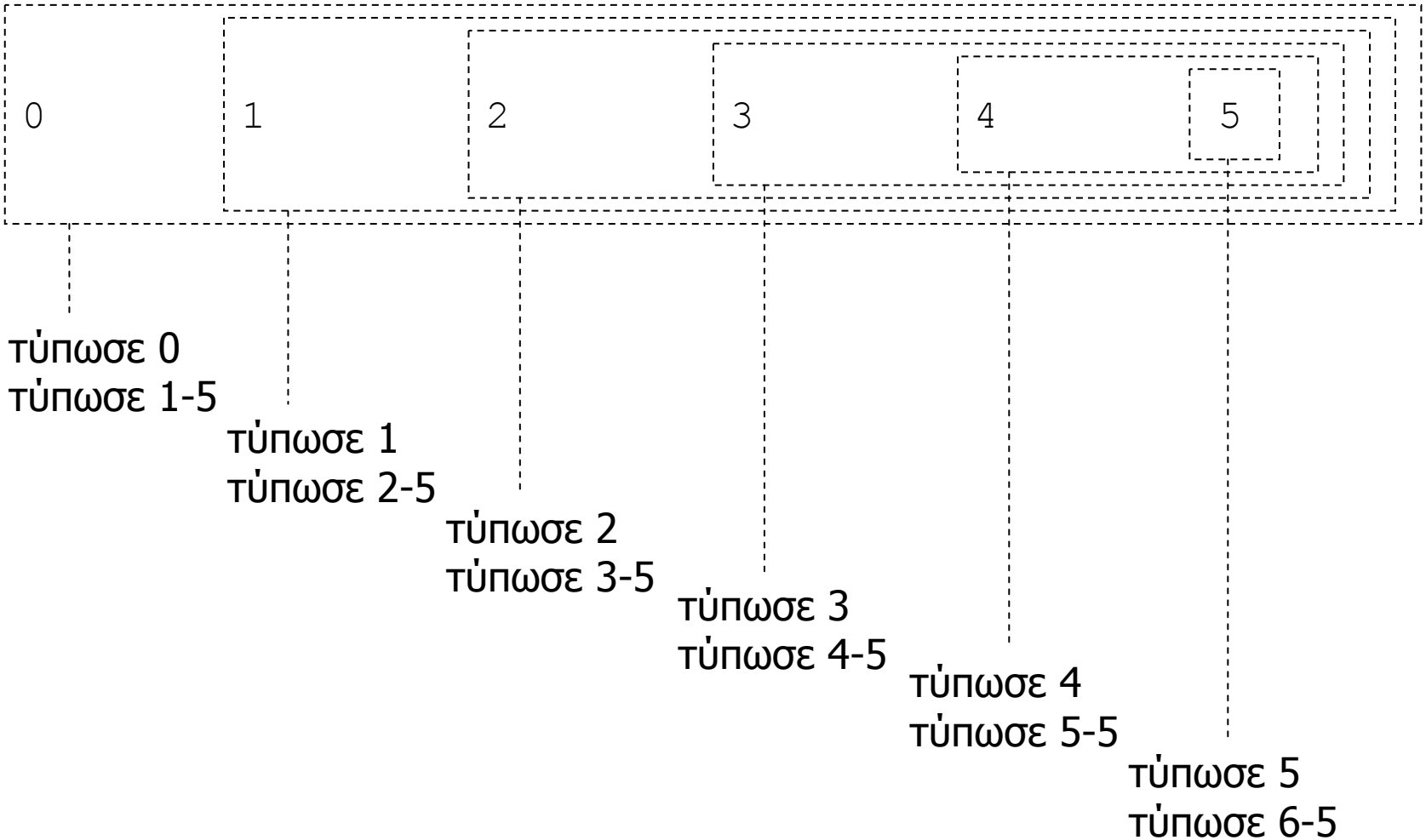
Αναδρομή

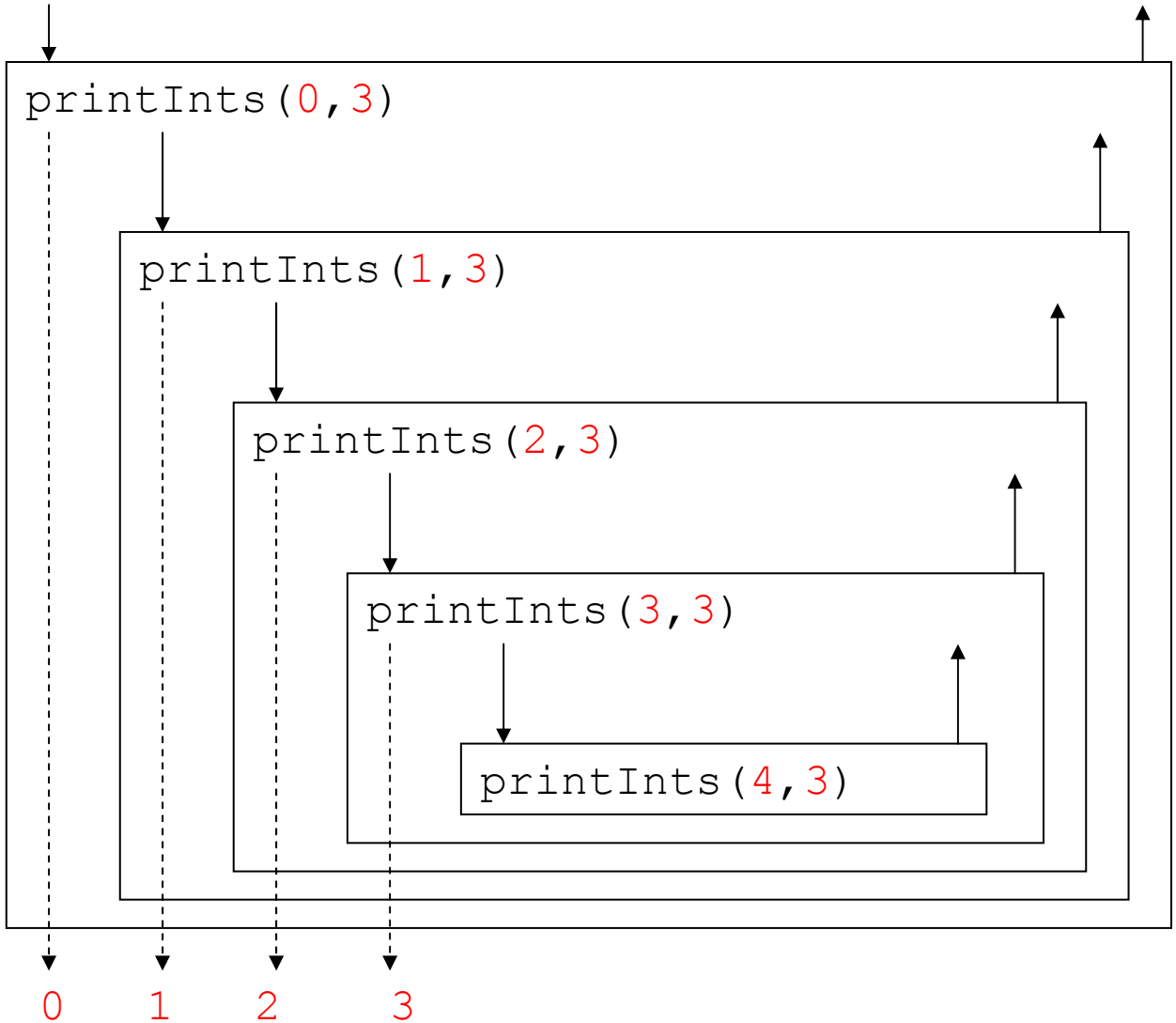
Συναρτήσεις που καλούν τον εαυτό τους

- Μια συνάρτηση ονομάζεται **αναδρομική** όταν καλεί τον **εαυτό** της – άμεσα ή έμμεσα (μέσα από άλλες συναρτήσεις).
- Μια αναδρομική συνάρτηση πρέπει να τερματίζει.
- Η ατέρμονη αναδρομή είναι προγραμματιστικό λάθος (αντίστοιχο με αυτό της ατέρμονης επανάληψης) και οδηγεί σε τερματισμό του προγράμματος, λόγω υπερχείλισης της στοίβας (stack overflow).
- Η αναδρομή μπορεί να θεωρηθεί σαν ένα ξεχωριστό **παράδειγμα προγραμματισμού** καθώς διάφορα (πολύπλοκα) προβλήματα μπορεί να λυθούν με φυσικό τρόπο χρησιμοποιώντας αναδρομή.

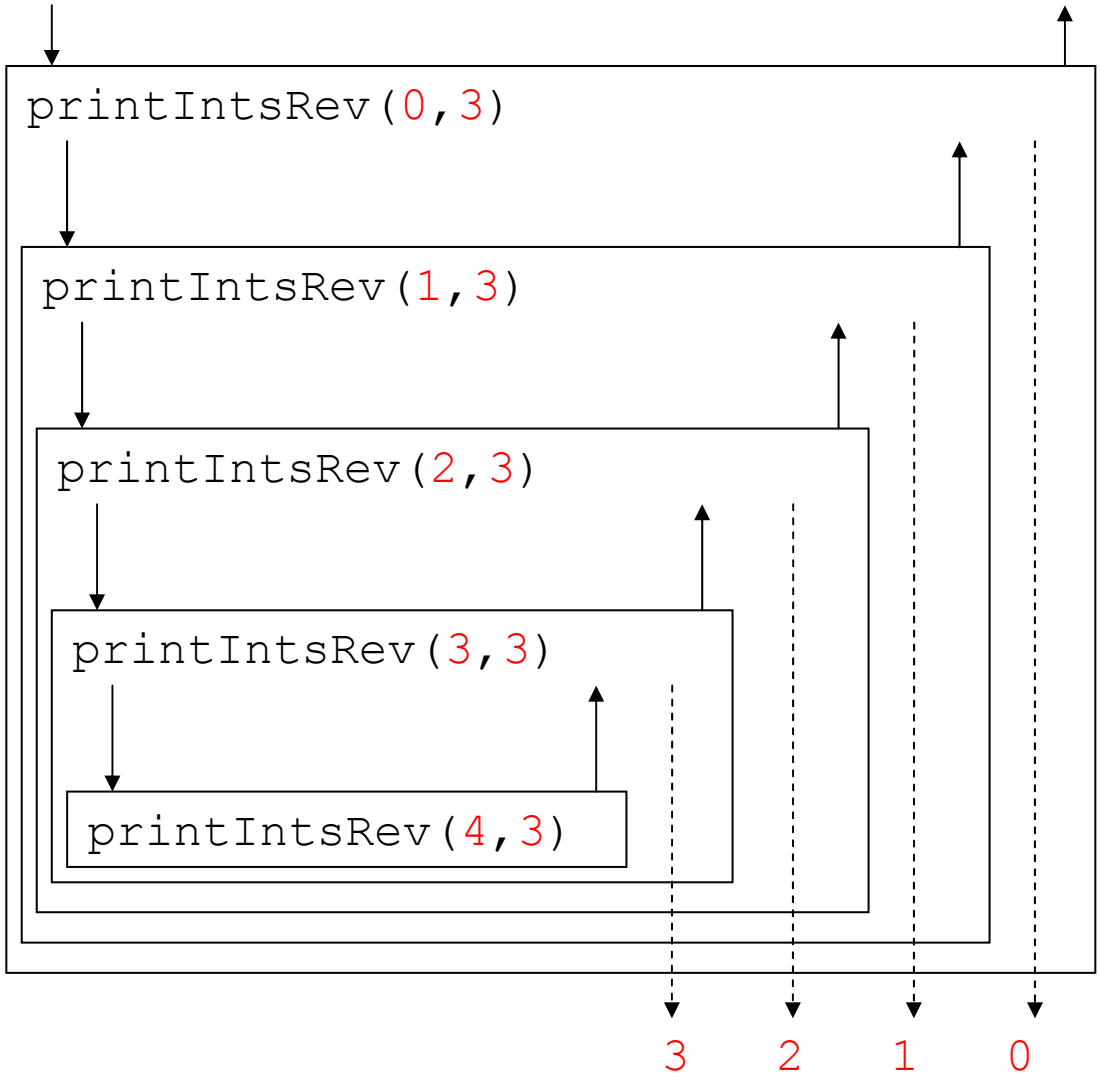
```
/* εκτύπωση τιμών από 0 μέχρι n */

#include <stdio.h>

void printInts(int from, int to) {
  int i;

  for(i=from; i <= to; i++) {
    printf("%d ",i);
  }
}


int main(int argc, char *argv[]) {
  int n;

  printf("enter int: ");
  scanf("%d",&n);
  printInts(0,n);
}
```


0
1
2
3
4
5
τύπωσε 0
τύπωσε 1-5
τύπωσε 1
τύπωσε 2-5
τύπωσε 2
τύπωσε 3-5
τύπωσε 3
τύπωσε 4-5
τύπωσε 4
τύπωσε 5-5
τύπωσε 5
τύπωσε 6-5

```
/* εκτύπωση τιμών από 0 μέχρι n */

#include <stdio.h>

void printInts(int from, int to) {
  if (from <= to) {
    printf("%d ",from);
    printInts(from+1,to);
  }
}

int main(int argc, char *argv[]) {
  int n;

  printf("enter int: ");
  scanf("%d",&n);
  printInts(0,n);
}
```


printInts(0,3)
printInts(1,3)
printInts(2,3)
printInts(3,3)
printInts(4,3)
0
1
2
3

```
/* εκτύπωση τιμών από n μέχρι 0 */

#include <stdio.h>

void printIntsRev(int from, int to) {
  if (from<=to) {
    printIntsRev(from+1,to);
    printf("%d ",from);
  }
}

int main(int argc, char *argv[]) {
  int n;

  printf("enter int: ");
  scanf("%d",&n);
  printIntsRev(0,n);
}
```


printIntsRev(0,3)
printIntsRev(1,3)
printIntsRev(2,3)
printIntsRev(3,3)
printIntsRev(4,3)
3
2
1
0

Αναδρομική σκέψη και λύση προβλημάτων

- Για να λυθεί ένα πρόβλημα με αναδρομή, πρέπει πρώτα να **εκφραστεί** με αναδρομικό τρόπο.
- Κλασική προσέγγιση:

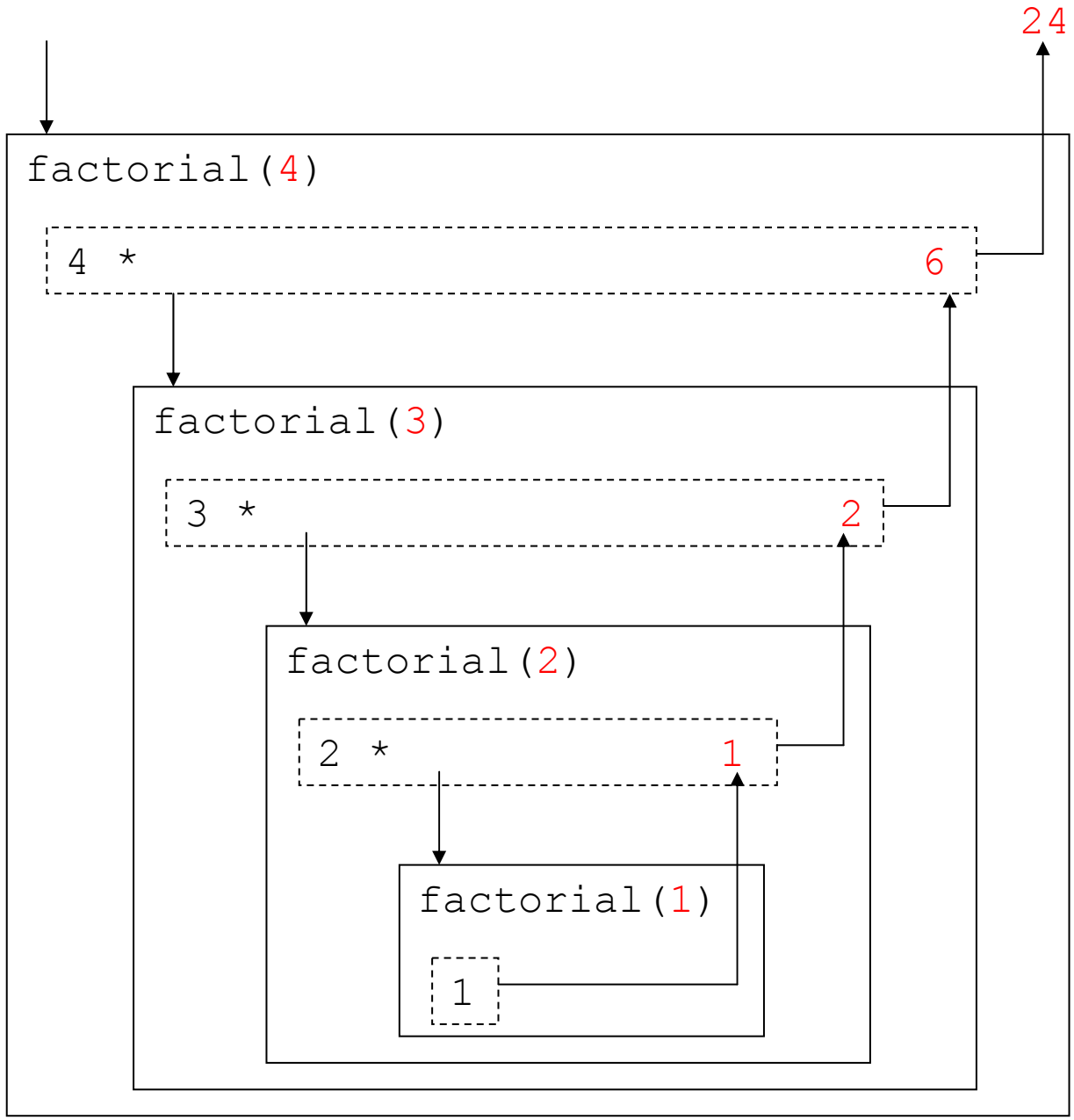
1. Προσπαθούμε να βρούμε την λύση του προβλήματος για την πιο απλή περίπτωση του.
2. Στην συνέχεια, ανάγουμε/κατασκευάζουμε την λύση της (αμέσως) πιο πολύπλοκης περίπτωσης, με βάση την λύση της πιο απλής περίπτωσης (αναδρομή).
3. Αν τα (1) και (2) γίνουν «σωστά», έχουμε **ήδη** κατασκευάσει την λύση στο πρόβλημα μας!

- Η συνθήκη τερματισμού είναι πολλές φορές το κλειδί στην ανεύρεση της πιο απλής περίπτωσης.

Υπολογισμός x!

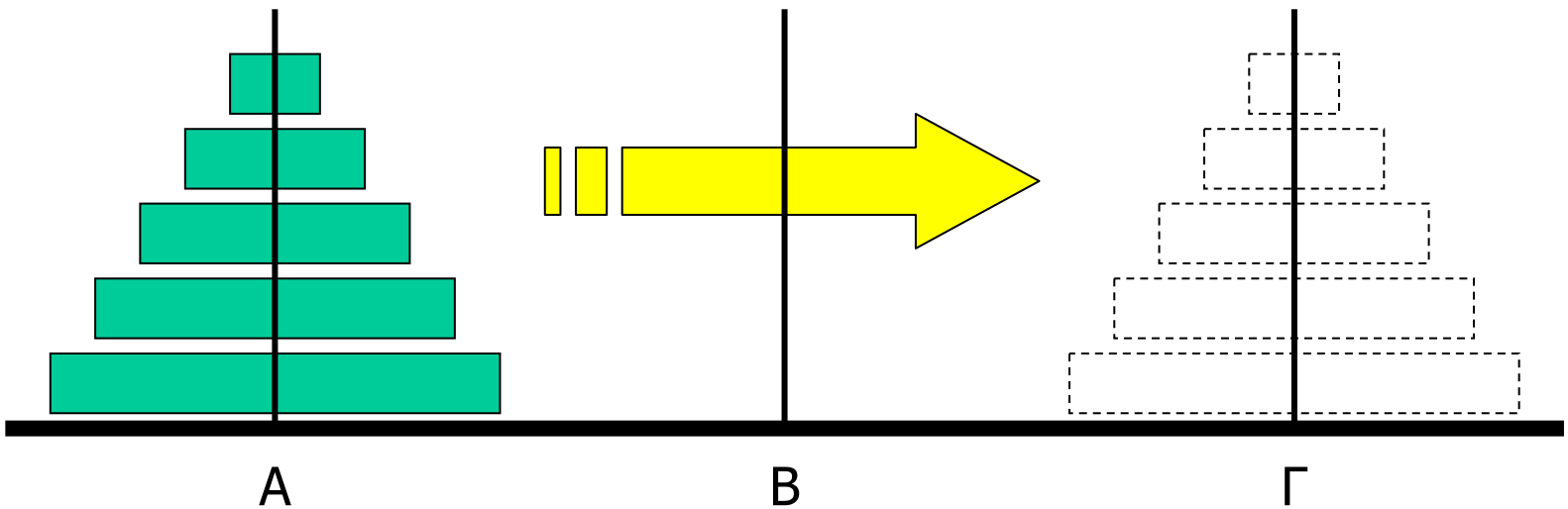
- Επιθυμούμε να υπολογίζουμε την έκφραση
  `x!  ==  1*2*3*…*(x-1)*x`
- Περίπτωση τερματισμού
  `x==0` ή `x==1`: επιστρέφεται `1`
- Γενική περίπτωση
  `x > 1`: επιστρέφεται `x*(x-1)!`
- Η «λύση» της γενικής περίπτωσης μπορεί να κατασκευαστεί με βάση την λύση του **ίδιου** προβλήματος, σε **μικρότερη** «κλίμακα».

```
/* υπολογισμός n! */

int factorial(int n) {

  if (n<=1) {
    return(1);
  }
  else {
    return(n*factorial(n-1));
  }

}
```

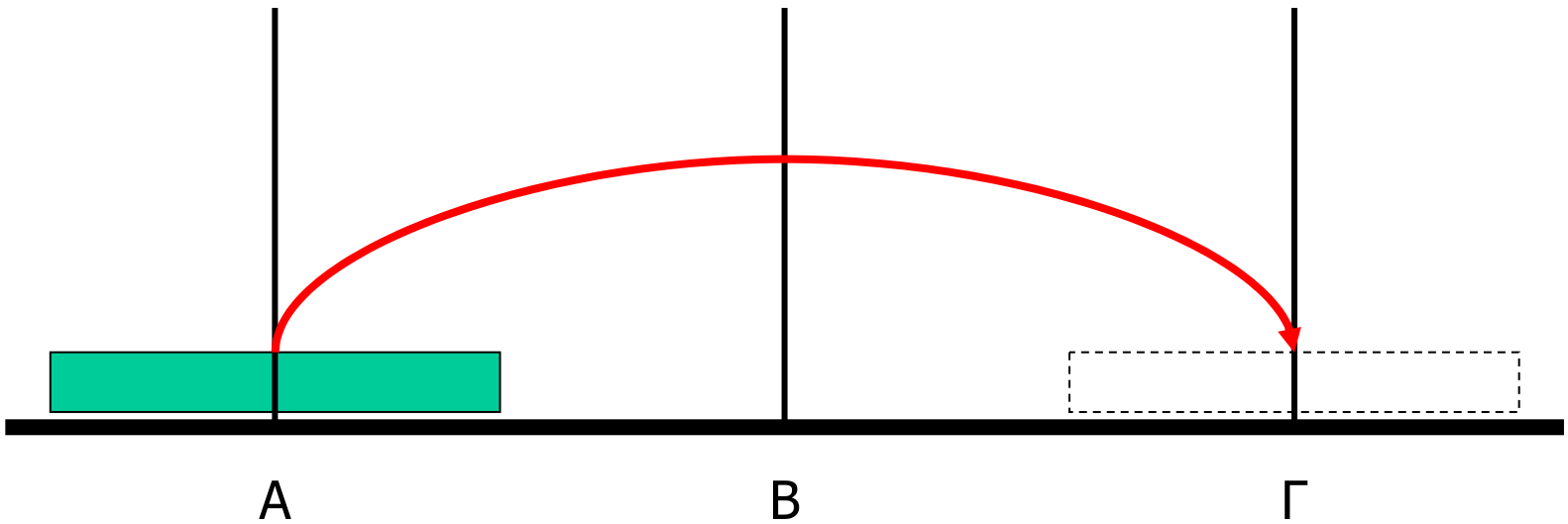

24
factorial(4)
4 *
6
factorial(3)
3 *
2
factorial(2)
2 *
1
factorial(1)
1

Οι πύργοι του Hanoi

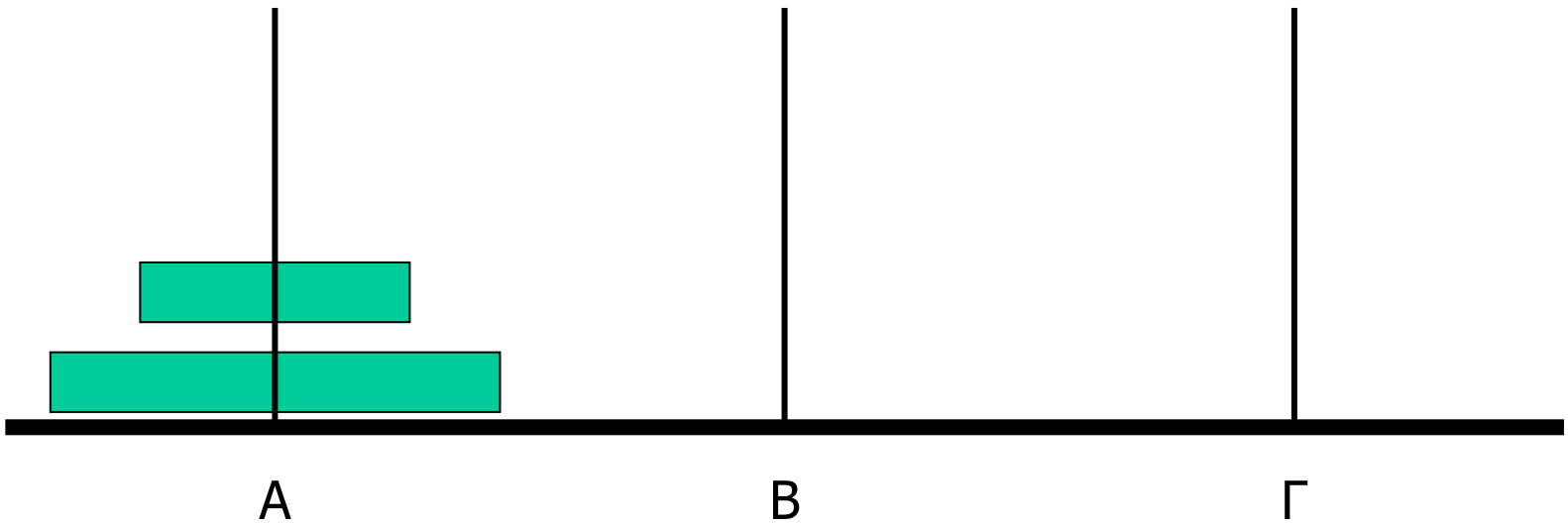
- Δίνεται ένα χώρος με τρείς θέσεις αποθήκευσης.
- Δίνεται μια στοίβα από Ν πλάκες σε φθίνων μέγεθος, σε μια από τις τρείς θέσεις.
- Ζητούμενο: μετακίνησε την στοίβα σε μια άλλη θέση χωρίς ποτέ να βάλεις μια μεγαλύτερη πλάκα πάνω από μια μικρότερη πλάκα.



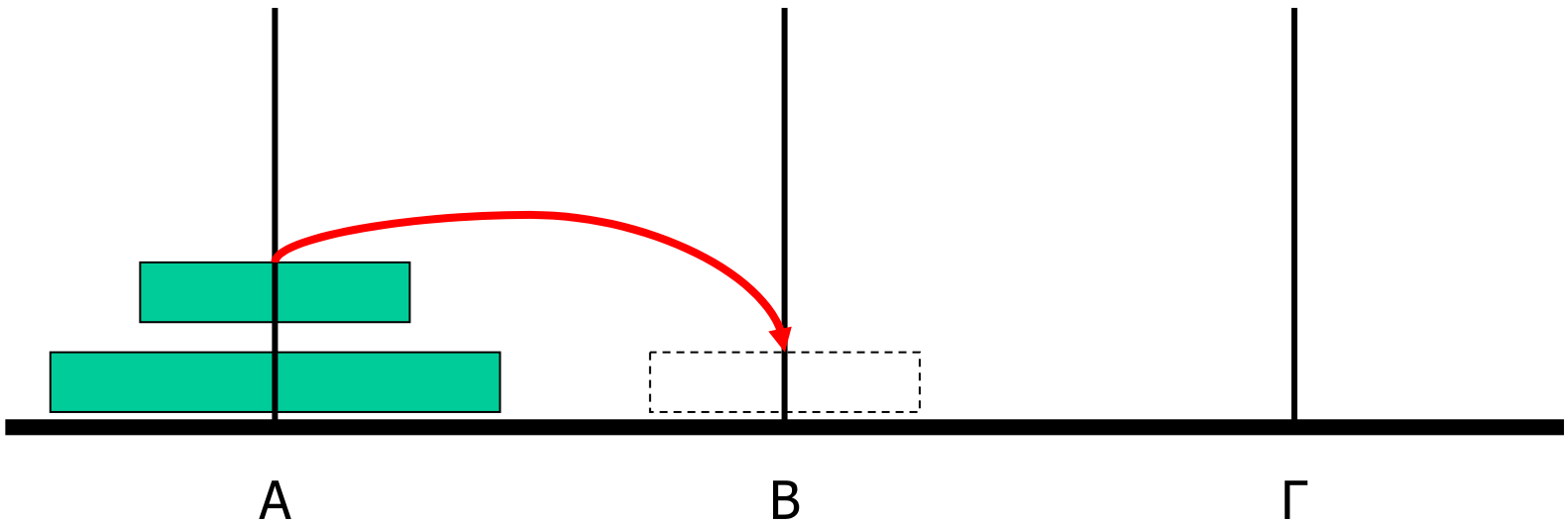
Λύση με 1 πλάκα



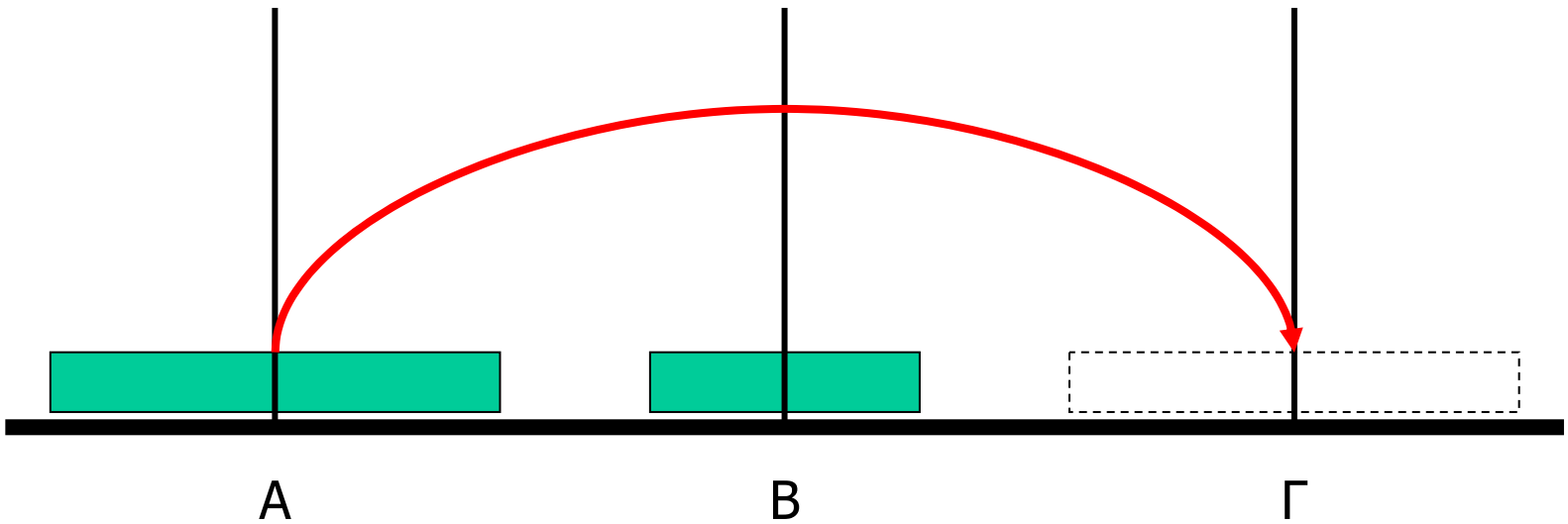
Λύση με 2 πλάκες



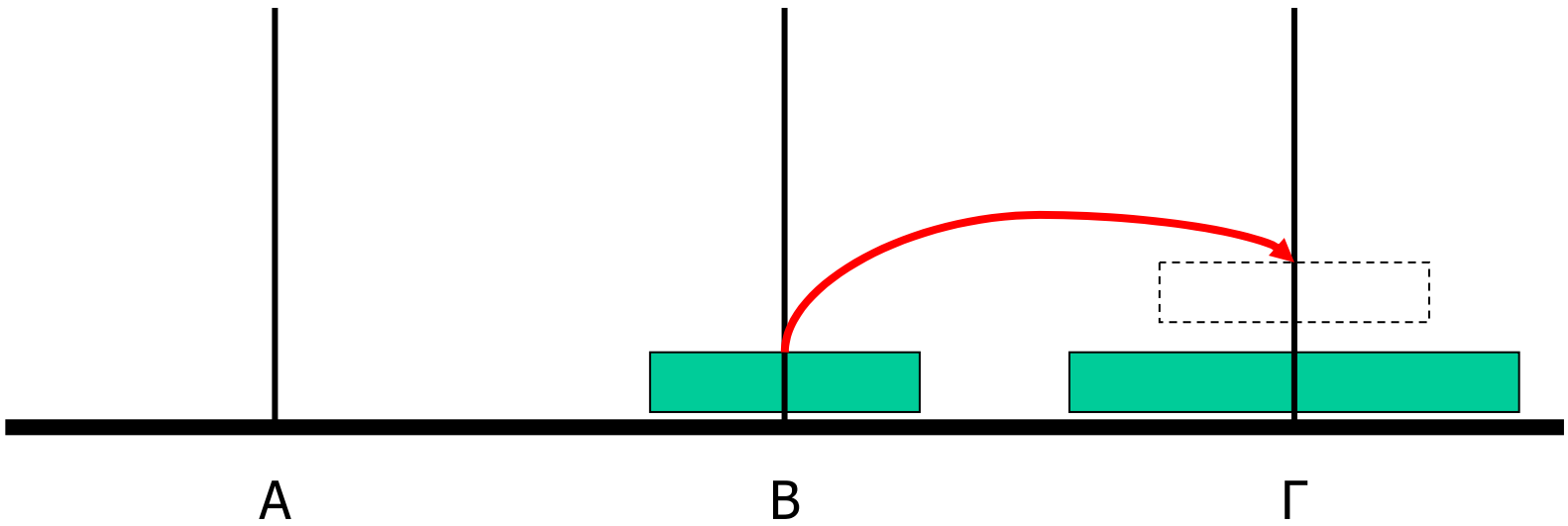
Λύση με 2 πλάκες



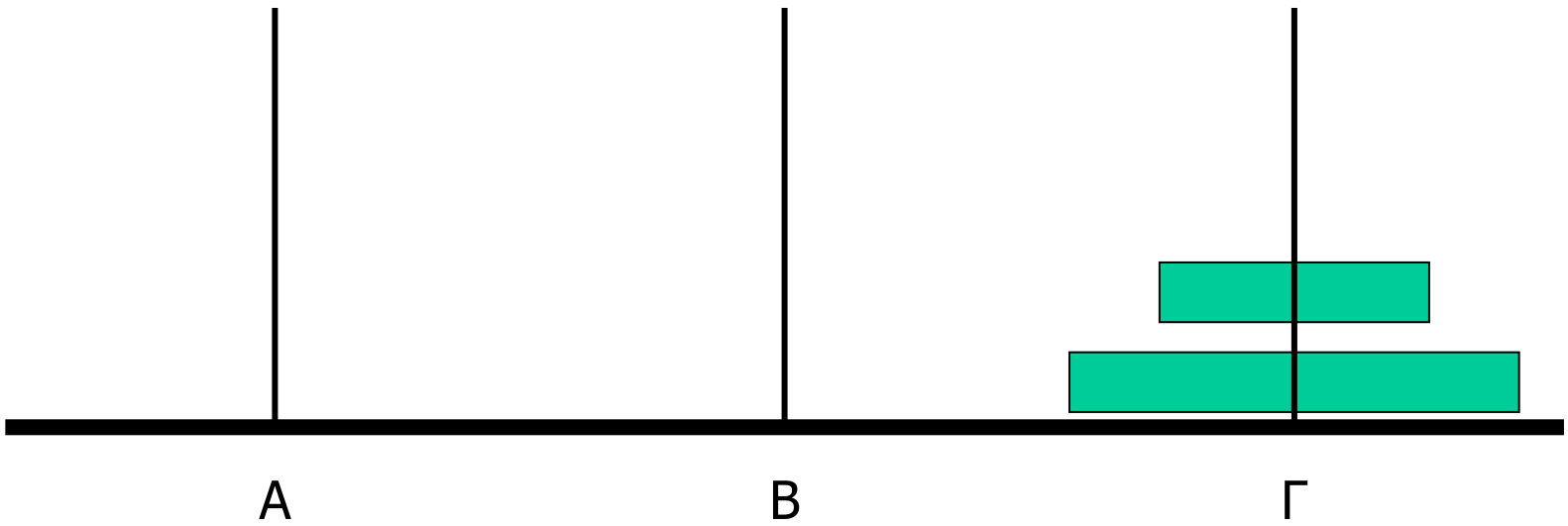
Λύση με 2 πλάκες



Λύση με 2 πλάκες

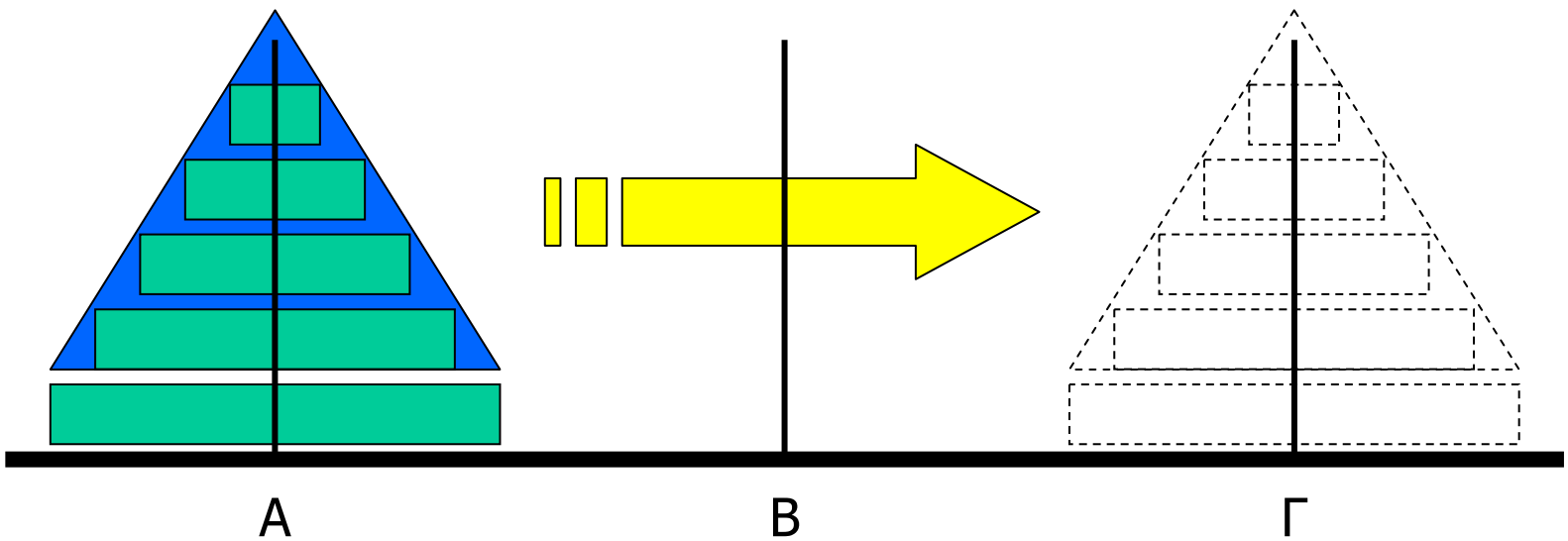


Λύση με 2 πλάκες



Λύση για Ν πλάκες

- Θεωρούμε ότι οι **μικρότερες Ν-1** πλάκες είναι **μια** πλάκα, η μετακίνηση της οποίας μπορεί να γίνει πλέον αναδρομικά (Ν-1 πλάκες πρέπει να μετακινηθούν από μια θέση σε μια άλλη μέσω μιας τρίτης).



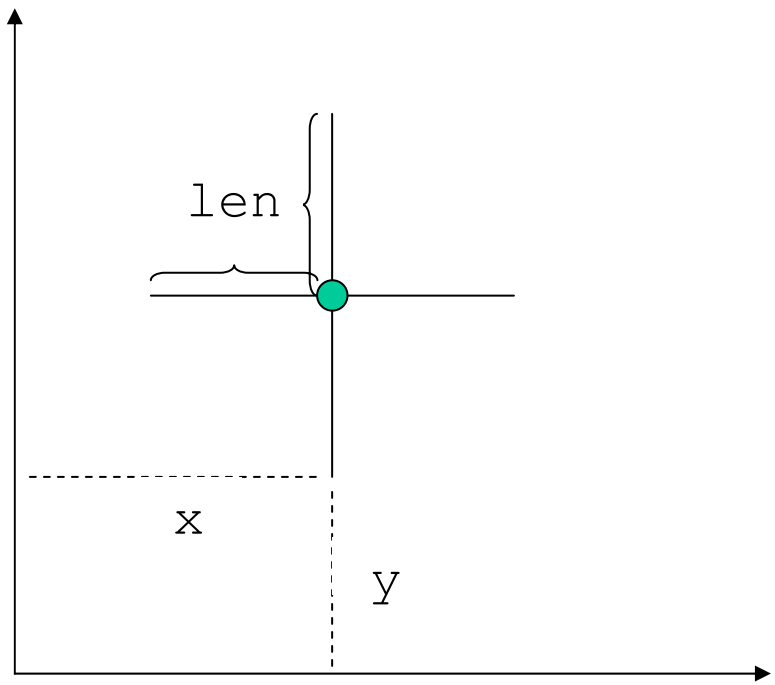
Μοντελοποίηση της λύσης

- Μετακίνησε Ν πλάκες από την θέση Α μέσω της θέσης Β στην θέση Γ:
- Απλή περίπτωση: αν Ν==1 τότε μεταφέρουμε την πλάκα από την θέση Α απ' ευθείας στην θέση Γ.
- Πολύπλοκη περίπτωση: αν Ν>1, τότε
  (α) μεταφέρουμε τις πάνω Ν-1 πλάκες από την θέση Α μέσω της θέσης Γ στην θέση Β,
  (β) μεταφέρουμε την (μια) πλάκα που απέμεινε από την θέση Α απ' ευθείας στην θέση Γ, και
  (γ) μεταφέρουμε τις Ν-1 πλάκες από την θέση Β μέσω της θέσης Α στην θέση Γ.

```
void moveTower(int a, int b, int c, int n) {
  if (n==1) {
    movePiece(a,c);
  }
  else {
    moveTower(a,c,b,n-1);
    movePiece(a,c);
    moveTower(b,a,c,n-1);
  }
}
```

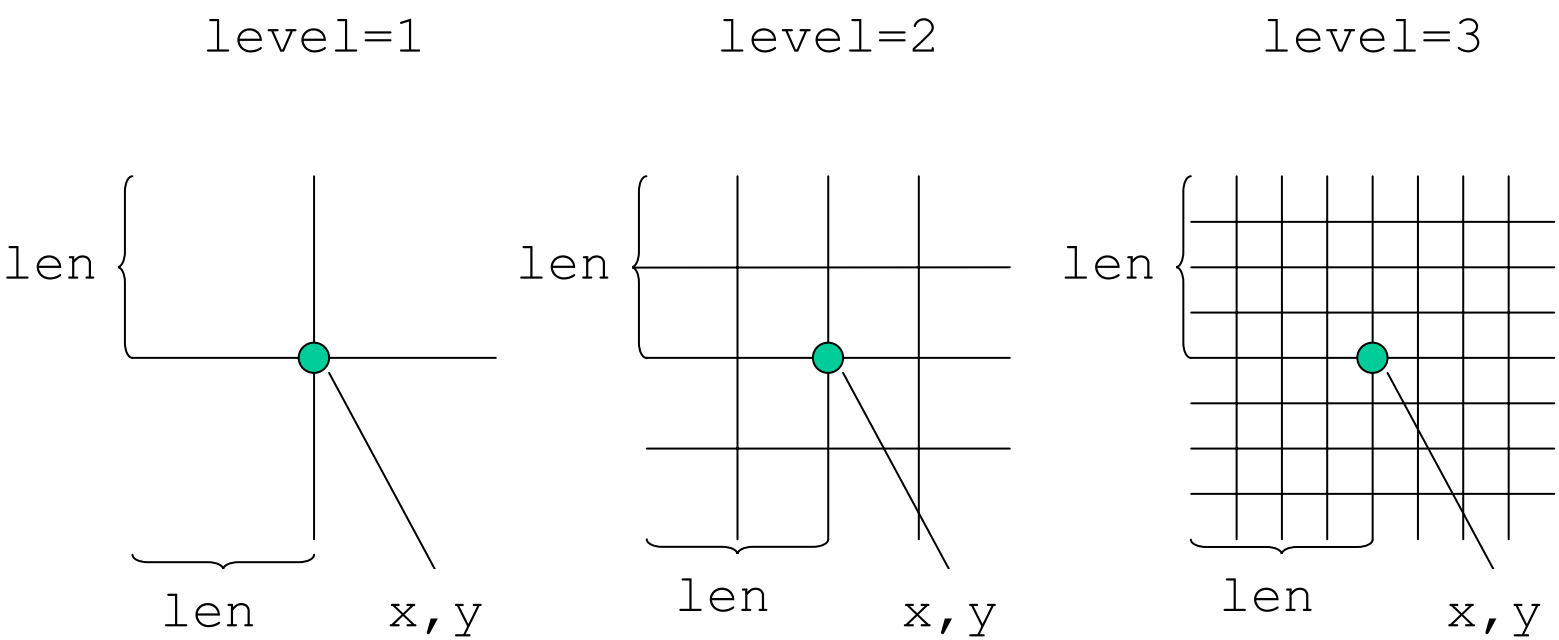
Σχεδίαση αναδρομικών σχημάτων

- Η `draw_cross(int x, int y, int len)` ζωγραφίζει ένα σταυρό, με κέντρο το σημείο (`x,y`) και μήκος `2*len`.



Σχεδίαση αναδρομικών σχημάτων

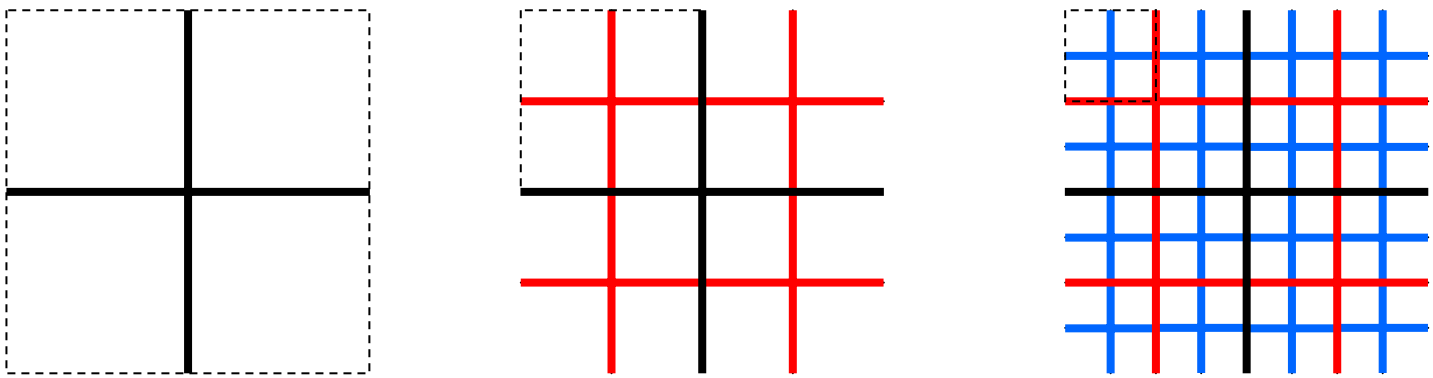
- Ζητούμενο: υλοποιήστε μια συνάρτηση που σχεδιάζει τα εξής σχήματα με βάση τις ακέραιες παραμέτρους `x`, `y`, `len` και `level`.



Σχεδίαση αναδρομικών σχημάτων

- Παρατήρηση: το ίδιο σχέδιο επαναλαμβάνεται σε μικρότερη κλίμακα σε όλα τα τεταρτημόρια του σχεδίου της αμέσως μεγαλύτερης κλίμακας.

`level=1` `level=2` `level=3`



```
void draw_grid(int x, int y, int len, int level) {
  if (level==1) {
    draw_cross(x,y,len);
  }
  else {
    draw_cross(x,y,len);
    draw_drid(x-len/2,y+len/2,len/2,level-1);
    draw_grid(x-len/2,y-len/2,len/2,level-1);
    draw_grid(x+len/2,y-len/2,len/2,level-1);
    draw_grid(x+len/2,y+len/2,len/2,level-1);
  }
}
```

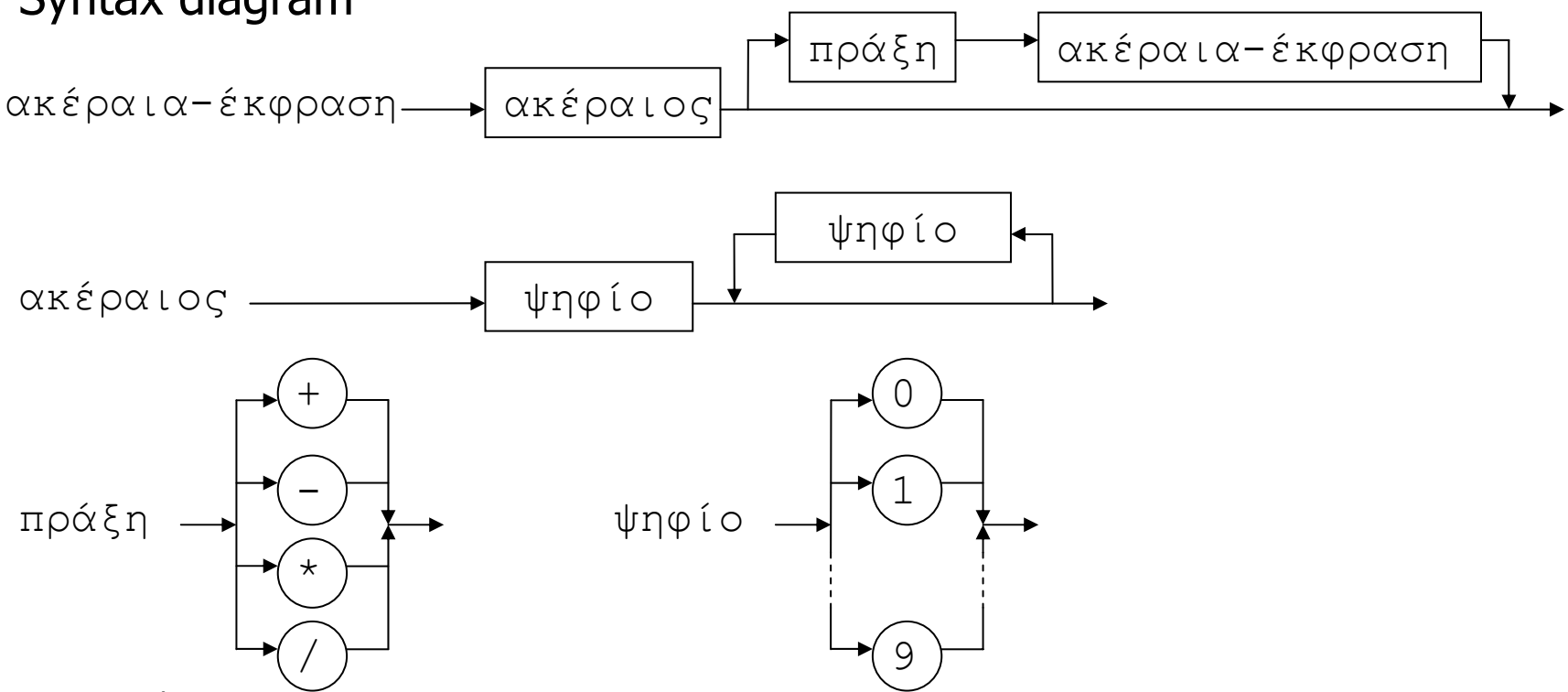
Έλεγχος και αποτίμηση εκφράσεων

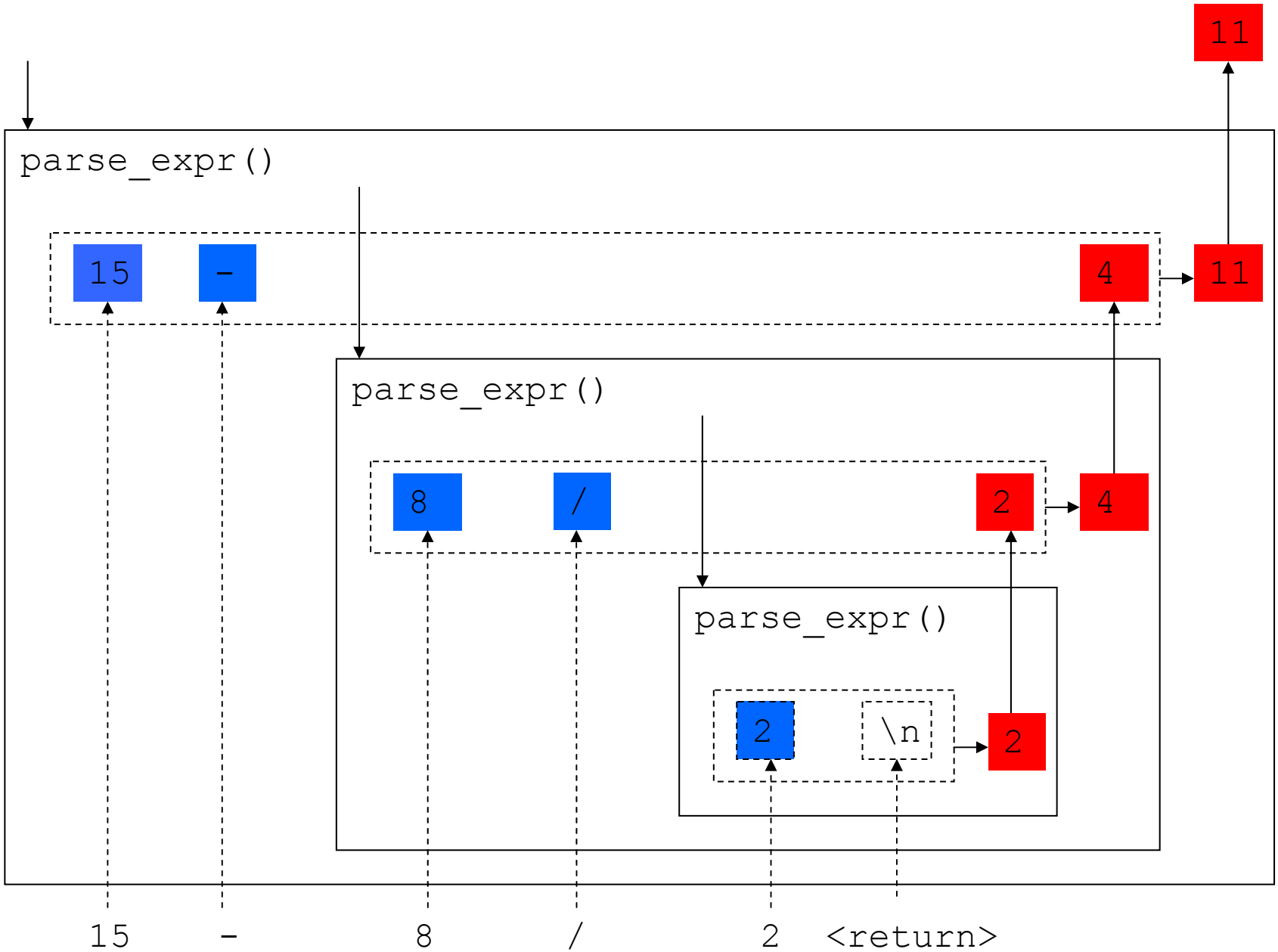
- Δίνεται το συντακτικό των επιτρεπτών αριθμητικών εκφράσεων (προτάσεων) με βάση EBNF.
- Δεν υπάρχουν συγκεκριμένες προτεραιότητες, και η αποτίμηση γίνεται **από δεξιά προς τα αριστερά**.
- Ζητούμενο: να υλοποιηθεί μια συνάρτηση που ελέγχει κατά πόσο η έκφραση που δίνεται από τον χρήστη σαν είσοδος αντιστοιχεί σε μια επιτρεπτή πρόταση, και αν ναι να υπολογίζει και να επιστρέφει το αποτέλεσμα.
- Επιτρέπονται κενά εκτός ανάμεσα στα ψηφία ενός αριθμού, και το τέλος της έκφρασης σηματοδοτείται από τον χαρακτήρα `'\n'`.

EBNF

```
ακέραια-έκφραση=ακέραιος[πράξη ακέραια-έκφραση]
ακέραιος=νούμερο{νούμερο}
νούμερο='0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
πράξη=='+'|'-'|'*'|'/'
```

Syntax diagram




11
parse_expr()
15
-
4
11
parse_expr()
8
/
2
4
parse_expr()
2
\n
2
15
-
8
/
2
<return>

Αναδρομή και επανάληψη

- Οι περισσότερες μορφές επανάληψης μπορεί να εκφραστούν με αναδρομή, αρκεί η συνάρτηση να περνά στον εαυτό της τις κατάλληλες παραμέτρους.
- Αντίθετα, υπάρχουν αρκετές μορφές αναδρομής που είναι δύσκολο να εκφραστούν με επανάληψη.
- Το κύριο πλεονέκτημα της αναδρομής είναι ότι η συνάρτηση κρατά **δυναμική ενδιάμεση** κατάσταση μέσα από τις τοπικές μεταβλητές (πλαίσιο εκτέλεσης) κάθε (αλυσιδωτής) αναδρομικής κλήσης.
- Παρόμοιο αποτέλεσμα μπορεί να επιτευχθεί και με συμβατικό τρόπο, αλλά η διαχείριση της κατάστασης πρέπει να γίνει από τον ίδιο τον προγραμμάτιστή (συνήθως χρησιμοποιώντας δυναμική μνήμη).

Συνδυαστικά προβλήματα

Επίλυση συνδυαστικών προβλημάτων

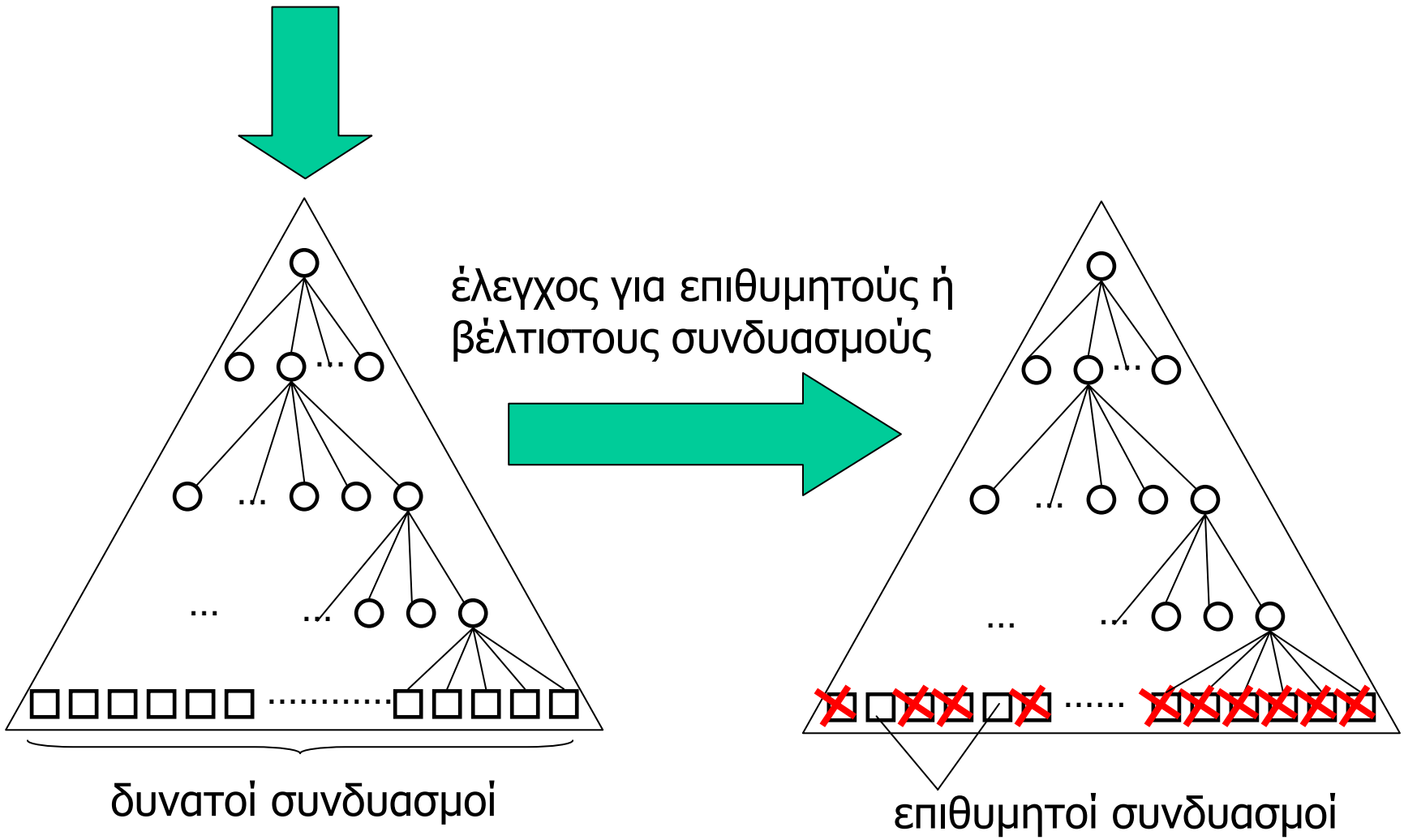
- Συχνά ένα πρόβλημα μπορεί να εκφραστεί ως μια «αναζήτηση» ενός επιτρεπτού συνδυασμού κάποιων μεταβλητών – μέσα από όλους τους δυνατούς συνδυασμούς που μπορεί να γίνουν.
- Κάποιες φορές, το ζητούμενο είναι να βρεθεί ο βέλτιστος συνδυασμός, που ελαχιστοποιεί ή μεγιστοποιεί κάποια συνάρτηση των μεταβλητών.
- Λύση με αναδρομή:

1. Κατασκευάζουμε το σύνολο των συνδυασμών αναδρομικά (με διεξοδική «αναζήτηση» σε βάθος).
2. Ελέγχουμε κάθε συνδυασμό για το κατά πόσο είναι επιτρεπτός ή/και καλύτερος σε σχέση με τους συνδυασμούς που έχουν κατασκευαστεί / εξεταστεί.

Δέντρα συνδυασμών και αποφάσεων

- Όλοι οι δυνατοί συνδυασμοί (μεταβλητών) μπορεί να μοντελοποιηθούν (αναπαρασταθούν) ως ένα δέντρο.
- Κάθε κόμβος του δέντρου σε επίπεδο L αντιστοιχεί σε ένα συγκεκριμένο συνδυασμό επιλογών (τιμών) για L διαφορετικές μεταβλητές.
- Ξεκινώντας από ένα κόμβο σε επίπεδο L, επιλέγουμε την επόμενη μεταβλητή, και για κάθε τιμή που αυτή μπορεί να λάβει κατασκευάσουμε ένα κόμβο παιδί σε επίπεδο L+1, κλπ.
- Ένας κόμβος αποτελεί «φύλλο» όταν δεν υπάρχουν άλλες «ελεύθερες» μεταβλητές –όλες έχουν λάβει συγκεκριμένες τιμές.

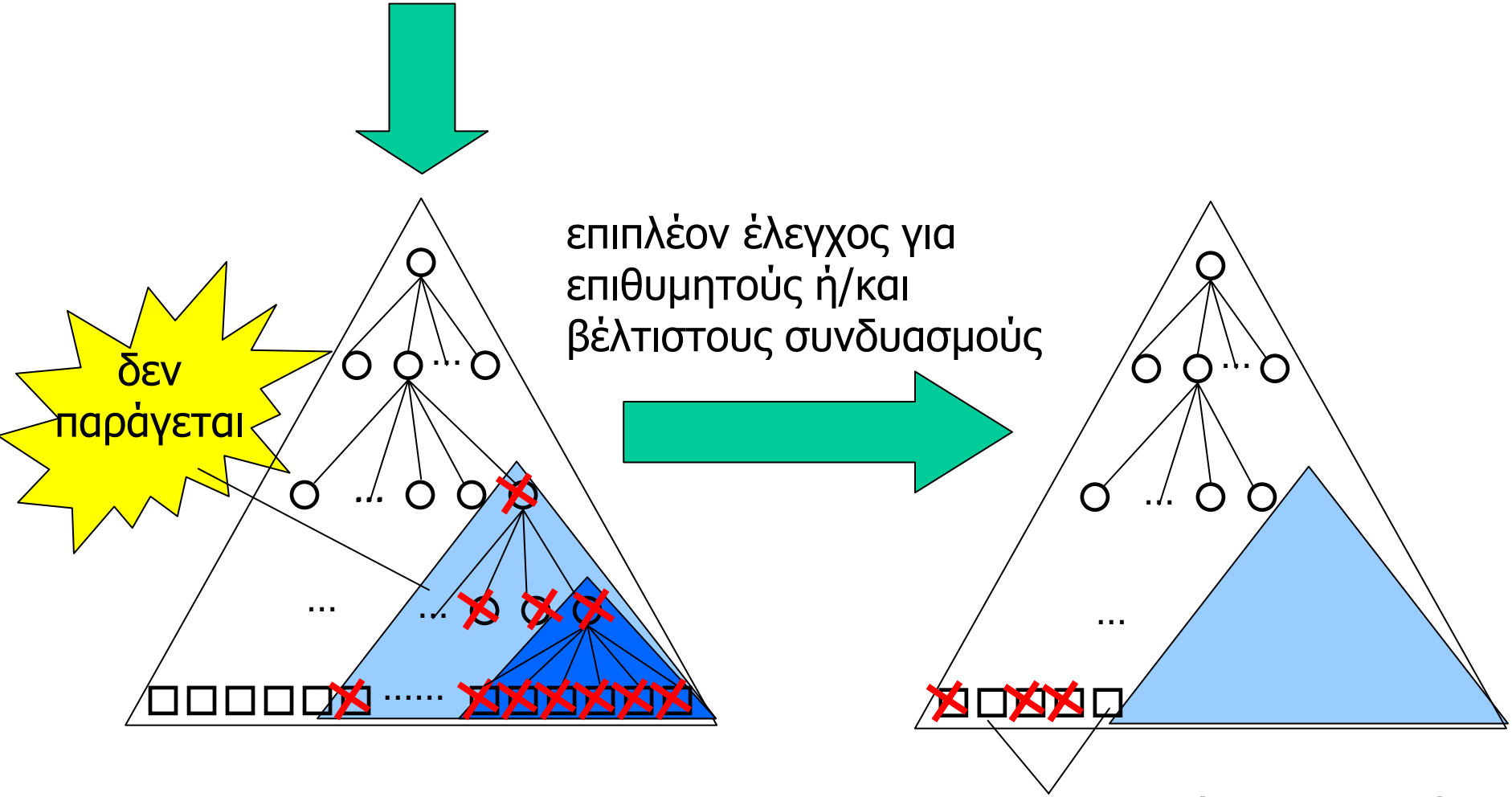
(αναδρομική) παραγωγή όλων
των δυνατών συνδυασμών



Αποφυγή της συνδυαστικής έκρηξης

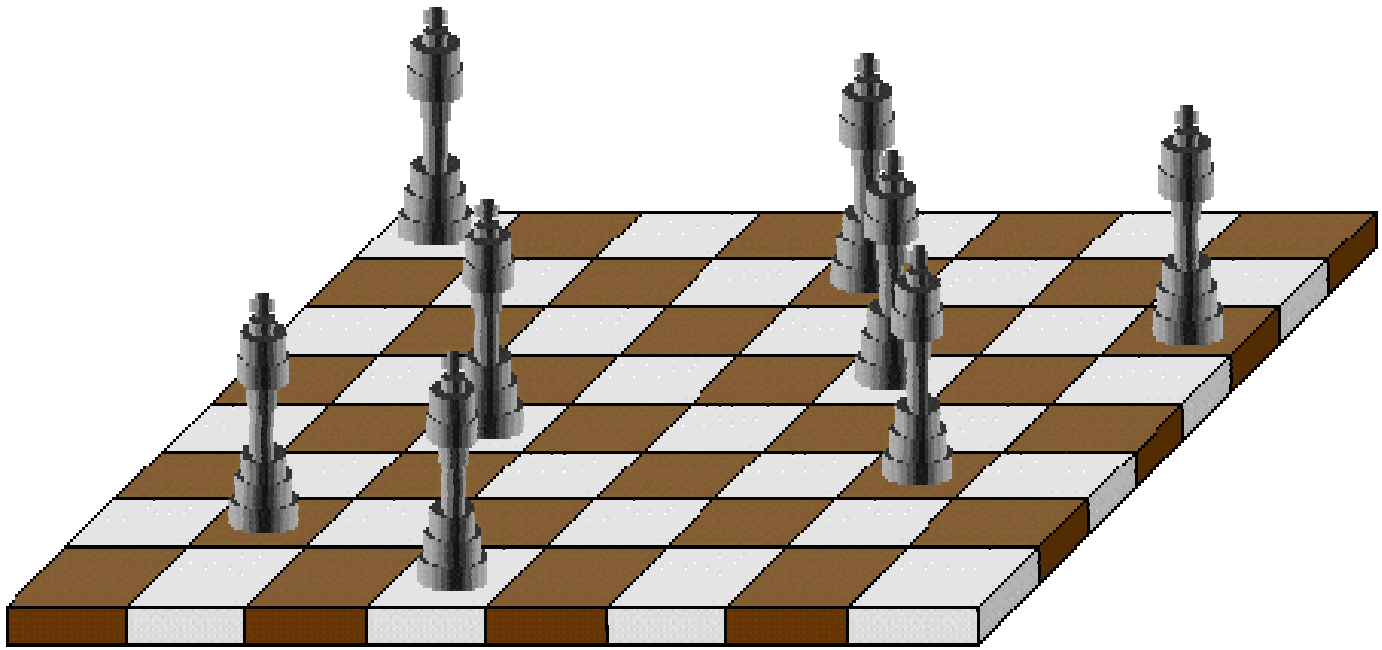
- Ο αριθμός των μεταβλητών μπορεί να είναι μεγάλος και οι πιθανές τιμές για κάθε μεταβλητή πολλές.
- Η παραγωγή όλων των δυνατών συνδυασμών απαιτεί πόρους (μνήμη, χρόνος) – μπορεί να είναι **πρακτικά ανέφικτη** (ακόμα και με τη σημερινή τεχνολογία).
- Επιπλέον, οι περισσότεροι από τους συνδυασμούς που παράγονται «στα τυφλά» είναι καταδικασμένοι **εκ των προτέρων** (πολύ πριν δημιουργηθούν οι τερματικοί κόμβοι «φύλλα» του δέντρου) να μην επιλεγούν ποτέ ως επιθυμητοί (ή βέλτιστοι).
- Βελτιστοποίηση: πρόωρος **αποκλεισμός** ολόκληρων υποδέντρων **από την ρίζα** τους χωρίς υπολογισμό των αντίστοιχων συνδυασμών (branch and bound).

παραγωγή των συνδυασμών
με **αποκλεισμό** όλων των
«αδιάφορων» υποδέντρων

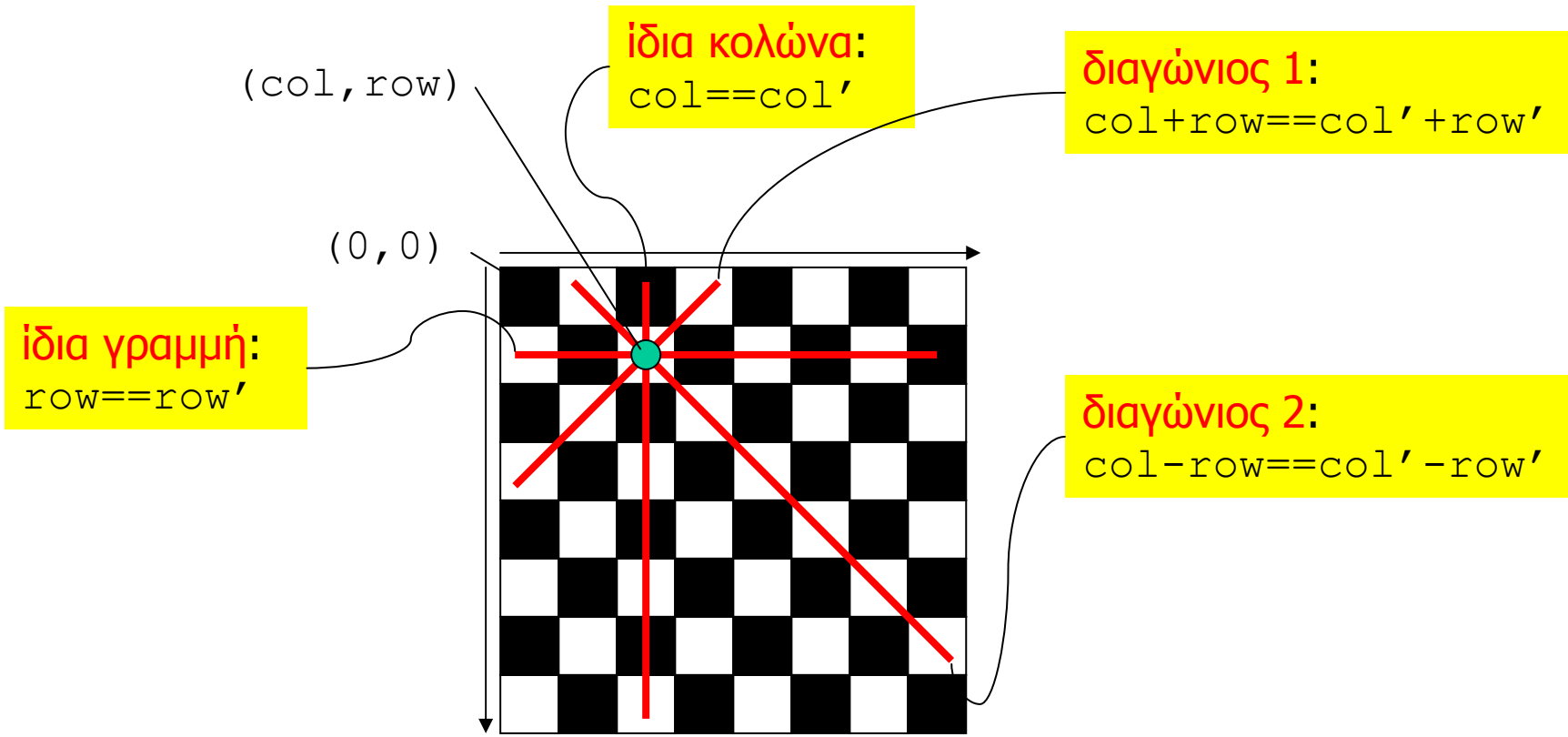


Απλό παράδειγμα - 8 Queens

- Ζητούμενο: να βρεθεί λύση στο εξής πρόβλημα: **να τοποθετηθούν 8 ντάμες σε μια σκακιέρα έτσι ώστε να μην απειλούνται μεταξύ τους**



Εντοπισμός σύγκρουσης




```
int check(int col1,int row1,int col2,int row2) {
  return((col1!=col2) && (row1!=row2) &&
         (col1+row1!=col2+row2) && (col1-row1!=col2-row2));
}
```

Δύο προσεγγίσεις

Προσέγγιση 1 (brute force):

1. Δημιουργούμε **όλους** τους συνδυασμούς θέσεων.
2. Διαγράφουμε τους συνδυασμούς όπου δύο ή περισσότερες ντάμες απειλούν η μια την άλλη.
3. Οι συνδυασμοί που απομένουν είναι αποδεκτοί.

Προσέγγιση 2 (b&b):

1. Για κάθε νέα ντάμα που τοποθετούμε, ελέγχουμε **προκαταβολικά** κατά πόσο απειλεί μια από τις ντάμες που έχουν **ήδη** τοποθετηθεί στη σκακιέρα.
2. Σε αυτή τη περίπτωση, τερματίζουμε την αναδρομή (όλοι οι περαιτέρω συνδυασμοί είναι μη αποδεκτοί).
3. Οι συνδυασμοί που μένουν είναι οι επιθυμητοί.

αριθμός στήλης για τις ντάμες 0<=i<n που έχουν τοποθετηθεί μέχρι στιγμής

αριθμός γραμμής για τις ντάμες 0<=i<n που έχουν τοποθετηθεί μέχρι στιγμής

```
void putNxtQueen(int cols[], int rows[], int n, int M) {
  …
}
```

αριθμός επόμενης ντάμας προς τοποθέτηση

τελικός αριθμός από ντάμες προς τοποθέτηση

```
/* κλήση */

#define M 8

int cols[M],rows[M];
putNxtQueen(cols,rows,0,M);
```

Προσέγγιση 1 (έλεγχος στο τέλος)

```
void putNxtQueen(int cols[], int rows[], int n, int M) {
  int i,j;

  if (n<M) {
    for (i=0; i<M; i++) {
      for (j=0; j<M; j++) {
        cols[n]=j; rows[n]=i;
        putNxtQueen(cols,rows,n+1,M);
      }
    }
  }
  else {
    for (i=0; i<M; i++) {
      for (j=i; j<M; j++) {
        if (!check(cols[i],rows[i],cols[j],rows[j])) {return;}
      }
    }
    for (i=0; i<M; i++) {printf("(%d,%d)\n",cols[i],rows[i]);}
  }
}
```

(βλακώδης) δημιουργία όλων των συνδυασμών (φύλλων του δέντρου)

απόρριψη συνδυασμού (φύλλου) αν υπάρχει κάποια σύγκρουση

εκτύπωση αποδεκτού συνδυασμού

Προσέγγιση 2 (έλεγχος στην τοποθέτηση)

```
void putNxtQueen(int cols[], int rows[], int n, int M) {
  int i,j,k,ok;

  if (n<M) {
    for (i=0; i<M; i++) {
      for (j=0; j<M; j++) {
        ok = 1;
        for (k=0; (k<n) && (ok); k++) {
          ok = check(j,i,cols[k],rows[k]);
        }
        if (ok) {
          col[n]=j; row[n]=i;
          putNxtQueen(cols,rows,n+1,M);
        }
      }
    }
  }
  else {
    for (i=0; i<M; i++) {printf("(%d,%d)\n",cols[i],rows[i]);}
  }
}
```

έλεγχος σύγκρουσης για την νέα τοποθέτηση j,i

εκτύπωση αποδεκτού συνδυασμού

Βελτιστοποίηση

- Ακόμα και ένας πρωτάρης σκακιστής, μπορεί να συμπεράνει ότι κάθε ντάμα πρέπει υποχρεωτικά να τοποθετηθεί σε ξεχωριστή γραμμή.
- Ιδέα: παράγουμε τους συνδυασμούς **υπό τον περιορισμό** ότι η i-οστή ντάμα τοποθετείται (κάπου) στην i-οστή γραμμή.
- Η λύση ακολουθεί τη φιλοσοφία της προσέγγισης 2, αλλά **αποφεύγει με ιδιαίτερα αποδοτικό** τρόπο **πάρα πολλούς** μη αποδεκτούς συνδυασμούς.
- Σημείωση: η συνάρτηση δεν χρειάζεται να δέχεται πλέον ως παράμετρο τους αριθμούς γραμμής από τις ντάμες που έχουν τοποθετηθεί, αφού `row[i]==i`.

αριθμός στήλης για τις ντάμες `0<=i<n` που έχουν τοποθετηθεί μέχρι στιγμής

ο αριθμός γραμμής για τις ντάμες `0<=i<n` που έχουν τοποθετηθεί μέχρι στιγμής είναι `row[i]==i`

```
void putNxtQueen(int cols[], int n, int M) {
   …
}
```

αριθμός επόμενης ντάμας προς τοποθέτηση

τελικός αριθμός από ντάμες προς τοποθέτηση

```
/* κλήση */

#define M 8

int cols[M];
putNxtQueen(cols,0,M);
```

Προσέγγιση 3

```
void putNxtQueen(int cols[], int n, int M) {
  int i,k,ok;

  if (n<M) {
    for (i=0; i<M; i++) {
      ok = 1;
      for (k=0; (k<n) && (ok); k++) {
        ok=check(i,n,cols[k],k);
      }
      if (ok) {
        col[n]=i;
        putNxtQueen(col,n+1,M);
      }
    }
  }
  else {
    for (i=0; i<M; i++) {printf("(%d,%d)\n",cols[i],i);}
  }
}
```

έλεγχος σύγκρουσης για τη νέα τοποθέτηση i,n

εκτύπωση αποδεκτού συνδυασμού

Σχόλιο

- Οι παραπάνω προσεγγίσεις βρίσκουν **όλους** τους επιτρεπτούς συνδυασμούς (συμπεριλαμβανομένων και ισοδύναμων «συμμετρικών» λύσεων).
- Όταν απορρίπτεται ένας συνδυασμός (φύλλο ή/και υποδέντρο) είναι **εγγυημένα** μη αποδεκτός.

Μονοπάτι σε γράφο

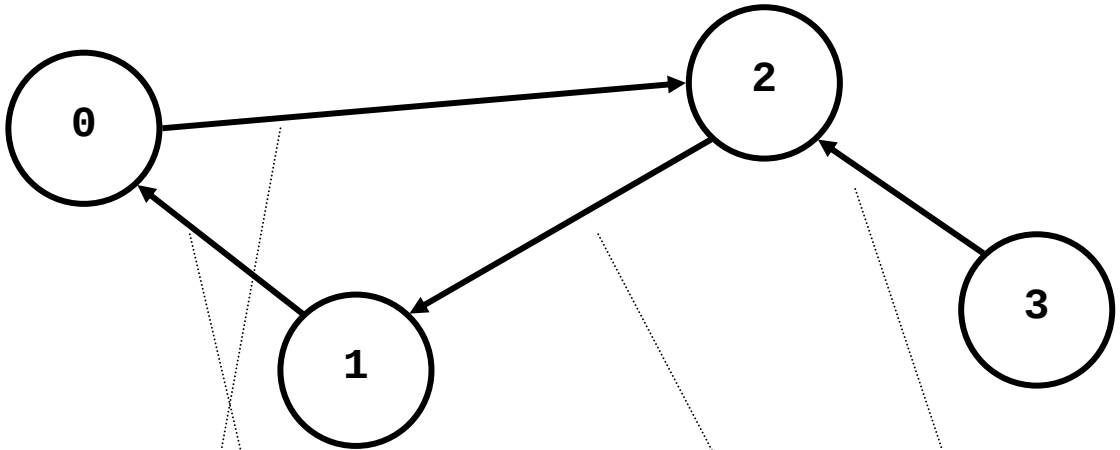
- Ένας γράφος κωδικοποιείται μέσω ενός 2-διάστατου πίνακα `c`, όπου `c[i][j]==1` αν υπάρχει ακμή από τον κόμβο `i` στον `j`, διαφορετικά `c[i][j]==0`.
- Επιθυμούμε να βρούμε ένα μονοπάτι από τον κόμβο `n1` προς ένα άλλο κόμβο `n2`: `path(n1,n2)`:
- Περίπτωση εύκολου τερματισμού:

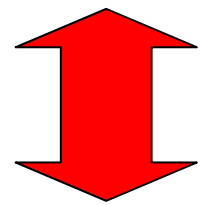
  `c[n1][n2]==1`

- Γενική περίπτωση:

  αν ∃ `n:[n1][n]==1` και `path(n,n2)` τότε το ζητούμενο μονοπάτι είναι `n+path(n,n2)`,

  διαφορετικά, δεν υπάρχει κανένα μονοπάτι που να οδηγεί από `n1` προς `n2`.





```
c[0][0]=0;
c[0][1]=0;
c[0][2]=1;
c[0][3]=0;
```

```
c[1][0]=1;
c[1][1]=0;
c[1][2]=0;
c[1][3]=0;
```

```
c[2][0]=0;
c[2][1]=1;
c[2][2]=0;
c[2][3]=0;
```

```
c[3][0]=0;
c[3][0]=0;
c[3][2]=1;
c[3][3]=0;
```

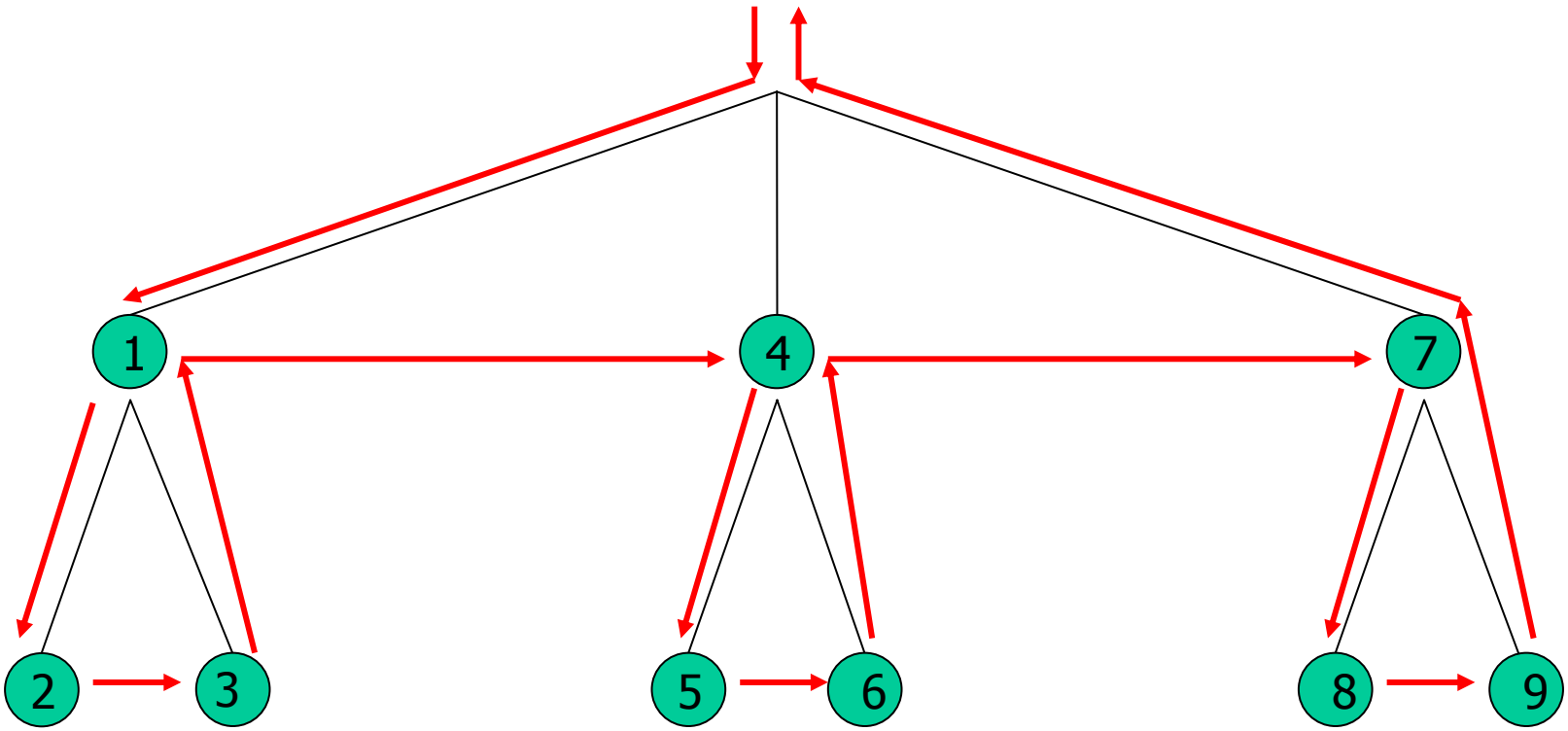
```
int path(int n1, int n2) {
  int n;

  if (c[n1][n2]) {
    return(1);
  }
  else {
    for (n=0; n<N; n++) {
      if ((c[n1][n]) && path(n,n2)) {
        return(1);
      }
    }
    return(0);
  }

}
```

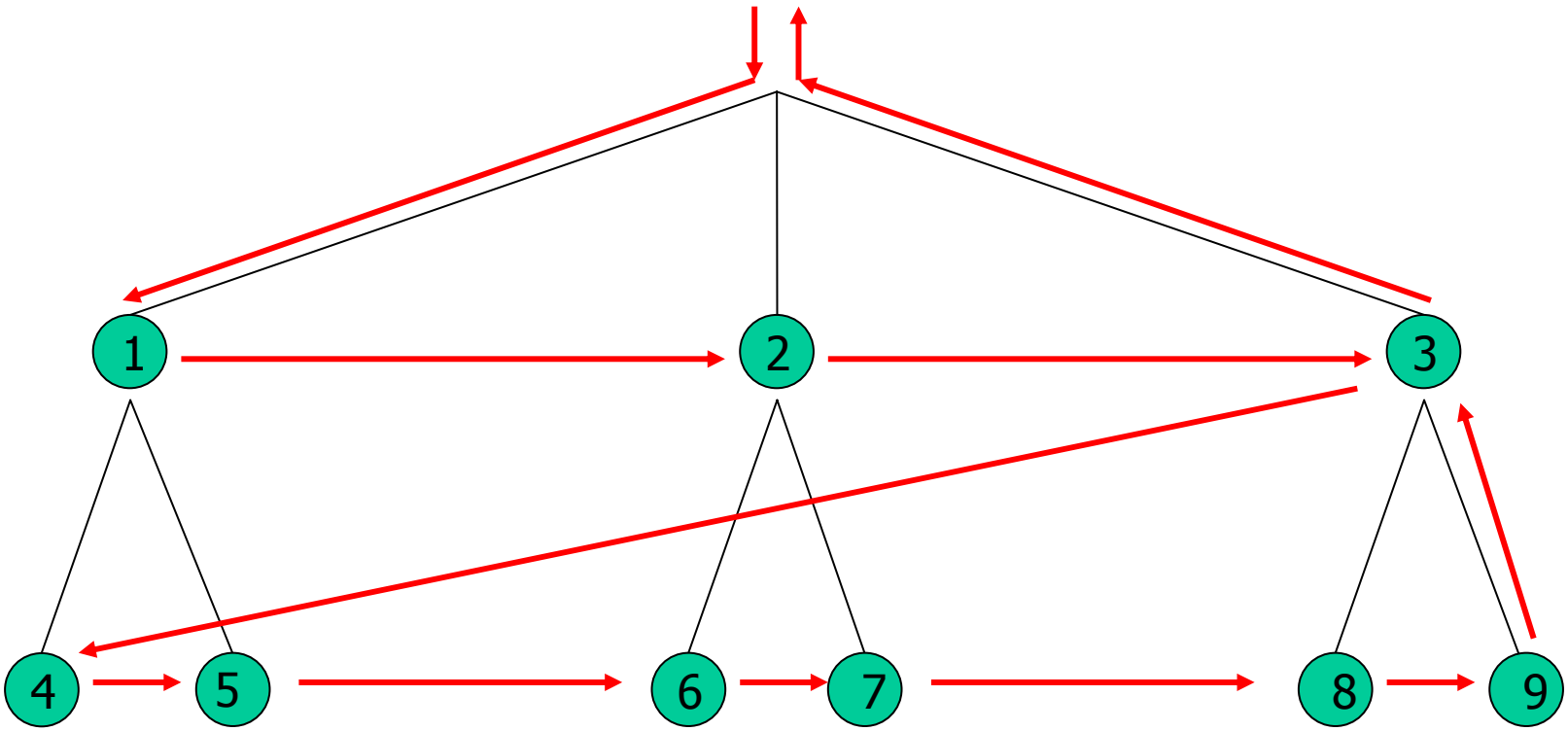
Σχόλια για την προηγούμενη λύση

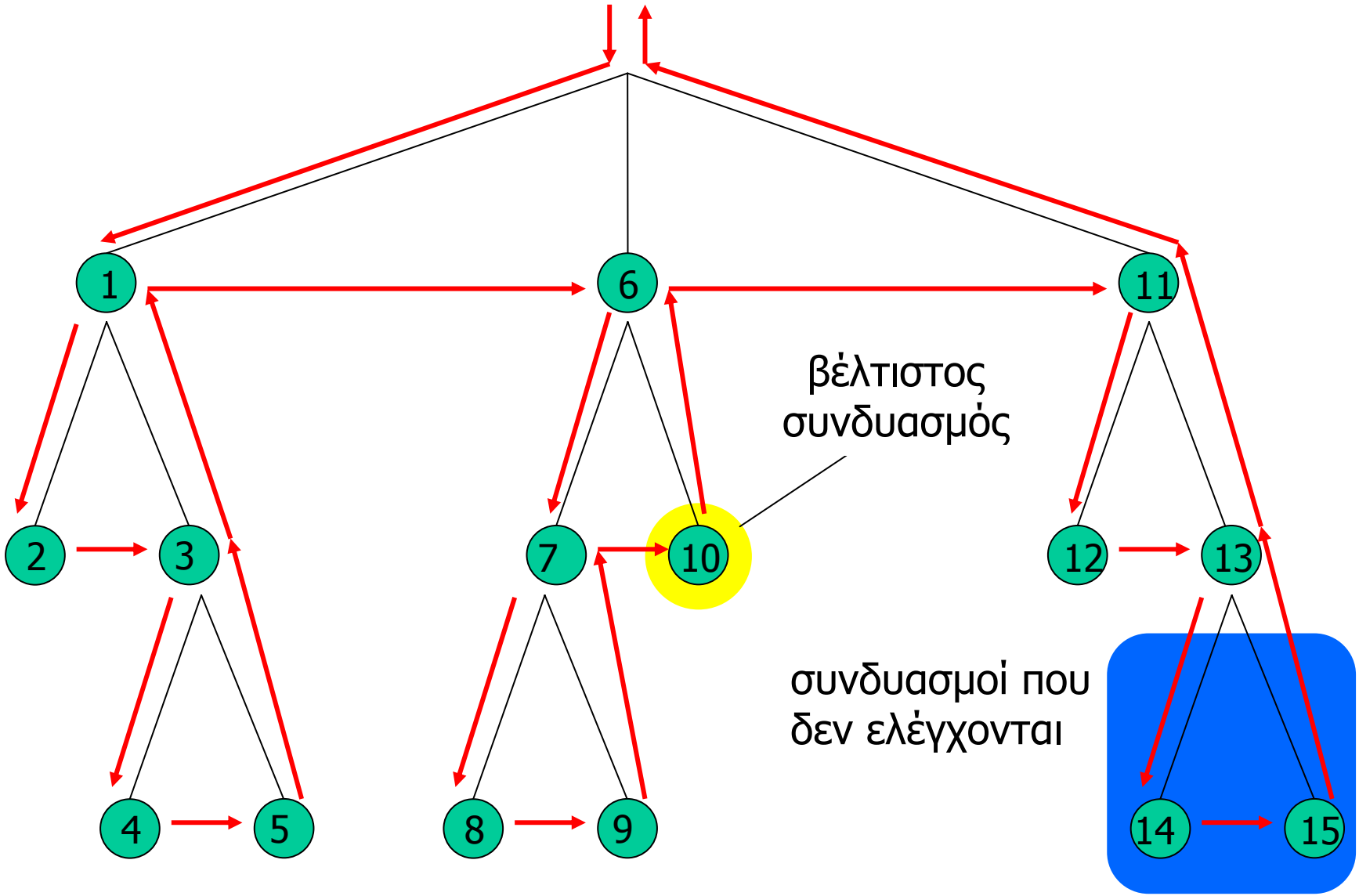
- Η αναζήτηση του μονοπατιού γίνεται «σε βάθος» (depth first) – υπάρχει και αναζήτηση «σε πλάτος».
- Επιστρέφεται το πρώτο μονοπάτι που θα βρεθεί, χωρίς να εξερευνούνται οι υπόλοιπες πιθανές λύσεις.
- Δεν επιστρέφεται εγγυημένα η πιο σύντομη διαδρομή (μόνο κατά τύχη, αν αυτή τυχαίνει να είναι η πρώτη διαδρομή που βρέθηκε μέσω της αναδρομής).
- Δεν λαμβάνονται υπ' όψη πιθανοί κύκλοι στον γράφο, οπότε υπάρχει πιθανότητα **ατέρμονης** αναδρομής.
- Η υλοποίηση μπορεί να επεκταθεί «σχετικά» εύκολα έτσι ώστε να αντιμετωπιστούν όλες οι παραπάνω αδυναμίες / ατέλειες.

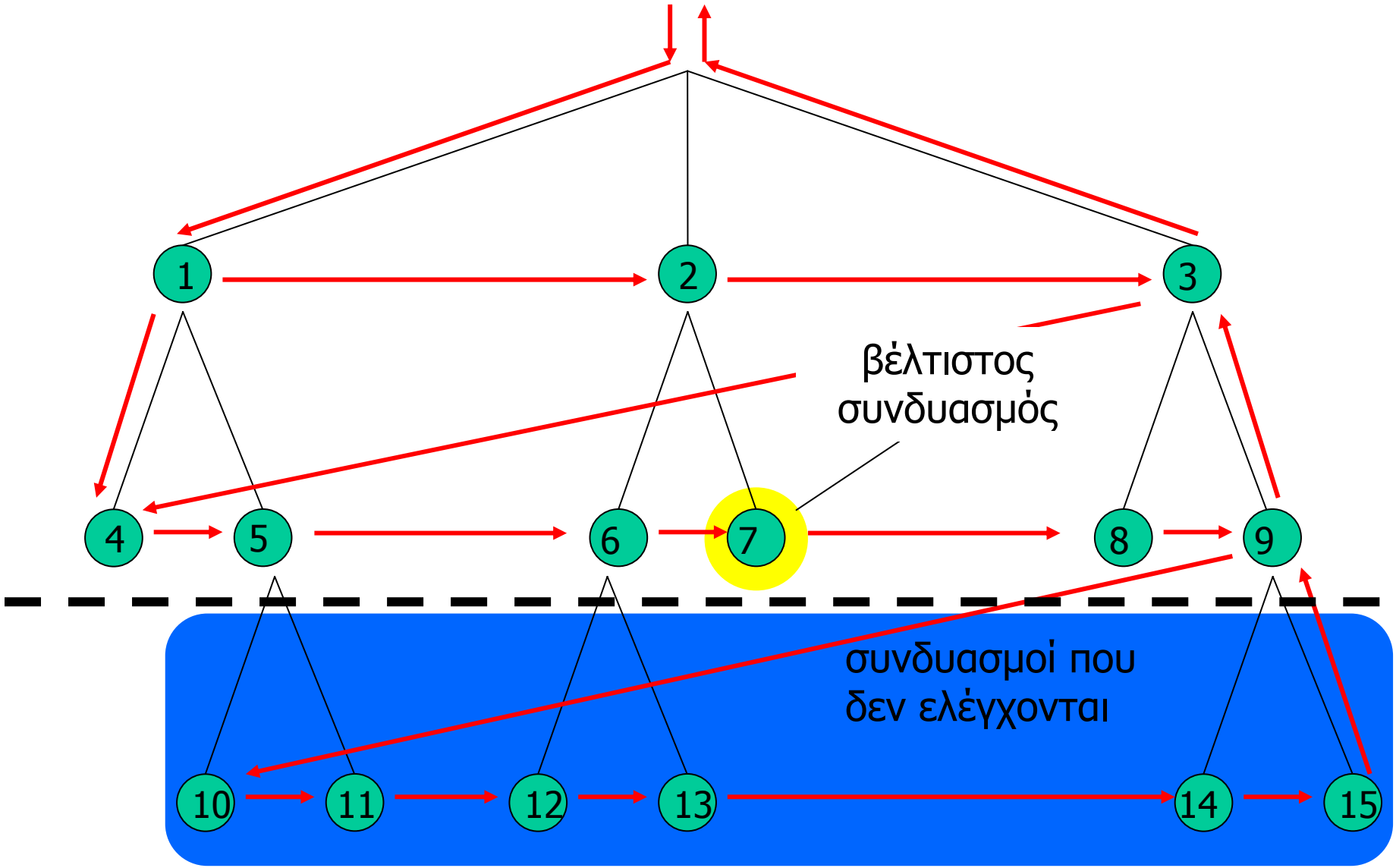
Κατασκευή «δέντρου» συνδυασμών

- Προσέγγιση «σε βάθος» (depth first): αρχίζουμε από τη ρίζα και κατασκευάζουμε / διερευνούμε τους κόμβους **ανά υποδέντρο** (από πάνω προς τα κάτω).
- Προσέγγιγη «κατά πλάτος» (breadth first): αρχίζουμε από τη ρίζα και κατασκευάζουμε / διερευνούμε τους κόμβους **ανά επίπεδο** (από αριστερά προς τα δεξιά).
- Η προσέγγιση σε βάθος μπορεί εύκολα να υλοποιηθεί αναδρομικά – σε αυτή την περίπτωση ονομάζεται και «**αναδρομική κάθοδος**» (recursive descent).
- Η προσέγγιση κατά πλάτος απαιτεί επιπλέον διαχείριση κατάστασης από το πρόγραμμα για την αποθήκευση των κόμβων έτσι ώστε να γίνεται έλεγχος με τη σωστή σειρά (ουρά FIFO).


1
4
7
2
3
5
6
8
9


1
2
3
4
5
6
7
8
9


1
6
11
2
3
7
10
12
13
4
5
8
9
14
15
βέλτιστος συνδυασμός
συνδυασμοί που δεν ελέγχονται

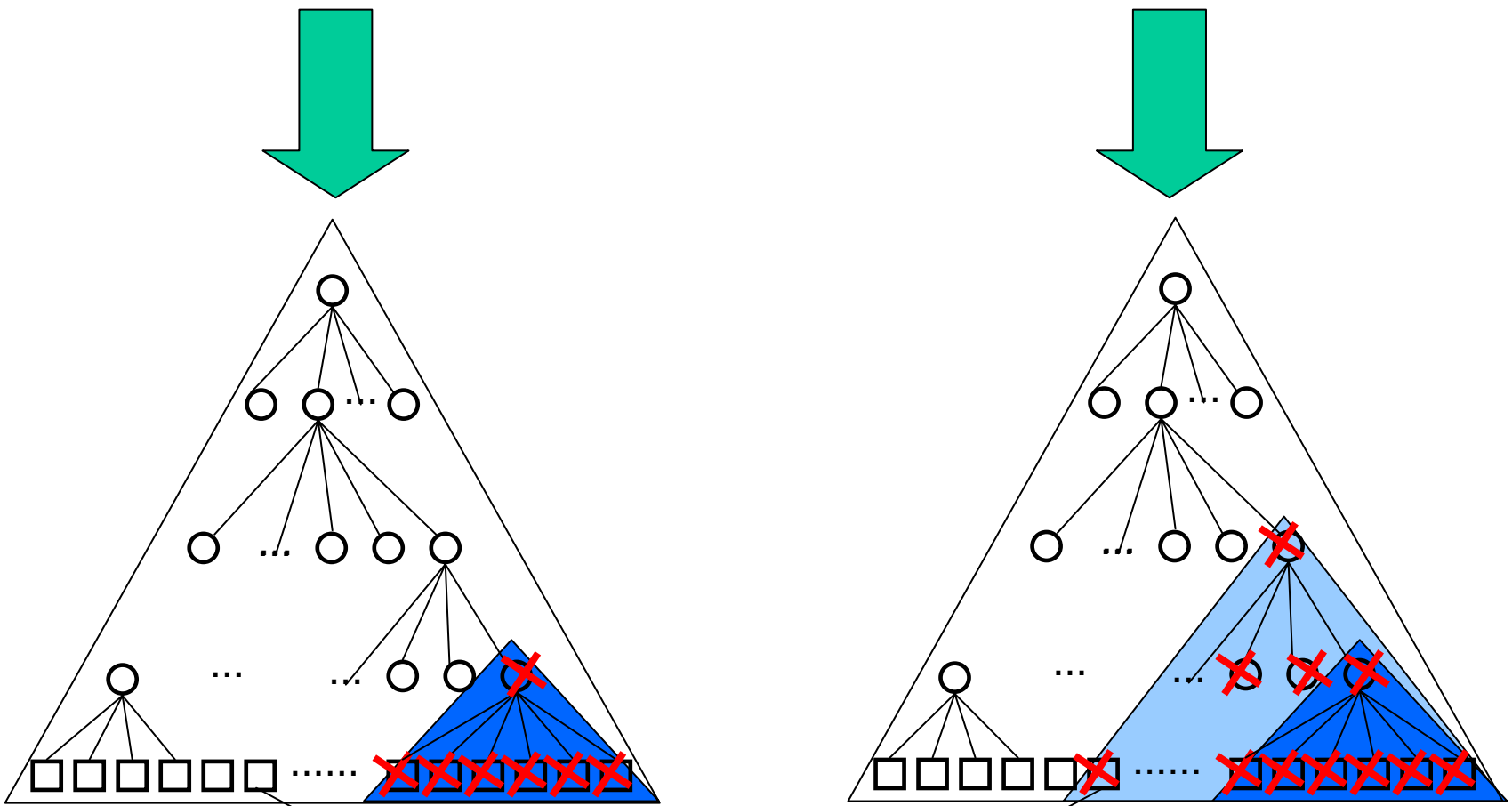

1
2
3
βέλτιστος
συνδυασμός
4
5
6
7
8
9
συνδυασμοί που
δεν ελέγχονται
10
11
12
13
14
15

Μη βέλτιστες (ευρεστικές) λύσεις

- Μερικά προβλήματα είναι δύσκολο να λυθούν διεξοδικά, με έλεγχο όλων των επιτρεπτών συνδυασμών (με b&b).
- Ακόμα και για «σχετικά μικρά» N, π.χ. 10000, ένας σύγχρονος Η/Υ μπορεί να χρειαστεί μέρες ή και βδομάδες να ολοκληρώσει τον υπολογισμό (αν στο μεταξύ δεν του έχει ήδη σωθεί η διαθέσιμη μνήμη).
- **Ευρεστικές** μέθοδοι: εκτός από τα υποδέντρα που σίγουρα δεν οδηγούν σε λύση, αποκλείουμε και τα υποδέντρα που **πιστεύουμε**, σύμφωνα με μια «κοινή» λογική, ότι δεν θα οδηγήσουν σε κάποια (καλή) λύση.
- Οι ευρεστικές μέθοδοι μειώνουν σημαντικά τον αριθμό των συνδυασμών προς έλεγχο, αλλά στην γενική περίπτωση **δεν** οδηγούν εγγυημένα σε βέλτιστη λύση.

αποκλεισμός υποδέντρων που **σίγουρα** δεν οδηγούν σε λύση

αποκλεισμός υποδέντρων που **μάλλον** δεν οδηγούν σε λύση



βέλτιστος συνδυασμός

Παράδειγμα – Knapsack / Rucksack

- Δίνεται: αντικείμενα O[i] με βάρος W[i] και αξία V[i].
- Ζητούμενο: να επιλεγούν τα αντικείμενα τα οποία μεγιστοποιούν την αξία ενός φορτίου με μέγιστο συνολικό βάρος ένα γνωστό άνω όριο maxWeight.
- Προσέγγιση 1 (brute force): κατασκευή **όλων** των συνδυασμών, και επιλογή του συνδυασμού με την μεγαλύτερη αξία στα πλαίσια του επιτρεπτού βάρους.
- Προσέγγιση 2 (branch & bound): όπως 1, αλλά **σε κάθε βήμα** ελέγχεται το βάρος του συνδυασμού και αυτός απορρίπτεται (πρόωρα) αν υπερβαίνει το όριο.
- Προσέγγιση 3 (heuristic b&b): όπως 2, αλλά σε κάθε βήμα επιλέγεται (**αμετάκλητα**) το αντικείμενο με το μεγαλύτερο **ειδικό βάρος** που χωρά στο φορτίο.

Σχόλιο

- Η προσέγγιση σε βάθος είναι κατάλληλη όταν όλες οι λύσεις (φύλλα του δέντρου) βρίσκονται στο **ίδιο** επίπεδο, π.χ. 8 ντάμες.
- Η προσέγγιση κατά πλάτος είναι κατάλληλη όταν κάποιες λύσεις (φύλλα του δέντρου) βρίσκονται πιθανώς σε **διαφορετικό** επίπεδο και **γνωρίζουμε** ότι ο επιθυμητός συνδυασμός βρίσκεται στο πιο ψηλό επίπεδο, π.χ. πιο σύντομο μονοπάτι εξόδου σε ένα λαβύρινθο.
- Σε αυτή τη περίπτωση ο αριθμός των συνδυασμών που ελέγχονται μπορεί να είναι κατά πολύ μικρότερος (το δέντρο των συνδυασμών κόβεται οριζόντια, κάτω από το επίπεδο της λύσης).