UNIVERSITY OF MICHIGAN-DEARBORN

Chapter 4 : Algorithms

UNIVERSITY OF MICHIGAN-DEARBORN

Return assignment 01

Introduction

Algorithms

- *Definition:* An algorithm is a step-by-step method of solving some problem. "Algorithm" typically refers to a solution that can be executed by a computer.
- Algorithms typically have the following characteristics:
  - ***Input :*** The algorithm receives *input*.
  - ***Output:*** The algorithm produce *output*.
  - ***Precision:*** The steps are precisely stated.
  - ***Determinism :*** The intermediate results of each step of execution are unique and determined only by the inputs and the results of preceding steps.
  - ***Finiteness:*** The algorithm *terminates*; that is, it stops after finitely many instruction have been executed.
  - ***Correctness:*** The output produced by the algorithm is correct; that is, the algorithm correctly solves the problem.
  - ***Generality:*** The algorithm applies to a set of inputs.

Algorithms

- **<u>Example:</u>** Let consider the following algorithm that finds the maximum of three numbers a, b and c :

    1. Large = a.
    2. If b > large, then large = b.
    3. If c > large, then large = c.

➢ The idea of algorithm is to inspect the numbers one by one and copy the largest value seen into a value *large*. At the conclusion of the algorithm, *large* will then be equal to the largest of the three numbers.

Algorithms

DEARBORN

- We show how the preceding algorithm executes for some specific values of a, b and c. Such a simulation is called a **trace**.

- **Example 1:** First suppose that

$$a = 1, \qquad b = 5, \qquad c = 3.$$

  - ❖ At line 1, we set *large* to a (1).
  - ❖ At line 2, b > *large* (5 > 1) is true, so we set *large* to b (5).
  - ❖ At line 3, c > large (3 > 1) is false, so we do nothing.

At this point *large* is 5, the largest of a, b and c.

Algorithms

- **Example 2:** Let suppose that

  a = 6, b = 1, c = 9.

  - At line 1, we set *large* to a (6).
  - At line 2, b < *large* (1 < 6) is false, so we do nothing.
  - At line 3, c > large (9 > 6) is true, so we set *large* to c (9).

At this point *large* is 9, the largest of a, b and c.

Conclusion: The algorithm receives the values a, b and c as *input* and produces the value *large* as *output*.

Algorithms

- The steps of the algorithm are stated sufficiently precisely so that the algorithm could be written in a programming language and executed by a computer.
- Given values for the input, each intermediate step of an algorithm produces a unique result. For example, given the values

$$a = 1, \qquad b = 7, \qquad c = 3.$$

At line 2, *large* will be set to 5 regardless of who executes the algorithm.

- The algorithm terminates after finitely many steps (three steps) correctly answering the given question (find the largest of the three values input).
- The algorithm is general; it can find the largest value of *any* three numbers.

Algorithms

- Although ordinary language is sometimes adequate to specify an algorithm, most mathematics and computer scientists prefer **pseudocode** because of its precision, structure and universality.
- **Pseudocode** is so named because it resembles the actual code of computer language such as C++ and Java.

Algorithms

- **Example 1:** Finding the Maximum of three Numbers.

This algorithm finds the largest of the numbers a, b and c.

Input : a, b, c

Output : large (the largest of a, b and c)

```
max3(a, b, c) {
large = a
if (b > large) // if b is larger than large, update large
large = b
if (c > large) // if c is larger than large, update large
large = c
return large
}
```

➢ *Our algorithms consist of a title, a brief description of the algorithm, the input to and the output from the algorithm, and the function containing the instructions of the algorithm.*

Algorithms

- **Example 2:** Finding the Maximum Value in a Sequence

This algorithm finds the largest of the numbers s1, s2, …, sn.

Input : s, n

Output : large (the largest value of the sequence s)

```
max(s, n) {
    large = s1
    for i = 2 to n
        if (si > large)
            large = si
    return large
}
```

Exercises

Exercise 1:

Write an algorithm that returns the smallest value in the sequence s1, …, sn.

Exercise 2:

Write an algorithm that returns the sum of the sequence numbers s1, …, sn.

Exercise 3:

Write an algorithm that receives as input of the matrix of a relation R and tests whether R is reflexive.

Examples of Algorithms

Example of searching

- **Algorithm :** Text search

This algorithm searches for an occurrence of the pattern p in text t. It returns the smallest index i such that p occurs in t starting at index i. If p does not occur in t, it returns 0.

```
Input: p (indexed from 1 to m), m, t (indexed from 1 to n), n
Output: i

text_search(p, m, t, n) {
    for i = 1 to n – m + 1 {
        j = 1

        // i is the index in t of the first character of the substring
        // to compare with p, and j is the index p

        // the while loop compares ti … ti+m-1 and p1 … pm
        while (ti+m-1 == pj)
        j = j + 1
        if (j > m)
            return i
        }
    }
    return 0
}
```

Example of searching

Explanation for the example :

The variable i marks the index in t of the first character of the substring to compare with p. The algorithm first tries i = 1, then i = 2, and so on. Index n–m+1 is the last possible value for i since, at this point, the string tn-m+1 tn-m+2 … tm has length exactly m.

After the value of i is set, the while loop compares ti … ti+m-1 and p1 … pm. If the characters match,

Ti+j-1 == pj

j is incremented

j = j + 1

and the text characters are compared. If j is m + 1, all m characters have matched and we found p at index i in t. In this case, the algorithm returns i :

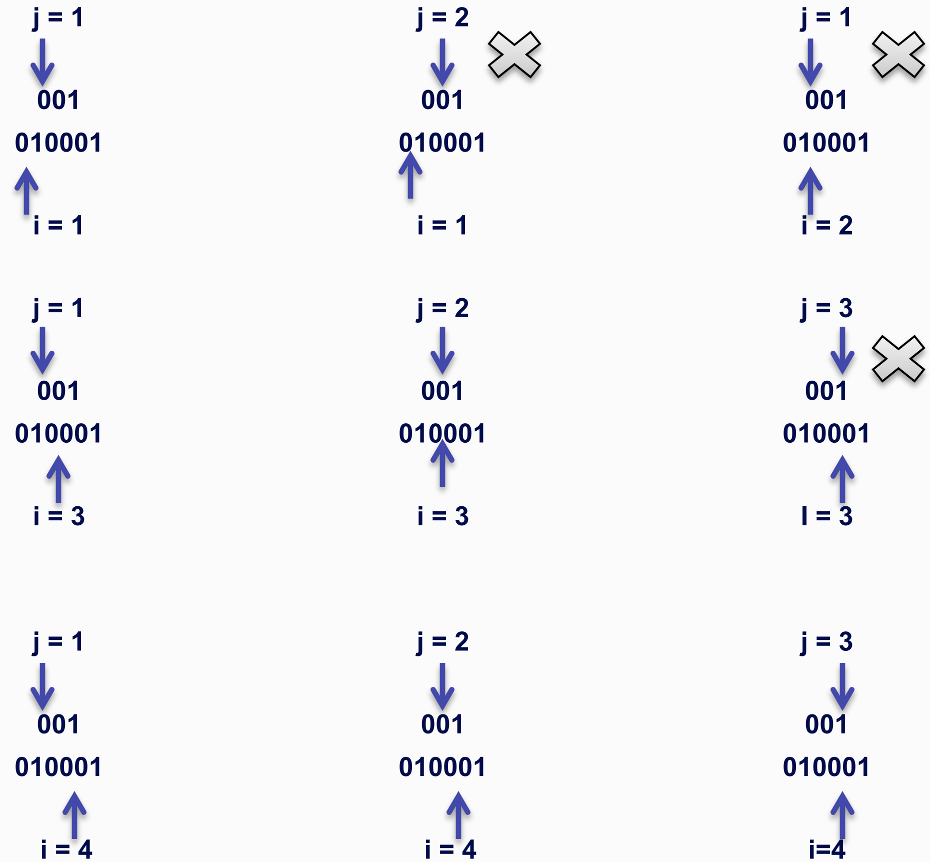
If (j > m)

return i

If the loop runs to completion, a match was never found; so the algorithm returns 0.

Example of searching

- **Example 2:** Show a trace of the Algorithm where we are searching for the pattern “001” in the text “010001”.



The cross marks a mismatch

Example of sorting

- To **sort** a sequence is to put in some order specified order. If we have a sequence of names, we might want the sequence sorted in non decreasing order according to dictionary order.
- For example, if the sequence is
    - *Jones, Johnson, Appel, Zamora, Chu*

After sorting the sequence in nondecreasing order, we would obtain

- *Appel, Chu, Johnson, Jones, Zamora*

- The fastest algorithms for sorting small sequence (less than 50 or so items) is **the insertion sort**.
- For example, we assume that the input to insertion sort is

$$s1, \ldots, sn$$

We suppose that the goal is to sort the data in nondecreasing order. At the *i*th iteration of the insertion sort, the first part of the sequence $s1, \ldots, si$

We can define the insertion sort as inserts $si+1$ in $s1, \ldots, si$

So we obtain **$s1, \ldots, si, si+1$** is sorted.

Example of sorting

- **For example**, suppose that i = 4 and s1, …, s4 is

8	13	20	27

If s5 = 16, after it is inserted, s1, …, s5 becomes

8	13	20	27	16

We can see that 20 and 27 are greater than 16. In this case, we must move one index to the right for 16.

we must insert 16 in

8	13	20	27

We first compare 16 with 27. Since 27 is greater than 16, 27 moves one index to the right:

8	13	20		27

We next compare 16 with 20. Since 20 is greater than 16, 20 moves one index to the right:

8	13		20	27

We next compare 16 with 23. Since 13 is less than or equal to 16, we insert 16 to the third index:

8	13	16	20	27

This subsequence is now sorted.

Example of sorting

- **Algorithm :** Insertion Sort

This algorithm sorts the sequence s1, …, sn in nondecreasing order.

```
Input: s, n
Output: s (sorted)

insertion_sort(s, n) {
    for i = 2 to n {
        val = si // save si so it can be inserted into the correct place
        j = i - 1
        // if val < sj, move sj right to make room for si
        while (j ≥ 1 ∧ val < sj) {
            sj+1 = sj
            j = j – 1
        }
        sj+1 = val // insert val
    }
}
```

Time and Space for Algorithms

- It is very important to know and be able to estimate *the time* (the number of steps) and *the space* (the number of variables, length of the sequences involved) required by algorithms.
- For example, if one algorithm takes n steps to solve a problem and another algorithm takes n^2 steps to solve the same problem, we would surely prefer the first algorithm, assuming that the space requirements are acceptable.
- If the input sequence is already sorted in nondecreasing order.

$$Val < sj$$

Will always be false and the body of the while loop will never be executed. We call this time the *best-case time*.

- If the input sequence is sorted in decreasing order.

$$Val < sj$$

Will always be true and the body of the while loop will execute the maximum number of times (the while loop will execute $i - 1$ times during the ith iteration of the for loop) . We call this time the *worst-case time*.

Randomized algorithms

- A *randomized algorithm* does not require that the intermediate results of each step of execution be uniquely defined and depend only on the inputs and results of the preceding steps.
- We shall assume that the existence of a function

rand (i, j)

Which returns a random integer between the integers i and j.

- As an example, we describe a randomized algorithm that shuffles a sequence of number, it inputs a sequence *A1, …, An* and moves the numbers to random positions. The algorithm first swaps *A1* and *Arand(1, n)*. At this point, the value of *A1* might be equal to any one of the original values in the sequence. Next, the algorithm swaps *A2* and *Arand(2, n)*. Now the value *A2* might be equal to any of the remaining values in the sequence. The algorithm continues in this manner until a swap *An-1* and *Arand(n-1, n)*.

➢ **Now the entire sequence is shuffled**.

Example of shuffle

- **Algorithm:** Shuffle

This algorithm shuffles the values in the sequence

A1, …, An

Input : A, n

Output : A (shuffled)

```
shuffle(A n) {
    for i = 1 to n - 1
        swap(Ai, Arand(i, n))
}
```

Example of shuffle

DEARBORN

- **<u>Example :</u>** Suppose that the sequence B

17	9	5	23	21

is input to shuffle. We first swap Bi and Bj, where i =1 and j = rand(1, 5). If j = 3, after swap we have

5	9	17	23	21
i		j		

Next, i = 2. If j = rand(2, 5) = 5, after the swap we have

5	21	17	23	9
	i			j

Next, i = 3. If j = rand(3, 5) = 3, the sequence does not change.

Next, i = 4. If j = rand(4, 5) = 5, after the swap we have

5	21	17	9	23
			i	j

Notice that the output depends on the random choices made by the random number generator.

Exercises

DEARBORN

Exercise 1: Trace the algorithm of the Text Search (slide 13) for the input

t = “balalaika” and p = “bala”

Exercise 2: Trace the algorithm of the Insertion Sort (slide 18) for the input

34 20 144 55

Exercise 3: Trace the algorithm of the Shuffle (slide 21) for the input

34 57 72 101 135

Assume that the values of *rand* are

rand(1, 5) = 5, *rand(2, 5) = 4*

rand(3, 5) = 3, *rand(4, 5) = 5*

Exercises

- **<u>Exercise 4:</u>** Write the algorithm that returns the index of the last occurrence of the value *key* in the sequence s1, …, sn. If *key* is not in the sequence, the algorithm returns the value 0. For example, if the sequence

12 11 12 23

And *key* is 12, the algorithm returns to 3.

- **<u>Exercise 5:</u>** The selection sort algorithm sorts the sequence s1, …, sn in nondecreasing order by first finding the smallest item, say si, and placing it first by swapping s1 and si. It then finds the smallest item in s2, …., sn, again say si, and places it second by swapping s2 and si. It continues until the sequence is sorted.

Write selection sort in pseudocode.

Analysis of Algorithms

Analysis of Algorithms

- *Analysis an algorithm* refers to the process of deriving estimates for the time and space needed to execute the algorithm.
- In this part of the chapter 4, we deal with the problem of estimating the time required to execute algorithms.
- An algorithm may be analyzed in terms of time or space that it takes.
  - **Time based analysis**: The efficiency of an algorithm may be analyzed in terms of execution time. (CPU TIME)
  - **Space based analysis**: The efficiency of an algorithm may be analyzed in terms of memory space it takes.

- *Algorithm Analysis* : → Time → non-recursive
  - Time → recursive
  - → Space (memory)

Analysis of Algorithms

- **Definition :** Let f and g be functions with domain {1, 2, 3, …}

We write $$f(n) = O(g(n))$$

We define *f*(n) is of **order at most** *g*(n) or *f*(n) is big oh of *g*(n) if there exists a positive constant C1 such that

$$|f(n)| \leq C1\, |g(n)|$$

For all but finitely many positive integers n.

We write $$f(n) = \Omega(g(n))$$

We define *f*(n) is of **order at least** *g*(n) or *f*(n) is omega of *g*(n) if there exists a positive constant C2 such that

$$|f(n)| \geq C2\, |g(n)|$$

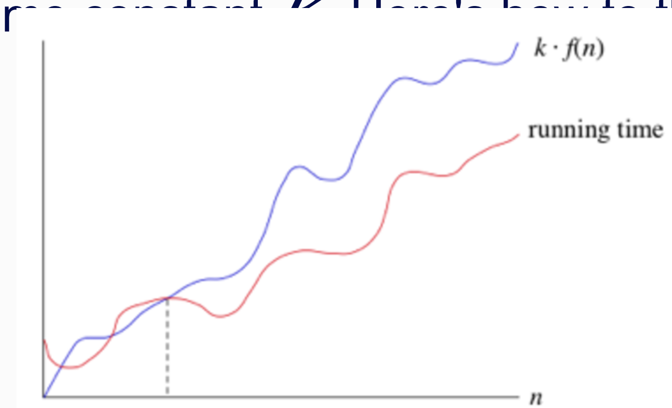
For all but finitely many positive integers n.

We write $$f(n) = \Theta(g(n))$$

We define *f*(n) is of **order** *g*(n) or *f*(n) is theta of *g*(n) if *f*(n) = $O(g(n))$ and *f*(n) = $\Omega(g(n))$.

Big-O

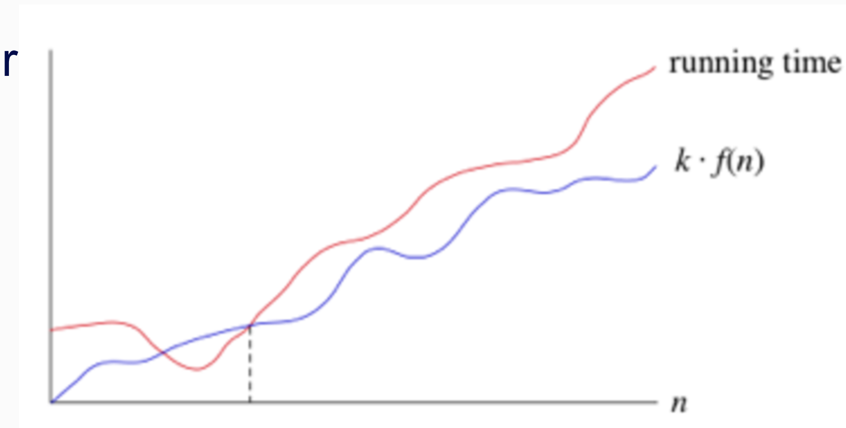
- If a running time is $O(f(n))$ then for **large enough** $\boldsymbol{n}$, the running time is at most $k.f(n)$ for some constant k. Here's how to think of a running time that is $O(f(n))$:



- We say that the running time is "big-O of $f(n)$" or just "O of $f(n)$ “. We use big-O notation for **asymptotic upper bounds**, since it bounds the growth of the running time from above for large enough input sizes.

Big- Ω

- If a running time is $\Omega(f(n))$, then for **large enough n**, the running time is at least $k.f(n)$ for



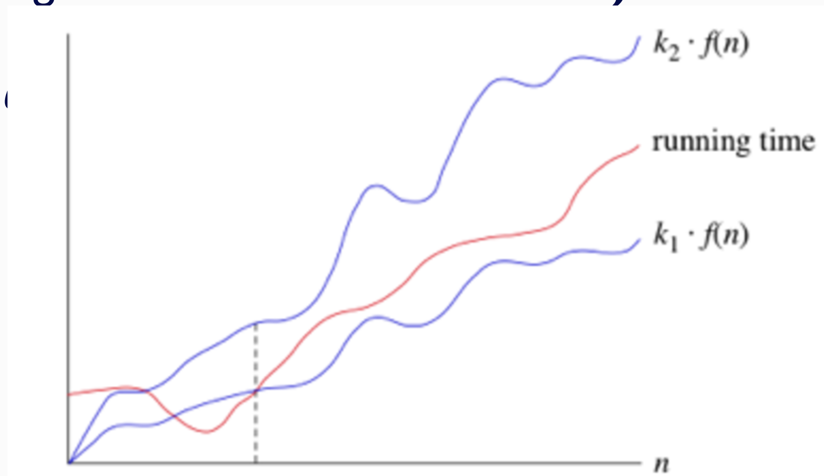
of a running time that is $\Omega(f(n))$:

- We say that the running time is "big-Ω of $f(n)$" We use big-Ω notation for **asymptotic lower bounds**, since it bounds the growth of the running time from below for large enough input sizes.

DEARBORN

Big- Θ

- When we say that a particular running time is $\Theta(n)$, we're saying that once n gets large enough, the running time is at least $k_1 \cdot n$, and at most $k_2 \cdot n$ for some constants k_1 an … $n)$:



- we have an asymptotically tight bound on the running time. "**Asymptotically**" because it matters for only large values of n. "**Tight bound**" because we've nailed

Analysis of Algorithms

- **Example:** Let consider

$$60\,n^2 + 5\,n + 1 \le 60\,n^2 + 5\,n^2 + n^2 = 66\,n^2 \qquad \text{for all } n \ge 1$$

We may take C1 = 66 to obtain

$$60\,n^2 + 5\,n + 1 = O(n^2)$$

Since $60\,n^2 + 5\,n + 1 \ge 60\,n^2$ for all $n \ge 1$

We may take C2 = 60 to obtain

$$60\,n^2 + 5\,n + 1 = \Omega(n^2)$$

Since $60\,n^2 + 5\,n + 1 = O(n^2)$ and $60\,n^2 + 5\,n + 1 = \Omega(n^2)$

$$\mathbf{60\,n^2 + 5\,n + 1 = \Theta(n^2)}$$

Analysis of Algorithms

- **Theorem :** Let consider

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + ,,, + a_1 n + a_0$$

be a polynomial in n of degree k, where each a_1 is nonnegative. Then

$$p(n) = \Theta(n^k)$$

- Proof: We first show that $p(n) = O(n^k)$. Let

$$C_1 = a_k + a_{k-1} + ,,, + a_1 + a_0$$

Then, for all n,

$$\begin{aligned} p(n) &= a_k n^k + a_{k-1} n^{k-1} + ,,, + a_1 n + a_0 \\ &\le a_k n^k + a_{k-1} n^k + ,,, + a_1 n^k + a_0 n^k \\ &= (a_k + a_{k-1} + ,,, + a_1 + a_0) n^k = C_1 n^k \end{aligned}$$

Therefore, $p(n) = O(n^k)$

Next, we show that $p(n) = \Omega(n^k)$. If for all n,

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + ,,, + a_1 n + a_0 \ \ge\ a_k n^k = C_2 n^k$$

Where $C_2 = a_k$. Therefore, $p(n) = \Omega(n^k)$

Since $p(n) = O(n^k)$ and $p(n) = \Omega(n^k)$, $p(n) = \Theta(n^k)$.

Analysis of Algorithms

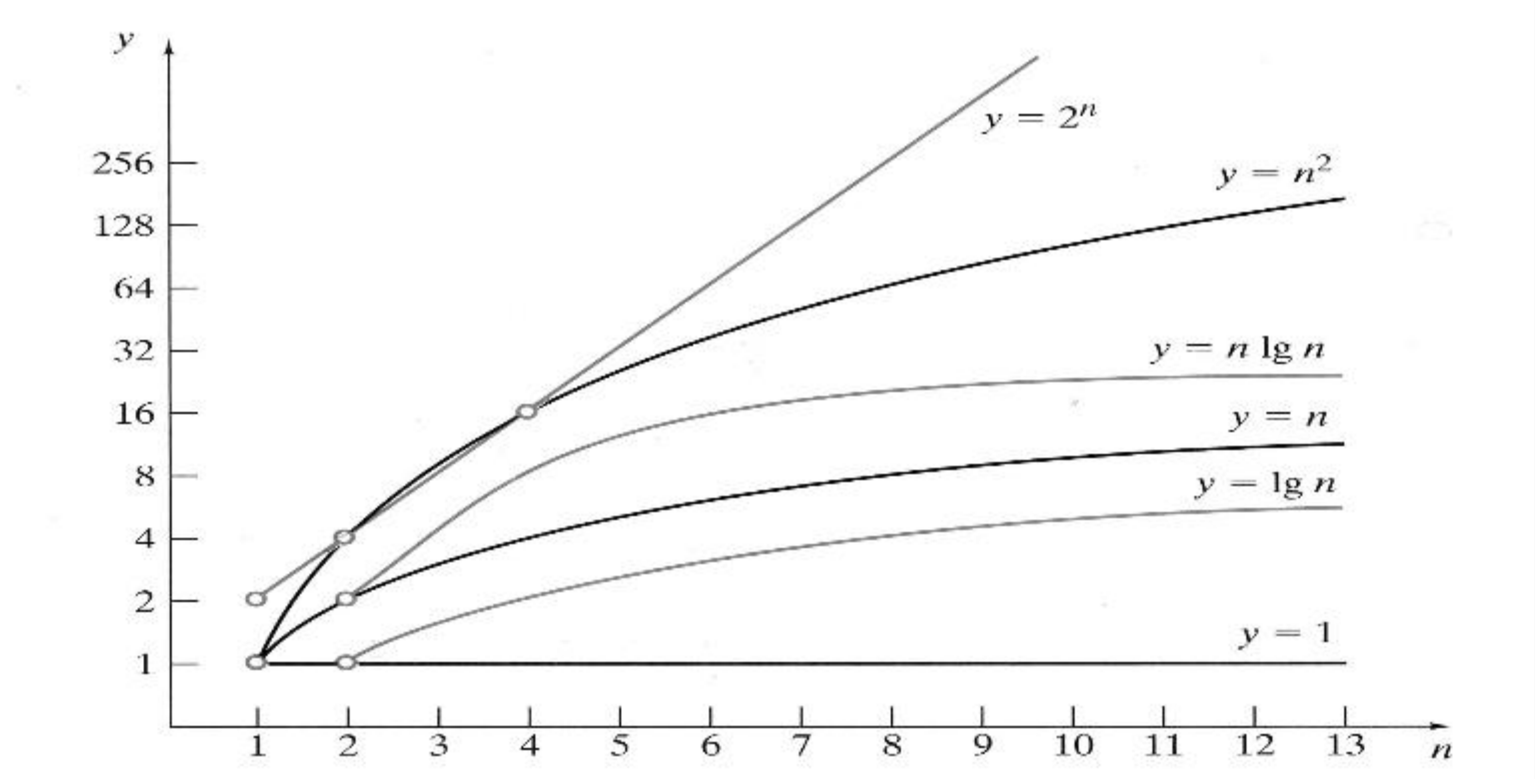
- **Example 1**: If we want to understand and apply asymptotic analysis, it is essential to have some idea of the rates of growth of some common functions.

The following table should help you understand the differences among the rates of grows of various functions:

n	log(n)	n*log(n)	n^2	n / log(n)
16	4	64	256	4.0
64	6	384	4096	10.7
256	8	2048	65536	32.0
1024	10	10240	1048576	102.4
1000000	20	19931568	1000000000000	50173.1
1000000000	30	29897352854	1000000000000000000	33447777.3

Analysis of Algorithms

- We let lg n denote $\log_2 n$ (the logarithm of n to the base 2). Since lg n < n for all n ≥ 1.



Graphs of some classical functions

Analysis of Algorithms

- **Example 2:** We let lg n denote $\log_2 n$ (the logarithm of n to the base 2). Since $\lg n < n$ for all $n \geq 1$.

$$2n + 3 \lg n < 2n + 3n = 5n \qquad \text{for all } n \geq 1$$

Thus,

$$2n + 3 \lg n = O(n)$$

Also,

$$2n + 3 \lg n \geq 2n \qquad \text{for all } n \geq 1$$

Thus,

$$2n + 3 \lg n = \Omega(n)$$

Therefore,

$$\mathbf{2n + 3 \lg n = \Theta(n)}$$

Analysis of Algorithms

- **Example 3:** if $a > 1$ and $b > 1$ (to ensure that $\log_b a > 0$), by the change-of-base formula for logarithms, we define :

$$\mathrm{Log}_b\, n = \log_b a \log_a n \qquad \text{for all } n \geq 1$$

Therefore,

$$\mathrm{Log}_b\, n \leq C\, \mathrm{Log}_a\, n \qquad \text{for all } n \geq 1$$

Where $C = \mathrm{Log}_b\, a$. Thus, $\mathrm{Log}_b\, n = O(\mathrm{Log}_a\, n)$.

Also,

$$\mathrm{Log}_b\, n \geq C\, \mathrm{Log}_a\, n \qquad \text{for all } n \geq 1$$

So $\mathrm{Log}_b\, n = \Omega(\mathrm{Log}_a\, n)$.

Since , $\mathrm{Log}_b\, n = O(\mathrm{Log}_a\, n)$ and $\mathrm{Log}_b\, n = \Omega(\mathrm{Log}_a\, n)$, we can conclude that

$$\mathbf{Log_b\, n = \Theta(Log_a\, n)}$$

Analysis of Algorithms

DEARBORN

- **Example 4:** If we replace each integer 1, 2, …, n by n in the sum 1 + 2 + … + n, the sum does not decrease and we have :

$$1 + 2 + \ldots + n \leq n + n + \ldots + n + n = n \times n = n^2 \qquad \text{for all } n \geq 1$$

It follows that $1 + 2 + \ldots + n = O(n^2)$

To obtain a lower bound, we might imitate the preceding argument and replace each integer 1, 2, …, n by 1 in the sum 1 + 2 + … + n to obtain

$$1 + 2 + \ldots + n \geq 1 + 1 + \ldots + 1 = n \qquad \text{for all } n \geq 1$$

In this case we conclude that $1 + 2 + \ldots + n = \Omega(n^2)$

and while the preceding expression is true, we cannot conclude a Θ-estimation for the sum 1 + 2 + … + n, since the upper bound n^2 and the lower bound n are not equal.

Analysis of Algorithms

DEARBORN

One way to get a sharper lower bound is to argue in the previous paragraph, but first throw away the first half of the terms. We obtain :

$$
\begin{aligned}
1 + 2 + \dots + n &\geq \lceil n/2 \rceil + \dots + (n-1) + n \\
&\geq \lceil n/2 \rceil + \dots + \lceil n/2 \rceil + \lceil n/2 \rceil \\
&= \lceil (n+1)/2 \rceil \lceil n/2 \rceil \geq (n/2)(n/2) = n^2/4
\end{aligned}
$$

For all $n \geq 1$. We can now conclude that

$$1 + 2 + \dots + n = \Omega(n^2)$$

Since , $1 + 2 + \dots + n = O(n^2)$ and $1 + 2 + \dots + n = \Omega(n^2)$, we can conclude that

$$\mathbf{1 + 2 + \dots + n = \Theta(n^2)}$$

Analysis of Algorithms

DEARBORN

- **Example 5:** If k is a positive integer, if we replace each integer 1, 2, …, n by n, we have :

$$1^k + 2^k + \ldots + n^k \le n^k + n^k + \ldots + n^k + n^k = n \times n^k = n^{k+1} \quad \text{for all } n \ge 1$$

For all $n \ge 1$; hence

$$1^k + 2^k + \ldots + n^k = O(n^{k+1})$$

As the previous example, we can also obtain a lower bound :

$$\begin{aligned} 1^k + 2^k + \ldots + n^k &\ge \lceil n/2 \rceil^k + \ldots + (n-1)^k + n^k \\ &\ge \lceil n/2 \rceil^k + \ldots + \lceil n/2 \rceil^k + \lceil n/2 \rceil^k \\ &= \lceil (n+1)/2 \rceil \lceil n/2 \rceil^k \ge (n/2)\ (n/2)^k = n^{k+1}/2^{k+1} \end{aligned}$$

For all $n \ge 1$, we can conclude that $1^k + 2^k + \ldots + n^k = \Omega(n^{k+1})$

Since , $1^k + 2^k + \ldots + n^k = O(n^{k+1})$ and $1^k + 2^k + \ldots + n^k = \Omega(n^{k+1})$, we can conclude that

$$\mathbf{1^k + 2^k + \ldots + n^k = \Theta(n^{k+1})}$$

Analysis of Algorithms

- **Example 6:** Using an argument similar to the previous examples, we show that :

$$\lg n! = \Theta(n \lg n)$$

By properties of logarithms, we have :

$$\lg n! = \lg n + \lg (n-1) + \ldots + \lg 2 + \lg 1$$

For all $n \geq 1$. Since lg is an increasing function.

$\lg n + \lg (n-1) + \ldots + \lg 2 + \lg 1 \leq \lg n + \lg n + \ldots + \lg n + \lg n = n \lg n$

For all $n \geq 1$. We can conclude that $\lg n! = O(n \lg n)$

For all $n \geq 4$, we have

$$\begin{aligned}
\lg n + \lg (n-1) + \ldots + \lg 2 + \lg 1 &\geq \lg n + \lg (n-1) + \ldots + \lg \lceil n/2 \rceil \\
&\geq \lg \lceil n/2 \rceil + \ldots + \lg \lceil n/2 \rceil \\
&= \lceil (n+1)/2 \rceil \lg \lceil n/2 \rceil \\
&\geq (n/2) \lg (n/2) \\
&= n \lg n/4
\end{aligned}$$

Since $(\lg n)/2 \geq 1$ for all $n \geq 4$, we can conclude that $\lg n! = \Omega(n \lg n)$

It follows that $\mathbf{\lg n! = \Theta(n \lg n)}$

Analysis of Algorithms

DEARBORN

- **Definition :**
  - ❖ If an algorithm requires t(n) units time to terminate in the best case for an input of size n and $t(n) = O(g(n))$
    - ➢ *We say that the best-case time required by the algorithm is of order at most g(n) or the best-case time required by the algorithm is O(g(n))*
  - ❖ If an algorithm requires t(n) units time to terminate in the worst case for an input of size n and $t(n) = O(g(n))$
    - ➢ *We say that the worst-case time required by the algorithm is of order at most g(n) or the worst-case time required by the algorithm is O(g(n))*
  - ❖ If an algorithm requires t(n) units time to terminate in the average case for an input of size n and $t(n) = O(g(n))$
    - ➢ *We say that the average-case time required by the algorithm is of order at most g(n) or the average-case time required by the algorithm is O(g(n))*

Analysis of Algorithms

- *By replacing O by Ω and "at most" by "at least", we obtain the definition of what it means for the best-case, worst-case, or average-case time of an algorithm to be of order at least g(n). If the best-case time required by an algorithm is O(g(n)) and Ω(g(n)), we say that the best-case time required by an algorithm is* ***Θ(g(n))*** *.*

Analysis of Algorithms

- **Example :** Find a theta notation in terms of n for the number of times the sequence x = x + 1 is executed

```
1.  for i = 1 to n
2.      for j = 1 to i
3.          x = x + 1
```

First, i is set to 1 and, as j runs from 1 to 1, line 3 is executed one time. Next, i is set to 2 as, as j runs from 1 to 2, line 3 is executed to times, and so on. Thus the total number of times, line 3 is executed is (See Example 4, slide 33-34)

$$1 + 2 + \ldots + n = \Theta(n^2)$$

Analysis of Algorithms

DEARBORN

- **Algorithm : Searching an unordered Sequence**

Given the sequence s1, …, sn and a value key, this algorithm returns the index of key, if key is not found, the algorithm returns to 0.

Input : s1, s2, …, sn, n, and key (the value to search)

Output : the index of key, or if key is not found, 0

```
Linear_search(s, n, key) {
      for i = 1 to n
         if (key ==s1)
             return i // successful search
      return 0 // successful search
}
```

- **Algorithm : Matrix Multiplication**

The algorithm computes the product C if the n × n matrices A and B directly from the definition of matrix multiplication.

Input : A, B, n

Output : C, the product of A and B

```
Matrix_product(A, B, n) {
    for i = 1 to n
        for j = 1 to n {
            C_ij = 0
            for k = 1 to n
                C_ij = C_ij + A_ik * B_kj
            }
    return C
}
```

Analysis of Algorithms

- To derive a theta notation, you must derive both big oh and omega notation.
- An other way to derive big oh, omega and theta estimations is to use known results :

Expression	Name	Estimate	Reference
$a_k n^k + a_{k-1} n^{k-1} + ,,, + a_1 n + a_0$	Polynomial	$\Theta(n^k)$	Slide 29
$1 + 2 + \ldots + n$	Arithmetic Sum (Case k = 1 for Next entry)	$\Theta(n^2)$	Slide 33-34
$1^k + 2^k + \ldots + n^k$	Sum of Powers	$\Theta(n^{k+1})$	Slide 35
lg n!	Log n Factorial	$\Theta(n \lg n)$	Slide 36

Exercises

DEARBORN

Exercise 1 : Find a theta notation for each expression

a) $6n + 1$

b) $3n^2 + 2n \lg n$

c) $2 + 4 + 6 + \ldots + 2n$

d) $2 + 4 + 8 + 16 + \ldots + 2^n$

Exercise 2 : Find a theta notation for the number of times the statement x = x +1 is executed

a)
```
for i = 1 to 2n
    x = x + 1
```

b)
```
for i = 1 to 2n
    for j = 1 to n
        x = x + 1
```

c)
```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            x = x + 1
```

DEARBORN

Exercises

Exercise 3 : Let consider that

$1 + 2 + \ldots + n = An^2 + Bn + C$ For all n and for some constant A, B and C.

① Assuming that this is true, plug in n = 1, 2, 3 to obtain three equations in the three unknowns A, B and C.

② Solve for A, B and C with the three equations obtained in the previous question.

③ Prove using the mathematical induction that the statement is true.

Exercise 4 : Let consider the formula

$$\frac{b^{n+1} - a^{n+1}}{b - a} = \sum_{i=0}^{n} a^i b^{n-1} \qquad 0 \le a \le b$$

Prove that

$$\frac{b^{n+1} - a^{n+1}}{b - a} < (n+1)b^n \qquad 0 \le a \le b$$

UNIVERSITY OF MICHIGAN-DEARBORN

Recursive Algorithms

Recursive Algorithms

- A *recursive algorithm* is an algorithm that contains a recursive function. Recursive is a powerful, elegant and natural way to solve a large class of problems.

Problem	Simplified Problem
5 !	5 × 4!
4!	4 × 3!
3!	3 × 2!
2!	2 × 1!
1!	1 × 0!
0!	None

Decomposing the factorial problem

Problem	Solution
0 !	1
1!	1 × 0! = 1
2!	2 × 1! = 2
3!	3 × 2! = 3 × 2 = 6
4!	4 × 3! = 4 × 6 = 24
5!	5 × 4! = 5 × 24 = 120

Combining subproblems of the factorial problem

Recursive Algorithms

DEARBORN

- We can write a recursive algorithm that computes factorials. The algorithm is a direct translation of the equation

$$n! = n \times (n-1)!$$

- **Algorithm :** Computing n Factorial

This recursive algorithm computes n!

Input : n, an integer greater than or equal to 0

Output : n!

```
factorial(n) {
    if (n ==0)
        return 1
    return n * factorial(n – 1)
}
```

Recursive Algorithms

- Explanation for the previous algorithm:

We show how the previous algorithm computes n! for several values of n. If $n = 0$, at line 3 the function correctly returns the value 1.

- ✧ If $n = 1$, we proceed to line 4 since $n \neq 0$. We use the function to compute 0!. We have just observed that the function computes 1 as the value of 0!. At line 4, the function correctly computes the value 1! :

$$n \times (n - 1)! = 1 \times 0! = 1 \times 1 = 1$$

- ✧ If $n = 2$, we proceed to line 4 since $n \neq 0$. We use the function to compute 1!. We have just observed that the function computes 1 as the value of 1!. At line 4, the function correctly computes the value 2! :

$$n \times (n - 1)! = 2 \times 1! = 2 \times 1 = 2$$

- ✧ If $n = 3$, we proceed to line 4 since $n \neq 0$. We use the function to compute 2!. We have just observed that the function computes 2 as the value of 2!. At line 4, the function correctly computes the value 3! :

$$n \times (n - 1)! = 3 \times 2! = 3 \times 2 = 6$$

Recursive Algorithms

DEARBORN

- We can use the *mathematical induction* to prove that the previous algorithm correctly return the value n! for any nonnegative integer n.

1. **Basic step (n =0)**

We have already observed that if $n = 0$, the algorithm of Computing n Factorial correctly return the value of $0! = 1$. We can say that the statement is true.

2. **Induction step**

If the algorithm of Computing n Factorial correctly return the value of $(n - 1)!$, $n > 0$. Now suppose that n is the input to the previous algorithm. Since $n \neq 0$, when we execute the function in the algorithm of Computing n Factorial we proceed to line 4. By the induction assumption, the function correctly computes the value $(n - 1)!$.

At line 4, the function correctly computes the value $(n - 1)! \times n = n!$.

Therefore, the algorithm of Computing n Factorial correctly return the value of n! for every integer $n \geq 0$.

Recursive Algorithms

DEARBORN

- **Definition :** The *Fibonacci sequence* $\{ f_n \}$ is defined by the equations

$$f_0 = 1$$
$$f_1 = 1$$
$$f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2} \qquad \text{for all } n \geq 3$$

The Fibonacci sequence begins

1, 1, 2, 3, 5, 8, 13, …

In mathematical terms, the sequence F_n of Fibonacci numbers is defined by the recursive relation $\mathbf{f_n = f_{n-1} + f_{n-2}}$

By definition, the first two numbers in the Fibonacci sequence are either 1 and 1, or 0 and 1, depending on the chosen starting point of the sequence, and each subsequent number is the sum of the previous two numbers.

Recursive Algorithms

- **Example :** Use the mathematical induction to show that

$$\sum_{k=1}^{n} f_k = f_{n+2} - 1$$ for all n ≥ 1

1. Basic step (n=1):

We must show that

$$\sum_{k=1}^{1} f_k = f_3 - 1$$

Since $\sum_{k=1}^{1} f_k = f_1 = 1$ and $f_3 - 1 = 2 - 1 = 1$, the equation is verified.

2. Inductive step : We assume the statement is true and we must prove case n+1

$$\sum_{k=1}^{n+1} f_k = f_{n+3} - 1$$

Recursive Algorithms

Now
$$\sum_{k=1}^{n+1} f_k = \sum_{k=1}^{n} f_k + f_{n+1}$$

$$= (f_{n+2} - 1) + f_{n+1}$$ by the induction assumption

$$= f_{n+2} + f_{n+1} - 1$$

$$= f_{n+3} - 1$$

The last equality is true because of the definition of the Fibonacci numbers:

$$f_n = f_{n-1} - f_{n-2}$$ for all n ≥ 3

Since the basic step and the inductive step have been verified, the given equation is true for all n ≥ 1.

Exercises

Exercise 1: Trace the algorithm of the computing n Factorial (slide 47) for $n = 4$.

Exercise 2: Let consider the formula

$$s_1 = 2, \qquad s_n = s_{n-1} + 2n \qquad \text{for all } n \geq 2$$

① Write the recursive algorithm that computes : $s_n = 2 + 4 + 6 + \ldots + 2n$.

② Proof using the mathematical induction that the recursive algorithm that computes s_n is correct.

Exercise 3: Write a recursive algorithm to find the maximum of a finite sequence of numbers. Give a proof using mathematical induction that your algorithm is correct.

Exercises

DEARBORN

Exercise 4: Use the mathematical induction to show that

$$f_n^2 = f_{n-1} f_{n+1} + (-1)^{n+1} \qquad \text{for all } n \geq 2$$

Exercise 5: Use the mathematical induction to show that

$$\sum_{k=1}^{n} f_k^2 = f_n f_{n+1} \qquad \text{for all } n \geq 1$$

Exercise 6: Let assume the formula for differentiating products:

$$\frac{d(fg)}{dx} = f\frac{dg}{dx} + g\frac{df}{dx} \qquad \text{for all } n \geq 1$$

Use mathematical induction to prove that

$$\frac{dx^n}{dx} = nx^{n-1} \qquad \text{for } n = 1, 2, \ldots.$$