CS 430 - Recitation Notes

Len Washington III

September 25, 2023

Contents

After Lecture 01 & 02 – Answer any questions on HW1
Practice Problems (all taken from previous exams)

1. Which of the following is not true of improved bubble sort (keep track of last swap position on the inner loop and use that to reduce outer loop iterations) on the case on input elements sorted?

   a) It is stable

   b) **Consumes less memory**. Optimized Bubble sort is one of the simplest sorting techniques and perhaps the only advantage it has over other techniques is that it can detect whether the input is already sorted. It is faster than other in case of sorted array and consumes less time to describe whether the input array is sorted or not. It consumes same memory than other sorting techniques. Hence it is not an advantage.

   c) Detects whether the input is already sorted

   d) Consumes less time

2.

Statement 1: In insertion sort, after m passes through the array, the first m elements are in sorted order.

Statement 2: And these elements are the m smallest elements in the array.

   a) Both of the statements are true.

   b) **Statement 1 is true but statement 2 is false**. There may be a smaller value $> m$ indexes from the start of the array, which is why statement 2 is false, however, the first m elements would be sorted amongst themselves.

   c) Statement 1 is false but statement 2 is true

   d) Both of the statements are false

3. Consider the following program that attempts to locate an element x in a sorted array $a[]$ using binary search. The program is erroneous. Under what conditions does the program fail?

Algorithm 1 Erroneous Binary Search

```
1: function BS
2:     int i ← 1, j ← 100, k, x                ▷ assume x is assigned a value to search for
3:     int[] a ← new int[100];                  ▷ assume values loaded in sorted order
4:     repeat
5:         k ← (i+j)/2
6:         if a[k] < x then
7:             i ← k;
8:         else
9:             j ← k;
10:        end if
11:    until ((a[k] == x) || (i ≥ j))
12:    if a[k] == x then
13:        System.out.println("x is in the array")
14:    else
15:        System.out.println("x is not in the array")
16:    end if
17: end function
```

a) x is the last element of the array $a[]$

b) x is greater than all elements of the array $a[]$

c) **Both of the Above** The above program doesn't work for the cases where element to be searched is the last element of $a[]$ or greater than the last element (or maximum element) in $a[]$. For such cases, program goes in an infinite loop because i is assigned value as k in all iterations, and i never becomes equal to or greater than j. So the while condition never becomes false.

d) x is less than the last element of the array $a[]$

4. What's the worst case of insertion sort if the correct position for inserting element is calculated using binary search?

a) $O(\log n)$

b) $O(n)$

c) $O(n \log n)$

d) **$O(n^2)$** The use of binary search reduces the time of finding the correct position from $O(n)$ to $O(\log n)$. But the worst case of insertion sort remains $O(n^2)$ because of the series of swaps required for each insertion.

5. The following routine takes as input a list of n numbers, and returns the first value of i for which $L[i] < L[i-1]$, or n if no such number exists.

```
int firstDecrease(int* L, int n){
        for(int i=2; i <= n && L[i] >= L[i-1]; i++){}
        return i;
}
```

5a) What is the big-O runtime for the routine, measured as a function of its return value i? $O(n) \rightarrow O(i)$. $T(i) = ai + b = O(i)$ (a and b are constants).

5b) If the numbers are chosen independently at random, then the probability that firstDecrease(L) returns i is $\frac{i-1}{i!}$, except for the special case of $i = n+1$ for which the probability is $\frac{1}{n!}$ Use this fact to write an expression for the expected value returned by the algorithm. (Your answer can be expressed as a sum, it does not have to be solved in closed form. Do not use O-notation.) Use expectation

$$\begin{aligned}\sum_{i=1}^{n+1} \frac{i-1}{i!} &= \frac{1-1}{1!} + \frac{2-1}{2!} + \frac{3-1}{3!} + \cdots + \frac{n-1}{n!} + \frac{(n+1)-1}{(n+1)!} \\ &= \frac{0}{1!} + \frac{1}{2!} + \frac{2}{3!} + \cdots + \frac{n-1}{n!} + \frac{n}{(n+1)!} \\ &= 0 + \frac{1}{2} + \frac{2}{6} + \cdots + \frac{n-1}{n!} + \frac{n}{(n+1)!}\end{aligned}$$

$$\begin{aligned}E_{\text{value of } i \text{ returned}} &= \sum_{i=2}^{n+1} [\text{Probability(return } i) \times i] \\ &= \sum_{i=2}^{n} [\text{Probability(return } i) \times i] + (\text{Probability(return } n+1) \times (n+1)) \\ &= \sum_{i=2}^{n} \left[\frac{i-1}{i!} \times i\right] + \frac{1}{n!}(n+1) \\ &= \sum_{i=2}^{n} \left[\frac{i-1}{(i-1)!}\right] + \frac{n+1}{n!} \\ &= \sum_{i=2}^{n} \left[\frac{1}{(i-2)!}\right] + \frac{n+1}{n!}\end{aligned}$$

5c) What is the big-O average case running time of the routine? Hint: Simplify the previous summation until you see a common taylor series.TODO: Check these corrections

$$\begin{aligned}\sum_{i=2}^{n} \left[\frac{1}{(i-2)!}\right] + \frac{n+1}{n!} &= \frac{1}{(2-2)!} + \frac{1}{(3-2)!} + \frac{1}{(4-2)!} + \frac{1}{(5-2)!} + \cdots + \frac{1}{(n-2)!} + \frac{n+1}{n!} \\ &= \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{(n-2)!} + \frac{n+1}{n!} \\ &= \frac{1}{1} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{(n-2)!} + \frac{n+1}{n!} \\ &= 1 + \frac{1}{1} + \frac{1}{2} + \frac{1}{6} + \cdots + \frac{1}{(n-2)!} + \frac{n+1}{n!} \\ &= O(e)\end{aligned}$$

6. Some sorting algorithms are NOT stable. However if every key in $A[i]$ is changed to $A[i] * n + i - 1$ (assume $1 \leq i \leq n$) then all the new elements are distinct (and therefore stability is

no longer a concern). After sorting, what transformation will restore the keys back to their original values? What is the effect on the runtime of any of the sorting algorithm if you add this transformation before executing the sort and un-transformation after the sort?

$$A[i] \rightarrow A[i] * n + i - 1$$

To transform back to the original keys replace each $A[i]$ with $\text{int}[A[i]/n]$, you cannot use $A[i] = (A[i] - i + 1)/n$ because the index i of each value has changed when we sorted. Add $O(2n)$ runtime, which does not affect the runtime of any sort because the growth of those is greater than $O(n)$.

7. a) Use pseudocode to specify a brute-force algorithm that determines when a sequence of n positive integers is given as input, whether there are two distinct terms of the sequence that have as sum a third term. The algorithm should loop through all triples of the sequence, checking whether the sum of the first two terms equals the third.

Algorithm 2 Brute Force Sum

```
1: function BruteForceSum(A:int[], n:int)
2:     for i ← 0...n − 1 do
3:         for j ← i0...n − 1 do
4:             for k ← j0...n − 1 do
5:                 if ~~A[i] ≠ A[j]~~ i ≠ j && j ≠ k && i ≠ k && A[i] + A[j] == A[k] then
6:                     return true
7:                 end if
8:             end for
9:         end for
10:    end for
11:    return false
12: end function
```

b) Give a big-O estimate for the complexity of the brute-force algorithm from part (a). $O(n) \times O(n) \times O(n) = O(n^3)$

c) Devise a more efficient algorithm for solving the problem that first sorts the input sequence and then checks for each pair of terms whether their sum is in the sequence.

Algorithm 3 Smart Sum

```
1: function SmartSum(A:int[], n:int)
2:     ImprovedQuicksort(A)                          ▷ Runtime: O(n lg n)
3:     for i ← 0...n − 1 do
4:         for j ← i...n − 1 do
5:             for k ← j...n − 1 do
6:                 if i ≠ j && j ≠ k && i ≠ k && A[i] + A[j] == A[k] then
7:                     return true
8:                 end if
9:             end for
10:        end for
11:    end for
12:    return false
13: end function
```

d) Give a big-O estimate for the complexity of this algorithm. Is it more efficient than the brute-force algorithm?

After Lecture 03 & 04 – Answer any questions on HW1
Practice Problems (all taken from previous exams)

1. Which of the following is time complexity of fun()?

```
int fun(int n){
        int count = 0;
        for(int i = 0; i < n; i++){
                for(int j = i; j > 0; j--){
                        count = count + 1;
                }
        }
        return count;
}
```

a) $O(n)$
b) $O(n^2)$
c) $O(n \log n)$
d) $O(n \log(n) \log(n))$

2. Consider the following function. What is the returned value of the above function?

```
int unknown(int n){
        int i, j, k=0;
        for(i = n/2; i <= n; i++){
                for(j=2; j <= n; j=j*2){
                        k = k + n/2;
                }
```

```
            }
            return k;
    }
```

a) $\Theta(n^2)$

b) **$\Theta(n^2 \log n)$** The outer loop runs $n/2$ or $\Theta(n)$ times. The inner loop runs $O(\log n)$ times (Note that j is multiplied by 2 in every iteration). So the statement "$k = k + n/2$;" runs $\Theta(n \log n)$ times. The statement increases value of k by $n/2$. So the value of k becomes $n/2 \times \Theta(n \log n)$ which is $\Theta(n^2 \log n)$.

c) $\Theta(n^3)$

d) $\Theta(n^3 \log n)$

3. What is the worst-case auxiliary space complexity (including stack space for recursion) of merge sort?

a) $O(1)$

b) $O(\log n)$

c) **O(n)** The worst case is every item is split into its own array at the lowest level of divide.

d) $O(n \log n)$

4. Choose the incorrect statement about merge sort from the following:

a) It is a comparison-based sort.

b) It's runtime is dependent on input order.

c) It is not an in-place algorithm (all the operations are on the original array).

d) It is a stable algorithm.

5. Use the definition of big-O to prove or disprove.

5a) is 2^{n+1}? $=?O(2^n)$True:

$$2^{n+1} = C2^n$$
$$2 = C \text{ if c } \geq 2$$

5b) is 2^{2n}? $=?O(2^n)$False:

$$2^{2n} = C2^n$$
$$2^n = C$$

However 2^n has an infinite range so it cannot be upperbounded

6. Although merge sort runs in $\Theta(n \lg n)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort make it faster for small n. Thus, it makes sense to use insertion sort within merge sort when sub-problems become sufficiently small. Consider a modification to merge sort in which n/k sub-lists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

6a) Show the n/k sub-lists; each of length k, can be sorted by insertion sort in $\Theta(nk)$ worst-case time.

$$k \to \Theta\left(k^2\right)$$
$$\frac{n}{k} \to \Theta\left(\frac{n}{k} \times k^2\right) = \Theta(nk)$$

6b) Show that the sub-lists can be merged in $\Theta(n \lg\left(\frac{n}{k}\right))$ worst-case time.

$$\frac{n}{k} \to \Theta\left(\frac{n}{2k}\right) \text{ for merging}$$
$$\Theta(2k) \text{ m?? for } 2k \text{ elements}$$
$$\Theta\left(\frac{n}{2k} \times 2k\right) = \Theta\left(n\right) \Theta\left(\frac{n}{4k} \times 4k\right) = \Theta\left(n\right)$$

6c) Given that the modified algorithm runs in $\Theta(nk + n \lg\left(\frac{n}{k}\right))$ worst-case time, what is the largest asymptotic (Θ-notation) value of k as a function of n for which the modified algorithm has the same asymptotic running time as standard merge sort?

$$nk + n \lg\left(\frac{n}{k}\right) \leq n \lg n$$
$$k + \lg\left(\frac{n}{k}\right) \leq \lg n$$
$$k + \lg(n) - \lg(k) \leq \lg n$$
$$k - \lg(k) \leq 0$$
$$k \leq \lg(k)$$

6d) How should k be chosen in practice?

7. The Fibonacci sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... is defined recursively as

Algorithm 4 The Fibonacci sequence

```
1: function FIB(n)    ▷ This mathematical definition leads naturally to a recursive algorithm
2:     if n ≤ 1 then
3:         return n
4:     else
5:         return FIB(n-1) + FIB(n-2)
6:     end if
7: end function
```

7a) Write the recurrence relation, $T(n)$, for the asymptotic runtime for procedure FIB(n) shown above, and solve the recurrence relation to show that $T(n) = O\left(2^{n-2}\right)$.

7b) Another recursive procedure which computes the nth Fibonacci number is below.

Algorithm 5

```
1: function F1(n)
2:     if n < 2 then
3:         return n
4:     else
5:         return F2(2, n, 1, 1)
6:     end if
7: end function
8:
9: function F2(i, n, x, y)
10:    if i ≤ n then
11:        F2(i + 1, n, y, x + y)
12:    end if
13:    return x
14: end function
```

Trace out the algorithm as it computes F1(1), F1(2), F1(3), F1(4), explain how the algorithm works, and then compare its asymptotic runtime to the time for procedure FIB(n).

8. Use mathematical induction to show that when n is an exact power 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2 \\ 2T(n/2) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

is $T(n) = n \lg n$

After Lecture 05 & 06 – Answer any questions on HW1
Practice Problems (all taken from previous exams)

1. What are the max number of levels in the recursion tree for this recurrence relation?

$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + n$$

   a) $\log_4(n)$
   b) $\log_2(n)$
   c) $\log_{\frac{4}{3}}(n)$
   d) $\log_{\frac{1}{3}}(n)$

2. Under what case of Master's Theorem will the recurrence relation of binary search fail?

   a) 1
   b) 2
   c) 3
   d) It cannot be solved using Master's Theorem.

3. What is the purpose of using randomized quick sort over standard quick sort?

   a) Improve the worst-case runtime

   b) To eliminate the possibility that a particular input order will always yied worst-case runtime

   c) To improve accuracy of output

   d) To improve average case time complexity

4. The non-recursive work in quicksort is done in which step of the divide-conquer-combine algorithm?

   a) divide

   b) conquer

   c) combine

   d) none

5. Give big-O bounds of T(n) in each of the following recurrences. Use induction, iteration or Master Theorem.

   5a)

$$T(n) = T(n-1) + n;\ T(1) = O(1)$$

$$T(n) = n + T(n-1)$$
$$T(n) = n + n - 1 + T(n-2)$$
$$T(n) = n + n - 1 + n - 2 + T(n-3)$$
$$T(n) = n + n - 1 + n - 2 + \cdots + T(1)$$
$$T(n) = n + n - 1 + n - 2 + \cdots + O(1)$$
$$T(n) = \sum_{1}^{n} n$$
$$T(n) = O\left(\frac{n^2 + n}{2}\right)$$
$$T(n) = O(n^2)$$

   5b)

$$T(n) = 2T\left(\frac{n}{4}\right) + n^{\frac{1}{2}};\ T(1) = O(1)$$

   5c)

$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + n^2;\ T(1) = O(1)$$

6. Throughout this course, we assume that parameter passing during procedure calls takes constant time, even if an N-element array is being passed. This assumption is valid in most systems because a pointer to the array is passed, not the array itself. This problem examines the implications of three parameter-passing strategies:

1. An array is passed by pointer. Time = $\theta(1)$.
2. An array is passed by copying. Time = $\theta(N)$, where N is the size of the array.
3. An array is passed by copying only the subrange that might be accessed by the called procedure. Time = $\theta(q - p + 1)$ if the subarray $A[p \ldots q]$ is passed. Use $n = q - p + 1$, where n is the size of the subarray passed.

Consider the recursive binary search algorithm for finding a number in a sorted array. Give recurrences for the worst-case running times of binary search when arrays are passed using each of the three methods above, and give good upper bounds on the solutions of the recurrences. Let N be the size of the original problem and n be the size of a subproblem. Binary search works by comparing the element for which you are searching to the elemenet at index $\frac{p-r}{2}$ of a subarray of size n, where p is the first index of the subarray and r is the last index (integer division is used). Therefore, the array passed inot binary search is continually divided in hald.

1)

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

The array is passed by pointer, which is constant time. Therefore, the time involved in tha

2)

7. Use the definition of Θ and induction to prove that the recurrence $T(n) = T(N-1) + \theta(n)$ (worst case Quicksort) has the solution $T(n) = \Theta(n^2)$. Since we are not given any boundary conditions, we cann assume the basis step for the inductive proof. Assume the claim is true for $n = k$.
$T(k) = \theta(k^2)$, in other words assume $c_1 k^2 \leq T(k) \leq c_2 k^2$ for some $c_1 > 0$, $c_2 > 0$ and k large enough.
Use that to prove the claim

8. What is you are sorting a collection of data that can have multiple entries of some of the values. When calling Quicksort's PARTITION(A, p, r), where do elements equal to the pivot end up and why?
How could we modify Quicksort and Partition (write pseudocode) so that if we happen to partition on a pivot that had many duplicate values, we can improve the runtime of Quicksort by having smaller recursive calls?

Algorithm 6 Quicksort1

```
1: function QUICKSORT1(A, p, r)                          ▷
2:     if p < r then
3:         (q1, q2) = PARITION1(A, p, r)                 ▷ two return values
4:         (()A, p, q1 − 1)
5:     end if
6: end function
```

Algorithm 7 Partition1

```
1: function PARTITION1(A, p, r)                    ▷ Two return values
2:     endLow ← p − 1, endEqual ← p − 1
3:     pivot ← A[r]
4:     for j=p to r-1 do
5:         get function
6:     end for
7: end function
```

After Lecture 07 & 08 – Answer any questions on HW2 (due today)
Practice Problems (all taken from previous exams)

1. Which one of the following is false?

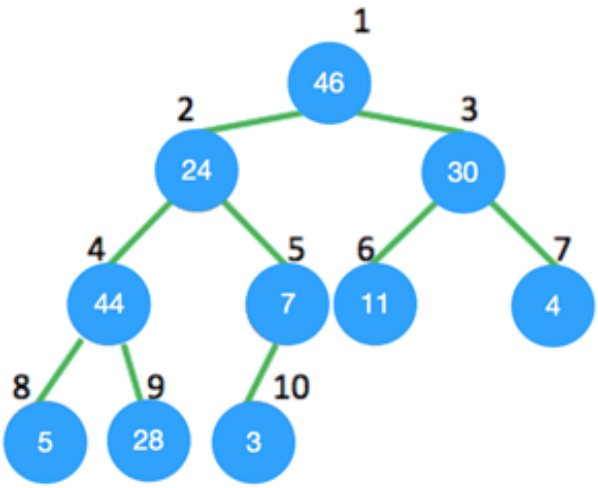
   a) Heap sort is an in-place algorithm.

   b) Heap sort has $O(n \log n)$ average case time complexity.

   c) Heap sort is a stable sort. Heap sort is an in-place algorithm as it needs $O(1)$ auxiliary space.

   d) Heap sort is a comparison-based sorting algorithm.

2. Consider the max heap shown below, the node with value 24 violates the max-heap property. Once heapify procedure is applied to it, which position will it be in?



   a) 5

   b) 8

   c) 9

   d) You cannot call heapify at the node with value 24

3. Counting sort can be used on any numeric data.

a) TRUE, it can be used, but it is a very bad idea. Counting sort requires a very large range of numbers, counting sort requires a very large array. This reduces its memory efficiency and increases space consumption. So while it possible to be used, it isn't a good idea to do so on any numeric data.

b) FALSE

4. Which of the following is not true about all comparison based sorting algorithms?

   a) The minimum possible runtime growth on a random input is $O(n \log n)$.

   b) Can be made stable by also using position when two elements are compared.

   c) Counting Sort is not a comparison-based sorting algorithm.

   d) Merge Sort is a comparison-based sorting algorithm.

5. The BUILD-MAX-HEAP discussed in class and shown to be $O(n)$ uses this process. Call Heapify from heap index position $\lfloor \frac{heapsize}{2} \rfloor$ down to heap index position 1. Building a heap can also be implemented by starting with an empty heap and repeatedly using MAX-HEAP-INSERT to insert the elements into the heap. Consider the following implementation:

```
1: function BUILD-MAX-HEAP1(A)
2:     H ← empty heap (of max size A.length)
3:     for i = 1 to A.length do
4:         H.MAX-HEAP-INSERT(A[i])
5:     end for
6: end function
```

   a) Do the procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP1 always create the same heap when run on the same input array? Prove that they do, or provide a counterexample. The procedures do not always create the same heap when run on the same input array.

   b) Show that in the worst case, BUILD-MAX-HEAP1 requires $\Theta(n \lg n)$ time to build an n-element heap. Since all but the last level is always filled, the height h of an n element heap is boundend because $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1 = \ldots$

6. The operation HEAP-DELETE(A, i) deletes the item in node i from heap A. Give an implementation of HEAP-DELETE that runs in $O(\lg n)$ time for an n-element max-heap.

Algorithm 8 HEAP-DELETE

```
1: function HEAPDELETE(A, i)
2:     swap A[i] with A[heapsize]  ▷ move the item to be deleted to the last position in the heap
3:     decrement heapsize
4:     key ← A[i]
5:     if key ≤ A[PARENT(i)] then
6:         HEAPIFY(A, i)
7:     end if
8:     ...
9: end function
```

7. Professor Fermat has the policy of giving A's to the top $n^{\frac{1}{2}} = \sqrt{n}$ students of his class, where n is the number of students. The algorithm that he uses to determine the top $n^{\frac{1}{2}}$ students first sorts the list of students by their numerical, real-valued grade, and then picks the top $n^{\frac{1}{2}}$ students from the sorted list. This algorithm has time-complexity $O(n \log n)$ because of the sort. Can you suggest a more efficient algorithm that has time-complexity $O(n)$? Describe your algorithm informally in English and justify its time-complexity. You can use a MaxHeap to effectively sort the students based off of the students grades which takes $O(n)$. Now perform RemoveMax $\sqrt{n}$ times. Each RemoveMax operation takes time $O(\log N)$. Total time complexity of this algorithm is $O(n) + \log(n)\sqrt{n} = O(n)$ because $\log(n)\sqrt{n} = O(n)$, because $\log(n) = O\left(\sqrt{n}\right)$