Dynamic & Greedy

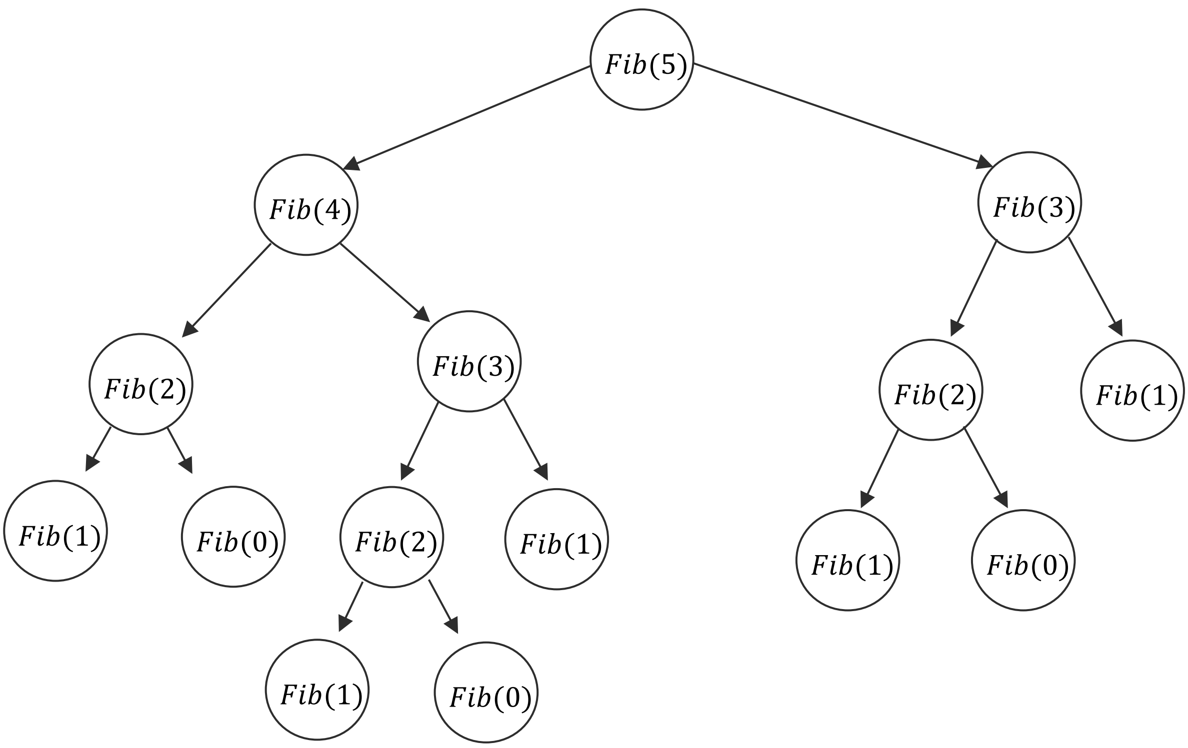
د. ابي صندوق

الخوارزمية الديناميكيّة

تُستخدم الديناميكيّة عندما يكون هناك تكرار للمشاكل الجزئيّة (sub problems)، وبالتالي تكرار في الاستدعاءات الجزئيّة، حيث نقوم بتخزين الحلول السابقة بدلاً من القيام بحسابها من جديد.

تذكرة: متتالية فيبوناتشي – The Fibonacci Sequence:

نعلم أنّ متتالية فيبوناتشي هي الأعداد: ... ,0,1,1,2,3,5,8,13، حيث أن عدد فيبوناتشي من أجل عدد n يحسب بالشكل التالي: $fib(n) = fib(n-1) + fib(n-2)$، ابتداءً من: $fib(0) = 0, fib(1) = 1$.



Dynamic solution:	Recursive solution:
$fib(n)$ $res[n]: array\ of\ integers$ $res[0] = 0\ , res[1] = 1$ $for(i \leftarrow 0\ to\ n)$ $\quad res[i] \leftarrow res[i-1] + res[i-2]$ $return\ res[n]$	$fib(n)$ $if\ (n = 0\ or\ n = 1)$ $\quad return\ n$ $else$ $\quad return\ fib(n-1) + fib(n-2)$

نلاحظ أنّ هناك عدّة استدعاءات يتم إعادة حسابها في النموذج العودي؛ (قيمة x يتم إعادة حساب $fib(x)$ من أجلها). وقد قمنا بتجاوز هذه المشكلة عبر تحويل الخوارزميّة السابقة إلى الشكل الديناميكي. حيث قمنا بداية بتعريف مصفوفة من الحجم n (حجم الدخل)، وذلك لحفظ النتائج السابقة للرجوع إليها بدلاً من القيام بحسابها من البداية.

الخوارزمية الديناميكية تأخذ في كل مرحلة الحل الأمثل، معتمدة بذلك على الحلول المثلى السابقة التي حصلنا عليها.

مسألة $(Longest\ Increasing\ Subsquence)$ $:LIS$

المصطلح $Subsquence$ يعني متتالية جزئية مثال: $A\{1,2,3,4,5\}$ نعتبر أن العناصر $\{2,4,5\}$ تشكل متتالية جزئية إلا أن العناصر $\{2,1,4\}$ ليست كذلك لأنها لم تأخذ الترتيب بعين الاعتبار ضمن مسألتنا هذه نريد معرفة أطول سلسلة جزئية مرتبة تصاعديا ضمن سلسلة معطاة .

الحل العودي:

$$\{5, 2, 8, 5, 6, 5, 6, 3, 4, 9, 5, 6, 7\}$$

كما مر معنا سابقا في مسألة الحقيبة فغننا من أجل كل رقم المتتالية الرئيسية

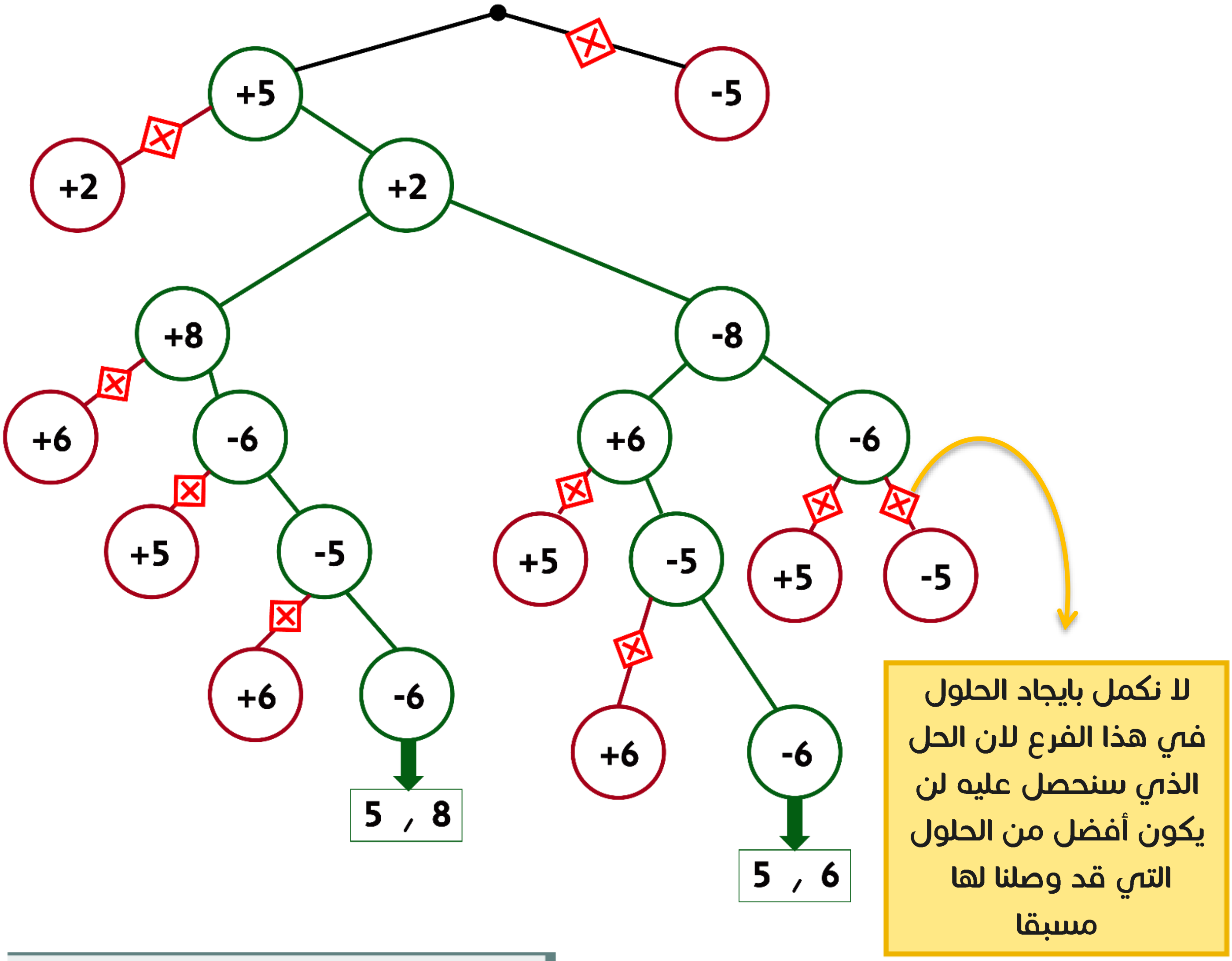
1. إما **نضيفه**:

في حال حقق الشرط المطلوب الا وهو أن تكون هذه المتتالية الجزئية مرتبة تصاعديا

أي: أن الرقم أكبر من أكبر رقم قمنا بإضافته

2. أو **نلغيه** (حيث يكون هذا الرقم أصغر من آخر رقم قمنا بإضافته للمتتالية الجزئية.)

((الحل كما مر معنا سابقا))



تعقيد هذا الحل $O(2^n)$

الحل الديناميكي:

نعتمد في الحل الديناميكي على طريقة تخزين الحلول الجزئية تدعى هذه الطريقة ب:(Memoiztion)

فلتكن لدينا المتتالية الرئيسية التالية: $\{5,2,8,5,6,5,6,3,4,9,5,6,7\}$

$\{5,2,8\}$

↲ 5

$\{5,2,8,6\}$

↲ 2 ↲ 5

$$\{5, 2, 8, 6, 5\}$$

$$5 \Lsh 2 \Lsh \hookrightarrow (5,8)$$ او $$(2,8)$$

$$\{5,\ \ 2,\qquad 8\qquad 6, 5, 6\quad ,, \ldots \ldots\qquad\qquad 5, 6, 7\}$$

$5 \Lsh 2 \Lsh \quad \hookrightarrow (5,8) \Rsh (5,6)$ وهكذا حتى نصل $\Lsh$ $2, 3, 4, 5, 6, 7$

وهو الحل الأمثل الصحيح.

مقارنة بين:

- الخوارزميّة الديناميكية - Dynamic
- الخوارزميّة الطموحة - Greedy

الخوارزميّة التراجعيّة – Backtracking

تهدف هذه الخوارزميّة إلى حساب (تجربة) جميع الاحتمالات الممكنة، والمقارنة بين الحلول لإعطاء الحل الأمثل.

تسلك التراجعية سلوكاً معيّناً (لها شكل ثابت).

$$try(\,)\{$$

$$if\ \left(\text{شرط القبول}\right)$$

إعادة النتيجة للمقارنة

$$try\ \left(\text{من أجل حالة جديدة}\right)\ \}$$

الخوارزميّة الطموحة – Greedy

تسعى الخوارزميّة الطموحة إلى انتقاء أفضل حل (ظاهريّ) لتعطي حلاّ صحيحاً ولكن ليس الحل الأمثل بالضرورة.

لنوضّح سلوك الخوارزميّة التراجعية والخوارزميّة الطموحة، وذلك من خلال المثال التالي.

يريد محاسب في محل تجاري إعادة الفكّة (الباقي) للزبائن وذلك بأقل عدد ممكن من العملات، حيث تتوافر العملات التالية:

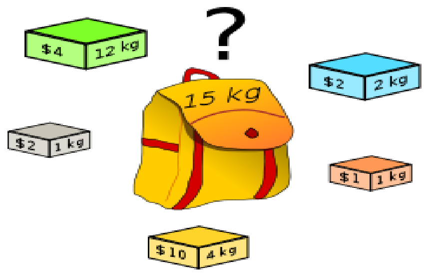
$Coins: \{1,5,10,21,25\}$.

	Greedy	Backtracking
Case 75	3×25	3×25
Case 63	$2 \times 25 + 10 + 3 \times 1$	3×21

نلاحظ أن كلاً من الخوارزميتين قد سلك سلوكاً مختلفاً. في حالة الدخل: 63، تقوم الخوارزميّة التراجعيّة بتجريب جميع الحالات المتاحة، أما الخوارزميّة الطموحة، فتقوم باختيار أكبر عملة متاحة طالما هناك باقٍ.

من خلال المثال السابق يتوضّح أنّ الخوارزميّة التراجعيّة قامت بتقديم الحل الأمثل.

Greedy algorithm



بالعودة الى مسألة الحقيبة التي تناولنها في الحاضرات السابقة سنعرض حلها التراجعي و من ثم سنحوله الى حل طموح:

يوجد مجموعة من الأغراض، حيث لكل غرض وزن وقيمة، و نريد أخذ مجموعة من هذه الأغراض بحيث أن يكون مجموع القيم لهذه الأغراض أكبر ما يمكن، وذلك ضمن حدود الوزن الأقصى الذي يمكن لحقيبته تحمّله.

الحل التراجعي

المتحوّلات وبنى المعطيات المستخدمة:

- عدد الأغراض الكلّي $n: integer$
- ومصفوفة لتخزين قيم الاغراض $array\ of\ integers\ [n] value$
- نعرف المصفوفة $size$ لتخزين حجم الاغراض $size: array\ of\ integers\ [n]$

فكرة الحل:

نقوم بالمرور على الأغراض ابتداءً من الغرض رقم n. عند انتهاء الأغراض، أو عندما لم يعد هناك سعة متاحة، فإن التابع يعيد القيمة 0، لأن القيمة التي يمكنه الحصول عليها هي 0. وإلّا، فنحن أمام خيارين:

1) ضم الغرض الحالي (i)، وهنا نطرح وزن الغرض من الوزن المتاح الحالي، ونخزّن في المتحوّل c_1، قيمة الغرض مضافاً إليها أكبر قيمة يمكن تحصيلها من الأغراض المتبقية وضمن حدود الوزن المتاح الحالي، وهي القيمة التي يردّها الاستدعاء من أجل الغرض التالي والوزن الجديد.
2) استبعاد عن الغرض الحالي، وهنا نخزّن في المتحول c_2 أكبر قيمة يمكن تحصيلها من الأغراض المتبقيّة وضمن حدود الوزن الحالي (دون تغيير)، وهي القيمة التي يردّها الاستدعاء من أجل الغرض التالي والوزن المتاح الحالي.

نختار الحل الذي يعود بالمردود الأكبر ونردّ قيمته.

Pseudocode:

```
main function( )
read(n)
read(MAX)
read_array(size)
read_array(value)
res = knapsack(MAX, n)
print(res)
```

```
knapsack (capacity, i)
if( i = 0 or capacity = 0 )
    return 0
else
    if( size [i] > capacity )
        return knapsack(capacity , i − 1 )
    else
        c1 = value[i] + knapsack(capacity − size[n], i − 1)
        c2 = knapsack( capacity , i − 1 )
        return max(c1, c2)
```

Greedy solution

إنّ خطوات الحل الطموح (greedy) هي مطابقة لخطوات الحل التراجعي، ولكن سنقوم بتخزين نتائج جميع الاستدعاءات لحذف الاستدعاءات المتكررة. وسنوظّف المصفوفة $memo$ كذاكرة للتخزين، ففي الحل التراجعي قمنا بالمقارنة بين الحلّين الناتِجَين عن ضمِّ العنصر الحالي واستبعاده، ونأخذ الـ max بينهما، ولكن هنا سنقوم بتخزين هذا الحل ثمّ ردّه بدلاً من ردّه مباشرة، وستشكّل القيمة المخزّنة الحل الأفضل الذي يمكن الوصول إليه ضمن المعطيات الحالية، (الوقوف عند العنصر الحالي والوزن المتاح الحالي)، بالتالي عند الاستدعاء مرّة أخرى من أجل المعطيات ذاتها، سيتم رد هذه القيمة مباشرة، ممّا يخفّف من التعقيد.

Pseudocode:

```
driver (MAX, n)
Initialize_memo( −∞)
 return knapsack(MAX, n)
```

```
knapsack(capacity, i)
if(memo[i] ≠ −∞)
   return memo[i][capacity]
else
    if(i = 0 or capacity = 0)
       q = 0
    else
       if(size[i] > capacity)
          q = knapsack(capacity, i − 1)
       else
          c1 = value[i] + knapsack(capacity_size[i], i − 1)
           c2 = knapsack(capacity, i − 1)
           q = max(c1, c2)
          memo[i][capacity] = q
    return q
```

انتهى المقرر النظري لمادة الخوارزميات1 ...

تنويه: تعد المحاضرات ليست كافية للاحاطة بأطراف المادة كاملة ... هي فقط نتاج اجتهدنا لتقديم الفائدة و الدعم العلمي ... وتعد طريقة لتغطية أفكار المقرر و التطرق الى معلوماته ليس الا...

نتمنى لكم التوفيق و النجاح ^.^