INT102
Algorithmic Foundations And Problem Solving
Algorithm efficiency + Searching/Sorting

Dr Yushi Li

Department of Computer Science

西交利物浦大學
Xi'an Jiaotong-Liverpool University

Learning outcomes

- See some examples of **polynomial** time and **exponential** time algorithms
- Able to carry out simple **asymptotic analysis** of algorithms
- Able to apply searching/sorting algorithms and derive their time complexities

More polynomial time algorithms - searching …

Searching

Input: a sequence of n numbers $a_0, a_1, \ldots, a_{n-1}$; and a number X

Output: determine whether X is in the sequence or not

Algorithm (Linear search):

1. Starting from i=0, compare X with a_i one by one as long as $i < n$.
2. Stop and report "Found!" when $X = a_i$.
3. Repeat and report "Not Found!" when $i >= n$.

Linear Search

To find 7

- 12 34 2 9 7 5 six numbers
  7 number X
- 12 34 2 9 7 5
  7
- 12 34 2 9 7 5
  7
- 12 34 2 9 7 5
  7
- 12 34 2 9 7 5
  7 found!

Linear Search (2)

To find 10

- 12 34 2 9 7 5
  10
- 12 34 2 9 7 5
  10
- 12 34 2 9 7 5
  10
- 12 34 2 9 7 5
  10
- 12 34 2 9 7 5
  10
- 12 34 2 9 7 5
  10 not found!

Linear Search (3)

```
i = 0
while i < n do
begin
   if X == a[i] then
         report "Found!" and stop
   else
         i = i+1
end
report "Not Found!"
```

Time Complexity

Important operation of searching: ***comparison***

How many comparisons this algorithm requires?

```
i = 0
while i < n do
begin
  if X == a[i] then
    report "Found!" & stop
  else
    i = i+1
end
report "Not Found!"
```

Best case: X is the 1st no., 1 comparison, O(1)

Worst case: X is the last no. OR X is not found, n comparisons, O(n)

Improve Time Complexity?

If the numbers are pre-sorted, then we can improve the time complexity of searching by **binary search**.

Binary Search

more efficient way of searching when the sequence of numbers is **pre-sorted**

Input: a sequence of n **sorted** numbers $a_0, a_1, \ldots, a_{n-1}$ in ascending order and a number X

Idea of algorithm:

- compare X with number in the middle
- then focus on only the first half or the second half (depend on whether X is smaller or greater than the middle number)
- reduce the amount of numbers to be searched by half

Binary Search (2)

To find 24

3	7	11	12	15	19	24	33	41	55
				24					
					19	24	33	41	55
							24		
					19	24			
					24				
						24			
						24			found!

Binary Search (3)

To find 30

3 7 11 12 15 19 24 33 41 55 10 nos

30

X

19 24 33 41 55

30

19 24

30

24

30

not found!

Binary Search (4)

```
first=0, last=n-1
while (first <= last) do
begin
    mid = ⌊(first+last)/2⌋
    if (X == a[mid])
        report "Found!" & stop
    else
        if (X < a[mid])
            last = mid-1
        else
            first = mid+1
end
report "Not Found!"
```

⌊ ⌋ is the floor function, truncate the decimal part

Time Complexity

Best case:

X is the number in the middle
=> 1 comparison, $O(1)$-time

Worst case:

at most $\lceil \log_2 n \rceil + 1$ comparisons, $O(\log n)$-time

Why? Every comparison reduces the amount of numbers by at least half

E.g., 16 => 8 => 4 => 2 => 1

```
first=0, last=n-1
while (first <= last) do
begin
  mid = ⌊(first+last)/2⌋
  if (X == a[mid])
    report "Found!" & stop
  else
    if (X < a[mid])
      last = mid-1
    else
      first = mid+1
end
report "Not Found!"
```

Binary search vs Linear search

Time complexity of linear search is O(n)

Time complexity of binary search is O(log n)

Therefore, binary search is ***more efficient*** than linear search

Search for a pattern

We've seen how to search a number over a sequence of numbers

What about searching a pattern of characters over some text?

Example

text: N O B O D Y _ N O T I C E _ H I M

pattern: N O T

substring: N O B O D Y _ **N O T** I C E _ H I M

String Matching

Given a string of **n** characters called the text and a string of **m** characters ($m \leq n$) called the pattern.

We want to determine if the text contains a substring matching the pattern.

Example

text: N O B O D Y _ N O T I C E _ H I M

pattern: N O T

substring: N O B O D Y _ **N O T** I C E _ H I M

Example

T[ ]:	N	O	B	O	D	Y	_	N	O	T	I	C	E	_	H	I	M
P[ ]:	**N**	**O**	T														
		N	O	T													
			N	O	T												
				N	O	T											
					N	O	T										
						N	O	T									
							N	O	T								
								N	**O**	**T**							

bolded: match
underlined: not match
un-bolded: not considered

The algorithm

The algorithm scans over the text position by position.

For each position **i**, it checks whether the pattern P[0..m-1] appears in T[i..i+m-1]

If the pattern exists, then report found

Else continue with the next position **i+1**

If repeating until the end without success, report not found

Match pattern with T[i..i+m-1]

```
j = 0
while (j < m && P[j]==T[i+j]) do
    j = j + 1
if (j == m) then
    report "found!" & stop
```

T[i]	T[i+1]	T[i+2]	T[i+3]	…	T[i+m-1]
P[0]	P[1]	P[2]	P[3]	…	P[m-1]

Match for all positions

```
for i = 0 to n-m do
begin

   // check if P[0..m-1] match with T[i..i+m-1]

end
report "Not found!"
```

Match for all positions

```
for i = 0 to n-m do
begin
   j = 0
   while (j < m && P[j]==T[i+j]) do
         j = j + 1
   if (j == m) then
         report "found!" & stop
end
report "Not found!"
```

Time Complexity

How many comparisons this algorithm requires?

Best case:

pattern appears in the beginning of the text, O(**m**)-time

Worst case:

pattern appears at the end of the text OR pattern does not exist, O(**nm**)-time

```
for i = 0 to n-m do
begin
    j = 0
    while j < m & P[j]==T[i+j] do
        j = j + 1
    if j == m then
        report "found!" & stop
end
report "Not found!"
```

More polynomial time algorithms - sorting …

Sorting

Input: a sequence of n numbers $a_0, a_1, \ldots, a_{n-1}$

Output: arrange the n numbers into ascending order, i.e., from smallest to largest

Example: If the input contains 5 numbers 132, 56, 43, 200, 10, then the output should be 10, 43, 56, 132, 200

There are many sorting algorithms: insertion sort, selection sort, bubble sort, merge sort, quick sort

Selection Sort

- find minimum key from the input sequence
- delete it from input sequence
- append it to resulting sequence
- repeat until nothing left in input sequence

Selection Sort - Example

- sort (34, 10, 64, 51, 32, 21) in ascending order

Sorted part	Unsorted part	Swapped
	34 **10** 64 51 32 21	10, 34
10	34 64 51 32 **21**	21, 34
10 21	64 51 **32** 34	32, 64
10 21 32	51 64 **34**	51, 34
10 21 32 34	64 **51**	51, 64
10 21 32 34 51	**64**	--
10 21 32 34 51 64		

Selection Sort Algorithm

```
for i = 0 to n-2 do
begin

// find the index of the minimum number
// in the range a[i] to a[n-1]

  swap a[i] and a[min]
end
```

Selection Sort Algorithm

```
for i = 0 to n-2 do
begin
  min = i
  for j = i+1 to n-1 do
    if a[j] < a[min] then
      min = j
  swap a[i] and a[min]
end
```

how to swap two entries of an array?
using a temp variable?
not using a temp variable?

Algorithm Analysis

The algorithm consists of a nested for-loop.

For each iteration of the outer i-loop,
there is an inner j-loop.

```
for i = 0 to n-2 do
begin
  min = i
  for j = i+1 to n-1 do
    if a[j] < a[min] then
      min = j
  swap a[i] and a[min]
end
```

Total number of comparisons
$= (n-1) + (n-2) + \ldots + 1$
$= n(n-1)/2$

$O(n^2)$-time

i	# of comparisons in inner loop
0	n-1
1	n-2
...	...
n-2	1

Bubble Sort

starting from the last element, swap adjacent items if they are not in ascending order

when first item is reached, the first item is the smallest

repeat the above steps for the remaining items to find the second smallest item, and so on

Bubble Sort - Example

round							
	(34	10	64	51	32	21)	
	34	10	64	51	**32**	**21**	
1	34	10	64	**51**	**21**	32	
	34	10	**64**	**21**	51	32	
	34	**10**	**21**	64	51	32	←don't need to swap
	34	**10**	21	64	51	32	
	10	34	21	64	**51**	**32**	
2	***10***	34	21	**64**	**32**	51	
	10	34	**21**	**32**	64	51	←don't need to swap
	10	**34**	**21**	32	64	51	
	10	***21***	34	32	64	51	

underlined: being considered

italic: sorted

Bubble Sort - Example (2)

round							
	10	*21*	34	32	<u>**64**</u>	<u>**51**</u>	
3	*10*	*21*	34	<u>**32**</u>	<u>**51**</u>	64	←don't need to swap
	10	*21*	<u>**34**</u>	<u>**32**</u>	51	64	
	10	*21*	*32*	34	<u>**51**</u>	<u>**64**</u>	←don't need to swap
4	*10*	*21*	*32*	<u>**34**</u>	<u>**51**</u>	64	←don't need to swap
	10	*21*	*32*	*34*	<u>**51**</u>	<u>**64**</u>	←don't need to swap
5	*10*	*21*	*32*	*34*	*51*	64	

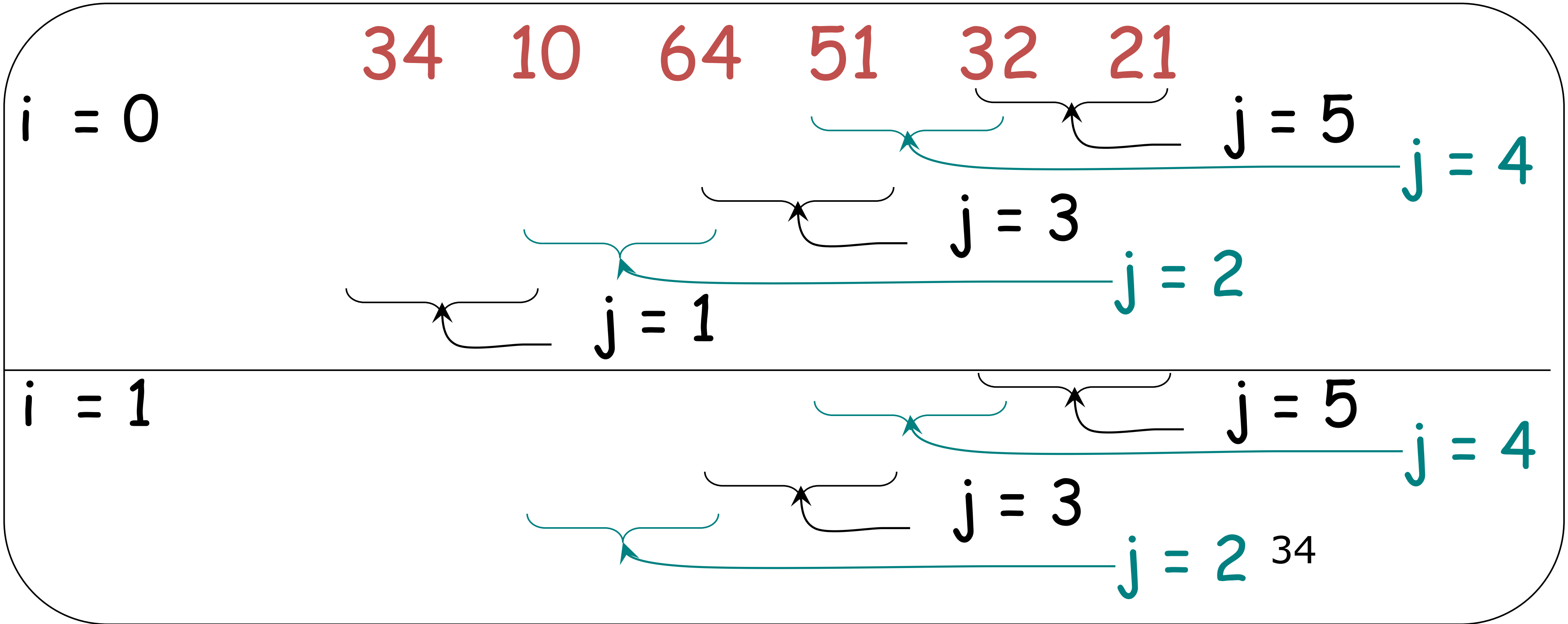
<u>**underlined**</u>: being considered
italic: sorted

Bubble Sort Algorithm

```
for i = 0 to n-2 do
  for j = n-1 downto i+1 do
    if (a[j] < a[j-1])
      swap a[j] & a[j-1]
```

the smallest will be moved to a[i]

start from a[n-1], check up to a[i+1]



Algorithm Analysis

The algorithm consists of a nested for-loop.

```
for i = 0 to n-2 do
  for j = n-1 downto i+1 do
    if (a[j] < a[j-1])
      swap a[j] & a[j-1]
```

Total number of comparisons
$= (n-1) + (n-2) + \ldots + 1$
$= n(n-1)/2$

$O(n^2)$-time

i	# of comparisons in inner loop
0	n-1
1	n-2
…	…
n-2	1

Sorting

Input: a sequence of n numbers $a_0, a_1, \ldots, a_{n-1}$

Output: arrange the n numbers into ascending order, i.e., from smallest to largest

We have learnt these sorting algorithms: selection sort, bubble sort

Next: insertion sort **(optional, self-study)**

Insertion Sort **(optional, self-study)**

look at elements one by one

build up sorted list by inserting the element at the correct location

optional

Example

- sort (34, 8, 64, 51, 32, 21) in ascending order

Sorted part	Unsorted part	int moved
	34 8 64 51 32 21	
34	**8** 64 51 32 21	-
8 34	**64** 51 32 21	34
8 34 64	**51** 32 21	-
8 34 51 64	**32** 21	64
8 32 34 51 64	**21**	34, 51, 64
8 21 32 34 51 64		32, 34, 51, 64

optional

Insertion Sort Algorithm

```
for i = 1 to n-1 do
begin
   key = a[i]
   pos = 0
   while (a[pos] < key) && (pos < i) do
      pos = pos + 1
   shift a[pos], …, a[i-1] to the right
   a[pos] = key
end
```

using linear search to find the correct position for key

i.e., move a[i-1] to a[i], a[i-2] to a[i-1], …, a[pos] to a[pos+1]

finally, place key (the original a[i]) in a[pos]

optional

Algorithm Analysis

```
for i = 1 to n-1 do
begin
  key = a[i]
  pos = 0
  while (a[pos] < key) && (pos < i) do
    pos = pos + 1
  shift a[pos], …, a[i-1] to the right
  a[pos] = key
end
```

Worst case input

– input is sorted in descending order

Then, for a[i]

– finding the position takes **i** comparisons

i	# of comparisons in the while loop
1	1
2	2
…	…
n-1	n-1

total number of comparisons

$= 1 + 2 + \ldots + n-1$

$= (n-1)n/2$

$O(n^2)$-time

optional

Selection, Bubble, Insertion Sort

All three algorithms have time complexity $O(n^2)$ in the worst case.

Are there any more efficient sorting algorithms? **YES**, we will learn them later.

What is the time complexity of the fastest comparison-based sorting algorithm? **O(n log n)**

Some exponential time algorithms – Traveling Salesman Problem, Knapsack Problem …

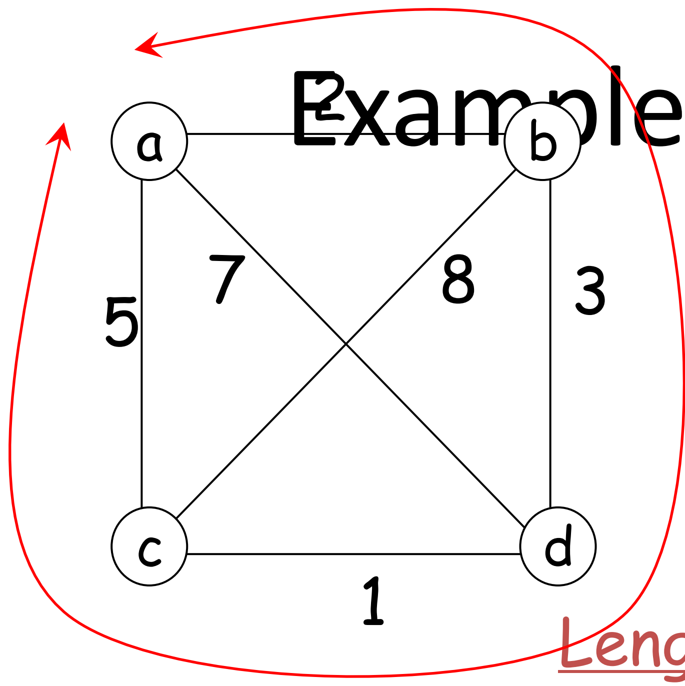
Traveling Salesman Problem

Input: There are n cities.

Output: Find the shortest tour from a particular city that visit each city exactly once before returning to the city where it started.

This is known as
Hamiltonian circuit

Example



To find a Hamiltonian circuit from a to a

Tour	Length
a -> b -> c -> d -> a	2 + 8 + 1 + 7 = 18
a -> b -> d -> c -> a	2 + 3 + 1 + 5 = **11**
a -> c -> b -> d -> a	5 + 8 + 3 + 7 = 23
a -> c -> d -> b -> a	5 + 1 + 3 + 2 = **11**
a -> d -> b -> c -> a	7 + 3 + 8 + 5 = 23
a -> d -> c -> b -> a	7 + 1 + 8 + 2 = 18

Idea and Analysis

A Hamiltonian circuit can be represented by a sequence of n+1 cities $v_1, v_2, \ldots, v_n, v_1$, where the first and the last are the same, and all the others are distinct.

Exhaustive search approach: Find all tours in this form, compute the tour length and find the shortest among them.

How many possible tours to consider?

$$\boxed{(n-1)! = (n-1)(n-2)\ldots1}$$

N.B.: (n-1)! is exponential in terms of n

Knapsack Problem

Input: Given n items with weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$, and a knapsack with capacity W.

Output: Find the most valuable subset of items that can fit into the knapsack.

Application: A transport plane is to deliver the most valuable set of items to a remote location without exceeding its capacity.

Example

capacity = 10

	item 1	item 2	item 3	item 4
w	w = 7	w = 3	w = 4	w = 5
v	v = 42	v = 12	v = 40	v = 25

knapsack

subset	total weight	total value	subset	total weight	total value
ϕ	0	0	{2,3}	7	52
{1}	7	42	{2,4}	8	37
{2}	3	12	**{3,4}**	**9**	**65**
{3}	4	40	{1,2,3}	14	N/A
{4}	5	25	{1,2,4}	15	N/A
{1,2}	10	54	{1,3,4}	16	N/A
{1,3}	11	N/A	{2,3,4}	12	N/A
{1,4}	12	N/A	{1,2,3,4}	19	N/A

Idea and Analysis

Exhaustive search approach: Try ***every subset*** of the set of n given items, compute total weight of each subset and compute total value of those subsets that do NOT exceed knapsack's capacity.

How many subsets to consider?

2^n, why?

Exercises (1)

Suppose you have forgotten a password with 5 characters. You only remember:

- the 5 characters are **all distinct**
- the 5 characters are **B, D, M, P, Y**

If you want to try **all possible combinations**, how many of them in total?

What if the 5 characters can be any of the 26 upper case alphabet?

Exercises (2)

Suppose the password also contains 2 digits, i.e., 7 characters in total

- all characters are **all distinct**
- the 5 alphabets are **B, D, M, P, Y**
- the digit is either **0 or 1**

How many combinations are there?

Exercises (3)

What if the password is in the form **`adaaada`?**

- **`a`** means alphabet, **`d`** means digit
- all characters are **all distinct**
- the 5 alphabets are **B, D, M, P, Y**
- the digit is either **0 or 1**

How many combinations are there?

Learning outcomes

- Able to carry out simple asymptotic analysis of algorithms
- Know some examples of polynomial time and exponential time algorithms
- Able to apply searching/sorting algorithms and derive their time complexities