Youssef
Naguib

Hw - 2

1) a) $T(n) \leq \begin{cases} d & \text{if } n = 1. \\ 2T(n/2) + d & \text{if } n > 1 \end{cases}$

b) Telescoping method.

$T(n) \leq 2T(n/2) + d$

$\leq 2(2(n/4) + d] + d = 4(n/4) + 3d$

$\leq 2(2(2(n/8) + d) + d) + d = 8(n/8) + 7d$

$\vdots$

$2^k (n/2^k) + 2^{k-1} d$

Let $2^k = n$, $k = \log(n)$

$T(n) = T(1) \cdot n + [2^k - 1) \cdot d$

$= n + (n-1)d = n + dn - d$

$T(n) \leq O(N)$

c) Lrun: Length of largest increasing sublist within the list. (adjacent indeces)

Rrun: Length of largest decreasing sublist within the list. Indeces of sublist must be adjacent.

maxRun: The length of the larger sublist between Lrun and Rrun.

2) a) $T(n) = 7T(n/8) + n$

$a = 7$ $\quad \log_b a = \log_8 7 = .9358$

$b = 8$

# of recursive calls is $O(n^{.9358})$

$f(n)$ is n^1

$n^{.9358} < n^1$ and in this case $f(n)$ dominates complexity, (case 3 from class)

$f(n)$ is $\Omega(n^{.9358})$

we can conclude that $T(n) = \Theta(f(n))$

so $T(n) = \Theta(n)$

b) $T(n) = 2T(n/4) + \sqrt{n}$

$a = 2$ $\quad \log_b a = \log_4 2 = \frac{1}{2}$ so # of recursive calls is $O(n^{\frac{1}{2}})$

$b = 4$

$f(n)$ is $\sqrt{n}$ or $n^{\frac{1}{2}}$ so $f(n)$ is $O(n^{\frac{1}{2}})$

In this case $f(n)$ and the number of recursive calls are "tied". (case 2 from class)

This, we can conclude that $T(n) = \Theta(n^{\log_b a} \cdot \log n)$

$$T(n) = \Theta(\sqrt{n} \cdot \log n)$$

c) $T(n) = 9T(n/3) + n^2$

$a = 9$ $\quad \log_b a = \log_3 9 = 2$ $\quad$ so # of recursive calls is $\Theta(n^{\log_b a})$
$b = 3$ $\quad = \Theta(n^2)$

$f(n) = n^2$

$f(n)$ and $n^{\log_b a}$ are "tied"
(case 2 from class)
we can conclude that $T(n) = \Theta(n^{\log_b a} \cdot \log n)$

so $T(n) = \Theta(n^2 \cdot \log n)$

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

$a = 4$ $\quad \log_b a = \log_2 4 = 2$ $\quad$ # of recursive calls $= n^{\log_b a}$
$b = 2$ $\quad = n^2$

$f(n) = n\log^2 n$

In this case the number of recursive calls dominates the complexity.

$f(n)$ is upper bounted by n^2.

Thus we can conclude that $T(n) = \Theta(n^{\log_b a})$

so $T(n) = \Theta(n^2)$

Youssef Naguib <ymn4543@rit.edu>
CS261
Write-up for hw-2 questions 3-5

3.

My approach to creating an algorithm that determines if there exists a sphere centered at the origin having two or more of the n input points on its surface was quite simple. For each point of the n 3d points being read in, I pass an x, y, and z coordinate (integers) into a function that calculates the distance of the point from the origin. This distance is calculated using the distance formula with an added z value. The value returned is stored into an array that contains all the total distances.

Once all points have been accounted for, this array gets sorted using a bucket-sort algorithm that works with floating points numbers. The bucket interval is calculated as the largest value in the array divided by the length of the array. Bucket sort runs in O(NlogN) time. After the array is sorted, I iterate through the values, and compare to the next index' value. If any two distances are the same, this means they both lie on the same sphere, and the program prints out YES and stops running. If no two equal distances are found, the program prints out NO and ends.

Running time analysis:
Iterate through input: O(N)
Calculate distance for each input (simple math operations) O(1)
Append distance to array O(1)
Bucket sort array after all distances added O(N)
Iterate through sorted array to check for duplicate distances O(N)
Total Complexity: O(N)

4.

This problem deals with food items that have a certain shelf life, which can be classified as a sort of priority value (The lower the shelf life, the higher the priority). It was clear to me that a heap would be the appropriate data structure to use for this algorithm. The program starts by creating an empty heap and setting a day counter to 0. The program then reads in the input file. Food items are stored as lists of 2 elements, the first being the arrival day and the second being the shelf life. Multiple food items can be added on the same day, but at least one has to be removed from the heap each day the heap is not empty. The shelf lives also decrease by one for each food item every day. If the root of the heap ever has a shelf live below 0, this means the food has expired and gone to waste, and the program prints out NO to indicate that it was not possible to not have food go to waste.

Once all input is read a final function is called on the heap named no_food_wasted(). This function checks for any expired items in the heap. For every element in the heap, the appropriate shelf life is decreased and it is popped from the root of the heap. If the value of the root shelf life ever goes below 0 , the program prints N0 and ends. If all food items are

successfully used within their expiration interval, the program prints YES to indicate that it was possible not to waste any food. The program then ends.

Running time analysis:

Make empty heap, initial variables. O(1)
Read in input and sort in heap based on priority.(Heap Sort) O(NLogN)
Remove a food item every day and decrease priorities O(1)
No_food_wasted() function O(N)

Total complexity: O(NlogN)

5.

This parade-sorting algorithm was very interesting to write. I actually originally tried to implement it with merge sort but ran into trouble with large input as I was sorting out of place. I rewrote the program with a modified merge sort that uses inversion counting to calculate the number of swaps that occur. The sorting is done in place and makes the program much more efficient. The program reads in the input and creates a double element list that represents a person. The first element of the list represents the patience of this person, and the second element their preferred position in the parade. The participants are added to an array in the order they are read in.

This is when my inversion_sort() function is called on the array. This function is a modification of merge sort, it splits the array in half and recursively runs on both halves, before sorting each sub list. While the lists are being sorted, inversions are calculated using index counter variables and the lengths of lists. Every time a person is moved, their patience is decreased by the correct number of inversions, and is checked to see if it is below 0. If a patience is below 0 the unhappy contestants counter gets increased by one. After the total array is sorted, the number of unhappy contestants (contestants who have lost their patience) is returned by the function and printed to stdout. The program then terminates.

Running time analysis:

Read in people and store in lists O(N)
Append to parade list O(1)
After all people added to parade line (array), run inversion_sort() O(NlogN)
Inversion_sort() is NlogN as it is a modified merge sort with some constant time operations added to it. Merge sort is O(NlogN)
Print result O(1)

Total complexity: O(NlogN)