* \_\_\_\_\_\_\_ / 100 points
* **DUE FRIDAY SEPTEMBER 12, 2014 10:30AM**
* Turn in **both** an electronic copy on Canvas and a printed copy in lecture.
* **Name** Nitin Chaurasia
* **ID** 0020223463
* 1. *[20 pts] Write an algorithm that sorts n integers, where each integer in the input appears exactly once and all integers are in the range 1 to k⋅n (inclusive), where k is a positive constant.****The algorithm must be linear (Θ(n)).***
* LinearSort(input[1…..n], temp[1…..kn])
* //Assumption : temp[1…k\*n] has been initialized with all elements as zero

//Place the I’th value in its respective index e.g. place 6 goes to //index 6, 9 goes to index 9 of temp array. Rest of the values in //temp array are ZERO

* for i=1 to k\*n

{

* temp[input[i]] = input[i];//Please see comments below this loop
* }//end for

//The Face value of the given integer becomes the array index of the temp array. That way we have a sorted array, with lots of ZEROES in the temp array of size k\*n.

* //The next for loop places the values from temp array back to the input array, in sorted order

int num = 0;//to control the index of input array

* for i=0 to k\*n{
* if (temp[i]!=0){
* input[num]=temp[i];
* num++;
* }//end if

}//end for

* PS : Appendix A has the java implementation of the same algorithm.
* 2. *[8 pts] Write the statement costs and times next to the algorithm above (similar to how the book shows on page 26) and give the running time T(n) in terms of those statement costs. Simplify T(n) as much as possible by collecting common terms. See bottom of page 26 for an example of how to simplify.*

|  |  |  |
| --- | --- | --- |
| * LinearSort(input[1..n], temp[1..kn]) | * Cost | * Times |
| * **for** i=1 to k\*n * temp[input[i]] = input[i]; * **int** num = 0 * **for** i=1 to k\*n{ * **if** (*temp*[i]!=0){ *input*[num]=*temp*[i]; * num++; * }//end if * }//end for | * c1 * c2   c3  c4  c5  c6  c7 | * k\*n+1 * (k\*n)   1  k\*n+1  k\*n  n *(only n non zero values)*  n |

* T(n) = c1(k\*n)+ c2(k\*n - 1) + c3(1) + c4(k\*n) + c5(k\*n) + c6(n) + c7(n)
* T(n) = [c1+ c2 + c4 + c5 ](k\*n) + [c6 + c7](n) + [c3 – c2]
* This takes the form (Linear function)
* T(n) = A(k\*n) + B(n) + C
* T(n) = (A\*k + B)\*n + C
* T(n) = D(n)+C

So in the worst case, the complexity goes to a maximum of constant times n. Thus this algorithm has a linear time complexity.

* 3. *[2 pts] Since the previous sorting algorithm is linear (and relatively simple!), give two (2) probable reasons why it is not used in real-world applications.*
* **Reason 1**: Uses extra memory location [(kn -n) + inputArraySize]. Most of the values are zero and only those indices are filled which has a match in the Input Array.
* **Reason 2** : This algorithm fails as soon as there is a repetition of a few integers. This algorithm will only keep one element and reject the rest of the repeated elements. In real world, having input of all unique values are rare.

4. *[20 pts] Consider a guessing problem, where you try to guess a secret positive integer in the range 1 to n. Use the divide and conquer approach to design an algorithm GUESS that takes n as a parameter and attempts to find the secret by calling the helper CHECK (shown below) to check if your guess is correct (returns 0), less than the secret (returns -1) or greater than the secret (returns +1). Your algorithm should make as few guesses as possible.*

* CHECK(p)
* {
* **if** p == secret **return** 0
* **if** p < secret **return** -1
* **if** p > secret **return** 1
* }
* GUESSRANGE(min, max)//initially min is 1 and max is n

{

* // calculate midpoint (guess is always a middle element) to cut set in half
* int guess = (min+max)/2;
* // three-way comparison
* if (CHECK(p) == 1)

{

* // guess is in lower subset (left side)
* GUESSRANGE(min, guess-1);

}

* else if (CHECK(p) == -1)

{

* // guess is in upper subset (Right Side)
* GUESSRANGE(guess+1, max);

}

* else
* // guess has been found
* return guess;

}

PS : The Java implementation of the above algorithm is added as appendix B.

* 5. [10 pts] Give the recurrence relation for the previous algorithm. Assume CHECK(p)= Θ(1).
* Divide : guesses (calculates) the middle element of the given range.It takes constant time.
* Conquer : There are two recursive calls to GUESSRANGE(min, max) and each time the guess is a number which is half the range (n/2), the recurrence relation thus is

(1) if n=1

(1) if n>1

T(n) =

* 6. [10 pts] Is 2n = big-O(2n/2)? **Prove** **your answer.**
* FALSE
* f(n) = 2n  g(n) = 2(n/2)
* 0<= 2n  <= c\*2n/2
* 0<= 2n/2  <= c (divide by 2n)

Thus the value of c is not a constant, and depends on n. Thus there doesn’t exist a Constant value for which f(n)<=c.g(n)

* 7. [10 pts] Is 2n = Ω(2n/2)? **Prove** **your answer.**
* TRUE
* f(n) = 2n  g(n) = 2(n/2)
* 0 <= c.g(n) <= f(n)
* 0<= c.2n/2  <= 2n
* Trivially, for n=n0 = 0 and c= 1,for which the equality holds true, but the definition says, there exists a **positive** constant c and n0 for which the relation must hold true.

For c=2, n0 = 2 (2.22/2  <= 22  ), the equality holds true and then the function 2n is ever increasing. Thus 2n = Ω(2n/2) holds true.

* 8. [10 pts] Is nc = big-O(cn)? Assume c > 1 is a constant. **Prove** **your answer.**
* **TRUE**
* f(n) = nc and g(n) = cn
* 0<= nc <= c1\*cn
* Taking , nc <= c1\*
* nc / cn<= c1

at c=n, c1 is greater than or equal to 1. The value of c1 depends upon the value of c and n

* Since n>0 and c>1, the polynomial function nc will grow slower than the exponential function cn. The value of n0 will exist because the polynomial function grows faster for smaller values than the exponential function, but after some values, growth of exponential is faster. There will be a point of intersection after which the exponential function grows faster.
* 9. [10 pts] Is nc = Ω(cn)? Assume c > 1 is a constant. **Prove** **your answer.**
* **FALSE**

As proven in the above question, the exponential function grows faster than the polynomial function when c>1