SORTED LIST MATCHING
&
EXPERIMENTAL RUN-TIME

COP 3502

Code Tracing Example

- Here is an example from a previous foundation exam:
  - **<u>Question:</u>** Find the value of x in terms of n after the following code segment below has executed.
    - You may assume that n is a positive even integer.

```
x = 0;
for (i = 1; i <= n*(8*n+8); i++) {
    for (j = n/2; j <=n; j++) {
        x = x + (n - j);
    }
}
```

Solved on the board

Sorted List Matching Problem – Approach #1

- Let's compare 3 different solutions to this problem and their runtimes.
  - **Problem:** Given 2 sorted lists of names, output the names common to both lists.
  - ***Obvious – Brute Force -*** way to do this:
    - For each name on list #1:
    1) Search for the current name in list #2.
    2) If the name is found, output it.

- This isn't leveraging the fact that we know the list is sorted,
  - it would take O(n) to do (1) and (2),
  - multiplied by the n names in list#1 gives a total of $O(n^2)$

Sorted List Matching Problem – Approach #2

- Let's use the fact that the lists are sorted!
  - For each name on list #1:
    1) Search for the current name in list #2.
    2) If the name is found, output it.
- For step (1) use a binary search.
  - We know that this takes
    - O(log n) time.
- Since we need to do this N times for each name in the first list,
  - Our total run time would be?
  - O(N log N)

Sorted List Matching Problem – Approach #3

- Can we do better?
  - We still haven't used the fact that list #1 is sorted!
  - Can we exploit this fact so that we don't have to do a full binary search for each name?

List #1	List #2
Albert	Cari
Brandon	Carolyn
Carolyn	Chris
Dan	Fred
Elton	Graham

Sorted List Matching Problem – Approach #3

- Formal Version of the algorithm:
  1) Start 2 “markers”, one for each list, at the beginning of both lists.
  2) Repeat the following until one marker has reached the end of its list:
     a) Compare the two names that the markers are pointing at.
     b) If they are equal, output the name and advance BOTH markers one spot.
     - If they are NOT equal, simply advance the marker pointing to the name that comes earlier alphabetically one spot.

Sorted List Matching Problem – Approach #3

- Algorithm Run-Time Analysis
  - For each loop iteration, we advance at least one marker.
  - The max number of iterations then , would be the total number of names on both list, N.
  - For each iteration, we are doing a constant amount of work.
    - Essentially a comparison, and/or outputting a name.
  - Thus, our algorithm runs in O(N) time – an improvement.

- Can we do better?
  - No, because we need to at least read each name in both lists, if we skip names, on BOTH lists we cannot deduce whether we could have matches or not.

Experimental Run-Time

- We can verify our algorithm analysis through running actual code
  - By comparing the experimental running time of a piece of code for different input sizes to the theoretical run-time.
- Assume T(N) is the experimental running time of a piece of code,
  - We'd like to see if T(N) is proportional to F(N) within a constant,
    - Where we've previously determined the algorithm to be O(F(N))

Experimental Run-Time

- One way to see if O(F(n)) is an accurate algorithmic analysis,
  - Is to compute T(N)/F(N) for a range of different values for N
    - Commonly spaced out by a factor of 2.
  - If the values for T(N)/F(N) stay relatively constant,
    - then our guess for the running time O(F(N)) was good.
  - Otherwise, if these T(N)/F(N) values converge,
    - our run-time is more accurately described by a function smaller than F(N).
  - And vice versa for if T(N)/F(N) diverges.

Experimental Run-Time – Example 1

- Consider the following table of data obtained from running an instance of an algorithm assumed to be cubic.
  - Decide if the Big-Oh estimate, $O(N^3)$ is accurate.

Run	N	T(N)
1	100	0.017058 ms
2	1000	17.058 ms
3	5000	2132.2464 ms
4	10000	17057.971 ms
5	50000	2132246.375 ms

The calculated values converge to a positive constant (1.0757×10^{-8})
– so the estimate of $O(n^3)$ is a good estimate.

- T(N)/F(N) = 0.017058/(100*100*100) = 1.0758×10^{-8}
- T(N)/F(N) = 17.058/(1000*1000*1000) = 1.0758×10^{-8}
- T(N)/F(N) = 2132.2464/(5000*5000*5000) = 1.0757×10^{-8}
- T(N)/F(N) = 17057.971/(10000*10000*10000) = 1.0757×10^{-8}
- T(N)/F(N) = 2132246.375/(50000*50000*50000) = 1.0757×10^{-8}

Experimental Run-Time – Example 2

- Consider the following table of data obtained from running an instance of an algorithm assumed to be quadratic.
  - Decide if the Big-Oh estimate, $O(N^2)$ is accurate.

Run	N	T(N)
1	100	0.00012 ms
2	1000	0.03389 ms
3	10000	10.6478 ms
4	100000	2970.0177 ms
5	1000000	938521.971 ms

The values diverge, so $O(n^2)$ is an ***underestimate.***

- T(N)/F(N) = 0.00012/(100 * 100) = 1.6×10^{-8}
- T(N)/F(N) = 0.03389/(1000 * 1000) = 3.389×10^{-8}
- T(N)/F(N) = 10.6478/(10000 * 10000) = 1.064×10^{-7}
- T(N)/F(N) = 2970.0177/(100000 * 100000) = 2.970×10^{-7}
- T(N)/F(N) = 938521.971/(1000000 * 1000000) =9.385×10^{-7}

Array Sum Algorithm

- Let's say we have 2 sorted lists of integers,
  - And we want to know if we can find a number in the 1^{st} array when summed with a number in the 2^{nd} array gives us our target value.
  - This is similar to the sorted list matching algorithm we talked about earlier, there are 3 solutions:

1) Brute force look at each value in each array and see if the target sum is found
   - **– $O(n^2)$**
2) Look at each value in the 1^{st} array (number1) and binary search for target –number1 in the 2^{nd} array.
   - **O(n logn)**
3) A smarter algorithm – O(n), where we only need to look at each value in each array once.
   - **O(n)**

Linear Array Sum Algorithm

- Linear Algorithm:
  - Target = 82
    - We start 2 markers, 1 at the bottom of Array1, the other at the top of Array2
    - Then if the sum of the values < Target, move marker 1 up, otherwise more marker 2 down, until we find the target sum.

Array 1:

1	3	5	6	7	9	13	45	56	99

Sum = Target, Done!

Array 2:

5	8	14	28	69	75	88	92	93	94

Determine if the Experimental Run-Time matches the Theoretical

Run	N	T(N)
1	100,000	37 s
2	200,000	149 s
3	400,000	593 s

- Brute Force ArraySum Alg.
  - $O(n^2)$

Run	N	T(N)
1	100,000	0.01 s
2	200,000	0.023 s
3	400,000	0.048 s

- Binary Search ArraySum Alg.
  - $O(n \log n)$

Run	N	T(N)
1	100,000	0.001 s
2	200,000	0.001 s
3	400,000	0.002 s

- Linear ArraySum Alg.
  - $O(n)$

Determine if the Experimental Run-Time matches the Theoretical

Run	N	T(N)	$F(N) = N^2$	T(N)/F(N)
1	100,000	37 s	$100{,}00^2$	3.7×10^{-7}
2	200,000	149 s	$200{,}000^2$	3.7×10^{-7}
3	400,000	593 s	$400{,}000^2$	3.7×10^{-7}

Since T(N)/F(N) converges to a value,
We know O(F(N)) was an accurate analysis.

Run	N	T(N)	F(N) = N logN	T(N)/F(N)
1	100,000	0.01 s		
2	200,000	0.023 s		
3	400,000	0.048 s		

I'll leave it as an exercise to determine if the other timing results verify the theoretical analysis.

Run	N	T(N)	F(N) = N	T(N)/F(N)
1	100,000	0.001 s		
2	200,000	0.001 s		
3	400,000	0.002 s		

Experimental Run-Time Practice Problem

- Given the following table, you have to determine what O(F(N)) would be, you are also given that it is either log n, n, or n^2.

Run	N	T(N)
1	100	0.11 ms
2	200	0.43 ms
3	400	1.72 ms
4	800	6.88 ms
5	1600	27.54 ms