CS-3250

Section 01

-

Homework #1

Patrick Pei

Professor Jeremy Spinrad

Due 1-27-2017

Exercises

1. Prove or disprove: For every fixed positive integer k, $1^k + 2^k + ... + n^k$ is $O(n^{k+1})$.

   Since every term in the sequence of 1 to n is less than or equal (which is the very definition of O in order notation) to n, it is clear that $1^k + 2^k + ... + n^k$ is $O(n^k + n^k + ... + n^k)$, where each base number is replaced by n. Since there are n total terms in the sequence from 1 to n, this is equivalent to $O(n * n^k)$. Finally, this can be rewritten as $O(n^1 * n^k)$ and by basic algebraic product laws of multiplying terms with the same base, this can be further simplified to $O(n^{k+1})$.

2. Prove or disprove: For every fixed positive integer k, $1^k + 2^k + ... + n^k$ is $\Omega(n^{k+1})$.

   Similar to the proof in lecture for proving the sum of the sequence of 1 to n to be $\Omega(\frac{n^2}{4})$, the numbers in the sequence of $1^k + 2^k + ... + n^k$ can be rewritten as $1^k + 2^k + ... + (\frac{n}{2})^k + (\frac{n+1}{2})^k + ... \ n^k$. By taking half of the elements in this sequence, it is clear that

   $$1^k + 2^k + ...(\frac{n}{2})^k + (\frac{n+1}{2})^k + ... \ + n^k = \Omega((\frac{n}{2})^k + (\frac{n+1}{2})^k + ... \ n^k)$$

   Furthermore, since there are $\frac{n}{2}$ elements in this new sequence, and each element is less than or equal to $(\frac{n}{2})^k$, it is clear that

   $$\begin{aligned} 1^k + 2^k + ...(\frac{n}{2})^k + (\frac{n+1}{2})^k + ... \ + n^k &= \Omega(\frac{n}{2} * (\frac{n}{2})^k) \\ &= \Omega((\frac{n}{2})^1 * (\frac{n}{2})^k) \\ &= \Omega((\frac{n}{2})^{k+1}) \\ &= \Omega(\frac{n^{k+1}}{2^{k+1}}) \end{aligned}$$

   Thus, since 2^{k+1} is a constant because k is a "fixed positive integer ", $1^k + 2^k + ... + n^k$ is $\Omega(n^{k+1})$.

3. Someone suggested that the reason you learn bubblesort is that it is good if the list is "almost sorted". To formalize this notion, we can say that the number of inversions ($I(L)$) in a list L is the number of pairs of elements x, y such that x comes before y in L, but x is greater than y.
   Show that bubblesort does not run in $O(n + I(L)$ time.

   In the following implementation of bubblesort from Cormen, Leiserson, Rivest, and Stein - Introduction to Algorithms (3rd edition), the runtime is obviously $O(N^2)$ since it is the sum of the sequence $n + n - 1 + ... + 1$ (it is the number of times the inner loop (lines 2 - 4) is executing, which we know to be $O(N^2)$, which is not $O(n + I(L))$.

```
1  for i = 1 to A.length - 1
2          for j = A.length downto i + 1
3                  if A[j] < A[j - 1]
4                          swap(A[j], A[j - 1])
```

   However, assuming an implementation that implements early termination such as

```
1  for i = 1 to A.length - 1
2          didSwap = False
3          for j = A.length downto i + 1
4                  if A[j] < A[j - 1]
5                          swap(A[j], A[j - 1])
6                          didSwap = True
7          Terminate if not didSwap
```

   the runtime is still not $O(n + I(L))$ time. This can be shown by way of contradiction with a simple counterexample. A case of this is the list L that is "almost sorted", where L = [2, 3, 4, 5, 1] and $I(L) = 4$. Tracing through the algorithm on this sample input, it is clear that in bubblesort, the largest element "bubbles "to the top, and specifically in this case, the 1 moves backward one place at a time toward its proper position as the first element. While it is true that the number of actual swaps is equivalent to $I(L)$, the contradiction is apparent in the number of times the input has to be scanned

(comparisons to check whether termination is possible if the work of sorting is complete). Since a swap is performed each time, the inner loop has to be executed $n-1$ times, and since the outer loop is executing n times, this particular case proves that bubblesort does not run in $O(n+I(L))$ time.

4. Show that insertion sort runs in $O(n+I(L))$ time. Recall that insertion sort adds the next element x at the back of the list, and swaps the element with the previous element until the previous element is smaller than x.

   The idea behind insertion sort is to constantly maintain a sorted array and to insert the next element into its appropriate place, hence the name. In analyzing the following implementation of insertion sort as per Cormen, Leiserson, Rivest, and Stein - Introduction to Algorithms (3rd edition), it is clear that the outer loop (lines 1 - 3, 7) will iterate through the entire input giving a bound of $O(n)$.

```
1  for j = 2 to A.length
2          key = A[j]
3          i = j - 1
4          while i > 0 and A[i] > key
5                  A[i + 1] = A[i]
6                  i = i - 1
7          A[i + 1] = key
```

   Next, it will be shown that the inner while loop (lines 4 - 6) will be executed $I(L)$ times. The idea behind this is extremely clear when presented in the following way: track the number of inversions I at each index i as the number of pairs of elements e_i, e_j, such that $i < j$ but $e_i > e_j$. (e_i is the i^{th} element of the input). It is clear from this that the aforementioned while loop (lines 4 - 6) will be executed I_i times for each element (this is the number of times it has to be swapped) and finally $I(L)$ is equivalent to the sum of all the individual I_i's. Thus, the runtime of only the inner while loop (lines 4 - 6) is $I(L)$ and the runtime of insertion sort is $O(n+I(L))$.