول بسيطة للمشاكل البرمجية.

**والعودية مشتقة من الفكرة التالية:** لحل مشكلة ما نقوم بإيجاد الحل الأبسط لفكرة مصغرة عن المشكلة الكلية وبذلك ننتهي بتوابع تستدعي نفسها.

**الفرضية:** لحل نفس المشكلة بطريقة أصغر وأصغر حتى تصبح المشكلة الكبيرة الكلية صغيرة ومباشرة.

أحد الطرق العامة لشرح العودية هي طريقة فرق تسد وتعد تسمية أخرى للعودية.

## فرِّق تَسُد *Divide and conquer*:

تتكون هذه التقنية من خطوتين هما:

1- **فرّق**: أي تفريق المشكلة و تقسيمها إلى عدة مشاكل جزئية كي لا تصبح معقدة في الحل، وبذلك نكون قد بسطنا الحل إلى خطوات أسهل.

مثال:

$$\frac{5x + 3 - 20x^3(7x)}{log\, x + \sqrt{x-2}}$$

إن هذه الحالة هي مسألة معقدة ولكننا نستطيع تبسيطها إلى عدة خطوات أساسية كالجمع والطرح و ... إلى أن نصل في النهاية إلى حلول جزئية تشكل الحل العام.

**ملاحظة:** إذا كان حجم المدخلات أصغر من عتبة معينة فإنه يتم حلها مباشرة باستخدام طريقة واضحة ثم إنتاج الحل، وإلا سوف يتم تقسيم المشكلة إلى مجموعة من المشاكل الثانوية الصغيرة ⇐ تجزيء (تقسيم) مجموعة البيانات المدخلة إلى المشكلة.

2- تَسُد: يتم أخذ حلول المشاكل الصغيرة $solution1, solution2, \ldots$ ثم القيام بعملية الدمج لهذه الحلول من أجل الحصول على حل المشكلة الأصلية.

**هذه الفكرة مطروقة (مستخدمة) كثيراً في حل المشاكل وخاصة المعقدة منها.**

**ملاحظة: إن التعقيد الزمني لمشكلة كبيرة هو عبارة عن مجموع التعقيدات الزمنية للمشاكل الصغيرة المكونة لها.**

★ التابع الرياضي التالي:

$$f(n) = \begin{cases} 2 * f(n-1) & when\ n >= 1 \\ 1 & n = 0 \end{cases}$$

هذا الشكل تضمن الخاصية العودية لأن التابع هنا يستدعي نفسه وذلك بتغير القيمة من $n$ إلى $n-1$ فالشكل السابق يعني:

$$f(5) = 2 * f(4)$$
$$f(4) = 2 * f(3)$$
$$f(3) = 2 * f(2)$$
$$f(2) = 2 * f(1)$$
$$f(1) = 2 * f(0)$$
$$f(0) = 1$$

## من الأمثلة الشهيرة على العودية:

### تابع العاملي *Factorial Function*

- العاملي: $Factory(n!)$:

نعلم أن $n! = n(n-1)!$ .. إذاً: $n! = n(n-1)(n-2)!$

وهكذا ... حتى نقلل حجم البيانات (المدخلات) ونصل إلى الحالة القاعدية *Base case* التالية: $1! = 1$

بشكل تكراري: $n! = n * (n-1) * (n-2) * \ldots * 1$

```
1.long fact(int N)
2.{
3.int retval=1;
4.for(int i=N;i>0;i--)
5.{
6.retval=retval*i;
7.}
8.return retval;
9.}
```

وبذلك نحصل على العاملي من تكرار العملية

$$retval = retval * i$$

التي تعني رياضياً

$$retval = n * (n-1) * (n-2)..etc$$

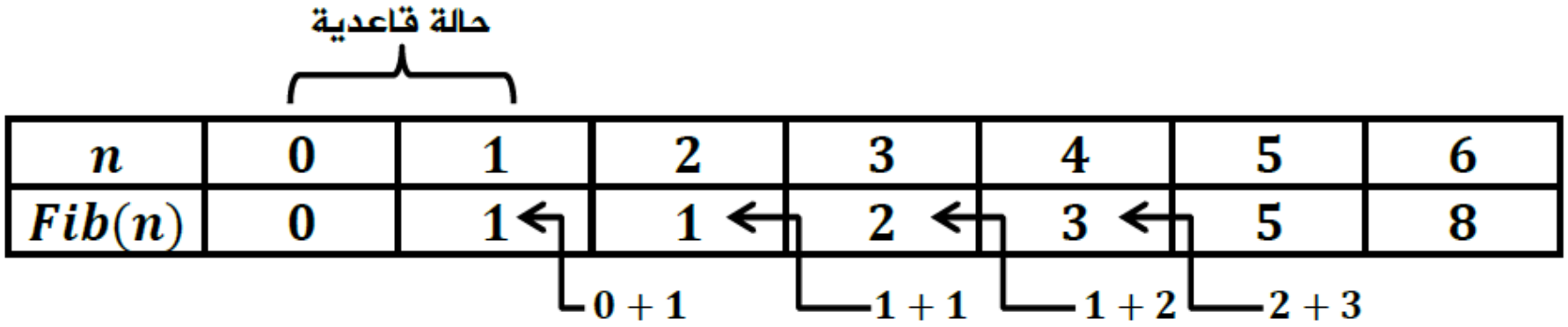
بشكل عودي:

```
1.int fact (int n)
2.{
3.if (n > 1)
4.return (n * fact (n-1));
5.else
6.return 1;
7.}
```

حيث قمنا بالشكل العودي بجعل التابع $fact$ يستدعي نفسه وذلك بإعادة القيمة

$n * fact(n-1)$ طالما $n > 1$ أما إذا كانت $n = 0$ تعيد عاملي الـ 0 وهو الـ 1.

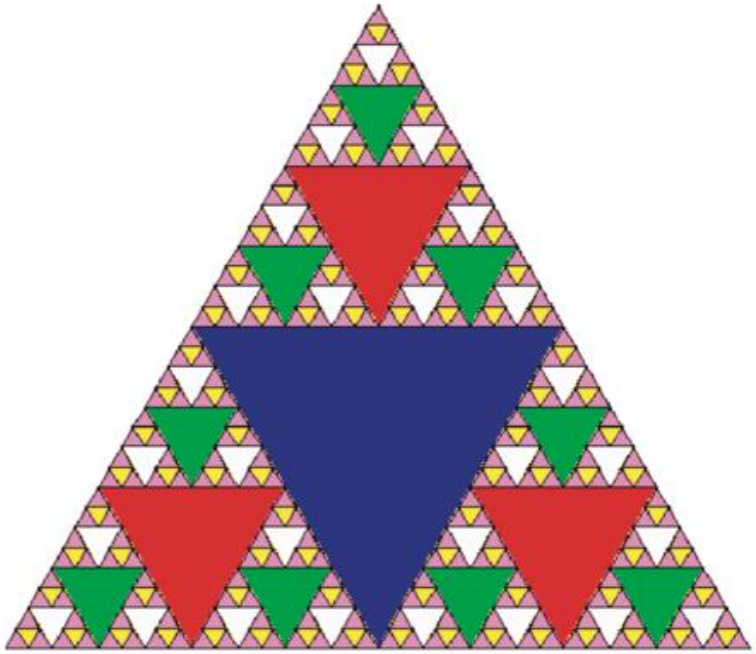
## - أرقام (سلاسل) فيبوناتشي *Fibonacci numbers*:




إن نتيجة $Fib$ تعتمد على جمع نتيجتي ما قبله وذلك بناءً على الحالة القاعدية:

$$Fib(0) = 0, Fib(1) = 1 \Rightarrow Fib(n) = Fib(n-1) + Fib(n-2)$$

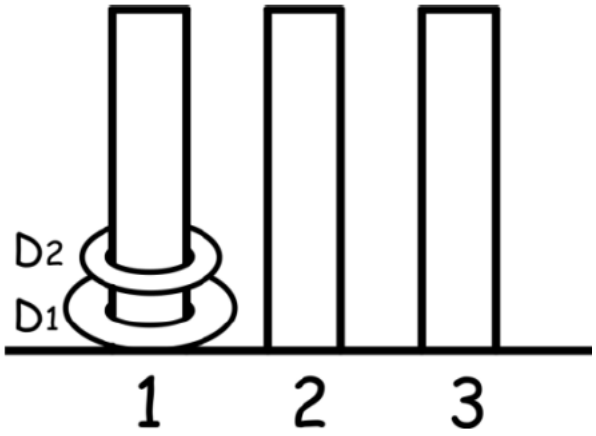
## - Sierpinski Triangle



## -أبراج هانوي Hanoi Towers

المطلوب من هذه العملية أن نرتب الأقراص وننقلها من 1 إلى 3 عبر العمود 2 بشرط عدم تحريك سوى قرص واحد في كل مرة وألا يقع قرص كبير فوق الصغير في أي مرحلة من مراحل الحل.

**الحل:**




حلها بسيط جداً وذلك بوضع القرص $D_2$ فى العمود 2 ثم نقل القرص $D_1$ إلى العمود 3 ثم نقل القرص $D_2$ مرة أخرى إلى العمود 3، ثم إذا زدنا عدد الأقراص نلاحظ أن المسألة تزداد تعقيداً وقد كانت هذه المسألة تستخدم قديماً لمعرفة مدى ذكاء الأشخاص.

فإذا فرضنا أن العمود الأول يحوي 9 أقراص مرتبة من الأسفل إلى الأعلى $(D_1 \rightarrow D_9)$ فهذه لا تحل بطريقة مباشرة، أما إذا اعتبرنا أن الـ 8 أقراص العليا هي عبارة عن قرص واحد وليكن $D_2$ يصبح لدينا في النهاية قرصان فقط وستصبح المسألة أقل تعقيداً لأننا سوف نصغر المشكلة جزئياً في كل مرة حتى نصل إلى الحل المطلوب.

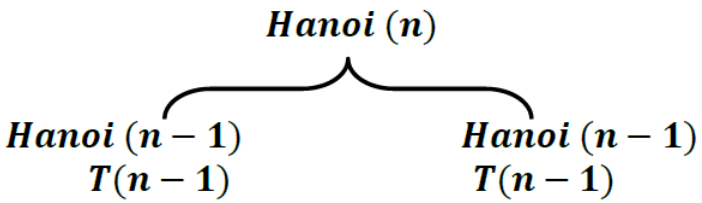
⇐ النتيجة: إن نقل قرصين هو حل بطريقة مباشرة ولكننا جزأناه مع أننا حللنا نفس المشكلة.

عمود موجود فيه  عمود سأنتقل إليه  عدد الأقراص

```
1. void Hanoi (int From,  int To  , int num) {
2. int Temp=6-From-To;
3. if (num==1)
4. cout<<"move"<<From<<"to"<<to;
5. else
6. Hanoi(From,Temp, num-1)
7. cout<<"…….  ;
8. Hanoi(Temp,To,num-1)
9. cout<<"……;}
```

(إن التعقيد الزمني $T(n)$ لـ $n$ قرص هو التعقيد الزمني لمجموع المشكلات)

⇐ التعقيد الزمني لخوارزمية هانوي:




$$T(n) = T(n-1) + T(n-1)$$
$$T(n) = 2 \cdot T(n-1)$$
$$T(n) = 2(\,2 \cdot T(n-2))$$
$$T(n) = 4\,T(n-2)$$
$$T(n) = \cdots$$
$$T(n) = 2 \cdot 2 \cdot 2 \cdot \ldots \ldots 2\,T(1)$$
$$T(n) = 2^n$$

$$\Longrightarrow T(n) = O(2^n)$$

وهو تعقيد زمني كبير جداً.

مثال رفع عدد إلى أس مثلاً $2^5$

```
1.#include <iostream.h>
2.long TwoRaisedTo(int N){
3.int retValue = 1;
4.for(int i=N; i>0; i--)
5.retValue = 2*retValue;
6.return retValue;}
7.int main(){
8.cout << "2 to the 5th is " << twoRaisedTo(5) << "\n";
9.return 0;}
```

```
1.  //Recursively computing powers of 2
2.  #include <iostream.h>
3.  int twoRaisedTo(int n)
4.  {
5.  if (n == 0) // special case
6.  return 1;
7.  else
8.  return 2 * twoRaisedTo(n-1);
9.  }
10. int main()
11. {
12. cout << "2 to the 5th is " << twoRaisedTo(5) << "\n";
13. return 0;
14. }
```

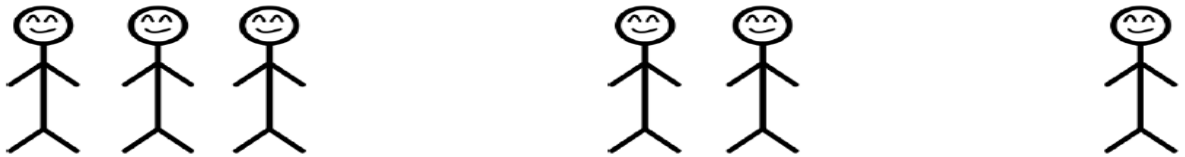
# أمثلة أخرى

الجمع – التوافيق – التراتيب – التعداد – الجذر التربيعي .....

**مثال: عدد المصافحات:**

إذا كان هنالك $n$ شخص في غرفة ما فما هو عدد المصافحات $h(n)$ في تلك الغرفة بفرض أن كل شخص يسلّم مرة واحدة فقط؟؟

الحل: سنمثل ذلك بدايةً برسم توضيحي بسيط:



| عدد الأشخاص | 1 | 2 | 3 |
|---|---|---|---|
| المصافحات | 0 | 1 | 3 |
| | $h(1) = 0$<br>$Base\ case$ | $h(2) = 1$<br>عدد المصافحات الجديدة + القديمة<br>$h(2) = h(1) + 1$ | $h(3) = 2 + 1 = 3$<br>$h(3) = h(2) + 2 = 3$ |

$\Rightarrow h(4) = h(3) + 3$

$h(n+1) = h(n) + n$

$h(n) = h(n-1) + (n-1)$

$h(n-1) = h(n-2) + (n-2)$

:

وهكذا حتى نصل إلى الحالة القاعدية

$h(1) = 0$

**مثال:** إيجاد القاسم المشترك الأكبر لعددين بالطريقة العودية: (تابع الـ$gcd$):

```
1. int gcd (int a,int b){
2. if (a==b)
3. return 0;
4. else if (a>b)
5. return gcd (b,a%b);
6. else
7. return gcd (a,b%a);}
```

```
1. int gcd (int a,int b){
2. if (a==b)
3. return 0;
4. else if (a>b)
5. return gcd (a-b,b);
6. else
7. return gcd (b-a,a);}
```

يأخذ التابع $gcd$ بارامترات هي $a$ وَ $b$ وحسب عدة شروط يقوم باستدعاء نفسه بعد تغيير البارامترات

ولفهم التابع لدينا المثال التالي:

نفرض $a = 60$ و$b = 24$ أي $gcd(60, 24)$

الشرط $a = b$ غير محقق والشرط $a > b$ محقق لذا ننفذ

التعليمة $return\ gcd(b, a\%b)$

وباقي قسمة $60/24$ هو 12

أي نعيد استدعاء التابع بالبارمترات $gcd(24, 12)$

ونعيد بذلك اختبار الشروط $a = b$ غير محقق و$a > b$ محقق

فنعيد استدعاء التابع و $24\%12 = 0$ أي أن التابع هو $gcd(12, 0)$

وبذلك فإن القاسم المشترك الأكبر هو 12.

## أي التوابع أفضل التكراري أم العودي؟

من حيث الأداء فإن التابع التكراري أفضل لأن العودي يقوم على استدعاء التابع نفسه وهذا يتطلب ذهاب للذاكرة وتخزين لحالة المعالجة الراهنة واستبدال السياق وتخزين قيم جديد خاصة بالتابع الذي استدعاه حتى لو كان نفسه لكن القيم تغيرت، وهذا كله يستغرق زمن وهذا الزمن يعد هدر.

**وبالتالي فان الخاصية العودية أبطأ من التكرارية**

أما من ناحية بساطة البرنامج فالتوابع العودية أفضل لأنها تحل المسألة بأبسط شكل كما أننا في أحيان كثيرة لا نستطيع حل المسائل بالطريقة التكرارية وبالتالي نكون محكومين بالطريقة العودية.

### التابع التركيبي *combinatorial Function*

وهو قانون التوافيق، ويكتب بهذا الكل عند استخدامه في أي خوارزمية.

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

# خوارزميات البحث العودية

## أولاً: البحث الخطي *Linear search*

```
1.int linear search (int a[], int n, int target){
2.if (n<=0)
3.Return -1;
4.else
5.if (a[n-1]==target)
6.return n-1;
7.else
8.return linear search (a, n-1, target);
9.}
```

نلاحظ الفرق في الخوارزمية السابقة عن خوارزمية البحث الخطي التي درسناها سابقاً هو في العودية أي استدعاء التابع لنفسه وذلك عندما يكون $n$ "عدد عناصر المصفوفة" أكبر من الصفر حيث نبدأ بالبحث عن $target$ في العنصر ذو الدليل $[n-1]$ فإذا كان غير موجود نستدعي التابع نفسه ولكن بتغيير البارامتر الثاني من $n$ إلى $n-1$ وبالتالي فعندما نبحث عن الـ $target$ فإن العملية تصبح $if\ ca\ [n-2] == target$ ونستمر في استدعاء التابع لنفسه إلى أن يجد الـ $target$.

سنرسم جدول التتبع للخوارزمية السابقة باستخدام المصفوفة: $a[5] = \{22, 11, 13, 81, 5\}$

التابع: $trace\ linear\ search\ (a, 5, 11)$

حيث: $a$: المصفوفة، 5: عدد الحدود، 11: $target$.

| $pass\ N$ | $n$ | $n <= 0$ | $a[n-1] == target$ | $reurn\ value$ |
|---|---|---|---|---|
| 1 | 5 | $F$ | $F$ | $ls\ (a, 4, 11)$ |
| 2 | 4 | $F$ | $F$ | $ls\ (a, 3, 11)$ |
| 3 | 3 | $F$ | $F$ | $ls\ (a, 2, 11)$ |
| 4 | 2 | $F$ | $T$ | 1 |

## ثانياً: البحث الثنائي *Recursive Binary search*

```
1.int binarysearch (int a[],int first,int last ,int target)
2.{
3.if (first>last)
4.return-1;
5.int mid=(first+last)/2;
6.if (a[mid]==target)
7.return mid;
8.else if (target<a[mid])
9.return binary search(a, first, mid-1, target);
10.else
11.Return binary search (a, mid+1, last,target);
12.}
```

الاختلاف في خوارزمية العودية هو في استدعاء التابع لنفسه فإذا كان الـ $target$ أصغر من $a[mid]$ يستدعي التابع نفسه يتغير البارامتر $last$ إلى $mid - 1$ وإذا كان $target$ أكبر من $a[mid]$ يستدعي التابع نفسه بتغيير البارامتر $first$ إلى $mid + 1$.

نرسم جدول لتتبع الخوارزمية البحث الثاني السابقة باستخدام المصفوفة

$$a = \{1, 6, 15, 22, 37, 45, 52\}$$

$$trace\ binary\ search\ (a, 0, 6, 37)$$

حيث: $a$: المصفوفة، 0: دليل اول عدد، 6: دليل آخر عدد، 37:الـ$target$.

| $passN$ | $f$ | $L$ | $f < l$ | $md$ | $a[mid] == target$ | $a[mid] > target$ | $retualue\ value$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 6 | $F$ | 3 | $F$ | $F$ | $Bs(4,6,37$ |
| 2 | 4 | 6 | $F$ | 5 | $F$ | $T$ | $Bs(4,5,37$ |
| 3 | 4 | 5 | $F$ | 4 | $T$ | – | 4 |

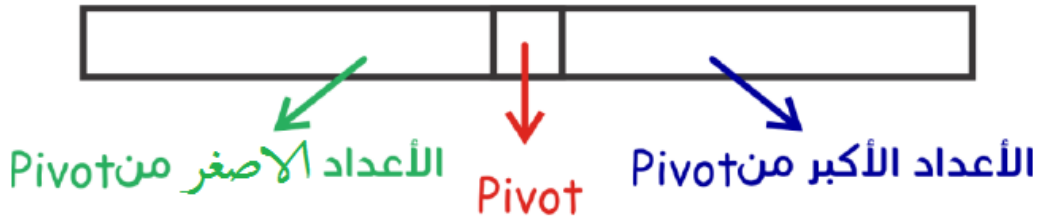
وضعنا حقول للمتغيرات $last$ و $first$ وللتعليمة $if(first > last)$ وللـ $mid$ وشرط $a[mid] > target$ والقيمة المعادة.

- مع أول محور كان $index$ أول عنصر هو 0 وآخر عنصر هو 6 وشرط الـ $f < l$ لم يتحقق "$False$"
- والـ $mid = \frac{6+0}{2} = 3$ ولم يتحقق شرط $a[mid] == target$ لم يتحقق "$False$"، وللشرط $a[mid] > target$ وبالتالي تكون القيمة المعادة هي التابع نفسه مع تغيير البارامترات $Bs(9,4,6,37)$، أي أن $first = mid + 1$ ونكمل في التتبع حتى نحصل على الـ $target$.

# خوارزميات الفرز العودية

## الفرز السريع *Quick sort*

- تعتمد هذه الخوارزمية على اختيار حد معين من المصفوفة ونسميه ***pivot*** (مثلاً) ثم نقوم بنقل جميع الأعداد التي أصغر منه قبله و نقل جميع الأعداد التي أكبر منه بعده وبالتالي يستقر ***pivot*** في موقعه الصحيح.
- وبعد هذه العملية يقوم التابع باستدعاء نفسه في المجالين "الأصغر من ***pivot*** والأكبر منه" ونستمر بهذه الاستدعاءات إلى أن نصل إلى نقطة خروج ونقطة الخروج هي أن تكون المجموعة الجديدة فارغة أو تحتوي على عنصر وحيد.




```
1.int quicksort (int a[], int first, int last)
2.{
3.if (first>=last)
4.return;
5.int split (partition (a, first, last));
6.quicksort (a, first, split-1);
7.quicksort (a, split+1, last);
8.}
```

أولاً التابع $partition$ هو التابع الذي اعتبرناه يقوم بعملية التقسيم للمصفوفة وتحديد مكان الـ$pivot$ أي تحديد $index$ الـ$pivot$ أما الـ$quick\ sort$ فهو يستدعي الـ$partition$ وله البارامترات:

$a$: المصفوفة

$first$: $index$ أول عنصري المجال الذي نطبق عليه الـ$quick\ sort$

$last$: $index$ آخر عنصري المجال الذي نطبق عليه الـ$quick\ sort$

علماً أن $first$ و $last$ هي قيم تختلف عند كل استدعاء للتابع كما هو مبين في الكود.

أما الـ$split$: هو عدد صحيح من نوع $int$ وهو الـ$index$ الـ$pivot$ لذلك نلاحظ أن القيمة المعادة من التابع $split = partition$

حيث أن التعليمة $int\ split\ (partition\ (a, first, last))$ تعني: $int\ split = partition\ (a, first, last)$

وبعد استدعاء الـ$partition$ يتحدد الـ$pivot$ وقيمته هي $a[split]$ ودليله $split$ ويتم وضع العناصر التي أصغر من الـ$pivot$ قبله وأكبر من الـ$split$ بعده، لم يتم استدعاء الـ$quick\ sort$ على المجال من $[a, split - 1]$ والمجال من $[split + 1, last]$ إلى أن تترتب كل المصفوفة.

## آلية عمل تابع التقسيم *partition*:

1) اختيار الـ *pivot*:

بالطريقة العامة يتم اختياره عشوائياً ثم نفرز باقي العناصر قبله وبعده كما وضحنا سابقاً ولكنها طريقة صعبة فتسهيلاً للعمل وللأكواد البرمجية نفرض أن الـ *pivot* هو $a[first]$ ونفرض أن آخر قيمة أصغر من الـ *pivot* هي $a[last\ small]$ بحيث عند انتهاء عملية المعالجة لكل العناصر نبدل بين $a[last\ small]$ والـ *pivot* فيصبح الـ *pivot* في مكانه الصحيح.

2) عملية المعالجة واختيار *last small*:

في البداية نفرض أن الـ *last small* هو $a[first]$ يصبح الـ $first$ يمثل الـ *index* الـ *pivot* والـ *last small* في بداية الأمر، تبدأ المعالجة من العنصر $a[first+1]$ فإذا كان أصغر من الـ *pivot* نضعه في الخانة $last\ small + 1$ ويصبح هو الـ *last small* من أجل المقارنات القادمة، وإذا كانت الـ $i$ أكبر من *pivot* يبقى مكانه.

```
1.int partition (int a[], int first, int last){
2.int lastsmall(first),i;
3.for (i=first+1;i<=1last;i++)
4.if (a[i]<=a[first]){
5.last small++;
6.swap Elements (a, lastsmall, i);}
7.swap Elements (a, first, lastsmall);
8.return lastsmall;
9.}
```