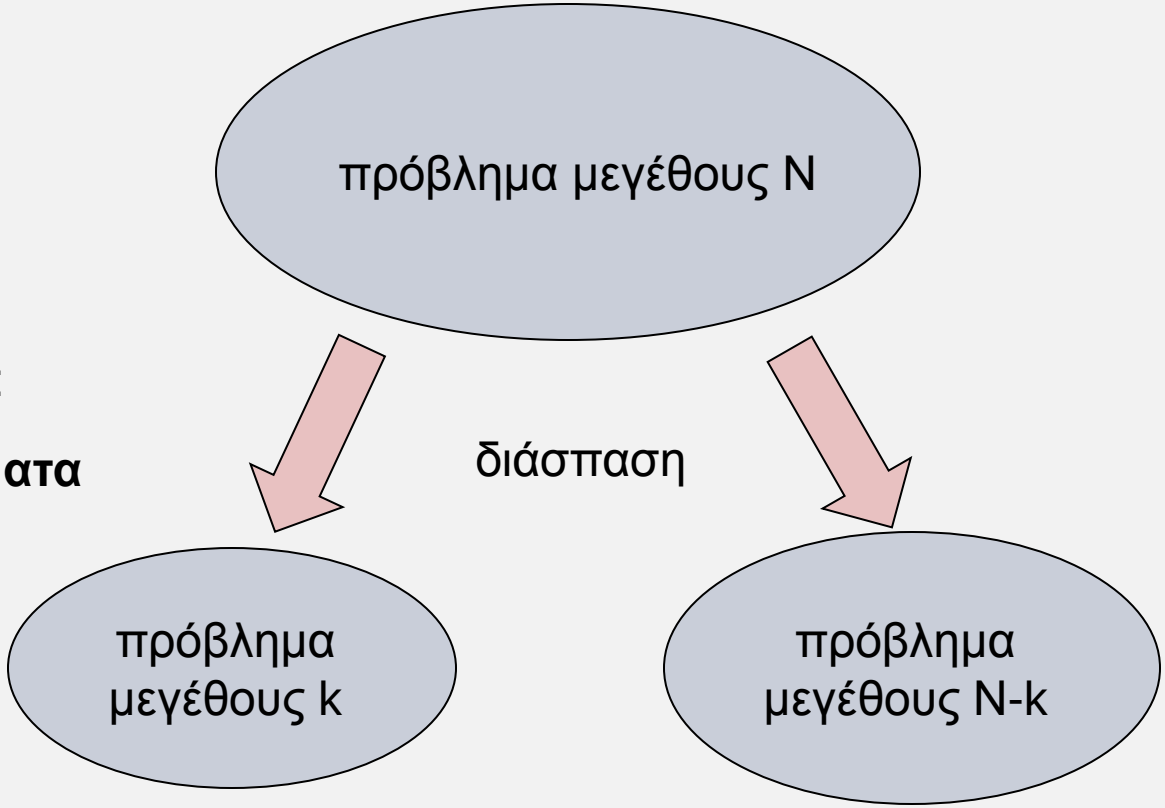
Δυναμικός Προγραμματισμός

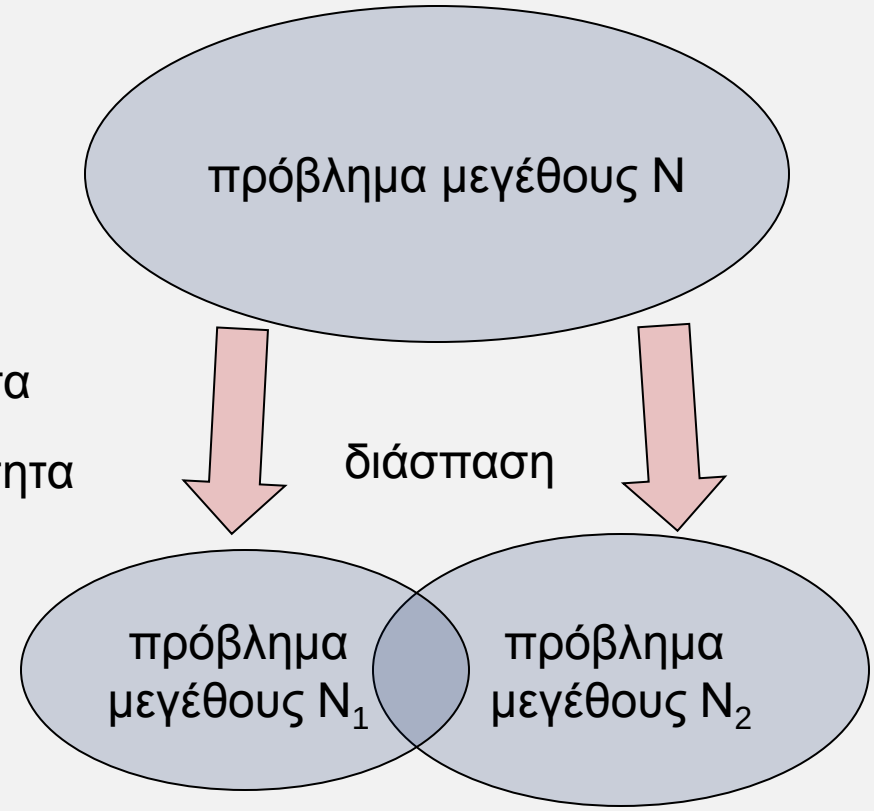
«Διαίρει και βασίλευε» :

ανεξάρτητα υποπροβλήματα



Δυναμικός Προγραμματισμός

Σε κάποιες περιπτώσεις όμως τα υποπροβλήματα δεν είναι ανεξάρτητα



Υπάρχουν επικαλυπτόμενα υποπροβλήματα, γεγονός που μπορεί να οδηγήσει σε πολύ μεγάλους χρόνους εκτέλεσης

Δυναμικός Προγραμματισμός

Παράδειγμα: Υπολογισμός της ακολουθίας Fibonacci

$$f_n = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f_{n-1} + f_{n-2}, & n \geq 2 \end{cases}$$

Η επίλυση της αναδρομικής εξίσωσης δίνει

$$f_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right]$$

Δυναμικός Προγραμματισμός

Παράδειγμα: Υπολογισμός της ακολουθίας Fibonacci

$$f_n = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f_{n-1} + f_{n-2}, & n \geq 2 \end{cases}$$

Μπορεί να υπολογιστεί με το ακόλουθο αναδρομικό πρόγραμμα :

```
int fibonacci(int n)
{
    if (n<1) return 0;
    if (n==1) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Δυναμικός Προγραμματισμός

Παράδειγμα: Υπολογισμός της ακολουθίας Fibonacci

$$f_n = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f_{n-1} + f_{n-2}, & n \geq 2 \end{cases}$$

Μπορεί να υπολογιστεί με το ακόλουθο αναδρομικό πρόγραμμα :

```
int fibonacci(int n)
{
    if (n<1) return 0;
    if (n==1) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

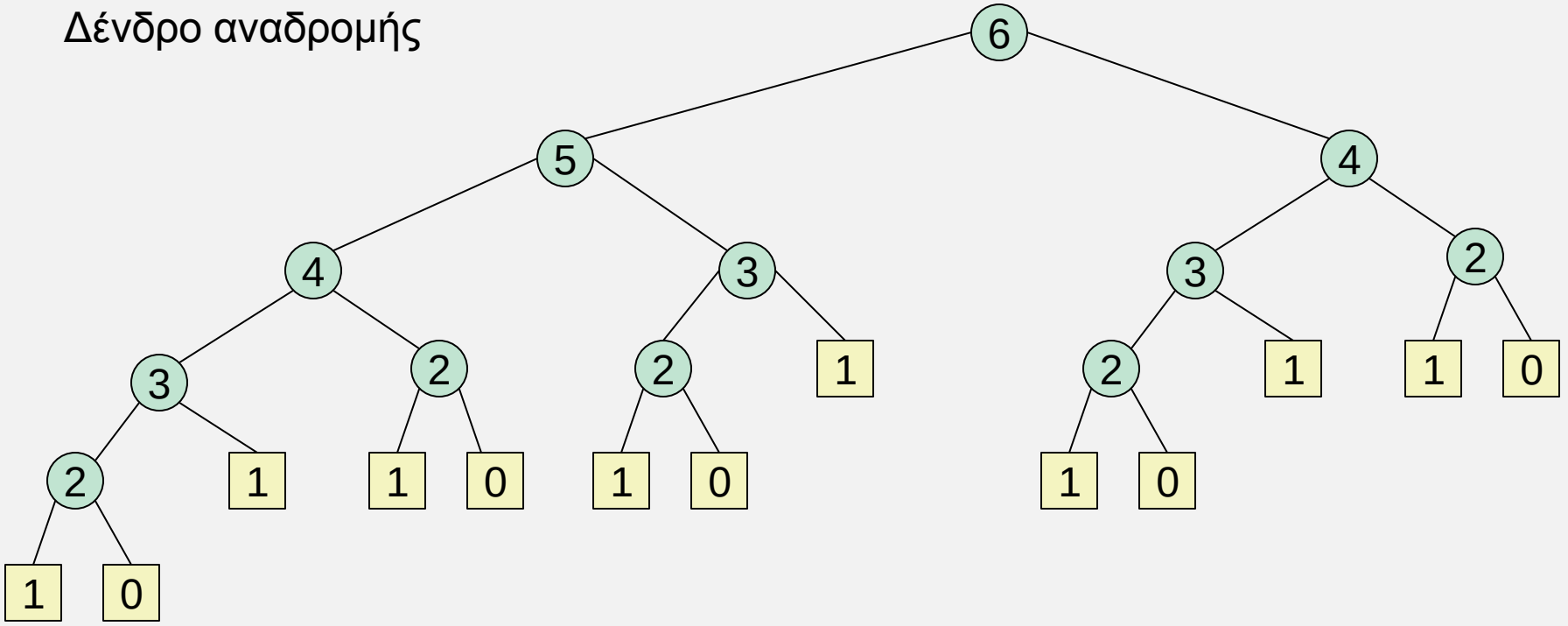
Χρόνος εκτέλεσης : $T(n) = T(n-1) + T(n-2) + \Theta(1)$

$$\geq 2T(n-2) + \Theta(1) \Rightarrow T(n) = \Omega((\sqrt{2})^n)$$

Δυναμικός Προγραμματισμός

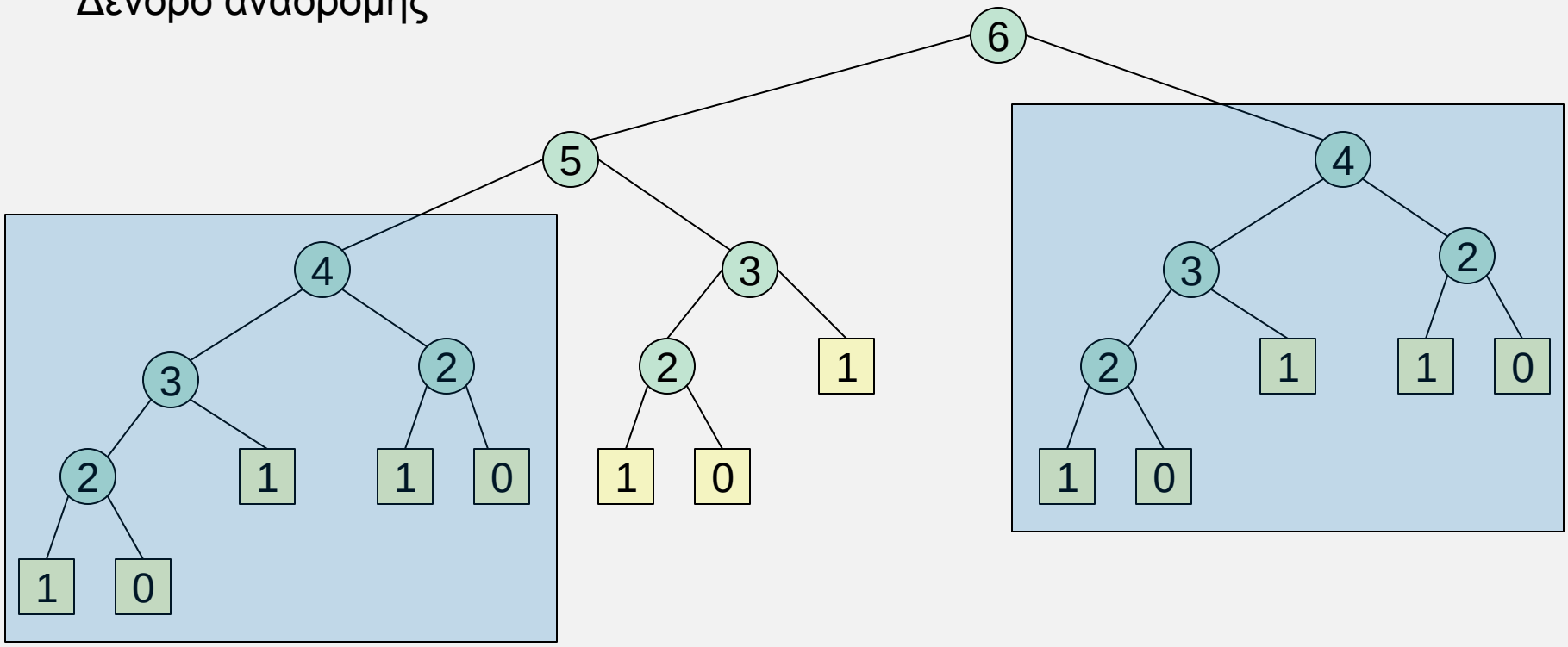
```
int fibonacci(int n)
{
    if (n<1) return 0;
    if (n==1) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Δένδρο αναδρομής



Δυναμικός Προγραμματισμός

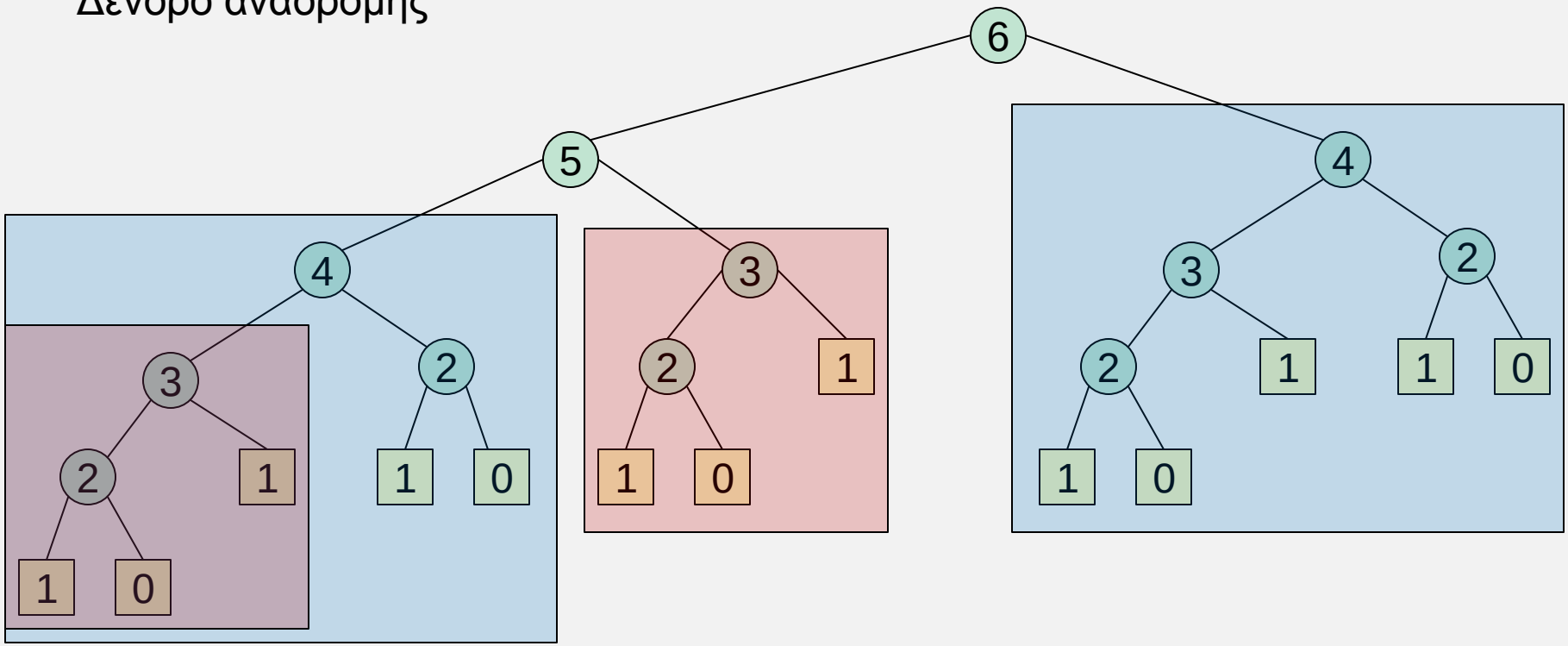
Δένδρο αναδρομής



Ιδέα: Αντί να υπολογίσουμε το f_4 δύο φορές, το υπολογίζουμε μία φορά και αποθηκεύουμε την τιμή του …

Δυναμικός Προγραμματισμός

Δένδρο αναδρομής



Ιδέα: Αντί να υπολογίσουμε το f_4 δύο φορές, το υπολογίζουμε μία φορά και αποθηκεύουμε την τιμή του …

ομοίως και για τις άλλες τιμές που χρειαζόμαστε

Δυναμικός Προγραμματισμός

Παράδειγμα: Υπολογισμός της ακολουθίας Fibonacci

$$f_n = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f_{n-1} + f_{n-2}, & n \geq 2 \end{cases}$$

Μπορεί να υπολογιστεί σε γραμμικό χρόνο με το ακόλουθο πρόγραμμα :

```
int fibonacci(int n)
{
    int i;
    int F[n+1];
    F[0]=; F[1]=1;
    for (int i=2; i<=n; i++)
       F[i] = F[i-1] + F[i-2];
    return F[n];
}
```

Δυναμικός Προγραμματισμός

Συνθετικός δυναμικός προγραμματισμός (bottom-up dynamic programming)

Υπολογίζει και αποθηκεύει όλες τις τιμές της συνάρτησης με τη σειρά ξεκινώντας τον υπολογισμό από τη μικρότερη τιμή του ορίσματος.

```
int fibonacci(int n)
{
    int i;
    int F[n+1];
    F[0]=; F[1]=1;
    for (int i=2; i<=n; i++)
       F[i] = F[i-1] + F[i-2];
    return F[n];
}
```

Δυναμικός Προγραμματισμός

Αναλυτικός δυναμικός προγραμματισμός (top-down dynamic programming)

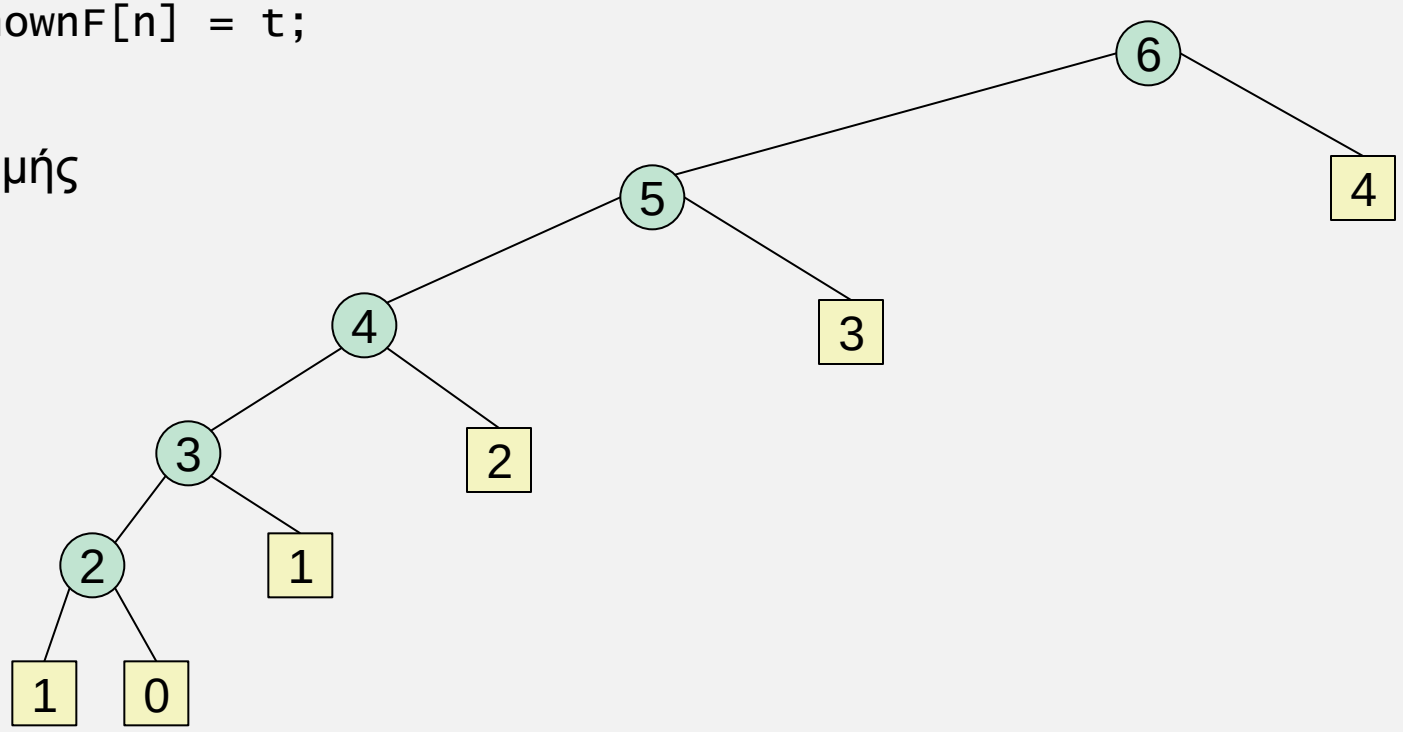
Το αναδρομικό πρόγραμμα αποθηκεύει τις τιμές που υπολογίζει και αποφεύγει τον υπολογισμό ήδη αποθηκευμένων τιμών.

```
int fibonacci(int n)
{
    int t;
    if (knownF[n] != unknown) return knownF[n];
    if (n==0) t = 0;
    if (n==1) t = 1;
    if (n > 1) t = fibonacci(n-1) + fibonacci(n-2);
    return knownF[n] = t;
}
```

Δυναμικός Προγραμματισμός

```
int fibonacci(int n)
{
    int t;
    if (knownF[n] != unknown) return knownF[n];
    if (n==0) t = 0;
    if (n==1) t = 1;
    if (n > 1) t = fibonacci(n-1) + fibonacci(n-2);
    return knownF[n] = t;
}
```

Δένδρο αναδρομής



Δυναμικός Προγραμματισμός

Συνθετικός δυναμικός προγραμματισμός (bottom-up dynamic programming)

Υπολογίζει και αποθηκεύει όλες τις τιμές της συνάρτησης με τη σειρά ξεκινώντας τον υπολογισμό από τη μικρότερη τιμή του ορίσματος.

Αναλυτικός δυναμικός προγραμματισμός (top-down dynamic programming)

Το αναδρομικό πρόγραμμα αποθηκεύει τις τιμές που υπολογίζει και αποφεύγει τον υπολογισμό ήδη αποθηκευμένων τιμών.

Γενικά ο αναλυτικός δυναμικός προγραμματισμός είναι προτιμητέος επειδή:

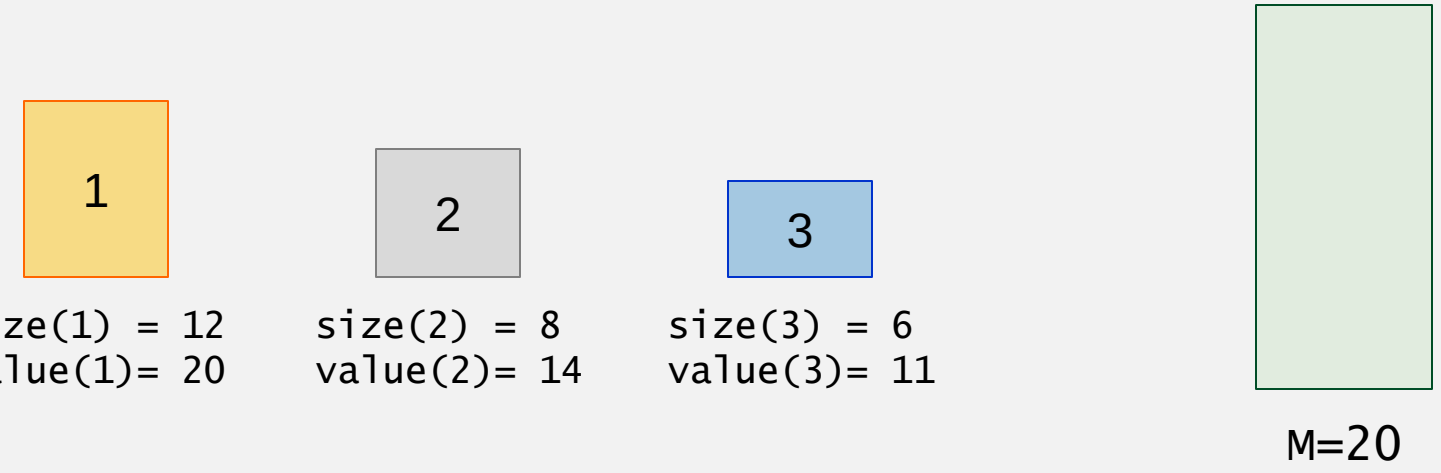
- Παρέχει μηχανικό τρόπο μετασχηματισμού της λύση ενός προβλήματος
- Ρυθμίζει αυτόματα τη σειρά υπολογισμού των υποπροβλημάτων
- Αποφεύγει την επίλυση υποπροβλημάτων που δε χρειάζονται

Δυναμικός Προγραμματισμός

Το πρόβλημα του σακιδίου (knapsack problem)

Έχουμε `N` τύπους αντικειμένων `{1,2,…,N}`. Το αντικείμενο k έχει μέγεθος `size(k)` και αξία `value(k)`.

Ο σκοπός μας είναι να γεμίσουμε με τα αντικείμενα αυτά ένα σακίδιο μεγέθους `M` έτσι ώστε η συνολική αξία των αντικειμένων να είναι η μέγιστη δυνατή.



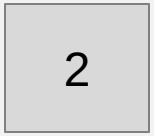
Δυναμικός Προγραμματισμός

Το πρόβλημα του σακιδίου (knapsack problem)

Έχουμε `N` τύπους αντικειμένων `{1,2,…,N}`. Το αντικείμενο k έχει μέγεθος `size(k)` και αξία `value(k)`.

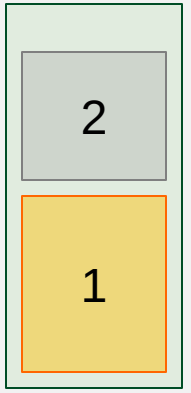
Ο σκοπός μας είναι να γεμίσουμε με τα αντικείμενα αυτά ένα σακίδιο μεγέθους `M` έτσι ώστε η συνολική αξία των αντικειμένων να είναι η μέγιστη δυνατή.

`size(1) = 12`
`value(1)= 20`



`size(2) = 8`
`value(2)= 14`

`size(3) = 6`
`value(3)= 11`



`size  = 20`
`value = 34`

`M=20`

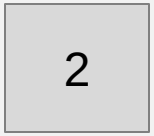
Δυναμικός Προγραμματισμός

Το πρόβλημα του σακιδίου (knapsack problem)

Έχουμε `N` τύπους αντικειμένων `{1,2,…,N}`. Το αντικείμενο k έχει μέγεθος `size(k)` και αξία `value(k)`.

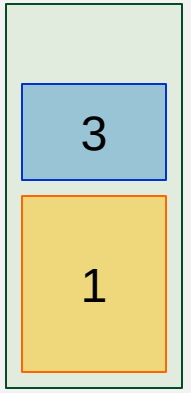
Ο σκοπός μας είναι να γεμίσουμε με τα αντικείμενα αυτά ένα σακίδιο μεγέθους `M` έτσι ώστε η συνολική αξία των αντικειμένων να είναι η μέγιστη δυνατή.

`size(1) = 12`
`value(1)= 20`



`size(2) = 8`
`value(2)= 14`

`size(3) = 6`
`value(3)= 11`



`size  = 18`
`value = 31`

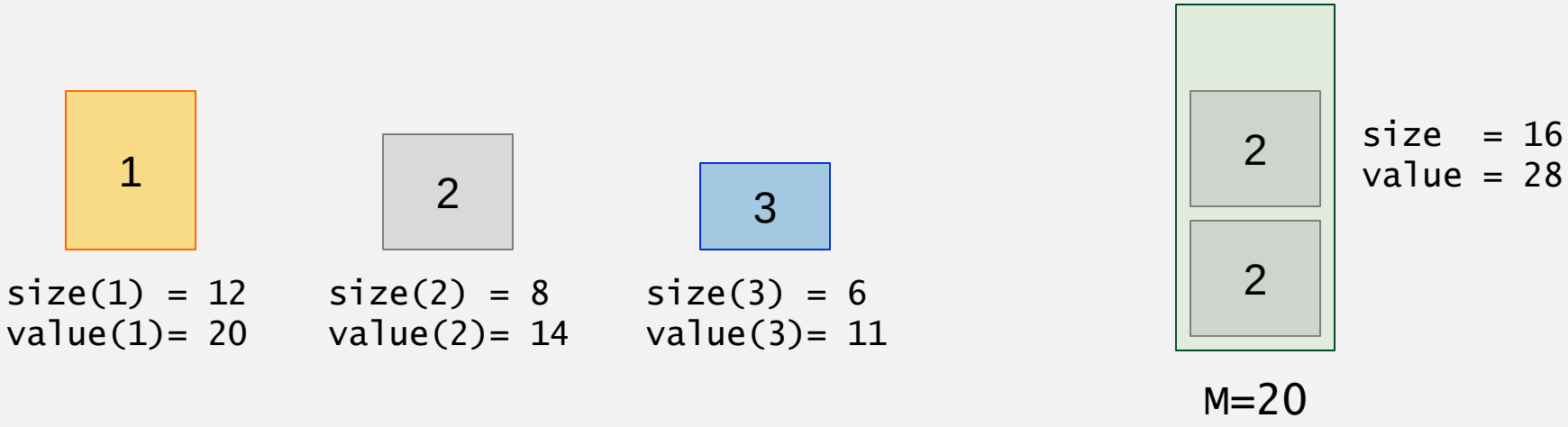
`M=20`

Δυναμικός Προγραμματισμός

Το πρόβλημα του σακιδίου (knapsack problem)

Έχουμε N τύπους αντικειμένων {1,2,…,N}. Το αντικείμενο k έχει μέγεθος size(k) και αξία value(k).

Ο σκοπός μας είναι να γεμίσουμε με τα αντικείμενα αυτά ένα σακίδιο μεγέθους M έτσι ώστε η συνολική αξία των αντικειμένων να είναι η μέγιστη δυνατή.

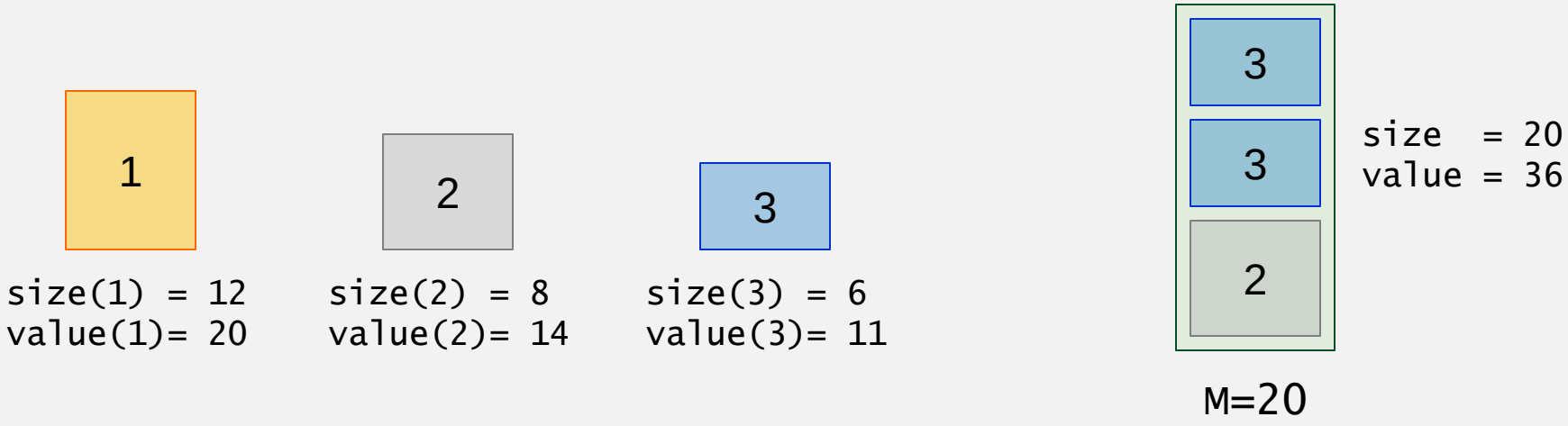


Δυναμικός Προγραμματισμός

Το πρόβλημα του σακιδίου (knapsack problem)

Έχουμε `N` τύπους αντικειμένων `{1,2,…,N}`. Το αντικείμενο k έχει μέγεθος `size(k)` και αξία `value(k)`.

Ο σκοπός μας είναι να γεμίσουμε με τα αντικείμενα αυτά ένα σακίδιο μεγέθους `M` έτσι ώστε η συνολική αξία των αντικειμένων να είναι η μέγιστη δυνατή.

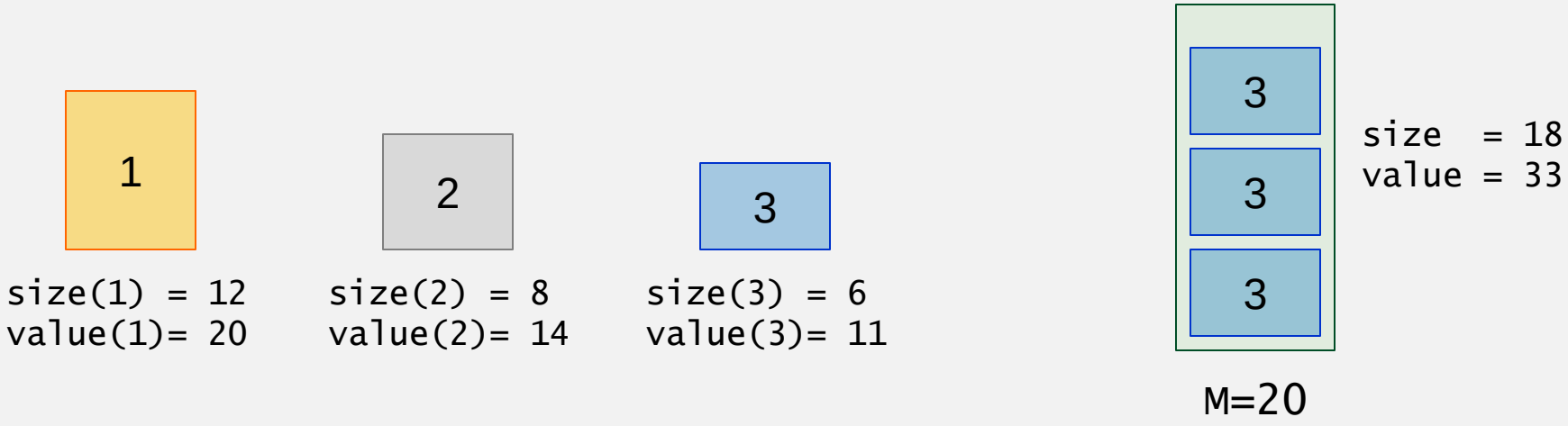


Δυναμικός Προγραμματισμός

Το πρόβλημα του σακιδίου (knapsack problem)

Έχουμε `N` τύπους αντικειμένων `{1,2,…,N}`. Το αντικείμενο k έχει μέγεθος `size(k)` και αξία `value(k)`.

Ο σκοπός μας είναι να γεμίσουμε με τα αντικείμενα αυτά ένα σακίδιο μεγέθους `M` έτσι ώστε η συνολική αξία των αντικειμένων να είναι η μέγιστη δυνατή.

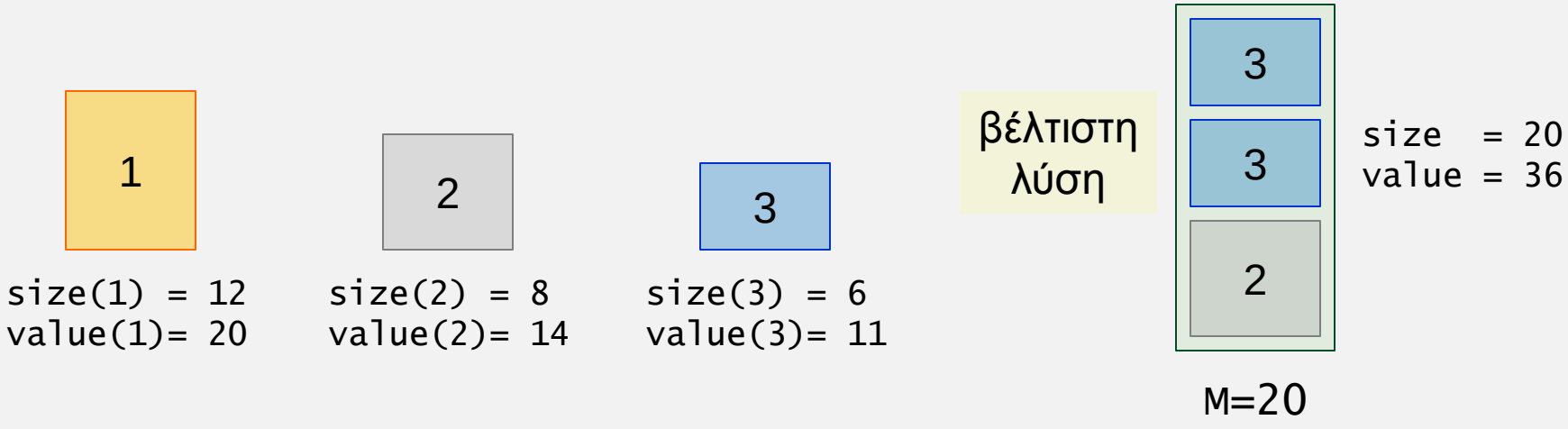


Δυναμικός Προγραμματισμός

Το πρόβλημα του σακιδίου (knapsack problem)

Έχουμε N τύπους αντικειμένων {1,2,…,N}. Το αντικείμενο k έχει μέγεθος size(k) και αξία value(k).

Ο σκοπός μας είναι να γεμίσουμε με τα αντικείμενα αυτά ένα σακίδιο μεγέθους M έτσι ώστε η συνολική αξία των αντικειμένων να είναι η μέγιστη δυνατή.



Δυναμικός Προγραμματισμός

Το πρόβλημα του σακιδίου (knapsack problem)

Έχουμε N τύπους αντικειμένων {1,2,…,N}. Το αντικείμενο k έχει μέγεθος size(k) και αξία value(k).

Ο σκοπός μας είναι να γεμίσουμε με τα αντικείμενα αυτά ένα σακίδιο μεγέθους M έτσι ώστε η συνολική αξία των αντικειμένων να είναι η μέγιστη δυνατή.

```
static class Item { int size; int val; }
Item item[N];

...

int knap(int M)
{
        int i, space, t, max = 0;
        for (i = 0; i < N; i++)
                if ( (space = M - items[i].size) >= 0 )
                        if ( (t = knap(space) + items[i].val > max )
                                max = t;
        return max;
}
```

Δυναμικός Προγραμματισμός

Το πρόβλημα του σακιδίου (knapsack problem)

Έχουμε N τύπους αντικειμένων {1,2,…,N}. Το αντικείμενο k έχει μέγεθος size(k) και αξία value(k).

Ο σκοπός μας είναι να γεμίσουμε με τα αντικείμενα αυτά ένα σακίδιο μεγέθους M έτσι ώστε η συνολική αξία των αντικειμένων να είναι η μέγιστη δυνατή.

```
static class Item { int size; int val; }
Item item[N];

...

int knap(int M)
{
        int i, space, t, max = 0;
        for (i = 0; i < N; i++)
                if ( (space = M - items[i].size) >= 0 )
                        if ( (t = knap(space) + items[i].val > max )
                                max = t;
        return max;
}
```

δοκιμάζει όλους τους εφικτούς συνδυασμούς

εκθετικός χρόνος εκτέλεσης!

Δυναμικός Προγραμματισμός

Το πρόβλημα του σακιδίου (knapsack problem)

Έχουμε N τύπους αντικειμένων {1,2,…,N}. Το αντικείμενο k έχει μέγεθος size(k) και αξία value(k).

Ο σκοπός μας είναι να γεμίσουμε με τα αντικείμενα αυτά ένα σακίδιο μεγέθους M έτσι ώστε η συνολική αξία των αντικειμένων να είναι η μέγιστη δυνατή.

```
int knap(int M)
{
        int i, space, t, maxi, max = 0;
        if ( maxKnown[M] != unknown )  return maxKnown[M];
        for (i = 0; i < N; i++)
                if ( (space = M - items[i].size) >= 0 )
                        if ( (t = knap(space) + items[i].val > max )
                        {       max = t;  maxi = i;     }
        maxKnown[M] = max; itemKnown[M] = items[maxi];
        return max;
}
```

Δυναμικός Προγραμματισμός

Το πρόβλημα του σακιδίου (knapsack problem)

Έχουμε N τύπους αντικειμένων {1,2,…,N}. Το αντικείμενο k έχει μέγεθος size(k) και αξία value(k).

Ο σκοπός μας είναι να γεμίσουμε με τα αντικείμενα αυτά ένα σακίδιο μεγέθους M έτσι ώστε η συνολική αξία των αντικειμένων να είναι η μέγιστη δυνατή.

```
int knap(int M)
{
        int i, space, t, maxi, max = 0;
        if ( maxKnown[M] != unknown )  return maxKnown[M];
        for (i = 0; i < N; i++)
                if ( (space = M - items[i].size) >= 0 )
                        if ( (t = knap(space) + items[i].val > max )
                        {       max = t;  maxi = i;     }
        maxKnown[M] = max; itemKnown[M] = items[maxi];
        return max;
}
```

Εφαρμόσαμε αναλυτικό δυναμικό προγραμματισμό!

Χρόνος εκτέλεσης : $O(M \cdot N)$