Complexity Overview

Defining measures for efficiency.
Improvement by factor vs. Improvement by order of magnitude.
Some examples of complexity analysis.
Intractable problems.
The limits of algorithms: some problems are unsolvable!

Different Solutions For the Same Problem

Sort
- Selection
- Insertion
- Merge
- Bubble

Search
- Linear
- Binary

Algorithmic Questions

- Is there an algorithm for this task (לשדדיה)?
- How can we know if an algorithm is correct?
- What is the time and the space complexity of the algorithm ( زمنו והמרחב)?
- What is the best algorithm for a given task?
- How do we measure efficiency?
More Questions…

- Is there something we can say about every algorithm which solves the problem?
  - For example, every algorithm must take at least x processing steps (sorting, searching), etc…
- When we implement the algorithm on a computer, will the problem be solved within a reasonable time?

Reasonable Time?

- Phone lookup (144) - few seconds
- Weather forecast - maximum one day
- Cruise missiles - real time (a late answer is useless…)
- Physical simulations - few days? Few weeks? Perhaps more?

How to Measure Time Efficiency?

Assume we have a problem P to solve, and two algorithms A1 and A2 that solve it.
We wish to compare A1 and A2’s efficiency.
What about the following test:

The algorithms were implemented and their running time was measured:
- Algorithm A1: 1.25 seconds
- Algorithm A2: 0.34 seconds

Conclusion: Algorithm A2 is better!
Questions We Must Ask

Were the algorithms tested on the same computer?
What were the inputs given to the algorithm? Were the inputs of equal size?
Did the same person implement both algorithms?
Is there a “benchmark” computer on which we test algorithms?

Input Size

The running time of an algorithm is dependent upon (is a function of) the size of input given to the algorithm:

• In a sorting algorithm - number of cells to sort
• In an algorithm for finding a word in a text - number of characters, or number of words
• In an algorithm that tests if a number is prime: size of number (number of bits which represent the number, or number of digits)

Efficiency Measure

Requirements

We would like to find a better way (than using a stopwatch for time) for measuring efficiency.

This measure should be:

• A function of the input size - n (i.e. T(n)).
• Independent of a particular computer.
• Independent of a particular implementation.
Efficiency Measure

A reasonable way to measure the time efficiency of an algorithm could be:

- Find out how many “steps” the algorithm performs for every input size (= as a function of the input size).

What could those “steps” be?

- Anything we find reasonable, as long as we know those “steps” take approximately “constant” time to run, that is, their running time is not a function of the input size.

Algorithmic Steps

Examples

- In a sorting algorithm: comparing elements, switching two cells.
- In a search algorithm: getting to the next cell, comparing elements.
- In an algorithm for testing if a number x is prime: find out if y divides x.
- In an algorithm for multiplying two numbers: multiply digits / add digits.

Meaning of ‘Constant’

Note that all these steps take “constant” time to perform.

Constant is any amount (large as it may be) which is not dependent upon (not a function of) the size of input.
Computer Independent Measure

If we wish to figure out what will be the running time of the algorithm on a particular computer, we’ll just have to:

- Estimate how long does it take to perform the “basic steps” we’ve defined on the particular computer.
- Multiply this measurement by the number of steps we’ve calculated for a specific input size.

Example: Character Search

Problem: Find out if the character c occurs in a given text.

```plaintext
found ← false
while (more characters) and (found == false)
    read the next character in the text
    if this character is c
        found ← true.
    if (end of text reached)
        print ("not found")
    else
        print ("found")
```

Solution 1 Time Analysis

Input size?
- Number of characters in text

What are the basic (constant) steps?
- Find out if end of text has been reached
- Read next character in text
- Test if character is c
Time Complexity of Solution 1 Search

What is the running time as function of input length $n$

- Depends on the particular text. But, in the worse case, no more than $n$ basic steps + constant (operations before and after loop)

$$T(n) \leq c_1 n + d_1$$

Character Search: Simple Optimization

```python
found = False
add c to end of text
while (found == False)
    read the next character in the text
    if this character is c
        found = True
    If (end of text reached)
        print("not found")
    else
        print("found").
Remove c from end of text.
```

Difference?

In solution 2, we have:

- Shortened the time it takes to perform each loop (only one Boolean check)
- Added a constant to the overall running time (add and remove a character c to the text)
Solution 2 Time Analysis

Solution 2 analysis looks the same. In the worse case, the running time of Solution 2 as a function of n is:

\[ T(n) \leq c_2 n + d_2 \]

however the use of the basic steps is slightly different, i.e. \( c_2 \) and \( d_2 \) are different:

\[ c_1 > c_2 \quad \text{and} \quad d_2 > d_1 \]

Example of Function Growth Tables

<table>
<thead>
<tr>
<th>Input Size</th>
<th>3n + 2</th>
<th>2n + 4</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>6</td>
<td>0.83</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>10</td>
<td>1.1</td>
</tr>
<tr>
<td>5</td>
<td>17</td>
<td>14</td>
<td>1.21</td>
</tr>
<tr>
<td>10</td>
<td>32</td>
<td>24</td>
<td>1.33</td>
</tr>
<tr>
<td>100</td>
<td>302</td>
<td>204</td>
<td>1.48</td>
</tr>
<tr>
<td>1000</td>
<td>3002</td>
<td>2004</td>
<td>-1.5</td>
</tr>
<tr>
<td>30000</td>
<td>90002</td>
<td>60004</td>
<td>-1.5</td>
</tr>
<tr>
<td>3000000</td>
<td>9000002</td>
<td>6000004</td>
<td>-1.5</td>
</tr>
</tbody>
</table>

Solution 1 Vs. Solution 2

In short texts, solution 1 will perform better than solution 2 (the improved solution).
However, as the text length grows, the constants \( d_1 \) and \( d_2 \) become less and less important and solution 2 will become better.

Better by a factor of \( \frac{c_2}{c_1} \)
**Improvement by Factor**

This type of algorithm improvement is called an improvement by factor, since the ratio between the complexity of both solutions, as \( n \) grows, converges to a constant:

\[
\lim_{n \to \infty} \left( \frac{T_1(n)}{T_2(n)} \right) = \text{Const}
\]

---

**Improvement by Factor: Is It Important?**

Many times (we shall soon see) we can improve more than by a factor. However, improvement by factor is still important:

- A program spends 80% of its time executing 20% of its code (the 80/20 rule).
- If we make an effort at optimizing specific “bottlenecks” in a program, we may gain a lot!
- Special programs called profilers help us in finding the “hot” spot areas in a program.

---

**Best, Average and Worse Cases**

Note that when we have counted the number of steps, we have analyzed the worst case, in which the character \( c \) is not in the text.

Other possibilities: best and average cases. Best case is not so interesting (why?). Average case computation might be complex to compute.
“GrowBy” Average Time Complexity

Assume there are \( n \) inserts in the sequence. For simplicity let’s assume we begin with 0 size array and we grow it by \( k \) each time. We must grow the array at most \( \frac{n}{k} \) times. This means at most \( \frac{n}{k} \) inserts will create an array copy of size \( k \) (the first we copy \( k \), the second we copy \( 2k \) etc.)

All other will cost constant time.

---

Average in Grow-by \( K \)

Assuming that \( n = 2k \), the cost of insert when we grow will be:

\[
1 + 2 + 3 + \cdots + \left( \frac{n}{k} - 1 \right) k = \frac{k}{2} \left( \frac{2k}{k} \right) \left( \frac{2k}{k} - 1 \right) = k \left( \frac{2k}{k} \right) \left( \frac{2k}{k} - 1 \right) = \frac{(2k - 1)n}{2}
\]

The total cost of inserts will be:

\[
\frac{(2k - 1)n}{2} + \frac{n}{k} - 1 = \frac{k + n - 2}{2k} n
\]

Or on Average \( \frac{n}{2k} = O(n) \)

---

“Doubling” Average Time Complexity

Assume there are \( n \) inserts in the sequence. For simplicity let’s assume we begin with 0 size array and each time there is no space we double it! We must grow the array at most \( \log(n) \) times. This means at most \( \log(n) \) inserts will create an array copy of size \( 2 \) (the first we copy 1, then 2, then 4 etc.)

All other will cost constant time.
**Average in Doubling**

Assuming that $n = 2^p$ the cost of insert when we grow will be:

$$1 + 2 + 4 + \cdots + 2^{p-1} = 2^p - 1 = n - 1$$

The total cost of inserts will be:

$$(n - 1) + (n - \log_2(n)) \approx 2n$$

Or on average $2 = O(1)$

---

**Problem of Doubling?**

*We may end up with half of the memory we allocated wasted (if we double and then insert only one more component).*

*Not really a problem!*

---

**Back To Binary Search**

*“Cut out” half of the search space in every step!*

- How to find a number in a phone book?
- How to find a lion in a desert?
**Binary Search Vs. Serial Search**

<table>
<thead>
<tr>
<th>Input Size</th>
<th>serial</th>
<th>binary</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>4</td>
<td>~2.5</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
<td>~14</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>10</td>
<td>~100</td>
</tr>
<tr>
<td>10000</td>
<td>10000</td>
<td>14</td>
<td>~714</td>
</tr>
<tr>
<td>100000</td>
<td>100000</td>
<td>17</td>
<td>~5883</td>
</tr>
<tr>
<td>1000000</td>
<td>100000</td>
<td>20</td>
<td>~50000</td>
</tr>
</tbody>
</table>

**Improvement by Order of Magnitude**

Recall, that when we have dealt with improvement in factor, the ratio between running times was constant. This time, we can evidently see the ratio between the number of steps is growing as the input size grows. This kind of improvement is called improvement by order of magnitude.

**Improvement As a Function of n**

\[
limit_{n \to \infty} \left( \frac{T_1(n)}{T_2(n)} \right) = f(n)
\]

In our example:

\[
f(n) = \frac{n}{\log(n)}
\]
What About the Duration of Basic Step?

When we have dealt with improvement in factor, the duration of a basic step was very interesting.

How important is it now?

Assume that the duration of a single step in serial search is 1 and that a single step in binary search takes 1000, would there still be an improvement?

---

Binary Vs. Serial, Steps of Different Duration

<table>
<thead>
<tr>
<th>Input Size</th>
<th>serial</th>
<th>binary</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>400</td>
<td>~0.025</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>700</td>
<td>~0.14</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1</td>
</tr>
<tr>
<td>10000</td>
<td>10000</td>
<td>1400</td>
<td>~7.14</td>
</tr>
<tr>
<td>100000</td>
<td>100000</td>
<td>1700</td>
<td>~58.8</td>
</tr>
<tr>
<td>1000000</td>
<td>1000000</td>
<td>2000</td>
<td>~500</td>
</tr>
</tbody>
</table>

---

Duration of Basic Step Is Negligible!

As we can see from the table, in this situation, for small input sizes (< 1000), serial search could be better. However, for larger input sizes, binary search still wins!

The reason is very simple: the ratio between duration of basic steps is constant, while the ratio between the number of basic steps grows as the input size grows.
Factor Vs. Order of Magnitude

Order of magnitude improvement is much more meaningful than improvement by factor.
This is why many times we want to “neglect” the small differences between two running time functions and get an impression of what is the “dominant” element in the functions.

R = \lim \left( \frac{T_1(n)}{T_2(n)} \right)

- R<0 cannot be (we are talking about timing)
- R=0: T_1 is better than T_2 in an order of magnitude
- 0<R<1: T_1 is better than T_2 by a factor 1/R
- R=1: They are the same
- \infty>R>1: T_2 is better than T_1 by a factor R
- R=\infty: T_2 is better than T_1 in an order of magnitude

Linear Order

For example, in serial search, any running time function will be of the form

T(n) = an + b

which is called a linear function of n.
We say that the complexity functions are of linear order, or that the complexity of the algorithms is linear.
Big-O Notation

Linear order can be symbolized by $O(n)$. We say that $T(n) = O(n)$. This is called the “Big-O notation”.

In general, we say that two functions are of the same order (have the same $O(f(n))$) if the ratio between their values is constant for large enough $n$.

Same Order of Magnitude

Example: $f(n) = n^2$ is of quadratic order, or $O(n^2)$.

All these functions are also of quadratic order:

$n^2$, $5n^2 + 6$, $5n^2 + 100n - 90$, $5000n^2$, $n^2/6$

Simply check that for any pair:

$$\lim_{n \to \infty} \left( \frac{f_1(n)}{f_2(n)} \right) = \text{Const}$$

Order of Magnitudes Classes

- $O(\log n)$ - logarithmic
- $O(n)$ – linear
- $O(n^2)$ - quadratic
- $O(n^k)$ (for $k > 2$) - polynomial
- $O(2^n)$ - exponential.
Why is it important?

Computers today are very fast, and perform millions of operations in seconds. Nevertheless, improvement in order of magnitude can reduce computation duration by seconds, hours and even days!

Wait For Better Computers?

For some problems, the only known algorithms take so many steps, that even the fastest computer that will ever exist(!), is unable to solve the problem in reasonable time!

Example: The traveling salesperson (TSP). Find the shortest path which starts at a city and traverses all cities.

The Traveling Salesperson Problem
Solutions to TSP

"Brute Force" algorithm:
- For each possible path, find its length
- Choose the path with minimum length

Number of possible paths
- At most (n-1)(n-2)...1 = (n-1)! (factorial)

Complexity of algorithm: (n-1)! = O(n!)

---

n! complexity

How long will it take to go over O(n!) paths for growing n?
Assume we have a computer which can compute a million paths per second!

---

TSP Computing Times

<table>
<thead>
<tr>
<th>cities</th>
<th># of paths</th>
<th>computing time</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>120</td>
<td>~8 milliseconds</td>
</tr>
<tr>
<td>11</td>
<td>3,628,800</td>
<td>~3.5 seconds</td>
</tr>
<tr>
<td>13</td>
<td>479,001,600</td>
<td>~8 minutes</td>
</tr>
<tr>
<td>16</td>
<td>1,307,674,368,000</td>
<td>~15 days</td>
</tr>
<tr>
<td>18</td>
<td>~3,355,000,000,000,000</td>
<td>~11 years</td>
</tr>
<tr>
<td>21</td>
<td>~2,430,000,000,000,000,000,000</td>
<td>~77,000 years!</td>
</tr>
</tbody>
</table>
TSP - an Intractable Problem!

* TSP evidently cannot be solved for even very small input sizes!
* The complexity of TSP $O(n!) \geq O(2^n)$ is exponential!
* Any exponential running time function is intractable.

Wait For Better Computers?

* If $T(n) = 2^n$ then $n = \log(T(n))$
* What is the input size $n$ we can solve with the following conditions?
  * Parallel computer with # of processors as the number of atoms in the universe
  * Time: Number of years since the big bang
* These are only constants!
* $n = \log(T(n)/k)$ still very very small!

Effect of Improved Technology

<table>
<thead>
<tr>
<th>Complexity</th>
<th>With Present Computer</th>
<th>With Computer 100 Times Faster</th>
<th>With Computer 1000 Times Faster</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$x$</td>
<td>100$x$</td>
<td>1000$x$</td>
</tr>
<tr>
<td>$n^2$</td>
<td>$x$</td>
<td>10$x$</td>
<td>31.6$x$</td>
</tr>
<tr>
<td>$n^3$</td>
<td>$x$</td>
<td>4.64$x$</td>
<td>10$x$</td>
</tr>
<tr>
<td>$n^4$</td>
<td>$x$</td>
<td>2.5$x$</td>
<td>3.98$x$</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$x$</td>
<td>$x + 6.64$</td>
<td>$x + 9.97$</td>
</tr>
<tr>
<td>$3^n$</td>
<td>$x$</td>
<td>$x + 4.19$</td>
<td>$x + 6.29$</td>
</tr>
</tbody>
</table>
According to the CIA…

The land area on earth is about 150 million square kilometers.
The population on Earth is about 6000 million, thus the average population density is about 40 people / square kilometer.
The current population growth is about 1.5% per year.

Exponential Is Bad!

1.5% may not sound like much growth, however: $1.015^{1000} \approx 2.9$ million.
Thus by the year 3000, if the population growth continues at 1.5% per year, the average population density will be around 120000 people per square meter(!).
By the year 4000 there will be 320 trillion people per square meter...

It Gets Even Worse...

It seems that computers can solve any problem by writing the appropriate program.
There are problems which cannot be solved by any computer!
Such problems were discovered and studied by the mathematician Alan Turing, the most famous: the halting problem (1937).
The Halting Problem

Given a program P and an input x, does the program P halt on the input x?

If everything is solvable than one can imagine a method

```java
boolean checkStop(P, x) { ... }
```

Of course P can be represented as a text file and x will be some representation of the input for P (maybe also a text file).

The checkStop Method

- reads the program P (which is just a text file)
- runs an algorithm which determines if the program P halts on the input x (such as simulating the way a computer will work with P)
- returns true if P halts on the specified input x
- returns false if P does not halt on the specified input x

Our Strange Program

```java
myProg(P) {
    // ...
    if (checkStop(P, P))
        loop forever;
    else
        print "halt";
}
```

What happens if we run myProg and give it as input the program myProg itself?

```java
myProg(myProg)
```
Option 1

Assume myProg(myProg) terminates and prints “halt”. This means:

- checkStop(myProg,myProg) returned false, which in turn means that myProg does not terminate on the input myProg!
- But we just assumed it does - A contradiction!

Option 2

Assume myProg(myProg) loops forever, which means:

- checkStop(myProg,myProg) returned true, which in turn means that myProg does terminate on the input myProg!
- Another contradiction!

A Paradox!

myProg(myProg)  
Prints “halt”  \hspace{1cm}  Infinite loop

testStop(myProg,myProg)  
Returned false  \hspace{1cm}  Returned true
Undecidability

Conclusion: Our assumption, that there exists a method \texttt{testStop}, which determines if a program halts on a specific input, cannot be true!
This can be formalized to prove that no algorithm can solve the halting problem!
We say that the Halting Problem is undecidable (הלולי ייווייטא).

Course Summary

- Algorithm/Model
- Program Code
- Compilers etc.
- Operating System
- Execution Cycle
- Hardware

Good Luck!

... And thank you for listening!