ion 1: [10 points] 7 8

Use the Master Theorem to solve the following recurrences. State which case of the Master Theorem you are using (and justify) for each of the recurrences

a. $T(n) = 2T\left(\frac{n}{8}\right) + \sqrt[3]{n}$

$a = 2 \quad b = 8 \quad d = \frac{1}{3}$

$a = b^d$ case 2

$\Theta(n^d \log n) = \Theta(n^{\frac{1}{3}} \log n)$ ✓

b. $T(n) = 4T\left(\frac{n}{2}\right) + n^3$

$a = 4 \quad b = 2 \quad d = 3$

$a < b^d$ case 1

$\Theta(n^d) = \Theta(n^3)$ ✓

Complete

a. The worst case running time for insertion sort is $\Theta(n^2)$ ✓ while the best case running time is $\Theta(n)$ sorted ✓.

b. Perfect hashing is better than a normal hash table because the number of expected collision is $< \frac{1}{2}$, insertion search $\Theta(1)$, storage $\Theta(n)$. Perfect hashing is done by using two levels of hashing by two hash fun, level 1, table size $b = n$, level 2 table size $m_i = n_i^2$ with uni hash fu

c. The brute-force algorithm for finding a longest common subsequence of two strings of lengths m and n takes time $\Theta(n2^m)$ ✓ while the dynamic programming algorithm takes time $\Theta(nm)$ ✓.

Question 2: [10 points] 8

1) A majority element in an array A of size N is an element that appears more than N/2 times.

a) Write an algorithm for function Has_Maj that determine if array A has a majority element or not.

bool Has_Maj (int A [], int N)

```
int maj=0;
for (int i=0; i<N; i++)
for (int j=0; j<N; j++
  if (A[i] == A[j])
  { maj++;
  if (maj >= N/2)
    return true;
  }
return false;
```

b) What is the time and space complexity of your algorithm?

time complexity $\Theta(n^2)$ ✓

space complexity 1 (in place) ✓

2) Consider the following algorithm:

```
PUZZLE(A :array of real numbers, l :integer, r :integer)
1  ▷ Assume l, r > 0 and l ≤ r
2  if l = r
3      then return A[l]
4  temp_1 ← PUZZLE(A, l, ⌊(l+r)/2⌋)
5  temp_2 ← PUZZLE(A, ⌊(l+r)/2 + 1⌋, r)
6  if temp_1 < temp_2
7      then return temp_1
8      else return temp_2
```

a) Describe in English the problem solved by this algorithm.

returns the minimum integer in a given array.

b) Formulate a recurrence describing the cost of running this algorithm on n element array: PUZZLE(A, 1, n).

$$T(n) = 2T\left(\frac{n}{2}\right) + 1$$

c) Solve the recurrence and give the resulting running time of the algorithm.

by back substitution

$T(n) = 2T\left(\frac{n}{2}\right) + 1$

$= 2\left[2T\left(\frac{n}{4}\right) + 1\right] + 1 = 4T\left(\frac{n}{4}\right) + 3$

$= 4\left[2T\left(\frac{n}{8}\right) + 1\right] + 3 = 8T\left(\frac{n}{8}\right) + 7$

$= kT\left(\frac{n}{k}\right) + (k-1)$ when $n = k$

$n = n-1 \quad 2n-1$

c) **True or False? Justify your answers**

1- Given a hash table with more slots than keys, and collision resolution by chaining, the worst case running time of a lookup is constant time.

(T) F

more slots than keys means that there is no collision

2- Given an array A[1:n] of integers, the running time of Counting Sort is polynomial in the input size n

T (F)

running time is function of input size & max element

$\Theta(n+k)$ k = max value

3- Heap sort can be used as the auxiliary sorting routine in radix so because it operates in place.

T

because it's stable sort

4- A longest common subsequence of "ABCBDAB" and "BDCABC is "ABC

T

BCBA

5- In radix sort, the algorithm sorts on individual digits starting from most significant digit then the least significant ones.

T

starting from LS digit to MS digit

Question 3: [10 points] (5)

1- Sort the array (15,2,13,4,15,1,7,18,9,10) using heap sort in descending order **show the algorithms steps in details**. What is the time complexity of the algorithm?

```
max_heapify(A, i)
{ l = left(i)
  r = right(i)
  if (A[l] > A[i])
    smallest = l;
  else smallest = i;
  if (A[r] < A[smallest])
    smallest = r;
  if (smallest ≠ i)
    swap(A[i], A[smallest])
  max_heapify(A, smallest);
}
```

```
build_heap(A)
{ for(i = ⌊A.size/2⌋; i>=0; i--)
    max_heapify(A, i)
}
```

→ $\Theta(n/2 \log n)$

```
heap_sort
{ build_heap(A)
  length = A.size;
  for(i = length; i>=0; i--)
    swap(A[i], A[0])
    A.size--;
    max_heapify(A,0);
}
```

→ $\Theta(n/2 \log n - n \log n)$

$= \Theta(n \log n)$

10

Question 4: [10 points]

On a positive integer, you can perform any one of the following 3 steps.
1) Subtract 1 from it. (n = n - 1) .
2) If its divisible by 2, divide by 2. (if n % 2 == 0 , then n = n / 2) .
3) If its divisible by 3, divide by 3. (if n % 3 == 0 , then n = n / 3) .

Now the question is, given a positive integer *n*, find the minimum number of steps that takes *n* to 1

For example

1) *For n = 1 , output: 0* 2) *For n = 4 , output: 2 (4 /2 = 2 /2 = 1)*
3) *For n = 7 , output: 3 (7 -1 = 6 /3 = 2 /2 = 1)*

a) Write a brute force algorithm to solve the above problem

```
int steps (int n)
{ int num = 0;  if (n == 1) return 0;
if (n % 2 == 0)
    num = min (num, steps(n/2) + 1);
if (n % 3 == 0)
    num = min (num, steps(n/3) + 1);

num = min (num, steps(n-1) + 1);

return num;
```

b) Using dynamic programming write a top down or bottom-up algorithm solving the above problem. **Show the recursive relation**
What is the complexity of your algorithm?

using top down

Global arr S[0 -> n] // S Values = -1

```
int steps (int n)
{ int num = 0;
if (n == 1) return 0;
if ( S[n] != 0)
    return S[n];
if (n % 2 == 0)
    num = min(num, steps(n/2) + 1);
if (n % 3 == 0)
    num = min(num, steps(n/3) + 1);
```

recursive relation

$T(n) = T(\frac{n}{3}) + T(\frac{n}{2}) + T(n-1) + 1$

with bottom up algorithm

S[0 ... n] has values = 0

```
int steps(n)
{ for (int i=2; i<=n; i++)
  { min = S[i-1] + 1;
    if (i % 2 == 0)
        min = min(min, S[i/2] + 1);
    if (i % 3 == 0)
        min = min(min, S[i/3] + 1);

    S[i] = min;
  }
  return S[n];
}
```

complexity $O(n)$