Αλγόριθμοι και πολυπλοκότητα: 1η σειρά γραπτών ασκήσεων

Ονοματεπώνυμο: Τσαγκαράκης Στυλιανός **ΑΜ:** 03115180

Άσκηση 1: Ασυμπτωτικός Συμβολισμός, Αναδρομικές Σχέσεις

(α)

Συνάρτηση	Τάξη Μεγέθους
n^2	$\Theta(n^2)$
$2^{(\log_2 n)^4}$	n^4 = $\Theta(n^4)$
$\frac{\log(n!)}{\log(n)^3}$	$\Theta(\frac{n}{log^2 n})$ γιατί $\log(n!) = \Theta(n \log n)$
$n * 2^{2^{2^{100}}}$	$cn = \Theta(n)$
$\log\binom{n}{\log n}$	$\Theta(\log^2 n)$ γιατί $\left(\frac{n}{k}^k < \log\binom{n}{k} < \frac{ne}{k}^k\right)$
$\frac{\log^2 n}{\log \log n}$	$> \frac{\log^2 n}{\log n} = \Omega(\log n) \subset O(poly(\log n))$
$\log^4 n$	$= \Theta(poly(\log n)) = O(\sqrt{n})$ $\forall \varepsilon > 0 : \log^d n = O(n^\varepsilon)$ για $\varepsilon = \frac{1}{2}$
$(\sqrt{n})!$	?
$\binom{n}{6}$	$= \frac{n1}{6!(n-6)!} = \Theta(n^6)$
$\frac{n^3}{\log^8 n}$	$\leq \frac{n^3}{\sqrt{n}} = \Theta(n^{\frac{5}{2}})$ ισχύει για $\frac{n^3}{\log^{10} n}$
$(\log_2 n)^{\log_2 n}$	$= \Theta(n^{\log \log n})$
$\log\binom{2n}{n}$	$\leq \log\left(\frac{(2n)^{2n}}{n^n (2n-n)^{2n-n}}\right) = \log 2^{2n} = \Theta(n)$ επίσης: $\binom{n}{k} \leq \frac{n^n}{k^2 (n-k)^{n-k}}$
$n \sum_{k=0}^{n} \binom{n}{k}$	$O(n2^n)$
$(\sqrt{n})^{\log_2 log_2(n!)}$	$\Theta(n^{logn})$
$\sum_{k=1}^{n} k2^{-k}$	$\Theta(1)$ αφού αποδεικνύεται ότι $2^{-k} = 2 - \frac{n+2}{2^n}$
$\sum_{k=1}^{n} k2^k$	$2(1 + (n-1)2^n) = O(n2^n)$ $\sum_{i=1}^{n} = \Theta(2^n)$

Τελικά έχουμε:

$$
\sum_{k=1}^{n} k2^{-k} \subseteq \log\binom{n}{\log n} \subseteq \frac{\log^2 n}{\log\log n} = \log^4 n \subseteq \frac{\log n!}{(\log n)^3} \subseteq
$$

$$
\log\binom{2n}{n} = n * 2^{2^{2^{100}}} = \Theta(n) \subseteq n^2 \subseteq \frac{n^3}{\log^8 n} \subseteq \binom{n}{6} \subseteq
$$

$$
(\log_2 n)^{\log_2 n} = \Theta\binom{\log\log^d n}{n} \subseteq 2^{\log_2 4} = \Theta(n^{log^3 n}) \subseteq
$$

$$
\sum_{k=1}^{n} k2^{k} = n\sum_{k=0}^{n}\binom{n}{k} = \Theta(n * 2^n) \subseteq (\sqrt{n})^{\log_2 log_2(n!)}
$$

(β)

1. $T(n) = 2T(\frac{n}{3}) + n\log n$ $a = 2, b = 3$ $2f(\frac{n}{3}) < f(n) \Rightarrow 2\frac{n}{3}\log\frac{n}{3} < n\log n$ $f(n) = n\log n = \Theta(n\log n)$ Περίπτωση 3: $\Theta(n\log n)$
2. $T(n) = 3T(\frac{n}{3}) + n\log n$ Με βάση το δέντρο αναδρομής βγαίνει το εξής: $T(n) = 3T(\frac{n}{3}) + n\log n$ $T(\frac{n}{3}) = 3T(\frac{n}{9}) + \frac{n}{3}\log\frac{n}{3} \dots T(\frac{n}{3^k}) = T(1) = \frac{n}{3^k}\log\frac{n}{3^k}$ όπου $k = \log_3 n$ άρα $T(n) = \sum_{i=0}^{k} n\log\frac{n}{3^k}$ άρα $T(n) = n\log_3 n(\log n - \log_3 n)$ και $\log_3 n = \frac{\log n}{\log 3}$ οπότε τελικά: $T(n) = \Theta(n\log^2 n)$
3. $T(n) = 4T(\frac{n}{3}) + n\log n$ $a = 4, b = 3$ Περίπτωση 1 $f(n) = n\log n \Rightarrow n^{\log_3 4} \Rightarrow T(n) = \Theta(n^{\log_3 4})$
4. $T(n) = T(\frac{n}{2}) + T(\frac{n}{3}) + n$ Επειδή $\frac{n}{2} + \frac{n}{3} < n$ υποθέτουμε ότι $\mathrm{T}(n) = \Theta(n)$ και αποδεικνύω με δέντρο. (όμοια με 2)
5. $T(n) = T(\frac{n}{2}) + T(\frac{n}{3}) + \mathrm{T}(\frac{n}{6}) + n$ Είναι $\frac{n}{2} + \frac{n}{3} + \frac{n}{6} = n$ Υποπτεύομαι ότι $Tau(n) = \Theta(n\log n)$ και αποδεικνύεται πάλι με δέντρο αναδρομής. Ύψος: $\Theta(\log n)$ Κορυφών: $\Theta(n)$ $\frac{\text{Χρόνος}}{\text{Επίπεδο}} = \Theta(n)$ από το 1.
6. $\mathrm{T}(n) = T(n^{\frac{5}{6}}) + \Theta(\log n)$ Θέτω $m = \log_6 n$ Άρα $T(m) = T(\frac{5m}{6}) + \Theta(m)$ Από Μ.Τ. $a = 1, b = \frac{6}{5} \Rightarrow T(m) = \Theta(m)$
7. $T(n) = T(\frac{n}{4}) + n^{\frac{1}{2}}$ Από Μ.Τ. $\log_b a = log_4 1 = 0, d = \frac{1}{2} \Rightarrow T(n) = \Theta(n^d) = \Theta(\sqrt{n})$

Άσκηση 2: Ταξινόμηση

(α)

1. Βρίσκουμε το μέσο του μη ταξινομημένου πίνακα. Το κάνουμε χρησιμοποιώντας τον παρακάτω αλγόριθμο της Quick Select (όπου $k = \frac{n}{2}$ στην περίπτωσή μας):

```
quickselect_modified(arrayA, left, right, k)
    if right - left + 1 > n/k
        start = 0
        end = SZ-1
        pivot = random element of arrayA
        for each element i in arrayA
            if i <= pivot
                arrayB[start] = i //B is an array of size "SZ"
                start = start + 1
            else
                arrayB[end] = i
                end = end - 1
        //it will finally be start=end;
        arrayB[end] = pivot
```

```
        if (end <= 1)
            quickselect_modified(arrayB, end+1, right, k)
        else if(end >= right-1)
            quickselect_modified(arrayB, left , end-1, k)
        else
            quickselect_modified(arrayB, end+1, right, k)
            quickselect_modified(arrayB, left , end-1, k)
    return
```

Η παραπάνω αναδρομική συνάρτηση θα εκτελέσει τις εντολές του for loop για n φορές την πρώτη φορά, $\frac{n}{2}$ την δεύτερη φορά κοκ. Συνολικά θα τρέξει $\log k$ φορές από n στοιχεία κάθε φορά. Άρα πολυπλοκότητα $O(n \log k)$. Με οπτικοποίηση του αλγόριθμου θα έχουμε ένα δέντρο ύψους $\log k$ και n στοιχείων σε κάθε επίπεδο. Ο αλγόριθμος είναι βέλτιστος γιατί πρέπει να κατασκευάζονται κάθε φορά $\frac{n!}{\left((\frac{n}{k})!\right)^k}$ φύλλα και ύψους $\log \varphi \acute{\upsilon} \lambda\lambda\omega\nu$. Άρα ο χρόνος εκτέλεσης: $\geq \log \frac{n!}{\left((\frac{n}{k})!\right)^k} = \log n! - k \log\left((\frac{n}{k})!\right) \Rightarrow (Stirling) = \Theta(n \log k)$

2. Γενικά η Quicksort θέλει $O(n \log n)$ στην μέση περίπτωση. Άρα ταξινομώ τους k υποπίνακες με κόστος $O(\frac{n}{k} * \log(\frac{n}{k}) * k)$ δηλαδή $O(n * \log(\frac{n}{k}))$. Κάθε συγκριτικός αλγόριθμος ταξινόμησης πρέπει να έχει $\Omega(n * \log(\frac{n}{k}))$, γιατί αλλιώς θα κατασκευάζαμε πλήρη συγκριτικό αλγόριθμο ταξινόμησης $O(n \log n) = \Theta(n \log k) + O(n \log(\frac{n}{k}))$.

(β)

- Έστω ότι έχουμε ένα υποπίνακα A' του A ο οποίος περιέχει M διαφορετικά στοιχεία του A σε αύξουσα σειρά.
- Διασχίζουμε τον A και για κάθε στοιχείο που συναντάμε κάνουμε δυαδική αναζήτηση στον A' για να βρούμε με ποιό στοιχείο του A' είναι ίσο.
- Αυξάνουμε ένα Counter σε εκείνο το σημείο και βάση αυτού μπορούμε να παράξουμε μια ταξινόμηση για τον A.
- Πολυπλοκότητα: $O(n \log M)$

Πρέπει να υπολογίσουμε τον A'. Θα χρησιμοποιήσουμε μια παραλλαγή της Mergesort.

- Χωρίζουμε τον A στην μέση και παίρνουμε τους πίνακες A_1' και A_2'.
- Αναδρομικά υπολογίζουμε τους A_1' και A_2', καθένας τους περιέχει το πολύ M στοιχεία.
- Συγχωνεύουμε τους A_1' και A_2' με την διαφορά οτι αν συναντήσουμε 2 ίσα στοιχεία πετάμε ο ένα.
- Ο πίνακας που προκύπτει είναι ο A'.
- Πολυπλοκότητα:

  $T(n) = 2T(\frac{n}{2}) + \Theta(\mathrm{M}) \Rightarrow T(n) = O(M \log n)$

<u>Συνολική Πολυπλοκότητα</u>: $O(n \log M + M \log n)$

<u>Συγκεκριμένα εδώ αφού έχουμε</u> $M = O(\log^d n)$:

$$O(n \log \log^d n) = O(d * n \log \log n) = O(n \log \log n)$$

Ουσιάστικά θέλουμε να υπολογίσουμε το πλήρθος των διαφορετικών ταξινομημένων ακολουθιών μήκους n με στοιχεία που παίρνουν τιμές στο $\{1,\ldots,M\}$, οπού $M=\max(A)$.

Έστω A_k το πλήθος των στοιχείων με τιμή k.

Αρκεί να υπολογίσουμε με πόσους τρόπους μπορούμε να αναθέσουμε τιμές από το 0 στο n, στους αριθμούς $\mathrm{A}_1,\mathrm{A}_2,\ldots,\mathrm{A}_\mathrm{M}$ έτσι ώστε $A_1+A_2+\ldots+A_M=n$.

Το παραπάνω μπορεί να γίνει με $\binom{n+M-1}{n}$ τρόπους.

Άρα κάθε συγκριτικός αλγόριθμος χρειάζεται τουλάχιστον $\Omega(\log\binom{n+M-1}{n}))=\Omega(n\log M)$.

Στην περίπτωση μας $M=\log^d n$ συνεπώς δεν έχουμε n διαφορετικά στοιχεία. Άρα δεν ισχύει το κάτω φράγμα του $\Omega(n\log k)$.

Παρακάτω είναι ο ψευδοκώδικας για την άσκηση:

```
merge_modified(arrayA, l, r)
    //arrayCounter is of size M
    //where M is the max element of arrayA
    initialize arrayCounter to 0
    copy arrayA to arrayAcopy

    mergeSort(arrayA, l, r)
    //now arrayA is A' and arrayAcopy is A

    for each element i of arrayA
        //find the position of each element of arrayA
        //in the Acount and increase counter by one
        index = binarySearch(arrayAcopy, l, r, i)
        arrayCounter[index] = arrayCounter[index] + 1

    //Create A sorted
    for each element i of arrayA
        for each element j of arrayCounter
            j = i

mergeSort(arrayA, l, r)
    if (l < r)
        m = (r-l)/2
        //Sort first and second halves
        mergeSort(A, l, m)
        mergeSort(A, m+1, r)
        merge_without_equals(A, l, m, r)

merge_without_equals(arrayA, l, m, r)
    arrayL = left half of arrayA
    arrayR = right half of arrayA

    while left in arrayL && right in arrayR
        if left < right
            add in arrayA the element left
            left = next element of arrayL
        else if left > right
```

```
            add in arrayA the element right
            right = next element of arrayR
        else //left == right
            if last element of arrayA == left || arrayA.empty()
                add left to arrayA
            left = next element of arrayL
            right = next element of arrayR

    for each remaining element i in arrayL
        add i to arrayA
    for each remaining element i in arrayR
        add i to arrayA
```

Άσκηση 3: Διάστημα ελάχιστου μήκος που καλύπτει όλους τους Πίνακες

(α)

```
min_diff_2(arrayA1, n1, arrayA2, n2)
    //n1 is the size of arrayA1
    //n2 is the size of arrayA2
    while i<n1 AND j<n2
        min = abs(A1[i]-A2[j]) //|A1 - A2|
        if A1[i] <= A2[j]
            i = i + 1
        else
            j = j + 1
        if i>n1 OR j>n2
            exit
```

Η παραπάνω συνάρτηση θα περάσει μια φορά τον κάθε πίνακα οπότε είμαστε στα πλαίσια της γραμμικής πολυπλοκότητας $\Rightarrow O(n)$.

Είμαστε σίγουροι ότι τα ζευγάρια που "χάνουμε" έχουν μεγαλύτερη διαφορά από αυτά που υπολογίζουμε αφού κάθε φορά προχωράμε το μικρότερο.

Απόδειξη:

Έστω $x_k \leq y_z$ και min η έως τώρα ελάχιστη διαφορά.

Έχουμε: $\forall(i,k) : i \leq k \rightarrow x_i \leq x_k$ και $\forall(z,j) : z \leq j \rightarrow y_z \leq y_j$ άρα $|x_i - y_j| \geq |x_k - y_z| \geq min$

(β)

```
min_diff_n(m arrays)
    while each element i_m in arraym
        min = min from i_m elements
        max = max from i_m elements
        diff = max - min
        if i1 <= i2 && i1 <= i3
            i1 = next element of array1
        else if i2 <= i1 && i2 <= i3
            i2 = next element of array2
        else if i3 <= i1 && i3 <= i2
            i3 = next element of array3
        else if ...
            ...
```

Ο παρακάτω αλγόριθμος έχει πολυπλοκότητα $O(mN)$ όπου $N = \sum_{k=1}^{m} n_k$.

Αυτό γιατί περνάει όλα τα στοιχεία του κάθε πίνακα, άρα N, και κάθε φορά περνάει το στοιχείο από τον κάθε πίνακα που επιλέγουμε άρα m. Συνολικά $O(mN)$.

(γ)

Θα μπορούσαμε να χρησιμοποιήσουμε heap για να αποθηκεύουμε το επιλεγμένο στοιχείο του κάθε πίνακα σε κάθε επανάληψη.

Έτσι η εισαγωγή/αφαίρεση του στοιχείου θα παίρνει $O(\log m)$.

Αντίστοιχα η εύρεση του min.

Η εύρεση του max μπορεί να παίρνει μόνο O(1) καθώς θα συγκρίνουμε το στοιχείο που αλλάζουμε σε κάθε επανάληψη με το ήδη max. Στην πρώτη επανάληψη θα κάνουμε (m-1) συγκρίσεις αλλά πάλι θεωρείται O(1) συνολικά.

Δηλαδή βελτιώνουμε το $O(mN)$ σε $O(N \log m)$

Άσκηση 4η: Αναζήτηση

(α)

Τις 1.000.000 φιάλες (διακριτές καταστάσεις) μπορούμε να τις κωδικοποιήσουμε με 20 bit ($2^{20} > 1.000.000$ ενώ $2^{19} \approx 500.000$). Έστω ότι κάθε bit ξεκινώντας από το LSB αναπαριστά έναν εθελοντή και όταν το bit αυτό γίνει 1 τότε πίνει ο αντίστοιχος εθελοντής.

> Για παράδειγμα η $1^η$ φιάλη είναι 0...01 και έχει 1 το $1^ο$ bit. Αυτό σημαίνει ότι από την φιάλη θα πιεί μόνο ο $1^{ος}$ εθελοντής.
>
> Η $2^η$ φιάλη είναι 0...010 και έχει 1 το $2^ο$ bit. Αυτό σημαίνει ότι από την $2^η$ φιάλη πίνει μόνο ο $2^{ος}$ εθελοντής.
>
> Στην $3^η$ (0...011)θα πιεί ο $1^{ος}$ και ο $2^{ος}$
>
> κ.ο.κ.

Έτσι θα μπορέσουμε να αντιληφθούμε με βάση συνδυασμών ποια φιάλη έχει το μαγικό φίλτρο.

(β)

```
find_best_split(arrayA, days)
    //lower search bound = max element -> O(n)
    //upper search bound = sum of all -> O(n)
    upper = 0 //sum
    lower = 0
    for each element i in arrayA
        upper = upper + i //calculate sum
        if i > lower
            lower = i

    temp[days] = 0 //array of 'days' positions initialized to 0
    while true
        test_number = (upper+lower)/2 //integer division
        if test_number = 0 //this means max - min < 1 then max is our answer
            return max //found result
        count = 0
        for each element i in arrayA
            if (temp[count] + i) < test_number
                temp[count] = temp[count] + i
            else
                count = count + 1
                if count > days
                    lower = test_number //lower upper bound
                    exit for loop
                else
                    temp[count] = temp[count] + i
        if count <= days
            upper = test_number
        continue //move to next test
```

Το άνω όριο αναζήτησης είναι το άθροισμα όλων των στοιχείων ενώ το κάτω όριο είναι το μεγαλύτερο στοιχείο. Με σταθερό k στο διάστημα [κάτω όριο, πάνω όριο] κάνουμε δυαδική αναζήτηση. Αν με το νούμερο που θα επιλέξουμε το ταξίδι είναι επιτυχές τότε διαιρούμε /2 το αριστερό μισό $(median \to upperbound)$ αλλιώς το δεξί μισό $(median \to lowerbound)$.

Πρακτικά ο αλγόριθμος είναι πολυπλοκότητας $O(upperbound) = O(\sum_{i=1}^{n} k_n)$ αφού κάνουμε δυαδική αναζήτηση και μας επηρεάζει το άνω όριο. Οποιαδήποτε βελτιστοποίηση άνω και κάτω ορίου δεν προσφέρει σημαντική βελτίωση στον αλγόριθμο.

Άσκηση 5: Επιλογή

(α)

Λύση:

- Ρωτάμε πλήθος στοιχείων μικρότερα από $(\frac{M}{2}), F_S(\frac{M}{2})$.
- Αν $F_S(\frac{M}{2}) < k$ συνεχίζουμε αναδρομικά στο $[\frac{M}{2}, M]$.
- Αν $F_S(\frac{M}{2}) > k$ συνεχίζουμε αναδρομικά στο $[0, \frac{M}{2}]$.
- Τελειώνουμε όταν βρούμε t τέτοιο ώστε $F_S(t-1) < k$ και $F_S(t) \geq k$.
- Επιστρέφουμε το t σε χρόνο $O(\log M)$.

Παρακάτω και σε ψευδοκώδικα:

```
return_kth_element(arrayA, k)
    upper = M
    lower = 0
    while true
        mid = (upper + lower)/2
        temp = Fs(mid) //external function
        if temp < k
            lower = upper/2
        else if temp > k
            upper = upper/2
        else
            return mid
```

Ο αλγόριθμος πάντα θα επιστρέφει το k-οστό στοιχείο καθώς κάθε φορά στον έλεγχο ψάχνουμε έναν αριθμό t τέτοιο ώστε $F_S(t) \geq k$ αλλά και $F_S(t-1) < k$ δηλαδή υπάρχουν k στοιχεία μικρότερα ή ίσα από τον t συμπεριλαμβανομένου του εαυτού του. Άρα είναι το ζητούμενο νούμερο. Στην χειρότερη περίπτωση θα γίνει $O(\log M)$ δηλαδή να χρειαστεί να ψάξουμε "όλο" τον πίνακα ή αλλιώς να μην πετυχαίνουμε το στοιχείο με κανένα κόψιμο παρά μόνο στο τελευταίο.