Δυναμικός Προγραμματισμός

Δημήτρης Φωτάκης

Τμήμα Μηχανικών Πληροφοριακών και Επικοινωνιακών Συστημάτων

Πανεπιστήμιο Αιγαίου

Τρίγωνο του Pascal

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

$$C(n,k) = \begin{cases} C(n-1,k-1) + C(n-1,k) & \text{αν } 0 < k < n \\ 1 & \text{διαφορετικά} \end{cases}$$

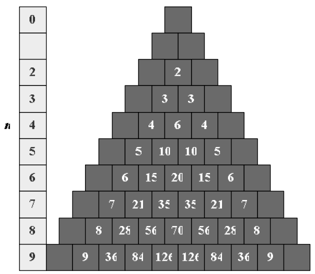
```
long Binom(int n, int k) {
  if ((k == 0) || (k == n)) return(1);
  return(Binom(n - 1, k - 1) + Binom(n - 1, k)); }
```

  - Χρόνος εκτέλεσης δίνεται από την ίδια αναδρομή! $T(n,k) = C(n,k) = \Omega((n/e)^k)$, $C(30,15) = 155117520$
  - Πρόβλημα οι επαναλαμβανόμενοι υπολογισμοί.
- Όταν έχω επικαλυπτόμενα στιγμιότυπα, χρησιμοποιώ **δυναμικό προγραμματισμό**.

Τρίγωνο του Pascal

$$C(n,k) = \begin{cases} C(n-1,k-1) + C(n-1,k) & \text{αν } 0 < k < n \\ 1 & \text{διαφορετικά} \end{cases}$$

- Όταν επαναλαμβανόμενα στιγμιότυπα, αποθηκεύω τιμές σε πίνακα και τις χρησιμοποιώ χωρίς να τις υπολογίζω πάλι.
  - Θεαματική βελτίωση χρόνου εκτέλεσης!
  - Σημαντικές απαιτήσεις σε μνήμη.



Τρίγωνο του Pascal

$$C(n,k) = \begin{cases} C(n-1,k-1) + C(n-1,k) & \text{αν } 0 < k < n \\ 1 & \text{διαφορετικά} \end{cases}$$

Binomial(n, k)
  $C[0,0] = C[1,0] = C[1,1] = 1;$
  for $i \leftarrow 2$ **to** n **do**
    $C[i,0] \leftarrow 1;$
    for $j \leftarrow 1$ **to** $\min\{i-1,k\}$ **do**
      $C[i,j] \leftarrow C[i-1,j-1] + C[i-1,j];$
    if $i-1 < k$ **then** $C[i,i] = 1;$
  return($C[n,k]$);

- Χρόνος εκτέλεσης $\Theta(nk)$ αντί για $\Omega((n/e)^k)$.
  - Μνήμη $\Theta(nk)$. Μπορεί να μειωθεί σε $\Theta(k)$.

Δυναμικός Προγραμματισμός

- Εφαρμόζουμε δυναμικό προγραμματισμό για προβλήματα συνδυαστικής βελτιστοποίησης με
  - Ιδιότητα των **βέλτιστων επιμέρους λύσεων.**
  - Κάθε τμήμα βέλτιστης λύσης αποτελεί βέλτιστη λύση για αντίστοιχο υπο-πρόβλημα.
    - Τμήματα συντομότερου μονοπατιού είναι συντομότερα μονοπάτια.
    - Ισχύει για τμήματα μακρύτερου μονοπατιού;
- Έστω βέλτιστες λύσεις για «μικρότερα» προβλήματα. Πως συνδυάζονται για βέλτιστη λύση σε «μεγαλύτερα»;
  - **Αναδρομική εξίσωση** που περιγράφει **τιμή** βέλτιστης λύσης.
  - Υπολογίζουμε λύση από μικρότερα σε μεγαλύτερα **(bottom-up).**

Πολλαπλασιασμός Πινάκων

- Γινόμενο πινάκων A (p × q) επί B (q × r) σε χρόνο Θ(p q r). (μετράμε μόνο πολ/μούς μεταξύ αριθμών).
- Συντομότερος τρόπος υπολογισμού γινομένου

$$\begin{matrix} A_1 & A_2 & A_3 & \ldots & A_{n-1} & A_n \\ (d_0 \times d_1) & (d_1 \times d_2) & (d_2 \times d_3) & \ldots & (d_{n-2} \times d_{n-1}) & (d_{n-1} \times d_n) \end{matrix}$$

  - Πολλαπλασιασμός πινάκων είναι πράξη προσεταιριστική (αποτέλεσμα ανεξάρτητο σειράς εκτέλεσης).
  - Ο χρόνος υπολογισμού εξαρτάται από τη σειρά!

$$\begin{matrix} A_1 & A_2 & A_3 \\ (1 \times 100) & (100 \times 3) & (3 \times 1) \end{matrix}$$

$$\begin{matrix} (A_1A_2) & A_3 \\ (1 \times 100 \times 3) & + (1 \times 3 \times 1) = 303 \end{matrix}$$

$$\begin{matrix} A_1 & (A_2A_3) \\ (1 \times 100 \times 1) & + (100 \times 3 \times 1) = 400 \end{matrix}$$

Πολλαπλασιασμός Πινάκων

$$\begin{matrix} A_1 & A_2 & A_3 & A_4 \\ (13 \times 5) & (5 \times 89) & (89 \times 3) & (3 \times 34) \end{matrix}$$

Σειρά Υπολογισμού	**Αριθμός Πολλαπλασιασμών**
$(((A_1A_2)A_3)A_4)$	$13 \times 5 \times 89 + 13 \times 89 \times 3 + 13 \times 3 \times 34 = 10582$
$((A_1A_2)(A_3A_4))$	54201
$((A_1(A_2A_3))A_4)$	2856
$(A_1((A_2A_3)A_4))$	4055
$(A_1(A_2(A_3A_4)))$	26418

Πολλαπλασιασμός Πινάκων

- Δίνονται *n* πίνακες:

$$\begin{matrix} A_1 & A_2 & A_3 & \ldots & A_{n-1} & A_n \\ (d_0 \times d_1) & (d_1 \times d_2) & (d_2 \times d_3) & \ldots & (d_{n-2} \times d_{n-1}) & (d_{n-1} \times d_n) \end{matrix}$$

  Με ποια σειρά θα υπολογιστεί το γινόμενο $A_1 A_2 \ldots A_n$ ώστε να ελαχιστοποιηθεί #πολ/μών μεταξύ στοιχείων.
- Πρόβλημα συνδυαστικής βελτιστοποίησης:
  - Κάθε σειρά υπολογισμού αντιστοιχεί σε #πολ/μών.
  - Ζητείται η σειρά που αντιστοιχεί στον ελάχιστο #πολ/μών.
- Αποδοτικός αλγόριθμος για υπολογισμό καλύτερης σειράς πολ/μου n πινάκων.

Εξαντλητική Αναζήτηση

- … δοκιμάζει όλες τις σειρές υπολογισμού και βρίσκει καλύτερη.
  - Κάθε σειρά αντιστοιχεί σε δυαδικό δέντρο με n φύλλα.
  - Χρόνος ανάλογος #δυαδικών δέντρων με n φύλλα:
    $$P(n) = \sum_{i=1}^{n} P(i)P(n-i)\,,\ P(1) = 1$$
  - Λύση (n-1)-οστός αριθμός Catalan: $P(n) = \frac{1}{n}\binom{2(n-1)}{n-1} = \Omega(\frac{4^n}{n^{3/2}})$
- Θα εφαρμόσουμε δυναμικό προγραμματισμό.

Βέλτιστες Επιμέρους Λύσεις

- Συμβολίζουμε $A_{i..j} = A_i \times \cdots \times A_j$
- Βέλτιστη λύση υπολογίζει $A_{1..i}$, $(d_1 \times d_i)$, και $A_{i+1..n}$, $(d_i \times d_n)$, για κάποιο $i,\ 1 \le i < n$, και τελειώνει με $A_{1..i} \times A_{i+1..n}$.
  - #πολ/μών = $d_0 \times d_i \times d_n$ + #πολ/μών($A_{1..i}$) + #πολ/μων($A_{i+1..n}$)
  - Επιμέρους γινόμενα $A_{1..i}$ και $A_{i+1..n}$ υπολογίζονται **βέλτιστα**.
- Συμβολίζουμε $m[i,j] = \beta έλτιστος\ \#πολ/μών(A_{i..j})$
- Έστω για κάθε $i,\ 1 \le i < n$, γνωρίζουμε $m[1,i]$ και $m[i+1,n]$
- Τότε $m[1,n] = \min_{1 \le i < n}\{m[1,i] + m[i+1,n] + d_0 d_i d_n\}$
- Γενική **αναδρομική σχέση**:

$$m[i,j] = \begin{cases} \min_{i \le k < j}\{m[i,k] + m[k+1,j] + d_{i-1}\, d_k\, d_j\} & \text{αν } i < j \\ 0 & \text{αν } i = j \end{cases}$$

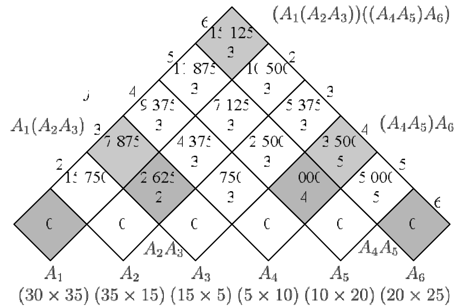
Δυναμικός Προγραμματισμός

- Bottom-up υπολογισμός *m*[1, *n*] από αναδρομική σχέση:

$$m[i,j] = \begin{cases} \min_{i \le k < j}\{m[i,k] + m[k+1,j] + d_{i-1}\, d_k\, d_j\} & \text{αν } i < j \\ 0 & \text{αν } i = j \end{cases}$$

- Υπολογίζω n(n – 1) / 2 τιμές m[i, j].
  - m[i, j] υπολογίζεται σε χρόνο O(n) από τιμές για γινόμενα μικρότερου εύρους.
  - Τιμές αποθηκεύονται σε πίνακα.

Παράδειγμα



Υλοποίηση (bottom-up)

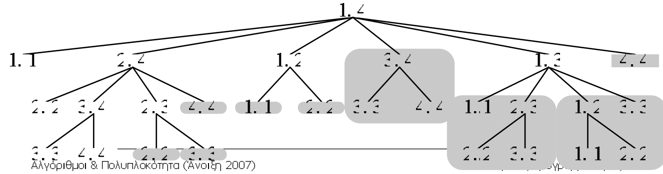
```
MatrixChainMultiplication(d[0, 1, ..., n])   /* A_i διάστασης d[i − 1] × d[i] */
    for i ← 1 to n do
        m[i, i] ← 0;
    for p ← 2 to n do
        for i ← 1 to n − p + 1 do
            j ← i + p − 1; m[i, j] ← ∞;
            for k ← i to j − 1 do
                q ← m[i, k] + m[k + 1, j] + d[i − 1]d[k]d[j];
                if q < m[i, j] then m[i, j] ← q;
    return(m[1, n]);
```

- Χρόνος $O(n^3)$ και μνήμη $O(n^2)$, μειώνεται σε $O(n)$.

Υλοποίηση (top-down)

Χρόνος **$O(2^n)$** !

```
RecMatrixChain(d[i − 1, ..., j])
    if i = j then return(0);
    m ← ∞;   /* Το m θα πάρει την τιμή m[i, j] */
    for k ← i to j − 1 do
        q ← RecMatrixChain(d[i − 1, ..., k]) +
            RecMatrixChain(d[k, ..., j]) + d[i − 1]d[k]d[j];
        if q < m then m ← q;
    return(m);
```



Αναδρομή με Μνήμη

- Ο αναδρομικός αλγόριθμος αποθηκεύει τιμές σε πίνακα. Κάθε τιμή υπολογίζεται μία φορά.
  - Συνδυάζει απλότητα top-down προσέγγισης με ταχύτητα bottom-up.

```
RecMemMatrixChain(d[0, ..., n])
    for i ← 1 to n do
        for j ← 1 to n do
            m[i, j] ← ∞;
    return(RecCM(d[0, ..., n]));
```

```
RecCM(d[i − 1, ..., j]);
    if m[i, j] < ∞ then return(m[i, j]);
    if i = j then m[i, j] = 0;
    else
        for k ← i to j − 1 do
            q ← RecCM(d[i − 1, ..., k]) +
                RecCM(d[k, ..., j]) +
                d[i − 1]d[k]d[j];
            if q < m[i, j] then m[i, j] ← q;
    return(m[i, j]);
```


ΔΠ vs ΔκΒ

- Δυναμικός Προγραμματισμός και Διαίρει-και-Βασίλευε επιλύουν προβλήματα συνδυάζοντας λύσεις κατάλληλα επιλεγμένων υπο-προβλημάτων.
- ΔκΒ είναι φύσει αναδρομική μέθοδος (top-down).
- ΔκΒ επιλύει υπο-προβ/τα ανεξάρτητα.
  - Εφαρμόζεται όταν παράγονται ανεξάρτητα υπο-προβ/τα.
  - Ειδάλλως ίδια υπο-προβ/τα λύνονται πολλές φορές: Σπατάλη υπολογιστικού χρόνου.

ΔΠ vs ΔκΒ

- ΔΠ «κτίζει» βέλτιστη λύση προβ/τος από βέλτιστες λύσεις υπο-προβ/των (bottom-up).
  - ΔΠ ξεκινά με στοιχειώση στιγμιότυπα.
  - Συνδυάζει λύσεις για να βρει λύσεις σε μεγαλύτερα.
- ΔΠ εφαρμόζεται όταν υπο-προβ/τα επικαλύπτονται. Αποθηκεύει επιμέρους λύσεις για να μην υπολογίζει πάλι.
  - «Προγραμματισμός» διαδικασία συμπλήρωσης πίνακα με ενδιάμεσα αποτελέσματα.
- ΔΠ εφαρμόζεται όταν ιδιότητα βέλτιστων επιμέρους λύσεων.
  - Επιτρέπει διατύπωση αναδρομικής εξίσωσης για βέλτιστη λύση.
  - Αναδρομική εξίσωση λύνεται bottom-up για βέλτιστη τιμή.
  - Επιλογές επίλυσης συνθέτουν βέλτιστη λύση.

Πρόβλημα Σακιδίου

- Δίνονται *n* αντικείμενα και **σακίδιο** μεγέθους B. Αντικείμενο *i* έχει **μέγεθος** και **αξία**: (s_i, p_i)
- Ζητείται συλλογή μέγιστης αξίας που χωράει στο σακίδιο.

$$\begin{array}{rl} \max & \sum_{i=1}^{n} f_i p_i \\ \text{υπό περιορισμούς} & \sum_{i=1}^{n} f_i s_i \leq B \\ & f_i \in \{0,1\} \quad \forall i \in [n] \end{array} \qquad f_i = \begin{cases} 1 & i \text{ εντός} \\ 0 & i \text{ εκτός} \end{cases}$$

  - Αντικείμενα: { (1, 0.5), (2, 5), (2, 5), (3, 9), (4, 8) } Μέγεθος σακιδίου: **4**.
  - Βέλτιστη λύση = { (2, 5), (2, 5) }
  - Αντικείμενα: { (3, 5), (2, 7), (4, 4), (6, 8), (5, 4) } Μέγεθος σακιδίου: **10**.
  - Βέλτιστη λύση = { (3, 5), (2, 7), (4, 4) }

Πρόβλημα Σακιδίου

- Πρόβλημα συνδυαστικής βελτιστοποίησης:
  - Συλλογή που χωράει **εφικτή λύση**. Αντιστοιχεί σε **αξία**.
  - Ζητούμενο: (**βέλτιστη**) συλλογή που χωράει με μέγιστη αξία.
- Εξαντλητική αναζήτηση:
  - #συλλογών = 2^n . Χρόνος $\Omega(n2^n)$
- Πρόβλημα Σακιδίου είναι **NP**-δύσκολο και δεν υπάρχει «γρήγορος» (πολυωνυμικός) αλγόριθμος.
  - Εφαρμογή δυναμικού προγραμματισμού.
  - Χρόνος Θ(n B). **Δεν** είναι πολυωνυμικός(;)!

Βέλτιστες Επιμέρους Λύσεις

- Αντικείμενα N = {1, …, *n*}, σακίδιο μεγέθους B. Βέλτιστη λύση $A^* \subseteq \{1, …, n\}$.
- Αγνοούμε αντικείμενο *n* :
  - $A^* \setminus \{n\}$ βέλτιστη λύση για $N \setminus \{n\}$ με σακίδιο $B - (f_n s_n)$.
- Αγνοούμε αντικείμενα {n, n – 1}:
  - $A^* \setminus \{n, n-1\}$ βέλτιστη λύση για $N \setminus \{n, n-1\}$ με σακίδιο $B - (f_n s_n + f_{n-1} s_{n-1})$.
- Αν γνωρίζουμε βέλτιστη αξία για αντικείμενα $N \setminus \{n\}$ και σακίδια μεγέθους B και $B - s_n$
  - … αποφασίζουμε αν αντικείμενο n στη βέλτιστη λύση!

Αναδρομική Εξίσωση

- *P*(*n*-1, *B*) βέλτιστη αξία για N \ {n} σε σακίδιο B
  P(*n*-1, *B* - s_n) βέλτιστη αξία για N \ {n} σε σακίδιο B - s_n

$$P(n,B) = \max\{P(n-1,B), P(n-1,B-s_n)+p_n\}$$

- Βέλτιστη αξία με αντικείμενα {1, …, *i* } και σακίδιο μεγέθους *b* : $P(i,b)$

$$P(i,b) = \begin{cases} 0 & \text{αν } b \leq 0 \\ 0 & \text{αν } i = 0 \\ \max\{P(i-1,b), P(i-1,b-s_i)+p_i\} & \text{για } i = 1,\ldots,n \\ & \text{και } b = 1,\ldots,B \end{cases}$$

Παράδειγμα

$$P(i,b) = \begin{cases} 0 & \text{αν } b \leq 0 \\ 0 & \text{αν } i = 0 \\ \max\{P(i-1,b), P(i-1,b-s_i)+p_i\} & \text{για } i = 1,\ldots,n \\ & \text{και } b = 1,\ldots,B \end{cases}$$

- Αντικείμενα: { (3, 5), (2, 7), (4, 4), (6, 8), (5, 4) }
  Μέγεθος σακιδίου: **10**.

$i \vert b$	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	5	**5**	5	5	5	5	5	5
2	0	0	7	7	7	12	**12**	12	12	12	12
3	0	0	7	7	7	12	12	12	12	16	**16**
4	0	0	7	7	7	12	12	12	15	16	16
5	0	0	7	7	7	12	12	12	15	16	16

Υλοποίηση

Knapsack($B, (s_1, p_1), \ldots, (s_n, p_n)$)
  for $i \leftarrow 0$ **to** n **do** $P[i, 0] \leftarrow 0$;
  for $b \leftarrow 1$ **to** B **do** $P[0, b] \leftarrow 0$;
  for $i \leftarrow 1$ **to** n **do**
    for $b \leftarrow 1$ **to** B **do**
      if $b - s_i \geq 0$ **then**
        $t \leftarrow P[i-1, b-s_i] + p_i$;
      else $t \leftarrow 0$;
      if $P[i-1, b] \geq t$ **then**
        $P[i, b] \leftarrow P[i-1, b]$;
      else $P[i, b] \leftarrow t$;
  return($P[n, B]$);

Χρόνος **O(nB)**
Μνήμη **O(nB)**

Ψευδοπολυωνυμικοί Αλγόριθμοι

- Το πρόβλημα του σακιδίου είναι **NP**-δύσκολο.
- Αλγόριθμος O(*n B*) δεν είναι πολυωνυμικού χρόνου;
  - Πολυώνυμο του μεγέθους εισόδου!
  - Μέγεθος εισόδου: $\mathrm{O}(n(\log_2 B + \log_2 P_{\max})$
  - Χρόνος πολυωνυμικός στο *n* αλλά εκθετικός στο $\log_2 B$
- Αριθμητικά προβλήματα:
  - Μέγεθος αριθμών πολύ μεγαλύτερο (π.χ. εκθετικό) στο πλήθος «βασικών συνιστωσών» (ότι συμβολίζουμε με n).
- Αλγόριθμος πολυωνυμικό χρόνου:
  $\mathrm{O}((n \max\_\mathrm{num})^k)$, σταθερά $k \geq 1$
- Αλγόριθμος **ψευδο-πολυωνυμικού** χρόνου:
  $\mathrm{O}((n\ \log \max\_\mathrm{num})^k)$, σταθερά $k \geq 1$