Chapter 2

Algorithm Analysis

2.1 Analysis of Algorithm

2.1.1 Definition

The analysis of algorithms is the determination of the amount of ***resources*** (such as ***time*** and ***storage***) necessary to execute them. Most algorithms are designed to work with inputs of arbitrary length. Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity).

We are often concerned about ***computational time*** as resource requirement but sometime we are also concerned about ***memory***, ***communication bandwidth***, and other costs are considered. But for this course we will always assume the resource we are most concerned is the computational time, hence, all our analysis is going to be in relation to that. When we analyze multiple viable candidates, we will be able to identify which are the most efficient once and which are not.

2.1.2 Model of Computation

Time efficiency estimates depend on what we define to be a step. For the analysis to correspond usefully to the actual execution time, the time required to perform a step must be guaranteed to be bound above by a constant. One must be careful here; for instance, some analyses count an addition of two numbers as one step. This assumption may not be warranted in certain contexts. For example, if the numbers involved in a computation may be arbitrarily large, the time required by a single addition can no longer be assumed to be constant.

Two cost models are generally used:

- The ***uniform cost model***, also called uniform-cost measurement, assigns a constant cost to every machine operation, regardless of the size of the numbers involved

- The ***logarithmic cost model***, also called logarithmic-cost measurement, assigns a cost to every machine operation proportional to the number of bits involved

The later is more cumbersome to use, so it's only employed when necessary, for example in the analysis of arbitrary-precision arithmetic algorithms, like those used in cryptography. Models should also set standards on other parameters such as number of processors and memory access pattern and time. Before we can analyze an algorithm we must have a model of the implementation technology that we will use.

Random Access Machine Model

For this course we use we assume a generic once processor, **random access machine (RAM)** model of computation. In the RAM model instructions are executed once after another with no concurrent operations.

Strictly speaking we should precisely define the instructions of the RAM model and their cost, but doing so would be tedious and offer little insight into algorithm design and analysis. Hence, we make assumptions about the instructions and leave out listing the available instructions and their cost explicitly. We should not abuse this model, though, by assuming instruction that would not realistically be available in real computers such as an instruction to sort. Hence, RAM model has a handful of instructions on arithmetic (add, subtract, multiply, divide, remainder, floor, and ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return).

The data types in RAM model are ***integer*** and ***floating point***. Some real computers sometimes have more instructions than those defined here and we consider these to be gray area for the analysis. For example x^y is a multiple instruction operation but when x = 2 it can be done with a single instruction which is shifting. Furthermore, in real computer there exists a multi-level memory hierarchy (cache, primary memory and virtual memory), which we do not consider in the RAM model.

External Memory Model

Most of the time when we analyze algorithms we assume Random Access Machine (RAM) as our model of computations. You are perhaps familiar with the complexities of many algorithms in internal memory. For example, it is well-known that N numbers can be sorted with $O(N \log N)$ time in the RAM model. What this statement says exactly is that, there is an algorithm able to solve the sorting problem by performing $O(N \log N)$ basic operations. In particular, each basic operation either performs some "standard" CPU work (e.g., +,,,/, comparison, taking the AND/OR/XOR of two words,etc.) or accesses a memory location.

Many applications in practice need to deal with data sets that are too large to fit in memory. While it is true that the memory capacity of a computer has been increasing rapidly, data set sizes have exploded at an even greater pace, such that it is increasingly unrealistic to hope that someday we could run all the applications entirely in memory. In reality, data still need to be stored in an external device, typically, a hard disk. An algorithm in such environments would need to perform many disk I/Os to move data between the memory and the disk. Since an I/O is rather expensive (at the order of 1-10 milliseconds), the overall execution cost may be far dominated by the I/O overhead.

This phenomenon has triggered extensive research in the past three decades on algorithms in the external memory (EM) model, which was proposed in 1988, and has been very successful in capturing the characteristics of I/O-bound algorithms. A computer of this model is equipped with a memory of M words, and a disk of an unbounded size. The disk has been formatted into disjoint blocks , each of which has the length of B words. An I/O either brings a block of data from the disk to the memory, or conversely writes B words in the memory to a disk block. The space complexity of a data structure or an algorithm is measured as the number of disk blocks occupied, while the time complexity is measured as the number of I/Os performed. CPU calculation can be done only on the data that currently reside in the memory, but any such calculation is charged with no cost. Accessing any data in the memory is also for free. The value of M is assumed to be at least 2 B , i.e., the memory can be as small as just 2 blocks. However, it is often acceptable to assume M B 2, which is known as the tall cache assumption. By fitting in some typical values of B in practice, you can convince yourself that a memory with B 2 words is available in almost any reasonable computer nowadays. For a data set of N elements 1, the minimum number of blocks required to store all the elements is (N/B). Therefore, linear cost should be understood as O(N/B), as opposed to O(N). 2

Cache Oblivious Model

The cache-oblivious model is an abstract machine (i.e. a theoretical model of computation). It is similar to the RAM machine model which replaces the Turing machine's infinite tape with an infinite array. Each location within the array can be accessed in time, similar to the Random access memory on a real computer. Unlike the RAM machine model, it also introduces a cache: a second level of storage between the RAM and the CPU. The other differences between the two models are listed below. In the cache-oblivious model:

- Memory is broken into lines of L words each
- A load or a store between main memory and a CPU register may now be serviced from the cache. If a load or a store cannot be serviced from the cache, it is called a cache miss.
- A cache miss results in one line being loaded from main memory into the cache. Namely, if the CPU tries to access word $\boldsymbol{w}$ and $\boldsymbol{b}$ is the line containing $\boldsymbol{w}$, then $\boldsymbol{b}$is loaded into the cache. If the cache was previously full, then a line will be evicted as well (see replacement policy below).
- The cache holds Z words, where $Z = \omega L^2$. This is also known as the tall cache assumption.
- The cache is fully associative: each line can be loaded into any location in the cache.
- The replacement policy is optimal. In other words, the cache is assumed to be given the entire sequence of memory accesses during algorithm execution. If it needs to evict a line at time, it will look into its sequence of future requests and evict the line that is accessed furthest in the future. This can be emulated in practice with the Least Recently Used policy, which is shown to be within a small constant factor of the offline optimal replacement strategy.

To measure the complexity of an algorithm that executes within the cache-oblivious model, we can measure the familiar (running time) work complexity $W(n)$. However, we can also measure the cache complexity, $Q(n, L, Z)$, the number of cache misses that the algorithm will experience.

The goal for creating a good cache-oblivious algorithm is to match the work complexity of some optimal RAM model algorithm while minimizing $Q(n, L, Z)$. Furthermore, unlike the external-memory model, which shares

many of the listed features, we would like our algorithm to be independent of cache parameters (L *and* Z)). The benefit of such an algorithm is that what is efficient on a cache-oblivious machine is likely to be efficient across many real machines without fine tuning for particular real machine parameters. Researchers showed that for many problems, an optimal cache-oblivious algorithm will also be optimal for a machine with more than two memory hierarchy levels.

2.1.3 Efficiency

Different algorithms can be devised to solve a single problem. Each of these algorithms will differ dramatically in their efficiency. Efficiency is the comparison of what is actually produced or performed with what can be achieved with the same consumption of resources (money, time, labor, etc.). This difference in efficiency is even much more significant than the hardware and software the solutions are implemented on.

For example, take two sorting algorithms insertion sort and merge sort. Insertion sort takes C_1N^3 time for an input size of N while merge sort takes $C_2N\log N$ time where C_1 *and* c_2 are constants independent of each other and N. It can be seen from *Figure* 2.1.3 that after a certain input size N $N \simeq 60\text{n}$ (n $= 60$) merge sort clearly outperforms (i.e., is much more efficient/runs faster than) insertions sort with $C_1 = 10 = 10$ and $C_2 = 100$. The detailed analysis is presented in *Table* 2.1.

2.1.4 Another Reason to Analyze Algorithms

Even if speed and cost were not an issue we still have to study algorithms to show they actually terminate and do so with the right answer (i.e., proof their correctness). In reality, though, both speed and cost associated with algorithms is an issue and we want to study it.

2.1.5 Example: Analysis of Insertion Sort

To recap, analyzing algorithm refers to figuring out the amount of resource required to execute it. Analysis of algorithm in relation to its time complexity, hence, refers to the time it takes to execute the algorithm. Further since we assume all elementary operations (i.e., those operations defined by the computational model) as to take a a single unit of time, counting the number of these operations performed by the algorithm will give as the total time required to execute the algorithm.

N	Insertion Sort	Merge Sort	Efficient Algorithm
2	40	400	Insertion
10	1000	6643.85619	Insertion
15	2250	11720.6718	Insertion
20	4000	17287.7124	Insertion
25	6250	23219.2809	Insertion
30	9000	29441.3436	Insertion
35	12250	35904.9811	Insertion
40	16000	42575.4248	Insertion
45	20250	49426.6779	Insertion
50	25000	56438.5619	Insertion
55	30250	63594.9568	Insertion
60	36000	70882.6871	Insertion
65	42250	78290.7816	Insertion
70	49000	85809.9622	Insertion
75	56250	93432.2804	Insertion
80	64000	101150.85	Insertion
85	72250	108959.646	Insertion
90	81000	116853.356	Insertion
95	90250	124827.257	Insertion
100	100000	132877.124	Insertion
110	121000	149189.914	Insertion
120	144000	165765.374	Insertion
130	169000	182581.563	Insertion
140	196000	199619.924	Insertion
150	225000	216864.561	Merge
160	256000	234301.699	Merge
170	289000	251919.292	Merge
180	324000	269706.711	Merge
190	361000	287654.513	Merge

Table 2.1: Running time for insertion sort and merge sort with $c_1 = 10$ *and* $c_2 = 200$

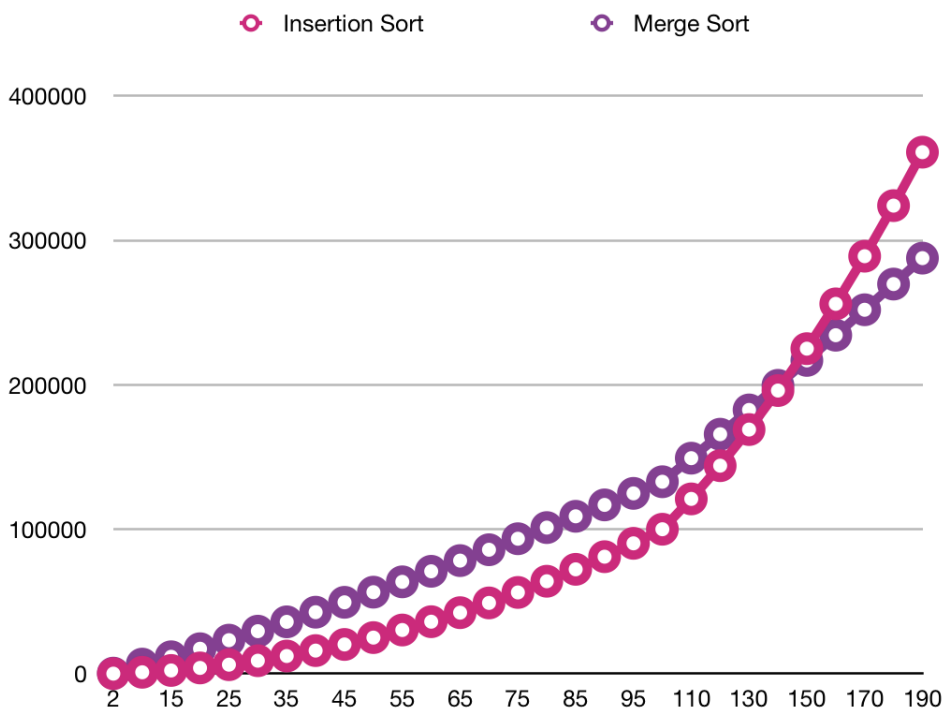

Figure 2.1: Asymptotic Comparison of running times of insertion sort and merge sort

```
procedure insertionSort( A : list of sortable items )
 for j = 2 to length [A]
   do key = A[j]
      // Insert A[j] into the sorted sequence A[1... j-1]
      i= j      1
      while i>0 and A[i]>key
         do A[I + 1]   A [i]
         j = i   1
      end while

      A[i+1] = key
   end for
end procedure
```

To illustrate this process in action, in this section, we will analyze the time complexity of insertion sort with an implementation presented above. The time taken by the insertion sort procedure depends on

- **Input size**: sorting a thousand numbers takes longer than ten
- **Input sort status**: sorting an input already nearly sorted would be faster than a reversely sorted input.

In general the running time of a program depends on the ***input size***. Hence we need to describe running time as a function of input size.The notion of input size depends on the problem being studied. For many problems it is the number of items in the input. For others such as multiplying integers the best measure is the total number of bits and in other cases such as graph problems the input will described by more than one number (i.e., number of vertices and edges). We shall indicate which input size measure is being used with each problem we study.

The running time of an algorithm on a particular input is the number of primitive operations or steps executed. It is convenient to define the notion of steps to make it machine-independent. It take a constant time to execute each line of our pseudocode but one line may take a different amount of time that the other. Let say line i of the pseudocode takes ci to execute and the inner while loop tests t_j times for the j^{th} item.

	INSERTIO-SORT (A)	Cost	Times
1	**for** j ← 2 **to** length [A]	C_1	N
2	**do** key ← A[j]	C_2	N-1
3	// Insert A[j] into the sorted sequence A[1... j-1]	0	N-1
4	i ← j – 1	C_4	N-1
5	**while** i > 0 **and** A[i] > key	C_5	$\sum_{j=2}^{n} t_j$
6	**do** A[I + 1] ← A[i]	C_6	$\sum_{j=2}^{n} t_j - 1$
7	j ← I – 1	C_7	$\sum_{j=2}^{n} t_j - 1$ - 1
8	A[i+1] ← key	C_8	N-1

Figure 2.2: Running Time Analysis of Insertion Sort

Based on Figure 2.1.5 the total running time of the algorithm is the sum of running times of each statement executed

$$T(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5 \sum_{j=2}^{n} t_j + C_6 \sum_{j=2}^{n} (t_j - 1) + C_7 \sum_{j=2}^{n} (t_j - 1) + C_8(n-1) \tag{2.1}$$

As stated in previously the running time of a program depends on the input size and in this case the level of sorting it already is in. For insertion

sort the best case is when the input is already sorted in which case for each $j = 2, 3...n$ $A[i] < key$ in the first iteration of the while loop ($i = j - 1$) hence $t_j = 1$ leading to the following equation.

$$T(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5(n-1) + C_8(n-1) \quad (2.2)$$

$$T(n) = (C_1 + C_2 + C_4 + C_5 + C_8)n - (C_2 + C_4 + C_5 + C_8) \quad (2.3)$$

If the array was in reverse order the algorithm will face its worst case results since we must compare each $A[j]$ with all the elements in the sorted portion of the array

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \quad (2.4)$$

$$\sum_{j=2}^{n} j - 1 = \frac{n(n-1)}{2} \quad (2.5)$$

$$T(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5(\frac{n(n-1)}{2} - 1) + C_6(\frac{n(n-1)}{2}) + C_7(\frac{n(n-1)}{2}) + C_8(n-1) \quad (2.6)$$

$$T(n) = (\frac{C_5}{2} + \frac{C_6}{2} + \frac{C_7}{2})n^2 + (C_1 + C_2 + C_4 + \frac{C_5}{2} - \frac{C_6}{2} + \frac{C_7}{2} + C_8)n - (C_2 + C_4 + C_5 + C_8) \quad (2.7)$$

So we can express the worst case running time as $an^2 + bn + c$ if we replace the coefficients in the above equation with constants a, b and c. We used some simplifying abstractions to ease our analysis by ignoring actual statement cost and even the abstracts costs C_i. We normally even further simplify this by only considering ***rate of growth*** or ***order of growth*** instead of the actual running time. We shall do this because what we really want to know is whom our algorithm performance compared to others for large input sizes.

We therefore only consider the leading term an^2 since the lower order terms (i.e., bn and c) will have relatively insignificant for the large values of n compared to the first once. We even ignore the coefficient a due to similar reasons as to why we ignored the other terms to be left with n^2 which we consider as our order of growth. We call this the theta notation and it is presented as $\Theta(n^2)$

2.2 Correctness of An Algorithm

2.2.1 Introduction

After designing algorithms in addition to analyzing the resource requirement one should proof its correctness. Proving the correctness means that showing that the algorithm halts and halts with correct answer for all valid inputs. In this course the primary technique we will use for proving correctness of an algorithm is ***mathematical induction***.

Mathematical Induction is a mathematical proof technique used to prove a given statement about any well-ordered set. Most commonly, it is used to establish statements for the set of all natural numbers. Mathematical induction is a form of direct proof, usually done in two steps. When trying to prove a given statement for a set of natural numbers, the first step, known as the **base case**, is to prove the given statement for the first natural number. The second step, known as the *inductive step*, is to prove that, if the statement is assumed to be true for any one natural number, then it must be true for the next natural number as well. Having proved these two steps, the rule of inference establishes the statement to be true for all natural numbers. In common terminology, using the stated approach is refers to as using the *Principle of Mathematical Induction*.

in addition to these two steps in the conventional mathematical induction, while proving correctness of algorithm we add a third step to check the termination of the algorithm.

2.2.2 Loop Invariant

A loop invariant is a property of a program loop that is true before (and after) each iteration. It is a logical assertion, sometimes checked within the ode by an assertion call. Knowing its invariant(s) is essential in understanding the effect of a loop. What is more, it is what we will use to prove in the base case and the inductive step.

In the pseudocode of insertion sort presented earlier, j indicates the current card being inserted. At the beginning of the outer loop (i.e., for loop), which is indexed by j, the sub array containing elements $A[1...j-1]$ consists of the currently sorted hand and elements $A[j+1...n]$ correspond to the pile of cards still on the table. In fact elements $A[1...j-1]$ are the elements originally positions 1 through $j-1$ but now in a sorted order.

We state these properties formally as ***loop invariants***

> At the start of each iteration of the for loop of lines 2-8, the subarray $A[1...j-1]$ consists of the elements originally in $A[1..j-$

1] ***but in a sorted order***

We use these loop invariants to help us understand why an algorithm is ***correct***.

2.2.3 Correctness Proof

We must show three things about a loop invariant to proof an algorithms correctness

- **Initialization**: It is true prior to the first iteration of the loop
- **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration
- **Termination**: when the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct

Notice the similarity of this approach to mathematical induction in which you proof the ***base case*** and then the ***inductive step***. Here showing the loop invariant holds for the first iteration is the base case and showing that the invariant holds from iteration to iteration is like the inductive case.

One important difference from mathematical induction is the third step in which we show the correctness for the termination case. In mathematical induction we show that the inductive step holds infinitely, here we stop the induction when the loop terminates.

2.2.4 Example: Insertion Sort

```
procedure insertionSort( A : list of sortable items )
 for j = 2 to length [A]
   do key = A[j]
      // Insert A[j] into the sorted sequence A[1... j-1]
      i= j      1
      while i>0 and A[i]>key
         do A[I + 1]   A [i]
         j = i   1
      end while

      A[i+1] = key
   end for
end procedure
```

Using these properties we will show the correctness of insertion sort algorithm.

- **Loop Invariant**: At the j^{th} iteration the elements $[1...j-1]$ are sorted relative to themselves.

- **Initialization**: We start by showing that the loop invariant holds before the first loop iteration when $j = 2$. The subarray $[1...j-1]$, therefore, consists of just the single element $A[1]$, which in fact it the original element $A[1]$. This array is sorted (trivial)

- **Maintenance**: In the algorithm the for loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$ and so on by one position to the right until the right position for $A[j]$ is found at which point the value of $A[j]$ is inserted. More formally, though, it is required to show the loop invariant of the inner loop as well. Here, however, we will not go into such detail for the time being

- **Termination**: The outer loop end when j exceeds n, i.e., when $j = n+1$ substituting $n+1$ for j in the wording of loop invariant we have the $subarray A[1...n]$ consists of the elements originally in $A[1..n]$ but in a sorted order. But the subarray $A[1...n]$ is the entire array which means the entire array is sorted

2.2.5 Example: Bubble Sort

```
procedure bubbleSort(A)
  for i = n-1 down to 0 do
    for j = 0 to i do
      if A[j+1] < A[j] then
        tmp = A[j+1]
        A[j+1] = A[j]
        A[j] = tmp
      end if
    end for
  end for
end procedure
```

Using mathematical induction we will show the correctness of bubble sort algorithm.

- **Loop Invariant**: At the i^{th} iteration the last i elements $[n-i, ...n]$ are sorted relative to themselves and are all greater than all elements in $[1, ...n-i-1]$.

- **Initialization**: We start by showing that the loop invariant holds before the first loop iteration when $i = n-1$. The subarray $[n-i...n]$, therefore, consists of no element as $n-n-1 = n$, and $A[n]$ is empty as we are using 0 based index. This array is sorted (trivial)

- **Maintenance**: In the algorithm the inner for loop bubbles the maximum element from $A[0]$, $A[1]$, $A[2]$, ... $A[i]$ to the end (i.e., the i^{th} position. Which means if we assume the elements i *to* n are sorted and they are all greater than all elements $0 - i - 1$ in the i^{th} iteration, then on the so on $(i+1)^{th}$ iteration the biggest element from $A[0]$, $A[1]$, $A[2]$, ... $A[i-1]$will be on $(i-1)^{th}$ location making the lats $(i+1)$ sorted hence maintaining the loop invariant.

- **Termination**: The outer loop end when i becomes 0 hence from the loop invariant we have when the loop terminates the last $n-0$ elements are sorted which means the entire list is sorted.

2.2.6 Exercises

1. Re-write the pseudocode of insertion sort procedure to sort into non-increasing instead of non-decreasing order

2. Analyze the running time of bubble sort algorithm presented in the previous chapter

3. Prove the correctness of selection sort algorithm presented in the previous chapter using mathematical induction.

4. Analyze the running time of selection sort algorithm presented in the previous chapter

5. Given the searching problem defined in previous class

   (a) Write the pesudocode for linear search, which scans through the sequence, looking for the item.

   (b) Using loop invariant, prove that your algorithms are correct

   (c) Write the pesudocode for binary search, which uses the divide and conquer technique.

6. Consider the problem of adding two n-bit binary integers, stored in two n-element arrays A and B. The sum of the two integers should be stored in binary form in an $(n+1)$ element array C. stat the problem formally and write pseudocode for adding the two integers