CSORE 4231 ANALYSIS OF ALGORITHMS I Homework 1

Francis Zhang UNI: xz3279

February 4, 2024

Problem 1

Ordering by asymptotic growth rates. CLRS 3-3.a
Rank the following functions by order of growth; that is, find an arrangement $g1, g2, ..., g30$ of the functions satisfying $g1 = \Omega(g2), g2 = \Omega(g3), ..., g29 = \Omega(g30)$. Partition your list into equivalence classes such that functions $f(n)$ and $g(n)$ are in the same class if and only if $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$\sqrt{2}^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(\frac{3}{2})^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln \ln n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg \lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2 \lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

Solution:

$g_1(n) = 2^{2^{n+1}}$

$g_2(n) = 2^{2^n}$

$g_3(n) = (n+1)!$

$g_4(n) = n!$

$g_5(n) = e^n$

$g_6(n) = n2^n$

$g_7(n) = 2^n$

$g_8(n) = (\frac{3}{2})^n$

$g_9(n) = (\lg n)^{\lg n}$

$g_{10}(n) = n^{\lg \lg n}$

$g_{11}(n) = (\lg n)!$

$g_{12}(n) = n^3$

$g_{13}(n) = n^2$

$g_{14}(n) = 4^{\lg n}$

$g_{15}(n) = n \lg n$

$g_{16}(n) = \lg(n!)$

$g_{17}(n) = 2^{\lg n}$

$g_{18}(n) = n$

$g_{19}(n) = (\sqrt{2})^{\lg n}$

$g_{20}(n) = 2^{\sqrt{2 \lg n}}$

$g_{21}(n) = \lg^2 n$

$g_{22}(n) = \ln n$

$g_{23}(n) = \sqrt{\lg n}$

$g_{24}(n) = \ln \ln n$

$g_{25}(n) = 2^{\lg^* n}$

$g_{26}(n) = \lg^* n$

$g_{27}(n) = \lg^*(\lg n)$

$g_{28}(n) = \lg(\lg^* n)$

$g_{29}(n) = n^{1/\lg n}$

$g_{30}(n) = 1$

Where $g_9(n) = (\lg n)^{\lg n}$ and $g_{10}(n) = n^{\lg \lg n}$, $g_{13}(n) = n^2$ and $g_{14}(n) = 4^{\lg n}$, $g_{15}(n) = n \lg n$ and $g16(n) = \lg(n!)$, $g_{17}(n) = 2^{\lg n}$ and $g_{18}(n) = n$, and $g_{29}(n) = n^{1/\lg n}$ and $g_{30}(n) = 1$ satisfies $f(n) = \Theta(g(n))$.
For $g_9(n) = \Theta(g_{10}(n))$, since $a^{log_b c} = c^{log_b a}$.
For $g_{13}(n) = \Theta(g_{14}(n))$, since $4^{\lg n} = n$.
For $g_{15}(n) = \Theta(g_{16}(n))$, since $\lg(n!) = \sum_{i=1}^{n} \lg i$ where

$$\underbrace{\lg \frac{n}{2} + ... + \lg \frac{n}{2}}_{\#\ of\ n\ is\ \frac{n}{2}} \le \lg \tfrac{n}{2} + \lg(\tfrac{n}{2}+1) + ... + \lg n \le \lg n + \lg(n-1) + ... + \lg 1 \le \underbrace{\lg n + ... + \lg n}_{\#\ of\ n\ is\ n}.$$ As

$\lg \frac{n}{2}^{\frac{n}{2}} = \frac{n}{2} \lg \frac{n}{2} = \frac{n}{2}(\lg n - 1) = \frac{n}{2} \lg n - \frac{n}{2} \ge \frac{n}{4} n \lg n$
For $g_{17}(n) = \Theta(g_{18}(n))$, since $2^{\lg n} = n$. For $g_{29}(n) = \Theta(g_{30}(n))$, since $n^{1/\lg n} = n^{\log_n 2} = 2 = \Theta(1)$.

Problem 2

Big-O. CLRS 3-4 b, d, e, g
Let $f(n)$ and $g(n)$ be asymptotically positive functions. Prove or disprove each of the following conjectures.
b. $f(n) + g(n) = \Theta(\min(f(n), g(n)))$.
d. $f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$.
e. $f(n) = O((f(n))^2)$.
g. $f(n) = \Theta(f(n/2))$.

Solution:
b. Disprove: Pick $f(n) = n$, $g(n) = 1$. When testing $f(n) + g(n) = O(\min(f(n), g(n)))$. Assume $\exists c > 0, N > 0$, $\forall n > N$ we can have $n + 1 \le c$ and $n \le c - 1$. However, as n has an upper bound now, this does not satisfy $\forall n > N$ contradicts the assumption.
d. Disprove: $f(n) = O(g(n))$ directly implies $f(n) \le c \cdot g(n)(\exists c > 0, N > 0, \forall n > N)$. And $f(n) \le c \cdot g(n)$ implies $2^{f(n)} \le 2^{c \cdot g(n)}$. To prove $2^{f(n)} = O(2^{g(n)})$, we need $2^{f(n)} \le c' \cdot 2^{g(n)}$. Now we can do some comparison between $2^{c \cdot g(n)}$ and $c' \cdot 2^{g(n)}$. Let $f(n) = 4n, g(n) = 3n, andc = 1.5$, we have $4n \le 1.5 \cdot 3n$ and $2^{4n} \le 2^{1.5 \cdot 3n}$. To make it easy, let $c' = c = 1.5$. We cannot say that $2^{4n} \le 1.5 \cdot 2^{3n}$ $\forall n > N forsomec, N$ as n's upper bound is $log_2 1.5$.

e. Disprove: Let $f(n) = \frac{1}{n}$. We cannot say that $\frac{1}{n} \leq c \cdot (\frac{1}{n})^2$. As $n \leq c$, n has an upper bound now.
g. Disprove: Assume $f(n) = O(f(n/2))$. Let $f(n) = 2^n$. Then $2^n \leq c \cdot 2^{\frac{n}{2}}$. That is $n \leq 2 \lg c$, n has an upper bound. Hence $f(n) \neq O(f(n/2))$, and $f(n) \neq \Theta(f(n/2))$.

Problem 3

Horner's Rule. CLRS 2-3
The following code fragment implements Horner's rule for evaluating a polynomial

$$\begin{aligned} P(x) &= \sum_{k=0}^{n} a_k x^k \\ &= a_0 + x(a_1 + x(a_2 + ... + x(a_{n-1} + xa_n)...)), \end{aligned}$$

given the coefficients $a_0, a_1, ..., a_n$ and a value for x :

```
1 $y = 0$
2 for $i = n$ downto 0
3     $y = a_i + x \cdot y$
```

a. In terms of Θ-notation, what is the running time of this code fragment for Horner's rule?

b. Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each term of the polynomial from scratch. What is the running time of this algorithm? How does it compare to Horner's rule?

c. Consider the following loop invariant:
At the start of each iteration of the **for** loop of lines 2-3,

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k.$$

Interpret a summation with no terms as equaling 0. Following the structure of the loop invariant proof presented in this chapter, use this loop invariant to show that, at termination, $\sum_{k=0}^{n} a_k x^k$.

d. Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by the coefficients $a_0, a_1, ..., a_n$.

Solution:
a. $n \cdot \Theta(1) = \Theta(n)$.
b.

```
1 $y = 0$
2 for $i = n$ downto 0
3     $z_i = 1$
4     for $j = i$ downto 1
5         $z_i = z_i \cdot x$
6     $y = a_i z_i + y$
```

The runing time is $\Theta(n^2)$, this is slower to Horner's rule.
c.
Invariant: P returns the summation of simplify power series of given $a_0, ..., a_n$.
Initialization: $y = 0$.
Maintenance:
$y = a_i + x \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k = a_i x^0 + \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^{k+1} = \sum_{k=-1}^{n-(i+1)} a_{k+i+1} x^{k+1} = \sum_{k=0}^{n-i} a_{k+i} x^k$.
Termination: $i = 0, y = \sum_{k=0}^{n-i} a_{k+i} x^k$.
d. We can conclude from the termination above that the given code correctly evaluates a polynomial characterized by the coefficients $a_0, a_1, ..., a_n$.

Problem 4

Recurrences. Give asymptotic upper bounds on the following recurrences. If the master method applies, you can use that. If not, draw a recursion tree and come up with an upper bound.

(a) $T(n) = 6T(n/9) + n$

(b) $T(n) = 3T(n/3) + n^{1.5}$

(c) $T(n) = T(n-2) + 1/n$

(d) $T(n) = 4T(n/2) + n^2 \log n$

(e) $T(n) = \sqrt{n}T(\sqrt{n}) + n$

Solution:
Master theorem:

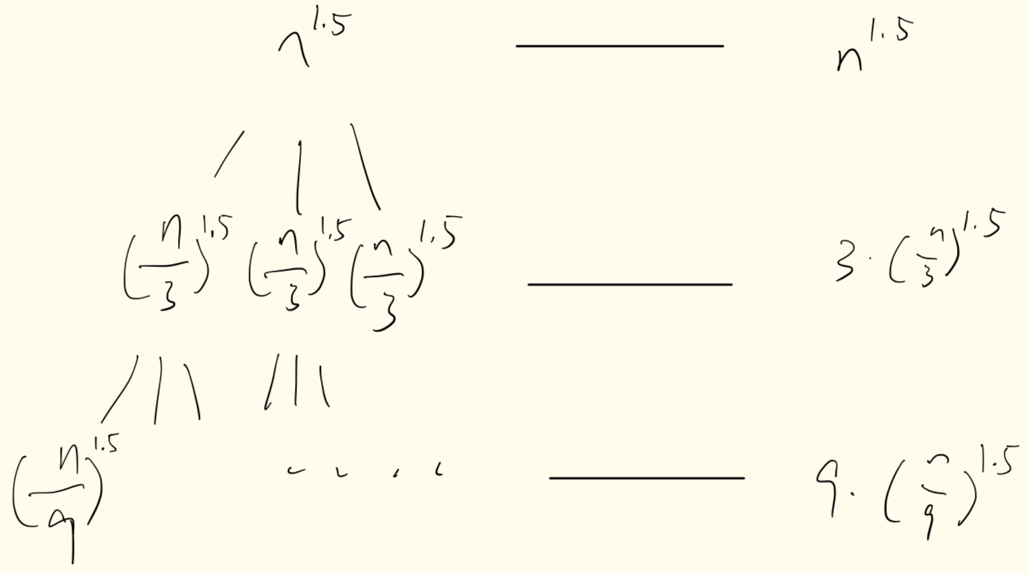
$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Cases	$f(n)$	$n^{\log_b(a)}$	
1	$\leq$		$\Rightarrow \Theta(n^{\log_b(a)})$
2	$=$		$\Rightarrow \Theta(n^{\log_b(a)} \log n)$
3	$\geq$		$\Rightarrow \Theta(f(n))$

Table 1: Master theorem

a. Applying Master theorem, $n \geq n^{\log_9 6} \Rightarrow$ Case 3 $\Rightarrow \Theta(n)$.

b.c.d. $n^{1.5}, \frac{1}{n}, n^2 \log n$ are not polynomials. We draw recursion trees below.


$n^{1.5}$
$n^{1.5}$
$(\frac{n}{3})^{1.5}$ $(\frac{n}{3})^{1.5}$ $(\frac{n}{3})^{1.5}$
$3 \cdot (\frac{n}{3})^{1.5}$
$(\frac{n}{9})^{1.5}$
$9 \cdot (\frac{n}{9})^{1.5}$

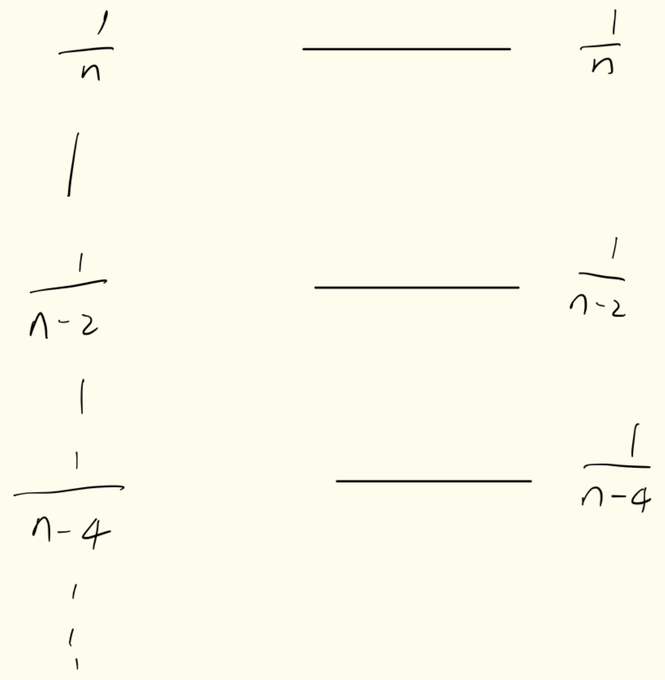
$$\text{Total} = n^{1.5}\left(1^{-0.5} + 3^{-0.5} + \cdots + \right)$$

$$= n^{1.5} \cdot \frac{1\left(1-\left(\frac{1}{\sqrt{3}}\right)^{n}\right)}{1-\frac{1}{\sqrt{3}}}$$

$$= n^{1.5} \cdot \frac{1-\frac{1}{(\sqrt{3})^{n}}}{1-\frac{1}{\sqrt{3}}}$$

$$\le n^{1.5} \cdot \frac{1}{1-\frac{1}{\sqrt{3}}} = O(n^{1.5})$$

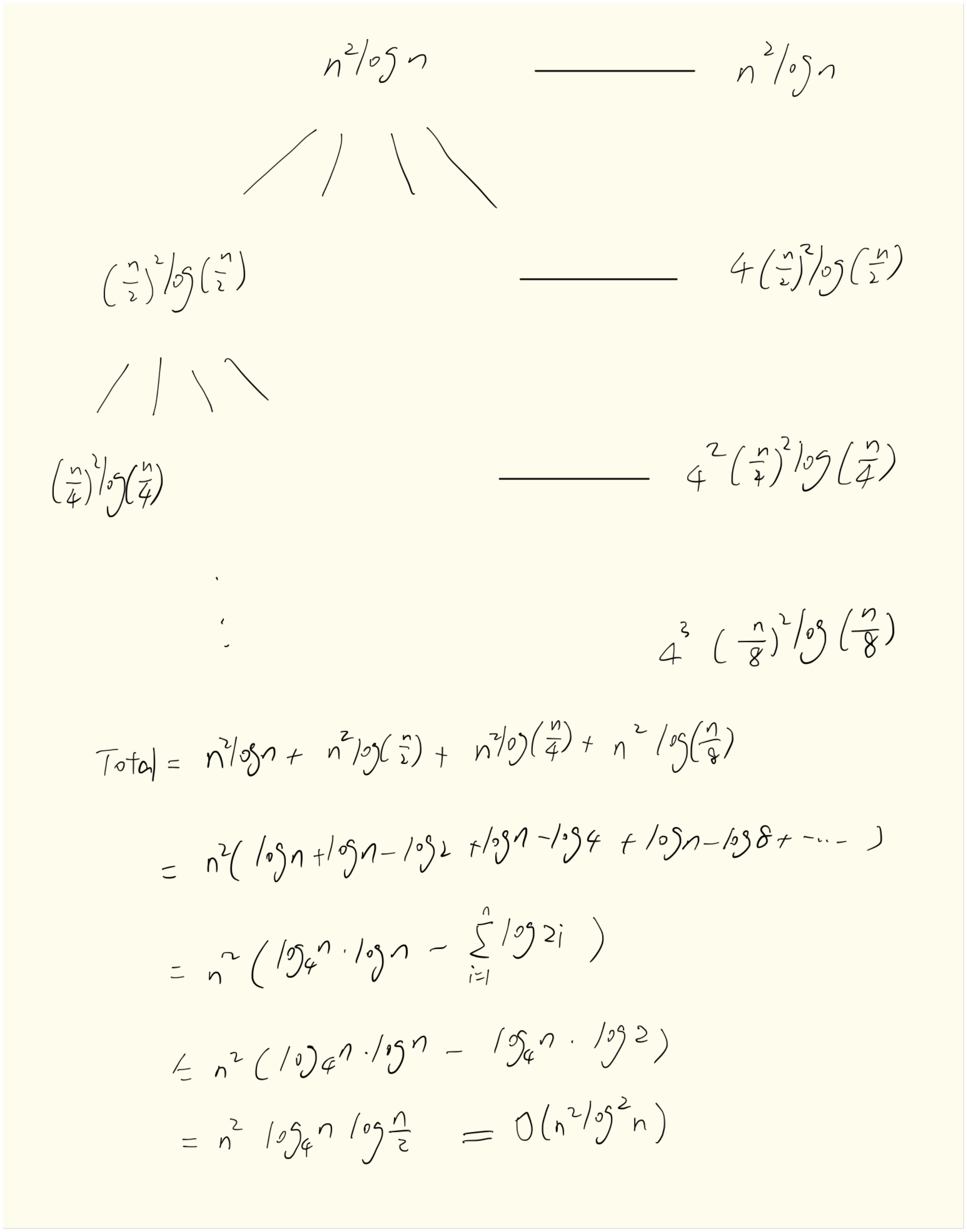
Figure 1: **b.** $T(n) = 3T(n/3) + n^{1.5}$



$$\text{Total} = \underbrace{\frac{1}{n} + \frac{1}{n-2} + \frac{1}{n-4} + \cdots}_{n/2 \text{ terms}}$$

$$\leq \frac{n}{2} \cdot (1) = O(\tfrac{n}{2})$$

Figure 2: **c.** $T(n) = T(n-2) + 1/n$


$n^2 \log n$
$n^2 \log n$
$(\frac{n}{2})^2 \log(\frac{n}{2})$
$4(\frac{n}{2})^2 \log(\frac{n}{2})$
$(\frac{n}{4})^2 \log(\frac{n}{4})$
$4^2(\frac{n}{4})^2 \log(\frac{n}{4})$
$4^3(\frac{n}{8})^2 \log(\frac{n}{8})$

$$\text{Total} = n^2\log n + n^2\log(\tfrac{n}{2}) + n^2\log(\tfrac{n}{4}) + n^2\log(\tfrac{n}{8})$$

$$= n^2(\log n + \log n - \log 2 + \log n - \log 4 + \log n - \log 8 + \cdots)$$

$$= n^2\left(\log_4 n \cdot \log n - \sum_{i=1}^{n} \log 2i\right)$$

$$\le n^2(\log_4 n \cdot \log n - \log_4 n \cdot \log 2)$$

$$= n^2 \log_4 n \log\frac{n}{2} = O(n^2\log^2 n)$$

Figure 3: **d.** $T(n) = 4T(n/2) + n^2 \log n$

e. We cannot find some integer a and b to use Master theorem here, however we can do some modifications to apply MT.
Let $n = 2^k$, $\sqrt{n} = 2^{k/2}$, then $k = \log n$. The equation becomes:

$$T(2^k) = 2^{\frac{k}{2}} T(2^{\frac{k}{2}}) + 2^k \tag{1}$$

Dividing (1) by 2^k, we get:

$$\frac{T(2^k)}{2^k} = \frac{T(2^{\frac{k}{2}})}{2^{k/2}} + 1 \tag{2}$$

Let $y(k) = \frac{T(2^k)}{2^k}$, we have:

$$y(k) = y(\frac{k}{2}) + 1 \tag{3}$$

And now we can use MT here. As $1 = k^{\log_2 1}$, it is Case 2. We have $y(k) = \Theta(\log k)$. Substituting
$T(n) = 2^k y(k) = 2^{logn} \Theta(\log k) = \Theta(n \log \log n)$.

Problem 5

Fun with Arrays.

(a) We are given sorted arrays A and B with n integers each, as well as an integer c. The problem is to determine if there exist indexes i, j with $1 \leq i$, $j \leq n$ such that $2A[i] + 3B[j] = c$. The output will be either "yes" or "no". Design and analyze an algorithm for this problem that runs in linear time (i.e., $O(n)$ time) in the worst case.

(b) Now we have one array A of length n. The problem is to find three integers i, j, k with $1 \leq i, j, k \leq n$ such that $2A[i] + 3A[j] = A[k]$. The output will be either i, j, k or "no". Design and analyze an algorithm for this problem that runs in $O(n^2)$ time.

Solution:
a.

```
Define the function below as FUNARRAY$(A, B, c)$
Assume $n \geq 1$. For empty arrays A, B, the answer is no here.
$i = 1, j = 1$
for $i, j \leq n$
    if $2A[i] + 3B[j] = c$, output yes.
    if $2A[i] + 3B[j] > c$, output no.
    else:
        if $2A[i] < 3B[j]$, $i = i + 1$.
        if $2A[i] > 3B[j]$, $j = j + 1$.
        else: $i = i + 1$, $j = j + 1$
            if $2A[i+1] + 3B[j] = c$ or $2A[i] + 3B[j+1] = c$, $i = i + 1$ or $j = j + 1$, output yes.
            if $2A[i+1] + 3B[j] > c$ and $2A[i] + 3B[j+1] > c$, output no.
            if $2A[i+1] + 3B[j] > c$ and $2A[i] + 3B[j+1] < c$, $j = j + 1$.
            if $2A[i+1] + 3B[j] < c$ and $2A[i] + 3B[j+1] > c$, $i = i + 1$.
```

```
15            if $2A[i+1] + 3B[j] < c$ and $2A[i] + 3B[j+1] < c$, $i = i+1$, $j = j+1$.
16 else: output no.
```

b. I'm using the function `FUNARRAY` from part **a** here

```
1 Assume $n \geq 1$. For empty array A, the answer is no here.
2 for $k = n$ downto 1
3     if FUNARRAY($A, A, A[k]$), output $i, j, k$.
4     else: $k = k - 1$.
5 else: output no.
```

Problem 6

Even more fun with arrays and weird collectables. You are given as input an array $A[1, ..., n]$, with n entries. Also, you know that this array consists of a series of 0's followed by a series of 1's, but you don't know how many of each there are. Your goal is to find the first 1 in the array. There is a cost associated with checking the value of an entry in the array. Each time you check a value and it turns out to be 0, you must pay 1 dollar. If value turns out to be 1, you have to pay 1 "Famous Algorithmist Trading Card (FATC)". In each of the parts of this question, you will have some number of dollars and some number of FATCs. You must give an algorithm which will return the index of the first (lowest indexed) 1 that works given your resources. For each part, write pseudocode and explain why your algorithm works, given your resources.

(a) You have n dollars and n FATC

(b) You have 1 dollars and n FATC

(c) You have $10 \log n$ dollars and $10 \log n$ FATC

(d) You have 2 dollars and $10n^{1/2}$ FATC

(e) You have k dollars and $10kn^{1/k}$ FATC, for some constant k

Solution:
a.

```
1 for $i = 1$ upto $n$
2     if $A[i] = 1$, return $i$.
3     else: $i = i + 1$.
```

As we have n dollars and n FATC, there is no limit to check 0's and 1's 1-by-1. That is from $A[1]$ to $A[n]$ 1-by-1 go through all 0's until the first index of 1.
b.

```
1 for $i = n$ downto 1
2     if $A[i] = 0$, return $i + 1$.
3     else: $i = i - 1$.
```

As we have 1 dollars and n FATC, there is no limit to check 1's but can only check 0 once. That is from $A[n]$ to $A[1]$ 1-by-1 go through all 1's until the first index of 0. Then the index of first 1 is the last index of 0 plus 1.
c.

```
$j = 0$, $B = A$
**for** $|B| \neq 1$
    **if** $B[\lceil \frac{n}{2} \rceil] = 0$, $B = B[\lceil \frac{n}{2} \rceil + 1, ..., n]$, $j = \lceil \frac{n}{2} \rceil$.
    **else**: $B = B[1, ..., \lceil \frac{n}{2} \rceil]$, $j = \lceil \frac{n}{2} \rceil$.
**if** $A[j] = 0$, return $j = j + 1$
**else**: return $j$.
```

As we have $10 \log n$ dollars and $10 \log n$ FATC, we just use binary search here. Set the pivot point first. As there cannot be 0's behind any 1's, once the pivot is 0, we just take a look at the part of the array behind that 0. Found a 1 as the pivot, we just check the array before(including that 1). The last steps just illustrate the index of 1. When the updated array has size 1, the second last updated array can be: "001", "01", "11". When the pivot point is 0, the index of 1 just 1 step behind that pivot. If pivot point is 1, then that is the 1 we are finding!
d.

```
$i = 0$
**for** $i = n, n - \sqrt{n}, n - 2\sqrt{n}$,... until $\sqrt{n}$
    **if** $A[i] = 0$, $i = i + \sqrt{n}$
    **else**: $i = i - \sqrt{n}$
**if** $i = 0$, $i = i + \sqrt{n}$
**for** $i = i$ **downto** $i - \sqrt{n} + 1$
    **if** $A[i] = 0$, return $i = i + 1$
    **else**: $i = i - 1$
**if** $i = i - \sqrt{n}$, $i = i + 1$
```

We have only two chance to test 0. We used 2 for loops to check 1's. which worst case is $O(\sqrt{n})$. By dividing $A[1, 2, ...n]$ to $\sqrt{n}$ parts, and find the right part to check every elements in that sub-array.
e.

```
$i = 0$
**for** $i = n, n - n^{\frac{k-1}{k}}, n - 2n^{\frac{k-1}{k}}$,... until $n^{\frac{k-1}{k}}$
    **if** $A[i] = 0$, $i = i + n^{\frac{k-1}{k}}$
    **else**: $i = i - n^{\frac{k-1}{k}}$
**if** $i = 0$, $i = i + n^{\frac{k-1}{k}}$
**for** $i = i, i - n^{\frac{k-2}{k}}, i - 2n^{\frac{k-2}{k}}$ until $n^{\frac{k-2}{k}}$
    **if** $A[i] = 0$, $i = i + n^{\frac{k-2}{k}}$
    **else**: $i = i - n^{\frac{k-2}{k}}$
**if** $i = i - n^{\frac{k-2}{k}}$, $i = i + n^{\frac{k-2}{k}}$
...
```

```
... for $i = i$ downto $i - n^{\frac{1}{k}} + 1$
...     if $A[i] = 0$, return $i = i + 1$
...     else: $i = i - 1$
... if $i = i - n^{\frac{1}{k}}$, $i = i + 1$
```

Similar idea just like part **d**. As we only have k possibilities to check 0's, the difference is we divide A to $n^{\frac{1}{k}}$ parts. After find the "right" part keep dividing it, until the length of it is $n^{\frac{1}{k}}$.