Notes for ECE 36800 - Data Structures and Algorithms

Shubham Saluja Kumar Agarwal

January 18, 2024

These are lecture notes for spring 2024 ECE 36800 at Purdue. Modify, use, and distribute as you please.

Contents

Course Introduction

Provides insight into the use of data structures. Topics include stacks, queues and lists, trees, graphs, sorting, searching, and hashing. The learning outcomes are:

- Advanced programming ideas, in practice and in theory
- Data structures and their abstractions: Stacks, lists, trees, and graphs
- Fundamentals of algorithms and their complexities: Sorting, searching, hashing, and graph algorithms
- Problem Solving

Introduction to Data Structures & Algorithms

Data Structures are methods of organizing information for ease of manipulation. Examples:

1. Dictionary
2. Check-out line or queues
3. Spring-loaded plate dispenser or stacked
4. Organizational Chart or tree

These are associated with methods known as algorithms to be manipulated
Algorithms are methods of doing something. Examples:

1. Multiplying two numbers
2. Making a sandwich
3. Getting dressed

The topics of interest within them are:

- Correctness
- Efficiency in time and space

Asymptotic Notation

The questions to be asked about an algorithm are the following:

- Is it correct?
- Is it as fast as possible?
- How many machine instructions (in terms of n) does it take?

Let us take the following algorithm to add the numbers form 1 to n:

```
total = 0;
for (i=1:n)
    total = total + i;
return total
```

The cost will be:

Cost	Frequency	Function
C_1	1	Assign initial value
C_2	n+1	For loop iterations and exit
C_3	n	Number additions
C_4	1	Return value

The total is then:

$$C_1 * 1 + C_2(n+1) + C_3(n) + C_4(1) = (C_2 + C_3)n + (C_1 + C_2) + C_4$$

However the $O(n)$ will only be n, as the constants and coefficients of these will be deprecated, as we will come to understand in more detail as this topic continues.
Let us take another example of some code that has a

$$\begin{aligned} T(n) &= n^2 + 10^7 n + 10^{10} \\ T(10^{11}) &= 10^{22} + 10^{18} + 10^{10} \\ T(2*10^{11}) &= 4*10^{22} + 2*10^{18} + 10^{10} \\ \implies \frac{T(2*10^{11})}{T(10^{11})} &\approx 4 = \left(\frac{2*10^{11}}{10^{11}}\right)^2 \end{aligned}$$

This goes to show that this algorithm has an $O(n) = n^2$, and all coefficients and lower order terms that are a part of the complexity are largely irrelevant for large n values. This is why this is called **asymptotic notation**.
Another example of a simple algorithm is

```
total = 0;
for (i=1:n):
    if (((i*i%3)==0)||((i*i%7)==0)):
        total = total+i*i;
return total;
```

Which has a cost table that looks like the following:

Cost	Frequency	Function
C_1	1	Assign initial value
C_2	n+1	For loop iterations and exit
C_3	n	Number of $i\%3$ comparisons
C_4	$n - \lfloor \frac{n}{3} \rfloor$	Number of $i\%7$ comparisons
C_5	$\lfloor \frac{n}{3} \rfloor + \lfloor \frac{n}{7} \rfloor - \lfloor \frac{n}{21} \rfloor$	Number of additions
C_6	1	Returning value

It can be noted that $O(n) = n$ for this function, despite all the other complexities in the algorithm. However, it is important to know how to calculate $T(n)$ as well.
Now, let us look at something more complicated, matrix multiplication of two lower triangular matrices.

```
for (i=1:n):
    for (j=1:i):
        C_ij = 0;
        for (k=j:i):
            C_ij = C_ij+A_ik*B_kj
return C
```

This has a cost table that looks like the following: Finally, we can

Cost	Frequency	Function
C_1	n+1	First loop
C_2	$\sum_{i=1}^{n}(i+1)$	Second loop
C_3	$\sum_{i=1}^{n}\sum_{j=1}^{i}1$	Number of assigns
C_4	$\sum_{i=1}^{n}\sum_{j=1}^{i}(i-j+2)$	Third loop
C_5	$\sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{k=j}^{i}1$	Number of assigns to matrix
C_6	1	Returning value

analyze an example that has logarithmic complexities.

```
i=2;
k=0;
while (i<n){
    i=i*i;
    k=k+1;
}
return i;
```

Which has a cost table that looks like the following:

Cost	Frequency	Function
C_1	1	Assign i
C_2	1	Assign k
C_3	$\lceil\log_2(\log_2(n))\rceil+1$	Number of while loop iterations
C_4	$\lceil\log_2(\log_2(n))\rceil$	number of i assigns
C_5	$\lceil\log_2(\log_2(n))\rceil$	Number of k assigns
C_6	1	Returning value

It can be noted that if line three was instead changed to

```
while (i<=n){
```

The table will instead be:

Cost	Frequency	Function
C_1	1	Assign i
C_2	1	Assign k
C_3	$\lceil \log_2(\log_2(n+1)) \rceil + 1$	Number of while loop iterations
C_4	$\lceil \log_2(\log_2(n+1)) \rceil$	number of i assigns
C_5	$\lceil \log_2(\log_2(n+1)) \rceil$	Number of k assigns
C_6	1	Returning value

As the loop break condition changed from $i \geq n$ to $i \geq n+1$ by simply changing.

Insertion and Shell Sort

Sorting is necessary to process items in sorted order. It speeds up the location of items, finding identical items, etc.
It is good to know that in real life, what is sorted is in fact the pointers of these structs, as the movement of structs have higher memory requirements.

Insertion Sort

Inserts an item into a sorted array. Compares the item with items in the sorted array, and if they are in the incorrect order, they are swapped. This is continued until everything has been succesfully sorted.
The code to sort n integers in an array r looks like this:

```
for (j=1:n-1){
    for (i=j:1){
        if (r[i-1]>r[i]){
            swap(r[i-1], r[i]);
        }
        else{
            break;
        }
    }
}
```

This is suboptimally inefficient due to the restriction of only swapping with neighbors, directly. However, it can be made even more efficient using the following algorithm:

```
for (j=1:n-1){
    temp = r[j];
    for (i=j:1){
        if (r[i-1]>temp_r){
            r[i] = r[i-1];
        }
        else{
            break;
        }
    }
    r[i] = temp_r;
}
```

This allows us to "move" items down without constant comparisons, saving us some assignments.
This can also be implemented using while loops, and thus avoiding break:

```
    for (j=1:n-1){
        temp=r[j];
        i=j;
        while (i>0 and r[i-1]>temp){
            r[i] = r[i-1];
            i-=1;
        }
        r[i]=temp_r;
    }
```

This has the following cost table in the best case: Which hs a $O(n) =$

Cost	Frequency	Function
C_1	n	For loop iterations
C_2	n-1	Assign temp
C_3	n-1	Assign i
C_4	n-1	It is checked once per iteration
C_5	0	Number of r[i] exchanges
C_6	0	Number of i decreases
C_7	n-1	Assign r[i]

n And the following in the worst case: Now, we will learn how to

Cost	Frequency	Function
C_1	n	For loop iterations
C_2	n-1	Assign temp
C_3	n-1	Assign i
C_4	$\frac{(n+2)(n-1)}{2}$	Number of time the while loop is checked
C_5	$\frac{(n)(n-1)}{2}$	Number of r[i] exchanges
C_6	$\frac{(n)(n-1)}{2}$	Number of i decreases
C_7	n-1	Assign r[i]

calculate the average performance of an algorithm like insertion sort.

Let us take a random j^{th} item. The probability of it not needing to be moved is $\frac{1}{j+1}$. And it will need a certain some number between 0 and j exchanges to get to its rightful position if not. This leads the expected total number of exchanges to be $\sum_{i=0}^{j} \frac{i}{j+1} = \frac{j}{2}$. Once we reach the $(n-1)^{th}$ element, this is $\frac{1}{2}\frac{n(n-1)}{2} \approx \frac{n^2}{4}$.

Average performance is seldom calculated for the intents and purposes of this course.
There are still some inefficiencies in insertion sort that can be imporved by using sentinels.

```
for (j=n-1:1){
    if (r[j]<r[j-1]){
        swap(r[j], r[j-1]);
    }
}
for (j=2:n-1){
    temp=r[j];
    i=j;
    while (r[i-1]>temp){
        r[i] = r[i-1];
        i-=1;
    }
    r[i]=temp_r;
}
```

By moving the smallest item to the beginning, we can avoid the (i>0) condition, slightly increasing efficiency.

Shell Sort

This improves insertion sort by allowing for swaps along larger distances between elements.
If we did 7-sorting and 3-sorting:
We would start with 7 subarrays with at most $\lceil \frac{n}{7} \rceil$ elements. These subarrays would need to be sorted within themselves.
We would then go over to having 3 subarrays with at most $\lceil \frac{n}{3} \rceil$ elements. These subarrays would once again need to be sorted within themselves.
Finally, we would conduct regular insertion sort. The complexity of shell sort changes based on the selected sequence.

- $1, 3, 7, 15, \ldots, 2^k - 1, \ldots$ has a complexity of $O(n^{1.5})$
- $1, 4, 13, \ldots, 3h(k-1), \ldots$ also thas a complexity of $O(n^{1.5})$
- $2^p 3^q$ has a complexity $O(n(\log(n))^2)$

The algorithm of shell sort is the following:

```
for (j=k:n-1){
    temp=r[j];
    i=j;
    while (i>=k and r[i-k]>temp){
        r[i]=r[i-k];
        i=i-k;
    }
    r[i]=temp;
}
```

To prove the complexity of the 2^p3^q complexity, we can visualize the following triangle:

```
1
2 3
4 6  9
8 12 18 27
```

which holds the values of k from the above algorithm the height and base of the triangle both have complexities of $\log(n)$, whilie the sorting of the subset make by each k has a complexity of n, which results in a total complexity of $n(\log(n))^2$.