1. Currently what is the fastest for the following. Give the name and running time

(a) matrix multiplication:

Algorithm: Francois Le Gall's improvement on Coppersmith and Winograd.

Running time: $O(n^{2.3728})$

Source: arxiv.org/abs/1401.7714

(b) Integer multiplication:

Algorithm: Schonhage-Strassen algorithm

Running time: $O(n \log n \log\log n)$

Source: en.wikipedia.org/wiki/Schonhage-Strassen_algorithm

(c) Polynomial multiplication:

Algorithm: Fast Fourier Transform

Running time: $O(n \log n)$

Source: Practical fast polynomial multiplication, Robert T Moenck

(d) Minimal spanning tree:

Algorithm: Kruskal's algorithm

Running time: $O(E \log V)$, E is the Edge & V is vertex

Source: CLLS, Introduction to Algorithms, 3rd edition, page 631

(e) Shortest path:

Algorithm: Dijkstra's algorithm

Running time: $O(|V|^2)$

Source: en.wikipedia.org/wiki/Dijkstra's_algorithm

(f) Factoring:

Algorithm: Pollard-Strassen Algorithm

Running time: $O(n E_B^{0.5})$

Source: math.stackexchange.com/questions/185524/pollard-strassen-algorithm

3-2 correctness of bubblesort

(a) we need to show that A is a copy of A, but probably the elements in arranged in different order.

(b) At the start of each iteration of the for loop the subarray A[j to n] will have smallest element in A[j]. In other words the first element on the left is the smallest (as we are adding elements in ascending/non-decreasing order). Also A[j to n] contains same number of elements as before, probably in different order.

Base loop invariant:

Initialization:

After the loop when A[j to n] contains only one element i.e the last element of A.

Maintenance:

If A[j] is smaller than A[j-1], we swap jth & j-1st elements in each loop. Since we only swap elements instead than adding any new elements (a) is still true. Similarly, among A[j] & A[j-1], we make sure A[j-1] is the smaller of the two (using loop in case they are not). Therefore in A[j to n] (b) holds.

Termination:

The loop terminates after j=i. This also signifies that A[i] is the smallest element in the subarray A[i to n] contains the original elements, where A[i] is the smallest.

(c) At the start of each iteration the array A[1 to i-1] contains elements in sorted order, which contains subset of elements from A[1 to n] & array A[i to n] contains elements which are larger or equal than elements in A[1 to i-1]

Base loop invariant

Initialization:

After according to (b) the loop starts A[i to i-1], when i=1 the elements A[1 to 0]. This is true since the array has only one element.

Maintenance:

From fact (b) loop invariant we can confirm A[1 to i-1] is in sorted order. After the second loop terminates, A[i] will be the smallest element of A[i to n], which also means A[1 to i] is in sorted order. That is, it works for this and next iteration.

Termination: The loop terminates when i = length+1. Simply, we can say that after the loop terminates the array A[1 to n] will be sorted in non-decreasing order.

(d) worst-case running time of bubble sort is $\Theta(n^2)$, because the inner loop may have to swap n items in each outer loop, which iterates n times. Although the inner loop does not loop n times, it is n (because few iteration less does not count). Insertion sort also has worst case running time of $\Theta(n^2)$.

2.3 (a) $\Theta(n)$ because there is one loop that goes from 0 to n

(b) let's assume A is an array with A.length elements

Pseudocode:

```
Polynomial(A, x):
    y = 0
    i = 0
    for i to A.length
     a = A[i]
     z = 1
     for j = 1 to i
         z *= x
     y += a * z
    return y
```

Since the code has two loops i.e; one outer and another inner which executes approximately n times each. Therefore the running time is $O(n^2)$. Compared to the previous code it is inefficient.

(c)

Initialization:

When the loop begins, $y = 0$ & $i = n$. Therefore the summation is from $k = 0$ to $n-(i+1)$ or $n-n-1 = -1$. In this case the loop does not get evaluated & y remains 0. Thus our loop invariant holds.

Maintenance:

At some iteration loop instance we have

$$y = a_i + x \cdot y$$

or $y' = a_i + x \cdot y$, where y' is the value computed from previous value of y

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

$$= a_i + \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^{k+1}, \text{ when } x \text{ goes inside}$$

$$= \sum_{k=-1}^{n-(i+1)} a_{k+i+1} x^{k+1}, \text{ this accomodates } a_i$$

If we reflect $k' = k+1$

$$y' = \sum_{k'=0}^{n-i} a_{k'+i} x^{k'}, \text{ with replacement and adjustment of range}$$

Let $i' = i-1$

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

At the end of the loop i is set to $i'-1$, before the next iteration. This way our loop invariant remains true for next iteration.

Termination: When the loop terminates

$i = -1$, because 0 is the last element

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

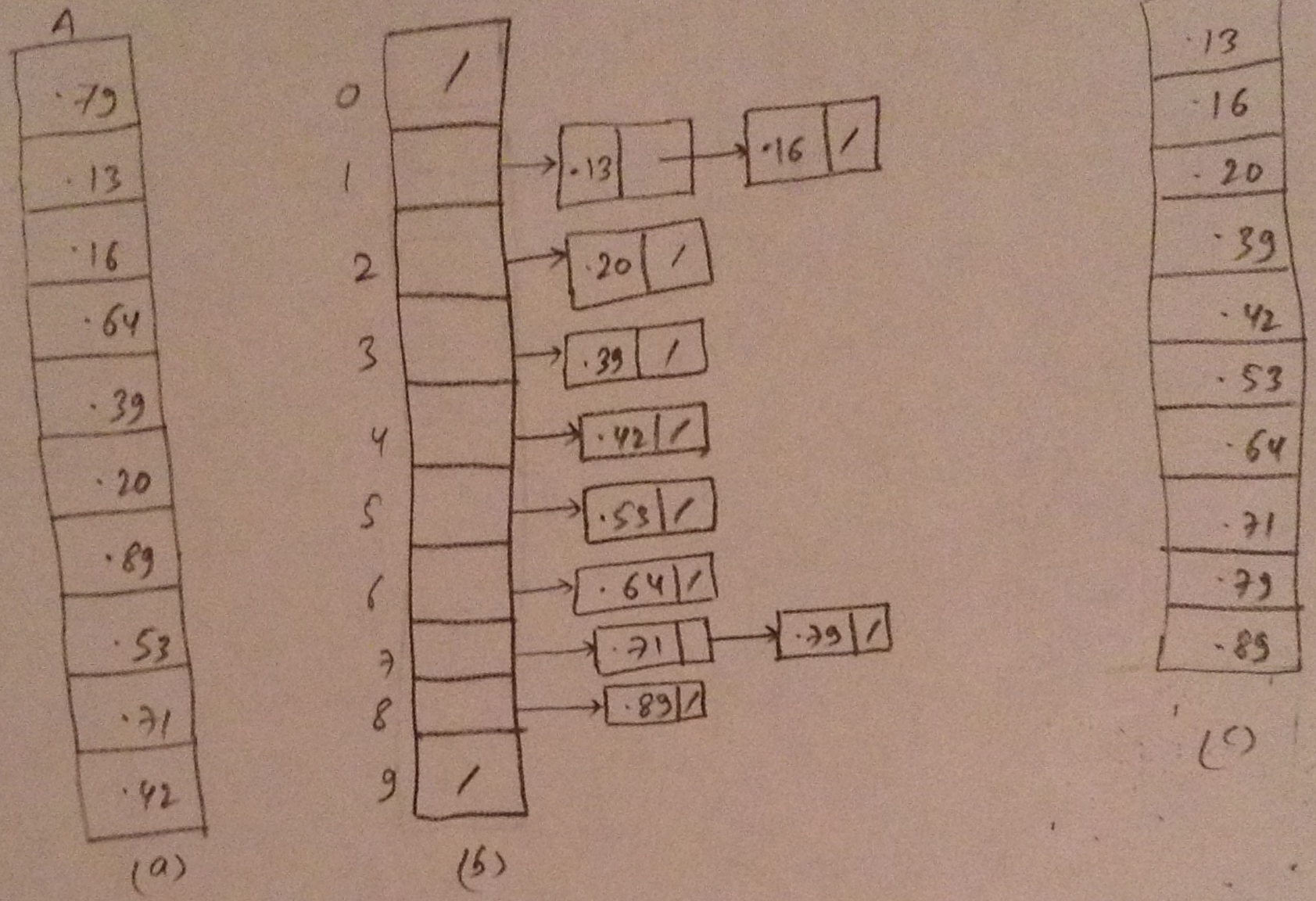
$$= \sum_{k=0}^{n} a_k x^k$$

After the loop terminates we get our polynomial evaluation.

(d) The polynomial characterized by the coefficients $a_0, a_1, \ldots, a_n$ initializes the solution to 0 ($\because y=0$). It then calculates summation when it enters the loop in successive iterations, using the Horner's rule. After the termination, as shown above it calculates the solution for the polynomial. We also know that i decreases by value 1 at each iteration & when it becomes less than 0, the loop terminates. At this stage it completely evaluates the polynomial.

8-4-1 Using figure 8.4 as a model illustrate the operation of BUCKET-SORT on the array

$A = \{.79, .13, .16, .64, .39, .20, .89, .53, .71, .42\}$



The operation of BUCKET-SORT for $n = 10$

(a) The input array $A[1 \text{ to } 10]$

(b) The output array $B[0 \text{ to } 9]$ of sorted lists (buckets) after execution of line 8 of the algorithm. The bucket i holds values in the half-open interval $[i/10, (i+1)/10)$.

(c) Final output after concatenation of the lists at each bucket.

8-4

(a) We can compare each red jug with each blue jug. One blue jug with each blue jug. Each blue jug may need to compare with every red jug to find if it matches. Therefore, the running time is $\Theta(n^2)$ in the worst case.

(b) Using decision tree we can compare for 3 cases (i.e the edges)

i) If both red and blue jugs are equal

ii) If red jug is larger than blue jug

iii) If red jug is smaller than blue jug

The height of the decision tree is equal to the case when it takes longest time or the worst-case no. of comparisons

If we label jugs from 1 to n i.e blue jugs from 1 to n & red jugs also from 1 to n, the outcome is as follows for the first blue jug.

$b_1, P(1)$, where P is permutations of all red jugs

In general form the pairing is

$(i, P(i))$, where $i = 1$ to n,

This pair contains pair of red & blue jugs that all matched. The permutation represents all the different outcomes, which is $n!$

Our node with 3 brackets (i, ii & iii) has 3^h leaves, where h is the height of the tree. Since we know the decision tree must have $n!$ leaves

$$3^h \geq n! \geq (n/e)^n \quad \text{implies}$$

$$h \geq n\log_3 n - n\log_3 e = \Omega(n \lg n)$$

Algorithms solving this problem use $\Omega(n \lg n)$ comparisons.

3.2.2 Minimum and maximum

$\lceil 3n/2 \rceil - 2$

when we compare the numbers for comparison to select the maximum or the minimum by one. If $a_1 < a_2$ & $a_3 < a_4$, we have excluded a_2 and a_3 from being minimum, but only one for maximum.

To optimize this we can split & compare the elements in pairs after $n/2$ comparisons we would have selected the maximum and minimum to $n/2$ each. Now we have two problem set of $n/2$ each one for minimum & another for maximum.

In the pair comparison, the smaller of the pair is compared to current min & the larger of the pair to current max. To do this we need 3 comparisons for each pair except the first pair, that takes only one comparison.

This results in

$\lceil 3n/2 \rceil - 2$ comparisons