The Growth of Functions

Section 3.2

Video 36: Growth of Functions

- Efficiency of Algorithms
- Characterizing growth of functions
- Big-O Notation

The Growth of Functions

- In computer science, we want to understand how quickly an algorithm can solve a problem as the size of the input grows.
  - We can compare the efficiency of two different algorithms for solving the same problem.
  - We can also determine whether it is practical to use a particular algorithm as the input grows.
- We characterize the **efficiency of an algorithm** by a function of its problem size
  - Efficiency is captured by the **growth of this function**
  - In both computer science and in mathematics, there are many cases where we care about how fast a function grows.

How much detail is needed?

When determining efficiency of algorithms we have two extremes

1. Precise count of everything involved
   - computer instructions, disk accesses, …
   - as a function of the problem size
     ➜ inconvenient, not always well-defined
2. “it took a few seconds on my laptop”
   - what if size doubles?
     ➜ not sufficiently informative

Example

Assume it took ***s* seconds** to find the maximum among ***n* unsorted items**

How to predict the time required to find the maximum among $2n$, $3n$, or m items?

Finding the maximum takes linear time

→ reasonable to predict **2*s*, 3*s*,** and **(*m*/*n*) *s* seconds**

Another example

Assume that, for some large n, sorting n items using bubble sort took s seconds

How to predict the time required to sort $2n$, $3n$, or m items using bubble sort?

Sorting using bubble sort takes quadratic time

$\Rightarrow$ reasonable to predict $\mathbf{2^2}\boldsymbol{s}$, $\mathbf{3^2}\boldsymbol{s}$, and $\mathbf{(m/n)^2}\,\boldsymbol{s}$ **seconds**

Example

Let $f(n) = 2n^2 + 240n + 9600$ be a function to estimate the time to solve problem of size n

Let $g(n) = 2n^2$, $h(n) = 240n$, $t(n) = 9600$

for small n:	$t(n)$ most significant
then:	$h(n)$ takes over
but ultimately:	**only $g(n)$ is relevant**

Observation

Let $f(n)$ estimate the time to solve problem of size n

if

$$f(n) = g(n) + h(n) + \ldots + t(n)$$

for functions $g, h, \ldots, t$: $\mathbf{N} \rightarrow \mathbf{R}$

then the "ultimately largest" of $g, h, \ldots, t$ determines f's behavior when n gets large

Observation

Let g(n) be f(n)'s "ultimately most relevant part"

Then f(n)'s growth rate is **independent of multiplicative constants** in g(n):

$$\frac{g(m)}{g(n)} = \frac{cg(m)}{cg(n)}$$

Consequence

When considering a runtime function f(n)

- Focus on part that grows “fastest” (for $n \to \infty$)
- Forget about multiplicative constants

- We do not care about small values of n
- We do not care about the absolute value, but about growth

Big-*O* Notation

Definition: Let *f* and *g* be functions from the set of integers or the set of real numbers to the set of real numbers. We say that ***f*(*x*) is *O*(*g*(*x*))** if there are constants *C* and *k* such that

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$.

- This is read as "*f*(*x*) is big-*O* of *g*(*x*)" or "*g* asymptotically dominates *f*.
- The constants C and k are called **witnesses** to the relationship *f*(*x*) is *O*(*g*(*x*)).
- Only one pair of witnesses is needed.

Summary

- Big-O notation
  - Abstract from details of how a function grows
  - Considers the fastest growing part for large values
- Used to characterize the efficiency of algorithms

Video 37: Big-O

- Illustration of Big-O
- Proofs for Big-O
- Examples for Big-O

Example

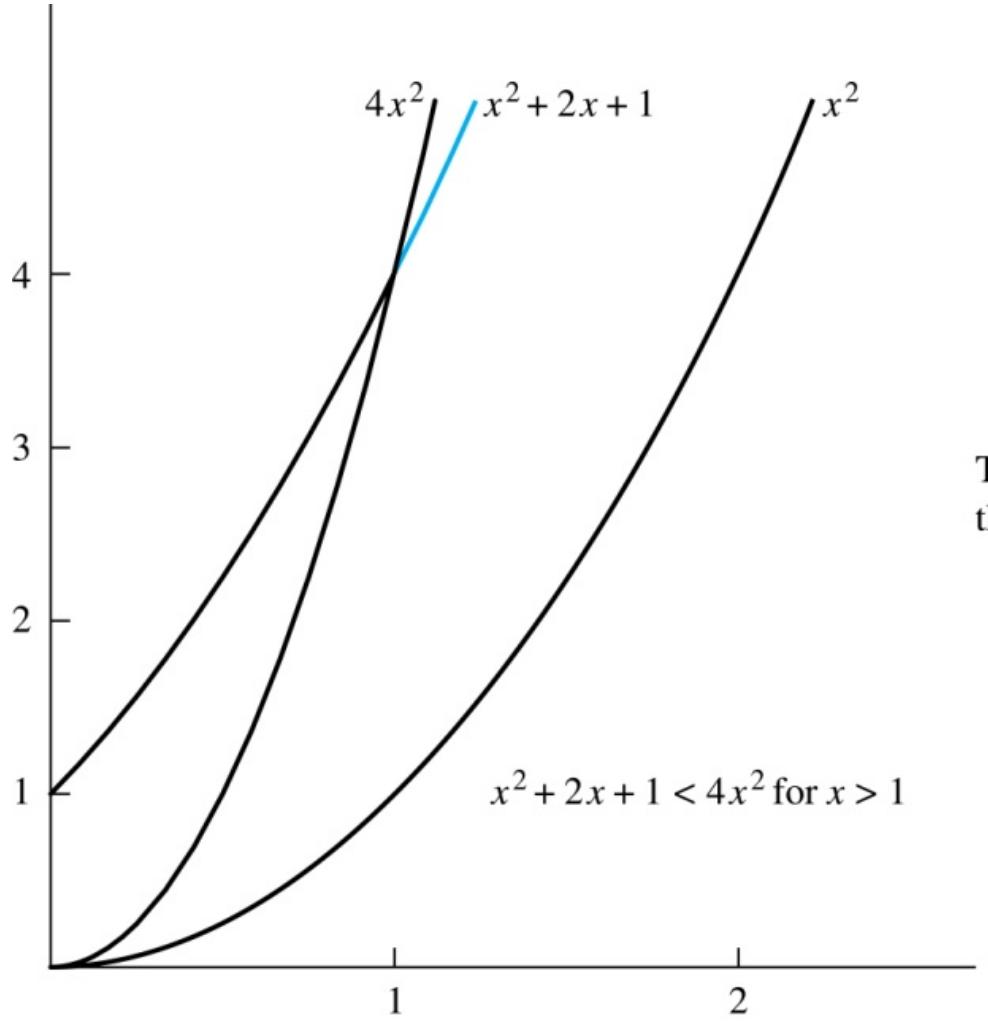
Show that $f(x) = x^2 + 2\mathrm{x} + 1$ is $O(x^2)$

Proof: If $\boldsymbol{x > 1}$, then $x < x^2$ and $1 < x^2$. Therefore

$$0 \leq f(x) = x^2 + 2\mathrm{x} + 1 \leq x^2 + 2x^2 + x^2 = \mathbf{4}x^2$$

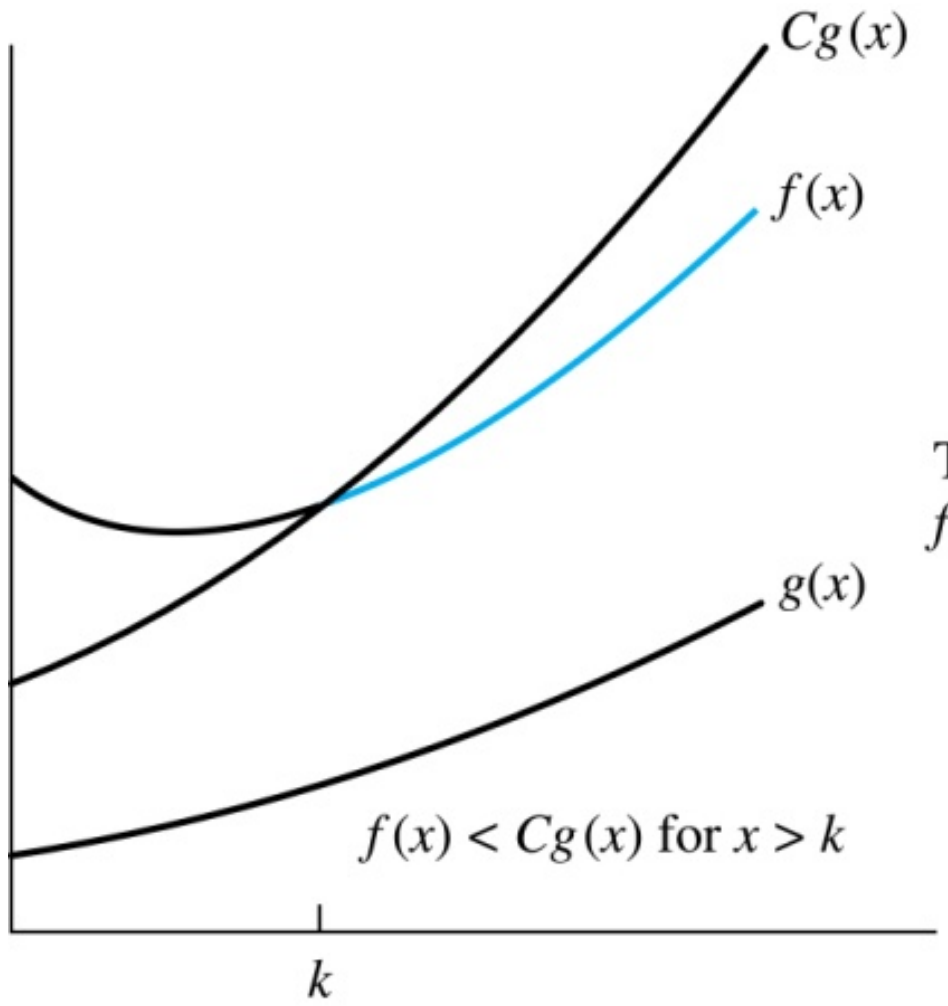
We choose $C = 4$ and $k = 1$ as witnesses to show that $f(x)$ is $O(x^2)$

Illustration of Big-*O* Notation



The part of the graph of $f(x) = x^2 + 2x + 1$ that satisfies $f(x) < 4x^2$ is shown in blue.

Illustration of Big-*O* Notation



f(*x*) is *O*(*g*(*x*))

The part of the graph of $f(x)$ that satisfies $f(x) < Cg(x)$ is shown in color.

Example

Show that x^2 is not $O(x)$.

Proof:

- Suppose there are constants C and k for which $x^2 \leq Cx$, whenever $x > k$.
- By dividing both sides of $x^2 \leq Cx$ by x, we obtain $x \leq C$, whenever $x > k$.
- This is a contradiction.

Big-*O* facts

75 is $O(1)$ and 1 is $O(75)$

1 is $O(x)$ but x is not $O(1)$

x is $O(x^2)$ but x^2 is not $O(x)$

x^2 is $O(x^2)$ and x^2 is $O(x^3)$

x^2 is $O(6x^2+x+3)$ and $6x^2+x+3$ is $O(x^2)$

$O(6x^2+x+3)$ and $O(75)$ are unusual

Examples

$f(x) = 2x^2 + 240x + 9600$ is $O(x^2)$

- $C = 4,\ k = 240$ are witnesses
- $\forall\ x > 240\ \ |f(x)| \leq 4|x^2|$

$r(x) = 0.0001x^2 + 24000x + 9600^{9600}$ is $O(x^2)$

- $C = 3,\ k = 9600^{4800}$ are witnesses
- $\forall\ x > 9600^{4800}\ |r(x)| \leq 3|x^2|$

$s(x) = 31\ (\sqrt{x})\ \log(x) + x\ \log_{10}(x) + 167x$ is $O(x\ \log(x))$

- $C = 2,\ k = 10^{167}$ are witnesses
- $\forall\ x > 10^{167}\ |s(x)| \leq 2|x\ \log(x)|$

Big-O Estimates for Polynomials

Theorem: Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0$ where $a_0, a_1, \ldots, a_n$ are real numbers with $a_n \neq 0$. Then $f(x)$ is is $O(x^n)$.

The leading term $a_n x^n$ of a polynomial dominates its growth.

Proof

$$|f(x)| = |a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0|$$

$$\leq |a_n| x^n + |a_{n-1}| x^{n-1} + \cdots + |a_1| x^1 + |a_0| \qquad \text{triangle inequality}$$

$$= x^n (|a_n| + |a_{n-1}| /x + \cdots + |a_1| /x^{n-1} + |a_0| / x^n)$$

$$\leq x^n (|a_n| + |a_{n-1}| + \cdots + |a_1| + |a_0|) \qquad \text{Assuming } x > 1$$

Take $C = |a_n| + |a_{n-1}| + \cdots + |a_0|$ and $k = 1$.

Then $f(x)$ is $O(x^n)$.

An Important Point about Big-*O* Notation

You may see " $f(x) = O(g(x))$" instead of " $f(x)$ is $O(g(x))$"

- This is an abuse of the equality sign

It is ok to write $f(x) \in O(g(x))$

- $O(g(x))$ represents the set of functions that are $O(g(x))$.

Summary

- Examples of Big-O
- Big-O for polynomials
- Use of Big-O notation

Video 38: Advanced Big-O Facts

- Big-O for more functions
- Big-O for combined functions

More big-*O* facts

$\forall u > v$, u, v constant:

n^v is $O(n^u)$ but n^u is not $O(n^v)$

$\forall a > 0, b > 0, u > v$, a, b, u, v constant:

$\log_b(n^v)$ is $O(\log_a(n^u))$

$\log_a(n^u)$ is $O(\log_b(n^v))$

and they are all $O(\log(n))$

Useful Big-*O* Estimates Involving Logarithms, Powers, and Exponents

If $d > c > 1$,
then n^c is $O(n^d)$, but n^d is not $O(n^c)$

If $b > 1$ and c and d are positive,
then $(\log_b n)^c$ is $O(n^d)$, but n^d is not $O((\log_b n)^c)$

If $b > 1$ and d is positive,
then n^d is $O(b^n)$, but b^n is not $O(n^d)$

If $c > b > 1$,
then b^n is $O(c^n)$, but c^n is not $O(b^n)$

Big-*O* Estimates for the Factorial Function

Factorial function

$$f(n) = n! = 1 \times 2 \times \cdots \times n \; .$$

$$n! = 1 \times 2 \times \cdots \times n \; \leq \; n \times n \times \cdots \times n \; = \; n^n$$

$$n! \text{ is } O(n^n) \text{ taking } C = 1 \text{ and } k = 1.$$

Logarithm of factorial function: log *n*!

Given that $n! \leq n^n$ then $\log(n!) \leq n \; log(n)$.

Hence, $\log(n!)$ is $O(n \; log(n))$ taking $C = 1$ and $k = 1$.

Combinations of Functions

If $f(x)$ is $O(g(x))$ and $g(x)$ is $O(h(x))$ then $f(x)$ is $O(h(x))$

If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$ then $(f_1 * f_2)(x)$ is $O(g_1(x)*g_2(x))$

If $f_1(x)$ and $f_2(x)$ are both $O(g(x))$ then $(f_1 + f_2)(x)$ is $O(g(x))$

If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$ then $(f_1 + f_2)(x)$ is $O(\max(|g_1(x)|, |g_2(x)|))$

Combinations of Functions

If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$ then $(f_1 + f_2)(x)$ is $O(\max(|g_1(x)|,|g_2(x)|))$

Proof: By the definition of big-O notation, there are constants C_1, C_2, k_1, k_2 such that $|f_1(x) \leq C_1|g_1(x)|$ when $x > k_1$ and $f_2(x) \leq C_2|g_2(x)|$ when $x > k_2$.

$|(f_1 + f_2)(x)| = |f_1(x) + f_2(x)| \leq |f_1(x)| + |f_2(x)|$ by the triangle inequality $|a + b| \leq |a| + |b|$

$|f_1(x)| + |f_2(x)| \leq C_1|g_1(x)| + C_2|g_2(x)|$

$\leq C_1|g(x)| + C_2|g(x)|$ where $g(x) = \max(|g_1(x)|,|g_2(x)|)$

$= (C_1 + C_2)|g(x)|$

$= C|g(x)|$ where $C = C_1 + C_2$

Therefore $|(f_1 + f_2)(x)| \leq C|g(x)|$ whenever $x > k$, where $k = \max(k_1,k_2)$.

Ordering Functions by Order of Growth

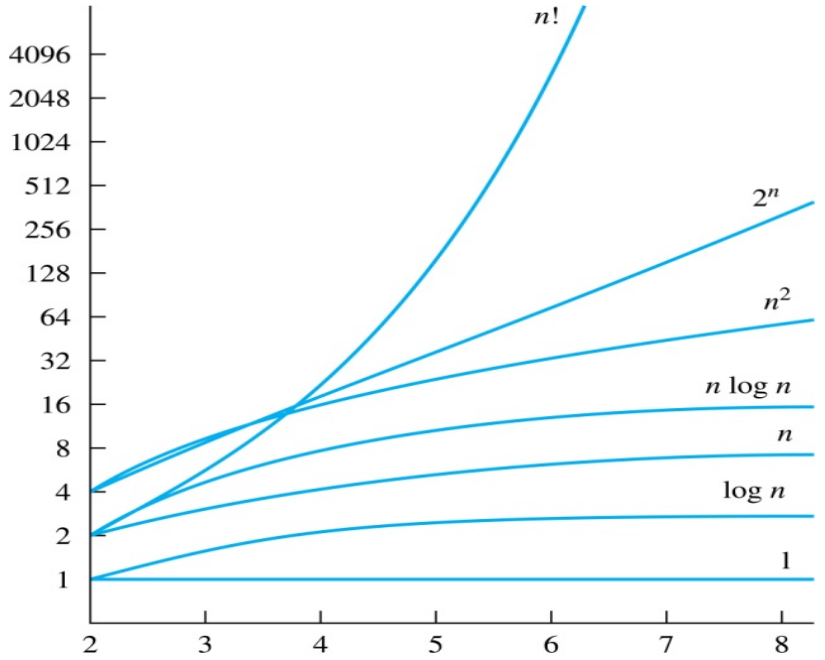
Put the functions in order so that each function is big-O of the next function on the list.

$f_1(n) = (1.5)^n$
$f_2(n) = 8n^3 + 17n^2 + 111$
$f_3(n) = (\log n)^2$
$f_4(n) = 2^n$
$f_5(n) = \log(\log n)$
$f_6(n) = n^2 (\log n)^3$
$f_7(n) = 2^n (n^2 + 1)$
$f_8(n) = n^3 + n(\log n)^2$
$f_9(n) = 10000$
$f_{10}(n) = n!$

The order is:

9, 5, 3, 6, 2, 8, 1, 4, 7, 10

Display of Growth of Functions



Note the difference in behavior of functions as *n* gets larger

Summary

- Big-O for powers, logarithms and factorials
- Big-O for sum and product of functions

Video 39: Big-Omega and Big-Theta

- Big-Omega
- Big-Theta
- little-o

Big-Omega Notation

Definition: Let *f* and *g* be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $\boldsymbol{f(x)}$ **is** $\boldsymbol{\Omega(g(x))}$ if there are constants *C* and *k* such that $|f(x)| \geq C|g(x)|$ when $x > k$.

- We say that "*f*(*x*) is big-Omega of *g*(*x*)."
- Big-*O* gives an upper bound on the growth of a function, while Big-Omega gives a lower bound
- Big-Omega tells us that a function grows at least as fast as another.
- *f*(*x*) is $\Omega(g(x))$ if and only if *g*(*x*) is *O*(*f*(*x*))

Example

$f(x) = 8x^3 + 5x^2 + 7$ is $\Omega(x^3)$

since $g(x) = x^3$ is $O(8x^3 + 5x^2 + 7)$

Big-Theta Notation

Definition: Let *f* and *g* be functions from the set of integers or the set of real numbers to the set of real numbers.
The function $\boldsymbol{f(x)}$ **is** $\boldsymbol{\Theta(g(x))}$ if $f(x)$ is $O(g(x))$ and $f(x)$ is $\Omega(g(x))$.

- We say that "f is big-Theta of *g*(*x*)"

  or "*f*(*x*) is of *order g*(*x* or "*f*(*x*) and *g*(*x*) are of the *same order*."
- $f(x)$ is $\Theta(g(x))$ if and only if there exists constants C_1 , C_2 and *k* such that $C_1 g(x) < f(x) < C_2\, g(x)$ if $x > k$.

Example

$f(x) = 8x^3 + 5x^2 + 7$ is $\Omega(x^3)$

$\mathrm{g}(x) = x^3$ is $\Omega(8x^3 + 5x^2 + 7)$

Therefore $f(x)$ is $\Theta\big(g(x)\big)$

Big-Theta Notation

Some further points to pay attention

- When $f(x)$ is $\Theta\big(g(x)\big)$ then also $g(x)$ is $\Theta\big(f(x)\big)$
- $f(x)$ is $\Theta\big(g(x)\big)$ if and only if $f(x)$ is $\mathrm{O}\big(g(x)\big)$ and $\mathrm{g}(x)$ is $\mathrm{O}\big(f(x)\big)$
- Sometimes people are careless and use the big-*O* notation with the same meaning as big-Theta.

Big-O Estimates for Polynomials

Theorem: Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0$ where $a_0, a_1, \ldots, a_n$ are real numbers with $a_n \neq 0$. Then $f(x)$ is is $O(x^n)$.

The leading term $a_n x^n$ of a polynomial dominates its growth.

Big-Theta Estimates for Polynomials

Theorem: Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0$ where $a_0, a_1, \ldots, a_n$ are real numbers with $a_n \neq 0$.

Then $f(x)$ is of order x^n (or $\Theta(x^n)$)

Example:

The polynomial $8x^3 + 5x^2 + 7$ is order of x^3 (or $\Theta(x^3)$)

Little-o

"$f(x)$ is $o(g(x))$" if $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$

- We also say that "*f* is *little-o* of *g*"

Example

x^2 is $o(x^3)$ but $x^2 + x + 1$ is not $o(x^2)$

$\lim_{x\to\infty} \frac{x^2}{x^3} = 0$ but $\lim_{x\to\infty} \frac{x^2+x+1}{x^2} = 1$

Little-o and Big-O

If $f(x)$ and $g(x)$ are functions such that $f(x)$ is $o(g(x))$, then $f(x)$ is $O(g(x))$.

However: if $f(x)$ and $g(x)$ are functions such that $f(x)$ is $O(g(x))$, then it does not necessarily follow that $f(x)$ is $o(g(x))$.

Example: $x^2 + x + 1$ is $O(x^2)$, but not $o(x^2)$

Summary

- Lower bounds on growth: Big-Omega
- Equal growth: Big-Theta
- little-o: different from Big-O

Complexity of Algorithms

Section 3.3

Video 40: Introduction to Complexity

- Computational Complexity
- Time and Space Complexity
- Worst and Average Case Complexity

The Complexity of Algorithms

Given an algorithm, how efficient is this algorithm for solving a problem given an input of a particular size (**computational complexity**)?

- How much time does this algorithm use to solve the problem for an input of a given size (**time complexity**)?
- How much computer memory does this algorithm use to solve the problem for an input of a given size (**space complexity**)?

Understanding complexity is important

- To understand whether it is practical to use an algorithm for inputs of a particular size
- To compare the efficiency of different algorithms for solving the same problem.

Time Complexity

We will focus on **time complexity**

- If the algorithm is sequential, all operations are executed in sequential order.
- Then time complexity corresponds to the **number of operations** performed.
- We will use big-*O* and big-Theta notation to describe the time complexity.

We **ignore implementation details** (including the data structures used and both the hardware and software platforms) because it is extremely complicated to consider them.

Determining Time Complexity

We determine the number of basic operations

- E.g., comparisons and arithmetic operations (addition, multiplication, etc.).
- We assume all operations use a constant time.
- The time for the basic operations can be different from one computer to the next.
- We ignore minor details, such as the “house keeping” aspects of the algorithm

Worst-Case Time Complexity

We will focus on the **worst-case time** complexity of an algorithm

- An upper bound on the number of operations an algorithm uses to solve a problem with input of a particular size.
- It is usually much more difficult to determine the **average case time complexity** of an algorithm, the average number of operations an algorithm uses to solve a problem over all inputs of a particular size.

Complexity Analysis of Algorithms

Example: Worst case time complexity of the algorithm for finding the maximum element in a finite sequence.

procedure $max(a_1, a_2, \ldots, a_n$: integers)

$max := a_1$

for $i := 2$ to n

if $max < a_i$ then $max := a_i$

return max

Counting the number of comparisons.

- The $max < a_i$ comparison is made $n - 1$ times.
- Each time i is incremented, a test is made to see if $i \leq n$.
- One last comparison determines that $i > n$.
- Exactly $2(n - 1) + 1 = 2n - 1$ comparisons are made.

Hence, the time complexity of the algorithm is $\Theta(n)$.

Summary

- Computational Complexity
  - Abstracts from implementation details
- Time and Space Complexity
  - Time complexity corresponds to number of operations
- Worst and Average Case Complexity
  - Worst case complexity in general easier to analyse

Video 41: Complexity Analyses

- Linear Search
- Binary Search
- Bubble Sort
- Insertion Sort

How much detail is needed?

procedure *max*(a_1, a_2, …., a_n: integers)

max := a_1

for *i* := 2 to *n*

    if *max* < a_i then *max* := a_i

return *max*

- Loop is executed c1(n) = n-1 times
  - In each execution 2 comparison
  - One extra comparison at the end
- Cost function
  - f(n) = 2*c1(n) + c2(n), c2(n) = 1
- c1(n) is O(n), Therefore also 2*c1(n) is O(n)
- c2(n) is O(1), Therefore 2*c1(n) + c2(n) is O(n)
- It is sufficient to count how often the loop is executed
  - Multiple operations per loop do not matter
  - Extra operations before or after do not matter

Worst Case Complexity of Linear Search

procedure *linearsearch*(x: integer, $a_1, a_2, \ldots, a_n$: distinct integers)

```
i := 1
while (i ≤ n and x ≠ a_i)
        i := i + 1
if i ≤ n then location := i
        else location := 0
return location
```

The test in the inner loop is executed n+1 times in the worst case, when the element is not found

f(n) = n + 1 is Θ(n)

Hence, the complexity is Θ(n).

Average Case Complexity of Linear Search

procedure *linearsearch*(x:integer, $a_1, a_2, \ldots, a_n$: distinct integers)

```
i := 1
while (i ≤ n and x ≠ a_i)
        i := i + 1
if i ≤ n then location := i
        else location := 0
return location
```

Assume the element x is in the list and that the possible positions are equally likely.

If $x = a_i$, the number of comparisons is 2i + 1

- Two comparison for each loop
- One comparison for the if statement

So the number of comparisons is:

$$\frac{3+5+7+\ldots+(2n+1)}{n} = \frac{2(1+2+3+\ldots+n)+n}{n} = \frac{2[\frac{n(n+1)}{2}]}{n} + 1 = n + 2$$

Hence, the average-case complexity of linear search is Θ(n).

Worst Case Complexity of Binary Search

procedure binary search(x: integer, $a_1, a_2, \ldots, a_n$: increasing integers)

$i := 1$

$j := n$

while $i < j$

$m := \lfloor (i + j)/2 \rfloor$

if $x > a_m$ **then** $i :=$ m + 1

else $j :=$ m

if $x = a_i$ **then** *location* := i

else *location* := 0

return *location*

Assume (for simplicity) $n = 2^k$ elements. Note that $k = \log_2 n$.

At the first iteration the size of the list is 2^k

- after the first iteration it is 2^{k-1}
- Then 2^{k-2}
- and so on until the size of the list is $2^1 = 2$.

Therefore k = log(n) iterations are performed

Therefore, the worst time complexity is $\Theta(\log(n))$, better than linear search.

Worst Case Complexity of Bubble Sort

procedure *bubblesort*(a_1,…,a_n: real numbers with $n \geq 2$)

for i := 1 to $n - 1$

for j := 1 to $n - i$

if $a_j > a_{j+1}$

then interchange a_j and a_{j+1}

A sequence of $n-1$ passes is made through the list.

On each pass i the inner loop is executed n-i times

Adding up the loops:

$(n-1) + (n-2) + \ldots + 2 + 1 = n*(n-1)/2 = n^2/2 - n/2$

The worst-case complexity of bubble sort is therefore $\Theta(n^2)$

Worst Case Complexity of Insertion Sort

procedure *insertion sort*(a_1,…,a_n: real numbers with $n \geq 2$)

```
for j := 2 to n
    i := 1
    while a_j > a_i and i < j
        i := i + 1
    m := a_j
    for k := 0 to j − i − 1
        a_{j-k} := a_{j-k-1}
    a_i := m
```

n-1 passes are executed for j = 2,…,n

In each pass

- The while loop is executed at most j times
- The for loop is executed at most j times

Since 2*(2+3+…+n) = n*(n+1) -2 the complexity is $\Theta(n^2)$.

Summary

- Linear Search: $\Theta(n)$
- Binary Search: $\Theta(\log(n))$
- Bubble Sort: $\Theta(n^2)$
- Insertion Sort: $\Theta(n^2)$

Video 42: Matrices

- Definition of Matrices
- Matrix Multiplication
- Complexity

Matrices

Definition: A **matrix** is a rectangular array of numbers. A matrix with *m* rows and *n* columns is called an $m \times n$ matrix.

- A matrix with the same number of rows as columns is called *square*.
- Two matrices are *equal* if they have the same number of rows and the same number of columns and the corresponding entries in every position are equal.

3×2 matrix $\quad \begin{bmatrix} 1 & 1 \\ 0 & 2 \\ 1 & 3 \end{bmatrix}$

Matrices are discrete structures that are used for many purposes

- Specifying linear transformations (a function)
- Specifying binary relationships

Matrix Notation

Let *m* and *n* be positive integers and let $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ . & . & & . \\ . & . & & . \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$

The ***i*th row** of **A** is the $1 \times n$ matrix $[a_{i1}, a_{i2}, \dots, a_{in}]$.

The ***j*th column** of **A** is the $\mathrm{m} \times 1$ *matrix:* $\begin{bmatrix} a_{1j} \\ a_{2j} \\ . \\ . \\ a_{mj} \end{bmatrix}$

The $(i,j)^{\text{th}}$ **element** or **entry** of **A** is the element a_{ij}.
We can write $\mathbf{A} = [a_{ij}]$ to denote the matrix with its $(i, j)^{\text{th}}$ element equal to a_{ij}.

Matrix Arithmetic: Addition

Definition: Let **A** = [a_{ij}] and **B** = [b_{ij}] be $m \times n$ matrices. The **sum** of **A** and **B**, denoted by **A** + **B**, is the $m \times n$ matrix that has $a_{ij} + b_{ij}$ as its $(i, j)^{th}$ element.

Example:

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 2 & -3 \\ 3 & 4 & 0 \end{bmatrix} + \begin{bmatrix} 3 & 4 & -1 \\ 1 & -3 & 0 \\ -1 & 1 & 2 \end{bmatrix} = \begin{bmatrix} 4 & 4 & -2 \\ 3 & -1 & -3 \\ 2 & 5 & 2 \end{bmatrix}$$

- We can write, **A** + **B** = [$a_{ij} + b_{ij}$].
- Note that matrices of different sizes can not be added.

Matrix Multiplication

Definition: Let **A** be an $m \times k$ matrix and **B** be a $k \times n$ matrix. The **product** of **A** and **B**, denoted by **AB**, is the $m \times n$ matrix that has its $(i, j)^{th}$ element

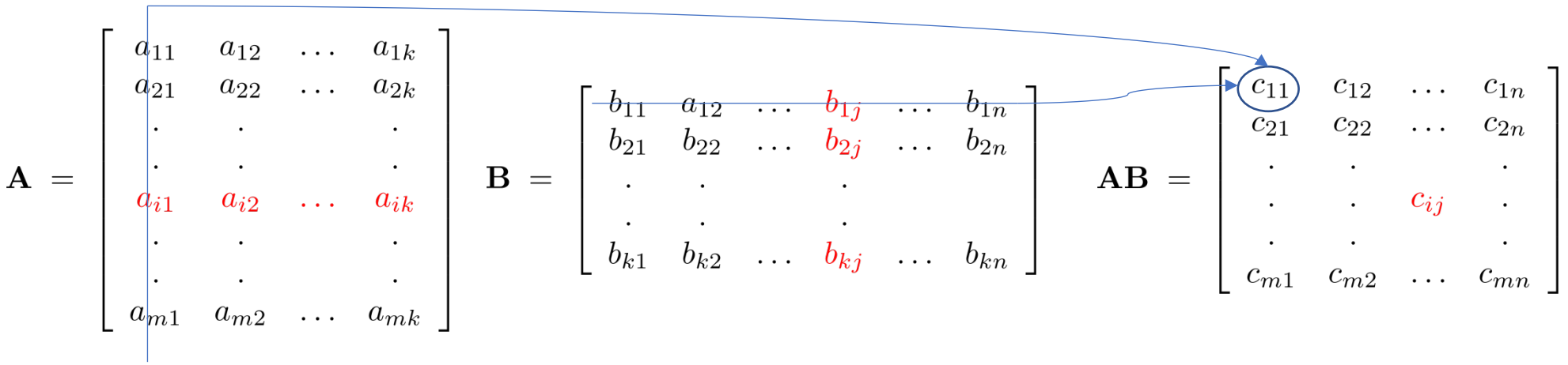
$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{kj}b_{2j}.$$

Example:

$$\begin{bmatrix} 1 & 0 & 4 \\ 2 & 1 & 1 \\ 3 & 1 & 0 \\ 0 & 2 & 2 \end{bmatrix} \begin{bmatrix} 2 & 4 \\ 1 & 1 \\ 3 & 0 \end{bmatrix} = \begin{bmatrix} 14 & 4 \\ 8 & 9 \\ 7 & 13 \\ 8 & 2 \end{bmatrix}$$

- The product of two matrices is undefined when the number of columns in the first matrix is not the same as the number of rows in the second.

Illustration of Matrix Multiplication



$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{ik}b_{kj}$$

Matrix Multiplication is not Commutative

Example: Let $\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix}$ $\mathbf{B} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$

$\mathbf{AB} = \begin{bmatrix} 2 & 2 \\ 5 & 3 \end{bmatrix}$ $\mathbf{BA} = \begin{bmatrix} 4 & 3 \\ 3 & 2 \end{bmatrix}$

AB ≠ BA

Matrix Multiplication Algorithm

We can write the definition of matrix multiplication **C** = **AB** as an algorithm

```
procedure matrixmult(A, B: matrices)
   for i := 1 to m
      for j := 1 to n
          c_ij := 0
          for p := 1 to k
              c_ij := c_ij + a_ip b_pj
return C
```

$\mathbf{A} = [a_{ij}]$ is a $m \times k$ matrix
$\mathbf{B} = [b_{ij}]$ is a $k \times n$ matrix

Complexity of Matrix Multiplication

Multiplying two $n \times n$ matrices

procedure *matrixmult*(**A**, **B**: $n \times n$ matrices)

    for *i* := 1 to *n*

        for *j* := 1 to *n*

            c_{ij} := 0

            for *p* := 1 to *n*

                $c_{ij} := c_{ij} + a_{ip}\, b_{pj}$

return C

There are n^2 entries in the product matrix **C**.

For each entry the inner loop is executed n times (performing an addition and multiplication)

Hence, the inner loop is executed n^3 times and the complexity of matrix multiplication is $\Theta(n^3)$.

Summary

- Matrices
- Matrix Addition and Multiplication
- Complexity of Matrix Multiplication: $\Theta(n^3)$

Video 43: Understanding Complexity

- Complexity Classes
- Tractable Problems
- Untractable Problems

Complexity Classes

TABLE 1 Commonly Used Terminology for the Complexity of Algorithms.

Complexity	*Terminology*
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

Effect of Complexity

TABLE 2 The Computer Time Used by Algorithms.

Problem Size	*Bit Operations Used*					
n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	3×10^{-11} s	10^{-10} s	3×10^{-10} s	10^{-9} s	10^{-8} s	3×10^{-7} s
10^2	7×10^{-11} s	10^{-9} s	7×10^{-9} s	10^{-7} s	4×10^{11} yr	*
10^3	1.0×10^{-10} s	10^{-8} s	1×10^{-7} s	10^{-5} s	*	*
10^4	1.3×10^{-10} s	10^{-7} s	1×10^{-6} s	10^{-3} s	*	*
10^5	1.7×10^{-10} s	10^{-6} s	2×10^{-5} s	0.1 s	*	*
10^6	2×10^{-10} s	10^{-5} s	2×10^{-4} s	0.17 min	*	*

A bit operation is assumed to take 10^{-11} seconds, i.e., in one second we perform 100 billion bit operations. Times of more than 10^{100} years are indicated with an *.

Tractable Problems

Commonly, a problem is considered **tractable**, if there exists an algorithm and some d such that the algorithm can for input of size n produce the result with $O(n^d)$ operations.

This is the class **P** of **polynomial complexity.**

If such an algorithm does not exist, the problem is considered **intractable**.

- For large d and large inputs, it is often not so clear that the problem is really tractable in a practical sense
- **Example**: multiplying two very large matrices

The Class NP

- The class **NP** consists of those problems for which the correcteness of a proposed solution can be verified in polynomial time
- **NP** stands for **nondeterministic polynomial time**
  - Every problem in **NP** can be solved in exponential time
  - Open problem (since 50 years): **P** = **NP**?
  - General assumption: **P** ≠ **NP**
  - **The Clay Mathematics Institute has offered a prize of $1,000,000 for a solution.**

Example of an NP Problem

Knapsack Decision Problem: Given a set S of n items, each item with a value and a weight, find the subset of items that exceeds a minimal value while fitting a maximal weight.

- No polynomial algorithm is known
- Checking correctness of solution is easy
- Exponential algorithm: try out all possible subsets

NP-Complete Problems

NP-complete problems are problems in NP, such that if a polynomial algorithm is found for the problem, all other problems in NP can also be solved in polynomial time.

Examples

- Knapsack Decision Problem
- 3-SAT

3-SAT

Satisfiability of Propositional Statements: NP-complete

3-SAT is a more special problem:

- Deciding satisfiability for formulas in conjunctive normal form, where each clause has at most 3 variables is NP-complete

Example: $(p \vee p \vee q) \wedge (\neg p \vee \neg q \vee \neg q) \wedge (\neg p \vee q \vee q)$ is a formula from the set of 3-SAT formulas

Summary

Tractable Problem: There exists a polynomial time algorithm to solve this problem. These problems are said to belong to the **Class P**.

Intractable Problem: There does not exist a polynomial time algorithm to solve this problem.

Unsolvable Problem: There does not exist any algorithm to solve the problem, e.g., halting problem.

Class NP: Solution can be checked in polynomial time. But no polynomial time algorithm has been found for finding a solution to problems in this class.

NP Complete Class: If you find a polynomial time algorithm for one member of the class, it can be used to solve all the problems in the class.