Κατ'οίκον Εργασία 1 – Σκελετοί Λύσεων

1. Ο αλγόριθμος διαίρει και βασίλευε για το πρόβλημα έχει ως εξής:
   - Μοίρασε τον πίνακα σε δύο μισά.
   - Υπολόγισε αναδρομικά τα μέγιστα και ελάχιστα στοιχεία των δύο υποπινάκων.
   - Επέστρεψε ως ελάχιστο του πίνακα το ελάχιστο των ελαχίστων στοιχείων των δύο υποπινάκων και ως μέγιστο του πίνακα το μέγιστο των μέγιστων στοιχείων των δύο υποπινάκων.

   Η διαδικασία minmax επιστρέφει στοιχεία (min, max) τύπου (int, int), όπου το min είναι το ελάχιστο και max το μέγιστο στοιχείο στοιχείο του πίνακα αντίστοιχα.

```
(int, int) minmax( int X[], int l, int r){

     if (r-l ≤ 1)                                  } Βασική
          if (X[l] > X[r])                         } Περίπτωση
               return(X[r], X[l]);                 }
          else                                     }
               return (X[l], X[r]);                }

     m=(l+r)/2;                                    }
     (min1, max1) = minmax(X,l,m);                 } Φάση διαίρει
     (min1, max2) = minmax(X,m+1,r);               }

     return (min(min1, min2),                      }
                  max(max1, max2))                 } Φάση συνδύασε
 }
```

Υποθέτοντας ότι το πλήθος των στοιχείων του πίνακα είναι δύναμη του 2, ο αριθμός συγκρίσεων της διαδικασίας δίνεται από την αναδρομική εξίσωση

$T(2) = 1$
$T(n) = 2 \cdot T(n/2) + 2$

Θα δείξουμε ότι $T(n) = 3n/2-2$ με τη μέθοδο της αντικατάστασης.

$$
\begin{aligned}
T(n) &= 2 \cdot T(n/2) + 2 \\
&= 2^2 T(n/2^2) + 2^2 + 2 \\
&= \ldots \\
&= 2^i T(n/2^i) + 2^i + \ldots + 2^2 + 2 \\
&= 2^{\lg n-1} T(n/2^{\lg n-1}) + 2^{\lg n-1} + \ldots + 2^2 + 2 \\
&= 2^{\lg n-1} T(2) + 2(2^{\lg n-2} + \ldots + 2^2 + 2 + 1) \\
&= 2^{\lg n-1} + 2(2^{\lg n-1} - 1) = n/2 + n - 2 \\
&= 3n/2 - 2
\end{aligned}
$$

Πως μπορούμε να επεκτείνουμε τον αλγόριθμο έτσι ώστε να δουλεύει για κάθε n (όχι μόνο για δυνάμεις του 2);
Με δεδομένο ένα πίνακα μεγέθους n προσδιορίζουμε ποια είναι η μεγαλύτερη δύναμη του 2, m, μικρότερη του n. Τρέχουμε στα πρώτα m στοιχεία του πίνακα τον αλγόριθμο minmax και επαναλαμβάνουμε, αναδρομικά, την ίδια διαδικασία για τον υπόλοιπο πίνακα (μεγέθους n-m). Αν (x,y) είναι το αποτέλεσμα της κλήσης της minmax και (x',y') το αποτέλεσμα της αναδρομικής κλήσης, επιστρέφουμε (min(x,x'), max(y,y')). Σε ψευδοκώδικα έχουμε:

```
(int, int) minmax2( int X[], int l, int r){
     if (l==r)                                          } Βασική
          return(X[r], X[l]);                           } Περίπτωση

     if (r-l+1) is a power of 2                         }
          return minmax(X,l,r)                          }
     else                                               }
          p = ⌊log(r-l+1)⌋;                             } Φάση διαίρει
          m = l + 2^p –1;                               }
          (min1, max1) = minmax(X,l,m);                 }
          (min1, max2) = minmax2(X,m+1,r);              }

          return (min(min1, min2),                      }
                    max(max1, max2))                    } Φάση συνδύασε
 }
```

Ο αριθμός συγκρίσεων της διαδικασίας δίνεται από την αναδρομική εξίσωση

$T(1) = 0$

$$T(n) = \begin{cases} 3n/2-2 & \text{αν } n=2^p \text{ για κάποιο } p \\ T(n-m) + 3m/2 -2 + 2 & \text{αν } m= 2^p < n < 2^{p+1} \text{ για κάποιο } p \end{cases}$$

Θα δείξουμε ότι $T(n) = \lceil 3n/2 \rceil -2$ με τη μέθοδο της μαθηματικής επαγωγής.

Βάση της επαγωγής
Για n=1, $T(1) = 0 = 2-2 = \lceil 3/2 \rceil -2$.

Υποθέτουμε ότι η πρόταση ισχύει για κάθε k<n.

Βήμα της επαγωγής
Αν $n = 2^p$ για κάποιο p, τότε το ζητούμενο έπεται.
Υποθέτουμε ότι για κάποιο p, $m = 2^p < n < 2^{p+1}$. Τότε

$T(n) = T(n-m) + 3m/2$ (εξ'ορισμού)
$= \lceil 3(n-m)/2 \rceil - 2 + 3m/2$ (από την υπόθεση της επαγωγής)
$= \lceil 3(n-m)/2 – 3m/2 \rceil - 2$
$= \lceil 3n/2 \rceil - 2$

όπως χρειάζεται. Αυτό ολοκληρώνει την απόδειξη.

2. (α) Ο αλγόριθμος έχει ως εξής:

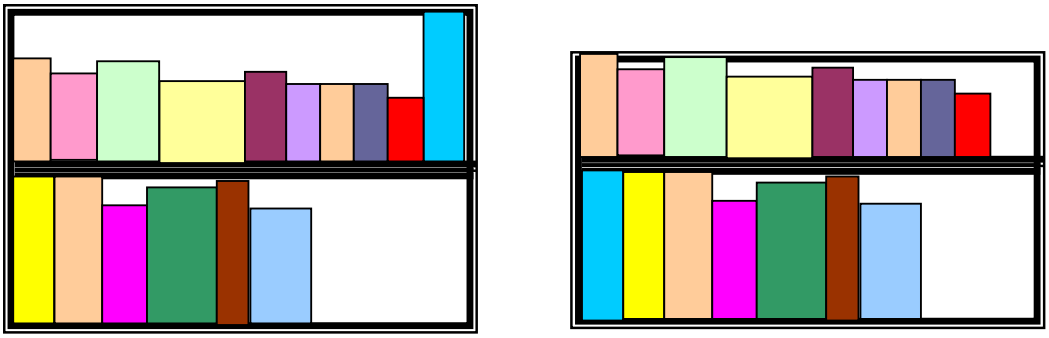
(ι) Ξεκίνα με το πρώτο ράφι.
(ιι) Εφόσον υπάρχει χώρος στο ράφι για το επόμενο στη σειρά βιβλίο τότε τοποθέτησε το βιβλίο στο ράφι. Επανάλαβε το βήμα.
(ιιι) Αν δεν υπάρχει χώρος για το επόμενο βιβλίο, τότε δημιούργησε ένα καινούριο ράφι και επέστρεψε στο βήμα (ιι).

Χρονική πολυπλοκότητα: Θ(n) όπου n είναι το πλήθος των βιβλίων.

Ορθότητα: Ο αλγόριθμος αυτός δημιουργεί τοποθέτηση των βιβλίων που ελαχιστοποιεί τον αριθμό χρησιμοποιούμενων ραφιών.

Ας υποθέσουμε πως η τοποθέτηση του αλγόριθμου χρησιμοποιεί k ράφια με την κατανομή βιβλίων σε ράφια $\langle b_{11},b_{12},...,b_{1r_1}\rangle,...,\langle b_{k1},b_{k2},...,b_{kr_k}\rangle$
και η βέλτιστη τοποθέτηση $\langle b_{11},b_{12},...,b_{1p_1}\rangle,...,\langle b_{m1},b_{m2},...,b_{mp_m}\rangle$
χρησιμοποιεί m ράφια. Θεωρήστε το r_1. Προφανώς, αφού ο αλγόριθμος δεν σταματά να γεμίζει κάθε ράφι εκτός και αν αυτό γεμίσει, συμπεραίνουμε ότι $r_1 > p_1$ και επίσης ότι η τοποθέτηση $\langle b_{11},b_{12},...,b_{1r_1}\rangle,...,\langle b_{m1},b_{m2},...,b_{mp_m}\rangle$ είναι επίσης βέλτιστη ως προς τον αριθμό ραφιών που χρησιμοποιεί.
Επαγωγικά τα ίδια επιχειρήματα ισχύουν για κάθε ράφι της τοποθέτησης που δημιουργεί ο αλγόριθμος. Επομένως k=m και ο αλγόριθμος πράγματι δίνει τη βέλτιστη λύση.

(β) Όχι, ο άπληστος αλγόριθμος δεν λύνει το νέο πρόβλημα. Αντιπαράδειγμα φαίνεται στο πιο κάτω σχήμα.



Το τελευταίο βιβλίο στο ψηλότερο ράφι της πρώτης τοποθέτησης μπορεί να τοποθετηθεί στο δεύτερο ράφι μειώνοντας έτσι το ύψος του ψηλότερου ραφιού.

(γ) Ας θεωρήσουμε τοποθέτηση των βιβλίων b_m, …, b_n, σε ράφια. Τότε, παρατηρούμε ότι για το πρώτο ράφι έχουμε αρκετές επιλογές: μπορούμε να τοποθετήσουμε από 1 μέχρι και k βιβλία, όπου k είναι ο μέγιστος ακέραιος για τον οποίο ισχύει ότι $t_m + \ldots + t_{m+k-1} \leq L$ (δηλαδή το μήκος του ραφιού δεν χωρεί περισσότερα από τα k πρώτα βιβλία). Αφού αποφασίσουμε πόσα βιβλία θα τοποθετήσουμε στο πρώτο ράφι, απομένει να τοποθετήσουμε τα υπόλοιπα βιβλία,

b_{m+k}, …, b_n, σε ράφια. Ανάμεσα σε αυτές τις επιλογές εμείς θέλουμε να διαλέξουμε αυτή που ελαχιστοποιεί το ύψος των ραφιών που χρησιμοποιεί. Επομένως το H[m] δίνεται ως εξής:

$$H[m] = \begin{cases} h_n & \text{if} \quad m = n \\ min_{m \le k \le l}(H[k+1] + max(h_m, ..., h_k)) & \text{if} \quad m \neq n \end{cases}$$

όπου το l είναι τέτοιο ώστε $h_m + ... + h_l \le L$ και $h_m + ... + h_{l+1} > L$.

Από αυτή την αναδρομική σχέση παρατηρούμε ότι για υπολογισμό κάθε H[m] απαιτείται γνώση κάποιων H[k], k>m. Η βασική περίπτωση δίνεται από το H[n] και το ζητούμενο είναι το H[1]. Επομένως χρησιμοποιούμε ένα πίνακα H μήκους n και γεμίζουμε τις θέσεις του ξεκινώντας από την H[n] και προχωρώντας προς τα αριστερά. Σε ψευδοκώδικα ο αλγόριθμος έχει ως εξής:

```
library(int H[n], int h[n], int b[n])
  H[1..n] = ∞;
  H[n]= h[n];
  for (m = n-1; m > 0; m--)
     k=m;
     t=t[m];
     h = h[m];
     while ( k <= n && t <= L )
          if H[m] > h + H[m+k]
               H[m] = h + H[m+k];
          k++;
          t = t+t[k];
          h = max(h, h[k]);
  return h[1];
```

Ο χρόνος εκτέλεσης του αλγόριθμου είναι $O(n^2)$.

(δ) Για υπολογισμό όχι μόνο του συνολικού ύψους της βέλτιστης τοποθέτησης αλλά και των στοιχείων της συγκεκριμένης τοποθέτησης, πρέπει να επεκτείνουμε τον αλγόριθμό μας έτσι ώστε όταν συναντά μια καλύτερη τοποθέτηση των βιβλίων να σημειώνει τον διαμερισμό των βιβλίων που πετυγχαίνει αυτή την τοποθέτηση. Ουσιαστικά στον αλγόριθμο μας ενδιαφέρει για κάθε H[m] το k για το οποίο τελικά H[m]= max(h[m], …, h[k]) + H[k+1], που απεικονίζει στον αριθμό του τελευταίου βιβλίου που τοποθετείται στο πρώτο ράφι της βέλτιστης τοποθέτησης. Φυλάγουμε αυτές τις πληροφορίες σε ένα νέο πίνακα B[n] και επεκτείνουμε τον αλγόριθμο ως εξής:

```
library2(int H[n], int h[n], int b[n])
  H[1..n] = ∞;
  H[n] = h[n]; B[n] = n;
  for (m = n-1; m > 0; m--)
     k=m;
     t=t[m];
```

```
        h = h[m];
        while ( k <= n && t <= L )
             if H[m] > h + H[m+k]
                  H[m] = h + H[m+k];
                  B[m] = k;
             k++;
             t = t+t[k];
             h = max(h, h[k]);
    return h[1];
```

Για επιστροφή της βέλτιστης τοποθέτησης χρησιμοποιούμε την πιο κάτω διαδικασία:

```
k=1;
while (B[k] != n)
   printf B[k];
   k=B[k+1];
```

Ο χρόνος εκτέλεσης της διαδικασίας παραμένει στην τάξη $O(n^2)$.

3. (α) Η ζητούμενη αναδρομική εξίσωσης είναι η
$T(n) = 6T(n/3) + 6(n/3), \qquad T(1) = 1$

Με τη μέθοδο της επανάληψης έχουμε

$$\begin{aligned}
T(n) &= 6T(n/3) + 2n \\
&= 6 \cdot T(n/3) + 2n \\
&= 6^2 T(n/3^2) + 6 \cdot (2n/3) + 2n \\
&= 6^2 T(n/3^2) + 4n + 2n \\
&= 6^3 T(n/3^3) + 6^2 \cdot (2n/3^2) + 4n + 2n \\
&= 6^3 T(n/3^3) + 2^3 n + 2^2 n + 2n \\
&= 6^i T(n/3^i) + 2^i n + \ldots + 2^2 n + 2n \\
&= 6^{\log_3 n} T(n/3^{\log_3 n}) + 2^{\log_3 n} n + \ldots + 2^2 n + 2n \\
&= 6^{\log_3 n} + 2n(2^{\log_3 n - 1} + \ldots + 2^2 + 2 + 1) \\
&= 6^{\log_3 n} + 2n(2^{\log_3 n} - 1)
\end{aligned}$$

Τώρα αφού $2^{\log_3 n} = n^{\log_3 2}$, $\quad 6^{\log_3 n} = 2^{\log_3 n} \cdot 3^{\log_3 n} = n^{\log_3 2} \cdot n$, και
$T(n) = n^{\log_3 2} \cdot n + 2n(n^{\log_3 2} - 1) \in \Theta(n^{1+\log_3 2}) = \Theta(n^{\log_3 6})$.

(β) Ο αλγόριθμος του φροντιστηρίου 1 έχει χρόνο εκτέλεσης $\Theta(n^{\lg 3})$. Ο προτεινόμενος αλγόριθμος έχει χρόνο εκτέλεσης που δίνεται από την πιο κάτω αναδρομική εξίσωση:

$T(n) = m \cdot T(n/3) + cn$
$T(1) = 1$

Χρησιμοποιώντας το θεώρημα γενικής χρήσης συμπεραίνουμε ότι

- Αν $n \in O(n^{\log_3 m - \varepsilon})$, δηλαδή $m>3$, $T(n) \in O(n^{\log_3 m})$.

- Αν $n \in \Theta(n^{\log_3 m})$, δηλαδή m=3, $T(n) \in O(n^{\log_3 m} \lg n) = O(n \lg n)$.
- Αν $n \in \Omega(n^{\log_3 m + \varepsilon})$, δηλαδή m<3, $T(n) \in O(n)$

Επομένως, για m≤3 ο αλγόριθμου είναι πιο αποδοτικός από τον γνωστό αλγόριθμο. Επίσης το ζητούμενο ισχύει όταν m>3 και

$$n^{\log_3 m} < n^{\lg 3}.$$

Δηλαδή

$$\log_3 m < \lg 3 \Leftrightarrow m < 3^{\lg 3} \Leftrightarrow m < 5.9$$

΄Αρα m=5 είναι ο μεγαλύτερος ακέραιος που ικανοποιεί το ζητούμενο.

4. (α) BAD[X-1, Y] = BAD[X,Y-1]= TRUE.

(β) Ο αλγόριθμος έχει ως εξής:

(i) Ξεκίνα από τη θέση (1,1) και μάρκαρε τη θέση ως θέση την οποία έχεις επισκεφθεί.

(ii) Προχώρησε προς μια από τις κατευθύνσεις Δεξιά, Αριστερά, Πάνω, Κάτω, προς την οποία δεν έχεις μέχρι στιγμής κινηθεί δεδομένου ότι (α) η διασταύρωση στην οποία φθάνεις υπάρχει στη σχάρα (β) δεν είναι κακόφημη και (γ) δεν την έχεις επισκεφθεί μέχρι τώρα.

(iii) Αν μια τέτοια κατεύθυνση υπάρχει τότε μάρκαρε τη θέση αυτή ως θέση την οποία έχεις επισκεφθεί και, αν η θέση αυτή είναι η (X,Y) τότε τερμάτισε την εκτέλεση. Διαφορετικά επανάλαβε από το βήμα (ii).

(iv) Αν μια τέτοια θέση δεν υπάρχει οπισθοδρόμησε μια θέση προς τα πίσω. Αν έχεις φθάσει στη θέση (1,1), απάντησε πως δεν υπάρχει μονοπάτι που να ικανοποιεί τις προδιαγραφές του προβλήματος. Διαφορετικά, επανάλαβε από το βήμα (ii).

Παρατηρούμε ότι όταν επισκεφθούμε για πρώτη φορά κάποια διασταύρωση επιστρέφουμε πίσω σε αυτή μόνο αν η προσπάθεια εύρεσης μονοπατιού από τη διασταύρωση προς κάποια κατεύθυνση πάνω στη σχάρα αποτύχει. Επομένως, αφού από κάθε διασταύρωση μπορούν να γίνουν το πολύ 4 αναζητήσεις, κάθε διασταύρωση συναντιέται το πολύ σταθερό αριθμό φορών. Επομένως, ο χρόνος εκτέλεσης του αλγόριθμου είναι της τάξης O(XY).

(γ) Ας γράψουμε K[n] για το σύνολο των κόμβων της σχάρας που βρίσκονται σε ελάχιστη απόσταση n από τον προορισμό (X,Y). Τότε έχουμε ότι

K[0] = {(X,Y)}

K[n] = {(i,j) | BAD[i,j]=FALSE, $(i,j) \notin$ K[0],…, K[n-1] και
{(i-1,j), (i+1,j), (i,j-1), (i,j+1)}$\cap$ K[n-1] $\neq \varnothing$}

Ζητούμενο του προβλήματος μας είναι το m για το οποίο ισχύει $(1,1) \in$ K[m].

Χρησιμοποιώντας αυτή την ιδέα μπορούμε να κτίσουμε το συντομότερο μονοπάτι που ικανοποιεί τις προδιαγραδές του προβλήματος, ξεκινώντας από το τέλος και

προχωρώντας προς τα πίσω, δηλαδή από τον προορισμό μας προχωρούμε στις ασφαλείς διασταυρώσεις που βρίσκονται ένα βήμα πιο πίσω (σύνολο K[1]), από τις διασταυρώσεις K[1], στις διασταυρώσεις K[2] που βρίσκονται ένα βήμα πιο πίσω από αυτές, και κατά συνέπεια δύο βήματα πιο πίσω από το (X,Y) και ούτω καθεξής. Χρησιμοποιούμε τους πίνακες Visited, Next, και Distance, για να φυλάγουμε για κάθε διασταύρωση Δ πληροφορίες για το αν έχουμε ήδη επισκεφθεί τη Δ και, αν ναι, τον επόμενο κόμβο στο συντομότερο μονοπάτι από τη Δ προς το (X,Y) και την απόσταση του μονοπατιού, αντίστοιχα.

```
labyrinth(int X, int Y, int BAD[x,y])
  queue K;
  Distance[1..X,1..Y]=∞;
  Visited[1..X, 1..Y]= False;

  Distance[X,Y]=0;
  Visited[X,Y]= True;
  K = [(X,Y)];

  while (!IsEmpty(K))
     p = Dequeue(K);
     if (p != (1,1))
       for all neighbors q of p
          if (BAD[q] == False)
               if (Visited[q] == False)
                    Next[q]= p;
                    Distance[q] = Distance[p] +1;
                    Enqueue(q,K);
  if (p==(1,1))
     print p;
     while (p!= (X,Y))
          print next(p);
          p = next(p);
```

Παρατηρούμε ότι επισκεπτόμαστε κάθε κόμβο ακριβώς μια φορά. Επομένως ο χρόνος εκτέλεσης του αλγόριθμου είναι O(XY).