

# Introduction to the Analysis of Algorithms

Manolis Koubarakis

# Outline

- How can we measure and compare algorithms meaningfully?
- $O$  notation.
- Analysis of a few interesting algorithms.

# Introduction

- **How do we measure and compare algorithms meaningfully** given that the same algorithm will run at different speeds and will require different amounts of space when run on different computers or when implemented in different programming languages?
- **Example:** let us consider a sorting algorithm for sorting an array  $A[0 : n-1]$ .

# Selection Sorting Algorithm

```
void SelectionSort (InputArray A)
{
    int MinPosition, temp, i, j;

    for (i=n-1; i>0; --i){
        MinPosition=i;
        for (j=0; j<i; ++j){
            if (A[j] < A[MinPosition]){
                MinPosition=j;
            }
        }
        temp=A[i];
        A[i]=A[MinPosition];
        A[MinPosition]=temp;
    }
}
```

# Comments

- The previous algorithm sorts the elements of an input array  $A$  in descending order using the technique of selection sorting.
- The algorithm proceeds in steps controlled by the outer `for` statement. In each such step the  $i$ -th element of the array is interchanged with the minimum element among elements  $0$  to  $i-1$ .
- The minimum element is computed by the inner `for` statement.
- The interchanging steps start with the last element of the array and proceed down to the first element.

# Running Times in Seconds to Sort an Array of 2000 Integers

Type of Computer	Time
Computer A	51.915
Computer B	11.508
Computer C	2.382
Computer D	0.431
Computer E	0.087

- Computers A, B, etc. up to E are progressively faster.
- The algorithm runs faster on faster computers.

# More Measurements

- In addition to trying different computers, we should try **different programming languages** and **different compilers**.
- Shall we take all these measurements to decide whether an algorithm is better than another one?

# A More Meaningful Criterion

- We can observe that algorithms usually **consume resources (e.g., time and space)** in some fashion that depends on the **size of the problem** solved.
- Usually, the bigger the size of a problem, the more resources an algorithm consumes.
- We usually use ***n*** to denote the size of the problem.
- **Examples of sizes:** the length of a list that is searched, the number of items in an array that is sorted etc.



# SelectionSort Running Times in Milliseconds on Two Types of Computers

Array Size $n$	Home Computer	Desktop Computer
125	12.5	2.8
250	49.3	11.0
500	195.8	43.4
1000	780.3	172.9
2000	3114.9	690.5

# Two Curves Fitting the Previous Data

- If we plot these numbers on a graph and try to fit curves to them, we find that they lie on the following two curves:

$$f_1(n) = 0.0007772n^2 + 0.00305n + 0.001$$

$$f_2(n) = 0.0001724n^2 + 0.00040n + 0.100$$

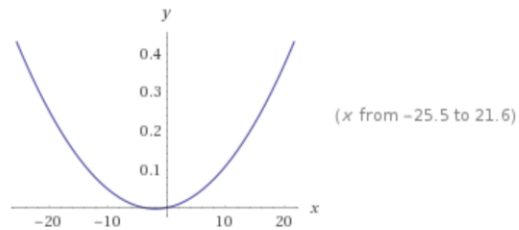
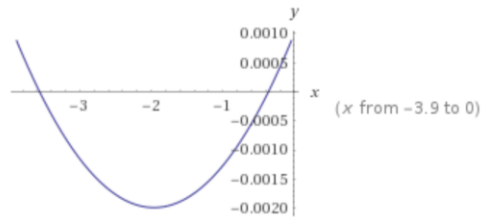
$$f_1(x)$$

Input:

$$0.0007772 x^2 + 0.00305 x + 0.001$$

[Open code](#)

Plots:



Geometric figure:

[Properties](#)

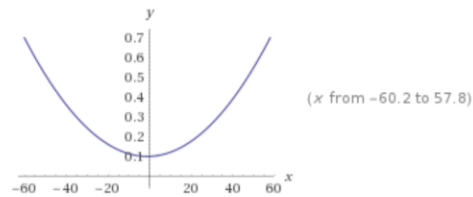
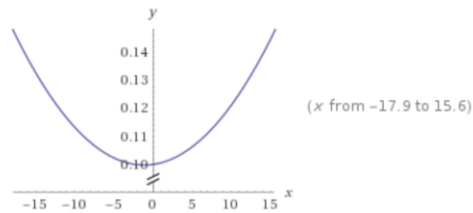
$$f_2(x)$$

Input:

$$0.0001724x^2 + 0.0004x + 0.1$$

[Open code](#)

Plots:



Geometric figure:

**parabola**

[Properties](#)

# Discussion

- The curves on the previous slide have the **quadratic** form  $f(n) = an^2 + bn + c$ .
- The difference between the two curves is that they have **different constants**  $a, b$  and  $c$ .
- Even if we implement `SelectionSort` on another computer using another programming language and another compiler, the curve that we will get will be of the same form.
- So, even though the particular measurements will change under different circumstances, **the shape of the curve** will remain the same.

# Complexity Classes

- The running times of various algorithms belong to different **complexity classes**.
- Each complexity class is characterized by a **different family of curves**.
- All of the curves in a given complexity class share **the same basic shape**. The shape is characterized by an equation that gives running times as a function of problem size.

# $O$ -notation

- This notation is used in Computer Science for talking about the time complexity of an algorithm.
- For `SelectionSort`, the time complexity is  $O(n^2)$ .
- We find this complexity by taking the **dominant term**  $an^2$  of the expression  $an^2 + bn + c$  and throwing away the constant coefficient  $a$ .

# $O$ -notation (cont'd)

- Let us consider the equation  $f(n) = an^2 + bn + c$  with  $a = 0.0001724$ ,  $b = 0.0004$  and  $c = 0.1$ . Then we have the following table:

$n$	$f(n)$	$an^2$	$n^2$ term as % of total
125	2.8	2.7	94.7
250	11.0	10.8	98.2
500	43.4	43.1	99.3
1000	172.9	172.4	99.7
2000	690.5	689.6	99.9



# $O$ -notation (cont'd)

- We conclude that **the lesser term  $bn + c$  contributes very little** to the value of  $f(n)$  even though  $c$  is 250 times more than  $b$  and  $b$  is more than two times  $a$ . Thus we can **ignore this lesser term**.
- We will also **ignore the constant of proportionality  $a$  in  $an^2$**  since we want to concentrate in the general shape of the curve.  **$a$  will differ for different implementations on different computers.**

# Some Common Complexity Classes

<i>O</i> -notation	Adjective Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n \log n)$	$n \log n$
$O(n)$	Linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
$O(10^n)$	Exponential
$O(2^{2^n})$	Doubly exponential

# Comparing Complexity Classes

- Let us assume that we have an algorithm A that runs on a computer that executes one step of this algorithm every microsecond.
- Let us assume that  $f(n)$  is the number of steps required by A to solve a problem of size  $n$ .
- Then we have the following table.

# Running Times for Algorithm A

$f(n)$	$n = 2$	$n = 16$	$n = 256$	$n = 1024$
1	1 $\mu$ sec	1 $\mu$ sec	1 $\mu$ sec	1 $\mu$ sec
$\log_2 n$	1 $\mu$ sec	4 $\mu$ sec	8 $\mu$ sec	10 $\mu$ sec
$n$	2 $\mu$ sec	16 $\mu$ sec	256 $\mu$ sec	1.02 ms
$n \log_2 n$	2 $\mu$ sec	64 $\mu$ sec	2.05 ms	10.2 ms
$n^2$	4 $\mu$ sec	25.6 $\mu$ sec	65.5 ms	1.05 secs
$n^3$	8 $\mu$ sec	4.1 ms	16.8 ms	17.9 min
$2^n$	4 $\mu$ sec	65.5 ms	$3.7 \times 10^{63}$ years	$3.7 \times 10^{294}$ years

# Size of Largest Problem Algorithm A Can Solve in $Time \leq T$

Number of steps is	$T = 1 \text{ min}$	$T = 1 \text{ hr}$
$n$	$6 \times 10^7$	$3.6 \times 10^9$
$n \log_2 n$	$2.8 \times 10^6$	$1.3 \times 10^8$
$n^2$	$7.75 \times 10^3$	$6.0 \times 10^4$
$n^3$	$3.91 \times 10^2$	$1.53 \times 10^3$
$2^n$	25	31
$10^n$	7	9

# Time Complexity Discussion

- **1**: This is the case when all the statements in a program are executed a **constant** number of times.
- **$\log n$** : When the time complexity of an algorithm is **logarithmic**, the algorithm runs a little bit slower when  $n$  increases. This time complexity is found in algorithms that solve a problem by transforming it into a series of smaller problems, reducing in each step the size of the problem by a constant amount. Every time  $n$  doubles,  $\log n$  increases only by a constant.
- **$n$** : When the time complexity of an algorithm is **linear** what happens usually is that a small part of the processing takes place for each element of the input. When  $n$  doubles, the running time of the algorithm doubles too. This time complexity is optimal for an algorithm that needs to process  $n$  inputs or to output  $n$  outputs.

# Time Complexity Discussion (cont'd)

- **$n \log n$** : This time complexity appears when an algorithm solves a problem by dividing it into smaller problems and combining the partial solutions. When  $n$  doubles, the time complexity more than doubles (but it is not very far from the double).
- **$n^2$** : When the time complexity is **quadratic**, the algorithm is practically useful only for small problems. Quadratic running times usually appear in algorithms that process pairs of elements of a problem (e.g., with two nested loops). When  $n$  doubles, the running time increases four times.
- **$n^3$** : Similarly, an algorithm that processes triples of elements of a problem (e.g., usually with three nested loops) has **cubic** running time. It is useful only for small problem sizes. When  $n$  doubles, the running time increases eight times.
- **$2^n$** : When the time complexity of an algorithm is **exponential**, the algorithm can be used in practice only for very small problem sizes. This is usually the case with algorithms that solve a problem by a brute-force method. When  $n$  doubles, the running time of the algorithm becomes the square of the previous time.

# Time Complexity

- We have to consider the following cases:
  - Worst case
  - Best case
  - Average case



# Examples of Well-Known Algorithms and Their Time Complexity

- Sequential searching of an array:  $O(n)$
- Binary searching of a sorted array:  $O(\log n)$
- Hashing (under certain conditions):  $O(1)$
- Searching using binary search trees:  $O(\log n)$
- SelectionSort, InsertionSort:  $O(n^2)$
- QuickSort, HeapSort, MergeSort:  $O(n \log n)$
- Address calculation sorting:  $O(n)$
- Parsing algorithms:  $O(n)$
- String pattern matching:  $O(n)$
- Multiplying two square  $n \times n$  matrices:  $O(n^3)$
- Traveling salesman, graph coloring:  $O(2^n)$

# Formal Definition of $O$ -notation

- We say that  $f(n)$  is  $O(g(n))$  if there exist two positive constants  $K$  and  $n_0$  such that  $|f(n)| \leq K|g(n)|$  for all  $n \geq n_0$ .

# Three Ways of Saying it in Words

- Let us assume that  $f$  and  $g$  are positive functions. Then:
  - $f(n)$  is  $O(g(n))$  provided the curve  $K \times g(n)$  can be made to lie above the curve for  $f(n)$  whenever we are to the right of some big enough value of  $n_0$ .
  - $f(n)$  is  $O(g(n))$  if there is some way to choose a constant of proportionality  $K$  so that the curve for  $f(n)$  is bounded above by the curve for  $K \times g(n)$  whenever  $n$  is big enough (i.e., when  $n \geq n_0$ ).
  - $f(n)$  is  $O(g(n))$  if for all but finitely many small values of  $n$ , the curve for  $f(n)$  lies below the curve for some suitably large constant multiple of  $g(n)$ .

# Example of a Formal Proof

- Let us suppose that a sorting algorithm A sorts a sequence of  $n$  numbers in ascending order with number of steps

$$f(n) = 3 + 6 + 9 + \cdots + 3n.$$

- We will show that the algorithm runs in  $O(n^2)$  steps.
- **Proof:** We will first find a closed form for  $f(n)$ .

# Proof (cont'd)

- Note that  $f(n) = 3 + 6 + 9 + \dots + 3n = 3(1 + 2 + \dots + n) = 3 \frac{n(n+1)}{2}$ .
- Then, choosing  $K = 3$ ,  $n_0 = 1$  and  $g(n) = n^2$ , we can show that for all  $n \geq 1$ , the following inequality holds:

$$\frac{3n(n+1)}{2} \leq 3n^2$$

# Proof (cont'd)

- Multiplying both sides of the above inequality with  $\frac{2}{3}$  gives  $n^2 + n \leq 2n^2$ .
- Subtracting  $n^2$  from both sides gives  $n \leq n^2$ .
- Dividing this inequality by  $n$  gives  $n \geq 1$ .
- The proof is now complete.

# Practical Shortcuts for Manipulating $O$ -notation

- In practice we can deal with  $O$ -notation in an easier way by separating the expression for  $f(n)$  into a **dominant** term and **lesser** terms and throwing away the lesser terms.
- In other words:  $O(f(n)) = O(\text{dominant term} \pm \text{lesser terms}) = O(\text{dominant term})$

# Scale of Strength for $O$ -notation

- We can rank the usual complexity functions on the following **scale of strength** so it is easy to determine the dominant term and the lesser terms:

$$O(1) < O(\log n) < O(n) < O(n^2) < O(n^3) < O(2^n) < O(10^n)$$



# Example

- $O(6n^3 - 15n^2 + 3n \log n) = O(6n^3) = O(n^3)$
- Let us see why we are allowed to do the above. Notice that we have  $6n^3 - 15n^2 + 3n \log n < 6n^3 + 3n \log n < 6n^3 + 3n^3 < 9n^3$
- This is the inequality that the definition of  $O$ -notation needs for  $K = 9$  and  $n \geq 1$ .

# Ignoring Bases of Logarithms

- When we use  $O$ -notation, we can **ignore the bases of logarithms** and assume that all logarithms are in base 2.
- Changing the bases of logarithms involves **multiplying by constants**, and constants of proportionality are ignored by  $O$ -notation.
- For example,  $\log_{10} n = \frac{\log_2 n}{\log_2 10}$ . Notice now that  $\frac{1}{\log_2 10}$  is a constant.

# $O(1)$

- It is easy to see why the  $O(1)$  notation is the right one for constant time complexity.
- Suppose that we can prove that an algorithm  $A$  runs in a number of steps  $f(n)$  that are always less than  $K$  steps for all  $n$ . Then  $f(n) \leq K \times 1$  for all  $n \geq 1$ . Therefore  $f(n)$  is  $O(1)$ .

# Some Algorithms and Their Complexity

- Sequential searching
- Selection sort
- Recursive selection sort
- Towers of Hanoi

# Analysis of Sequential Searching

- Suppose we have an array  $A[0:n-1]$  that contains distinct keys  $K_i$  ( $1 \leq i \leq n$ ) and assume that  $K_i$  is stored in position  $A[i-1]$ .
- **Problem:** we are given a key  $K$  and we would like to determine its position in  $A[0:n-1]$ .

# An Algorithm for Sequential Searching

```
#define n 100
typedef int Key;
typedef Key SearchArray[n];

int SequentialSearch(Key K, SearchArray A)
{
    int i;

    for (i=0; i<n; ++i) {
        if (K==A[i]) return i;
    }
    return (-1);
}
```

# Complexity Analysis

- The amount of work done to locate key  $K$  **depends on its position** in  $A[0:n - 1]$ .
- For example, if  $K$  is in  $A[0]$ , then we need only one comparison.
- In general, if  $K$  is in  $A[i - 1]$ , then we need  $i$  comparisons.

# Complexity Analysis (cont'd)

- **Best case:** This is when  $K$  is in  $A[0]$ . The complexity is  $O(1)$ .
- **Worst case:** This is when  $K$  is in  $A[n - 1]$ . The amount of work is  $an + b$  where  $a$  and  $b$  are constants. Therefore the complexity is  $O(n)$ .
- **Average case:** Let us assume that each key is equally likely to be used in a search. The average can then be computed by taking the total of all the work done for finding all the different keys and dividing by  $n$ .



# Complexity Analysis (cont'd)

- The work needed to find the  $i$ -th key  $K_i$  is of the form  $a i + b$  for some constants  $a$  and  $b$ .

Therefore:

$$\begin{aligned} Total &= \sum_{i=1}^n (ai + b) = a \sum_{i=1}^n i + \sum_{i=1}^n b = \\ &= a \frac{n(n+1)}{2} + bn \end{aligned}$$

- Now the average is:

$$Average = \frac{Total}{n} = a \frac{n+1}{2} + b = \frac{a}{2}n + \left(\frac{a}{2} + b\right)$$

- Therefore the average is  $O(n)$ .

# Selection Sorting Algorithm

```
void SelectionSort (InputArray A)
{
    int MinPosition, temp, i, j;

    for (i=n-1; i>0; --i){
        MinPosition=i;
        for (j=0; j<i; ++j){
            if (A[j] < A[MinPosition]){
                MinPosition=j;
            }
        }
        temp=A[i];
        A[i]=A[MinPosition];
        A[MinPosition]=temp;
    }
}
```

# Complexity Analysis of SelectionSort

- We start from the inner `for` statement. The `if` statement inside the `for` takes a constant amount of time  $a$ . Thus, the `for` statement takes  $ia$  time units.
- Let us now consider the outer `for`. The statements inside this `for`, except the inner `for`, take a constant amount of time  $b$ . Thus all the statements inside the outer `for` take time  $ai + b$ .

# Complexity Analysis (cont'd)

- The outer `for` takes time

$$\begin{aligned}\sum_{i=1}^{n-1} (ai + b) &= a \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} b = \\ &= a \frac{(n-1)n}{2} + (n-1)b = \\ &= \frac{a}{2}n^2 + \left(b - \frac{a}{2}\right)n - b\end{aligned}$$

- Therefore the time complexity of the algorithm is  $O(n^2)$ .

# Recursive SelectionSort

```
/* FindMin is an auxiliary function used by the Selection sort below */
int FindMin(InputArray A, int n)
{
    int i,j=n;
    for (i=0; i<n; ++i) if (A[i]<A[j]) j=i;
    return j;
}

void SelectionSort(InputArray A, int n)
{
    int MinPosition, temp;

    if (n>0){
        MinPosition=FindMin(A,n);
        temp=A[n]; A[n]=A[MinPosition]; A[MinPosition]=temp;
        SelectionSort(A, n-1)
    }
}
```

# Analysis of Recursive SelectionSort

- To use this recursive version of `SelectionSort` to perform selection sorting on the array  $A[0:n-1]$ , we make the function call `SelectionSort(A, n-1)`.
- The first thing we need to do is to analyze the running time of function `FindMin` which finds the position of the smallest element in the array  $A[0:n]$ .
- It is easy to see that the time for this function is  $an + b_1$  for suitable constants  $a$  and  $b_1$ .

# Analysis of Recursive SelectionSort (cont'd)

- We now analyze the running time of recursive function `SelectionSort`.
- Let  $T(n)$  stand for the cost, in time units, of calling `SelectionSort` on  $A[0:n]$ .
- Then the costs in `SelectionSort` are as follows:

```
if (n>0) {  
    Cost  $an + b_1$   
    Cost  $b_2$   
    Cost  $T(n - 1)$   
}
```

# Analysis of Recursive SelectionSort (cont'd)

- If  $b = b_1 + b_2$  then the following **recurrence relation** holds for  $n > 0$ :

$$T(n) = an + b + T(n - 1)$$

- The base case of this recurrence relation is  $T(0) = c$  where  $c$  is the cost of executing `SelectionSort(A, 0)`.
- To solve such recurrence relations, we can use a method called **unrolling**.



# Analysis of Recursive SelectionSort (cont'd)

$$T(n) = an + b + T(n - 1)$$

$$T(n) = an + b + a(n - 1) + b + T(n - 2)$$

$$T(n) = an + b + a(n - 1) + b + a(n - 2) + b + T(n - 3)$$

...

$$T(n)$$

$$= an + b + a(n - 1) + b + a(n - 2) + b + \cdots + a * 1 + b + T(0)$$

$$T(n)$$

$$= an + b + a(n - 1) + b + a(n - 2) + b + \cdots + a * 1 + b + c$$

# Analysis of Recursive SelectionSort (cont'd)

- Rearranging some of the terms so that all those with coefficients  $a$  and  $b$  are collected together, we have:

$$T(n) = (an + a(n - 1) + a(n - 2) + \cdots + a) + nb + c$$

$$T(n) = \sum_{i=1}^n (ai) + nb + c$$

$$= a \frac{n(n + 1)}{2} + nb + c$$

$$= \frac{a}{2}n^2 + \left(\frac{a}{2} + b\right)n + c$$

# Analysis of Recursive SelectionSort (cont'd)

- Therefore  $T(n)$  but also  $T(n - 1)$  is  $O(n^2)$ .

# A Recursive Solution to the Towers of Hanoi

```
void MoveTowers(int n, int start, int finish, int spare)
{
    if (n==1){
        printf("Move a disk from peg %ld to peg %ld\n", start,
finish);
    } else {
        MoveTowers(n-1, start, spare, finish);
        printf("Move a disk from peg %ld to peg %ld\n", start,
finish);
        MoveTowers(n-1, spare, finish, start);
    }
}
```

# Analysis of Towers of Hanoi

- Let  $n$  be the number of towers to be moved. Then the running time  $T(n)$  of the algorithm is given by the following recurrence relations:

$$T(1) = a$$

$$T(n) = b + 2T(n - 1)$$

- We will solve these recurrence relations using the technique of **unrolling plus summation**.

# Analysis of Towers of Hanoi (cont'd)

$$T(n) = b + 2T(n - 1)$$

$$T(n) = b + 2(b + 2T(n - 2))$$

$$T(n) = b + 2b + 2^2T(n - 2)$$

$$T(n) = b + 2b + 2^2(b + 2T(n - 3))$$

$$T(n) = b + 2b + 2^2b + 2^3T(n - 3)$$

...

$$T(n) = b + 2b + 2^2b + \cdots + 2^{(i-1)}b + 2^iT(n - i)$$

# Analysis of Towers of Hanoi (cont'd)

- When  $i = n - 1$ , we have:

$$T(n - i) = T(n - (n - 1)) = T(n - n + 1) = T(1) = a$$

- Therefore,  $T(n)$  can be expressed as follows:

$$\begin{aligned} T(n) &= 2^0 b + 2^1 b + 2^2 b + \dots + 2^{(n-2)} b + 2^{(n-1)} a = \\ &= \sum_{i=0}^{n-2} 2^i b + 2^{(n-1)} a = \\ &= b \sum_{i=0}^{n-2} 2^i + 2^{(n-1)} a \end{aligned}$$

# Analysis of Towers of Hanoi (cont'd)

- Now we can see that the sum is a standard **geometric progression**. So we will use the fact that  $\sum_{k=0}^m x^k = \frac{x^{m+1}-1}{x-1}$  to conclude the following:

$$\sum_{i=0}^{n-2} 2^i = \frac{2^{(n-1)} - 1}{2 - 1} = 2^{(n-1)} - 1$$



# Analysis of Towers of Hanoi (cont'd)

- Therefore:

$$\begin{aligned} T(n) &= b(2^{(n-1)} - 1) + 2^{(n-1)}a \\ &= (a + b)2^{(n-1)} - b = \frac{a + b}{2}2^n - b \end{aligned}$$

- Finally, we can see that  $T(n)$  is  $O(2^n)$ .

# What $O$ -notation Does Not Tell You

- $O$ -notation **does not apply to small problem sizes** because in this case the constants might dominate the other terms.
- One can use **experimental testing** to select the best algorithm in this case.
- Experimental testing is also useful if we want to compare algorithms that are **in the same complexity class**.

# Space Complexity

- In a similar way, we can measure the **space complexity** of an algorithm.

# Other notations

- There are also other complexity notations such as  $o(n)$ ,  $\Theta(n)$ ,  $\Omega(n)$ ,  $\omega(n)$ .
- More details in the Algorithms course.

# Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C.*  
Chapter 6.
- Robert Sedgewick. Αλγόριθμοι σε C.  
Κεφ. 2.