COMP108
Algorithmic Foundations

Polynomial & Exponential Algorithms

Prudence Wong

http://www.csc.liv.ac.uk/~pwong/teaching/comp108/201617

Learning outcomes

- See some examples of polynomial time and exponential time algorithms
  - Able to apply searching/sorting algorithms and derive their time complexities

Sequential search: Time complexity

```
i = 1, found = false
while (i <= n && found == false) do
begin
  if X == a[i] then
    found = true
  else
    i = i+1
end
if found==true then
  report "Found!"
else report "Not Found!"
```

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

Worst case: X is last OR
X is not found,
n comparisons, **O(n)**

Binary search: Time complexity

Best case:

X is the number in the middle
$\Rightarrow$ 1 comparison, **O(1)**-time

Worst case:

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

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

Binary search vs Sequential search

Time complexity of sequential search is O(n)

Time complexity of binary search is O(log n)

Therefore, binary search is ***more efficient*** than sequential 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: A C G G A A T A A C T G G A A C G

pattern: A A C

substring: A C G G A A T <u>A A C</u> T G G <u>A A C</u> G

String Matching

Given a string of **n** characters called the text and a string of **x** characters (x≤n) called the pattern.

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

Example

text:	A C G G A A T A A C T G G A A C G
pattern:	A A C
substring:	A C G G A A T <u>A A C</u> T G G <u>A A C</u> G

Example

T[ ]:	A	C	G	G	A	A	T	A	A	C	T	G	G	A	A	C	G
P[ ]:	**A**	~~A~~	C														
		~~A~~	A	C													
			~~A~~	A	C												
				~~A~~	A	C											
					A	**A**	~~C~~										
						A	~~A~~	C									
							~~A~~	A	C								
								A	**A**	**C**							

bolded: match
~~**crossed**~~: 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[1..x] appears in T[i..(i+x-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+x-1)]

```
j = 1
while (j<=x && P[j]==T[i+j-1]) do
   j = j + 1
if (j==x+1) then
   found = true
```

2 cases when exit loop:

- j becomes x+1
  - ✓ all matches

OR

- P[j] ≠ T[i+j-1]
  - X unmatched

T[i]	T[i+1]	T[i+2]	T[i+3]	…	T[i+x-1]
↕	↕	↕	↕		↕
P[1]	P[2]	P[3]	P[4]	…	P[x]

Match for each position

```
i = 1, found = false
while (i <= n-x+1 && found == false) do
begin

  // check if P[1..x] match with T[i..(i+x-1)]


  i = i+1
end
if found == true
  report "Found!"
else report "Not found!"
```

Algorithm

```
i = 1, found = false
while (i <= n-x+1 && found == false) do
begin
  j = 1
  while (j<=x && P[j]==T[i+j-1]) do
     j = j + 1
  if (j==x+1) then
     found = true
   i = i+1
end
if found == true
   report "Found!"
else report "Not found!"
```

Time Complexity

How many comparisons this algorithm requires?

Best case:

pattern appears at the beginning of the text, O(**x**)-time

Worst case:

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

```
i = 1
while (i <= n-x+1 && found == false) do
begin
   j = 1
   while (j<=x && P[j]==T[i+j-1]) do
      j = j + 1
   if (j==x+1) then
      found = true
   i = i+1
end
```

More polynomial time algorithms - sorting …

Sorting

Input: a sequence of n numbers $a_1, a_2, \ldots, a_n$

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:
bubble sort, insertion sort, merge sort, quick sort, selection sort

Bubble Sort

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

when last item is reached, the last item is the largest

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

Bubble Sort - Example

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

underlined: being considered

italic: sorted

Bubble Sort - Example (2)

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

underlined: being considered

italic: sorted

Bubble Sort Algorithm

```
for i = n downto 2 do
  // move the largest number in a[1]…a[i]
  // to a[i] by swapping neighbouring
  // numbers if they are not in correct order

  for j = 1 to i-1 do
    // compare a[j] and a[j+1]
    // swap them if incorrect order

    if (a[j] > a[j+1])
      swap a[j] & a[j+1]
```

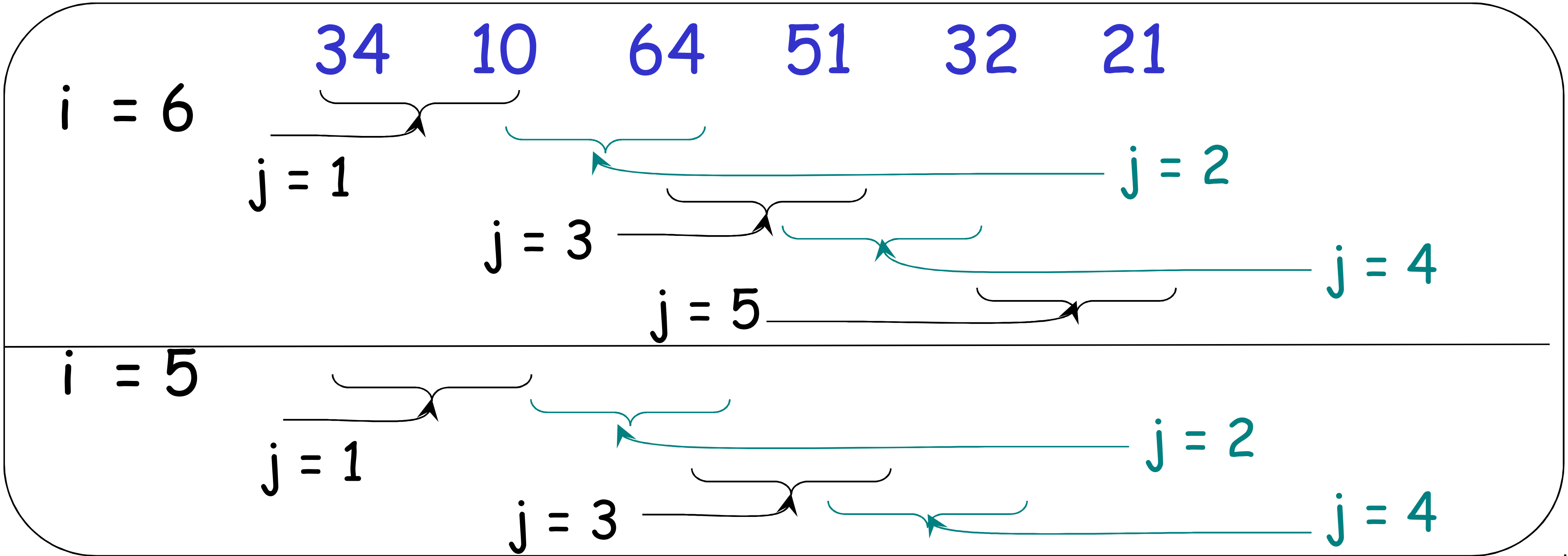
How to swap two variables?

Bubble Sort Algorithm

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

the largest will be moved to a[i]

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



Algorithm Analysis

The algorithm consists of a nested for-loop.

```
for i = n downto 2 do
  for j = 1 to 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(??)-time

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

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	To swap
	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 = 1 to n-1 do
begin

// find the index 'loc' of the minimum number
// in the range a[i] to a[n]

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

Selection Sort Algorithm

```
for i = 1 to n-1 do
begin // find index 'loc' in range a[i] to a[n]
   loc = i
   for j = i+1 to n do
      if a[j] < a[loc] then
        loc = j
   swap a[i] and a[loc]
end
```

How to swap two variables?

Algorithm Analysis

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

The algorithm consists of a nested for-loop.

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

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

O(??)-time

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

Insertion Sort **(self-study)**

look at elements one by one

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

Example

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

Sorted part	Unsorted part	int moved to right
	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

Insertion Sort Algorithm

```
for i = 2 to n do
begin
   key = a[i]
   loc = 1
   while (a[loc] < key) && (loc < i) do
      loc = loc + 1
   shift a[loc], …, a[i-1] to the right
   a[loc] = key
end
```

using sequential 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[loc] to a[loc+1]

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

Algorithm Analysis

Worst case input

- input is sorted in descending order

Then, for a[i]

- finding the position takes **i-1** comparisons

```
for i = 2 to n do
begin
    key = a[i]
    loc = 1
    while (a[loc] < key) && (loc < i) do
        loc = loc + 1
    shift a[loc], ..., a[i-1] to the right
    a[loc] = key
end
```

i	# of comparisons in the while loop
2	1
3	2
...	...
n	n-1

total number of comparisons
= 1 + 2 + ... + n-1
= (n-1)n/2

$O(n^2)$-time

Bubble, Selection, 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 ...

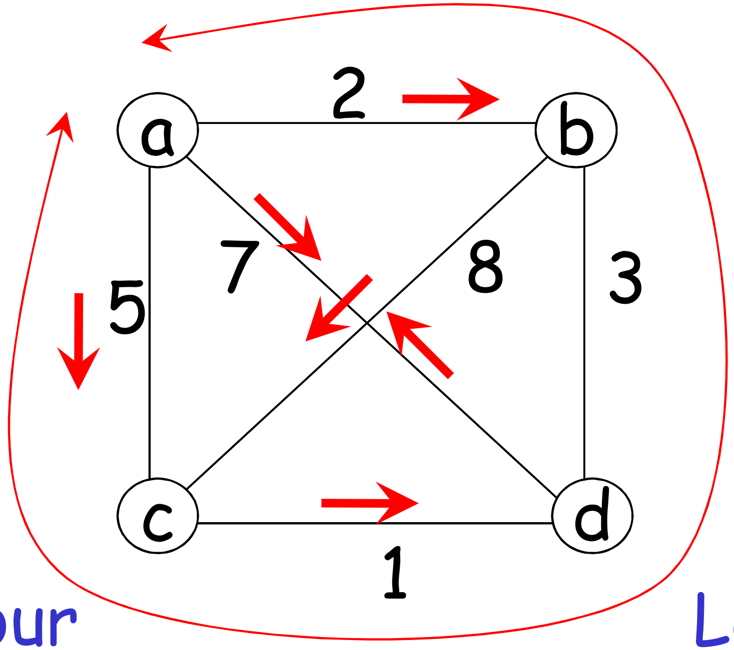
Traveling Salesman Problem

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)! grows faster than exponential in terms of n

[refer to notes on induction]

Knapsack Problem

What to take? so that…
1. Not too heavy
2. Most valuable

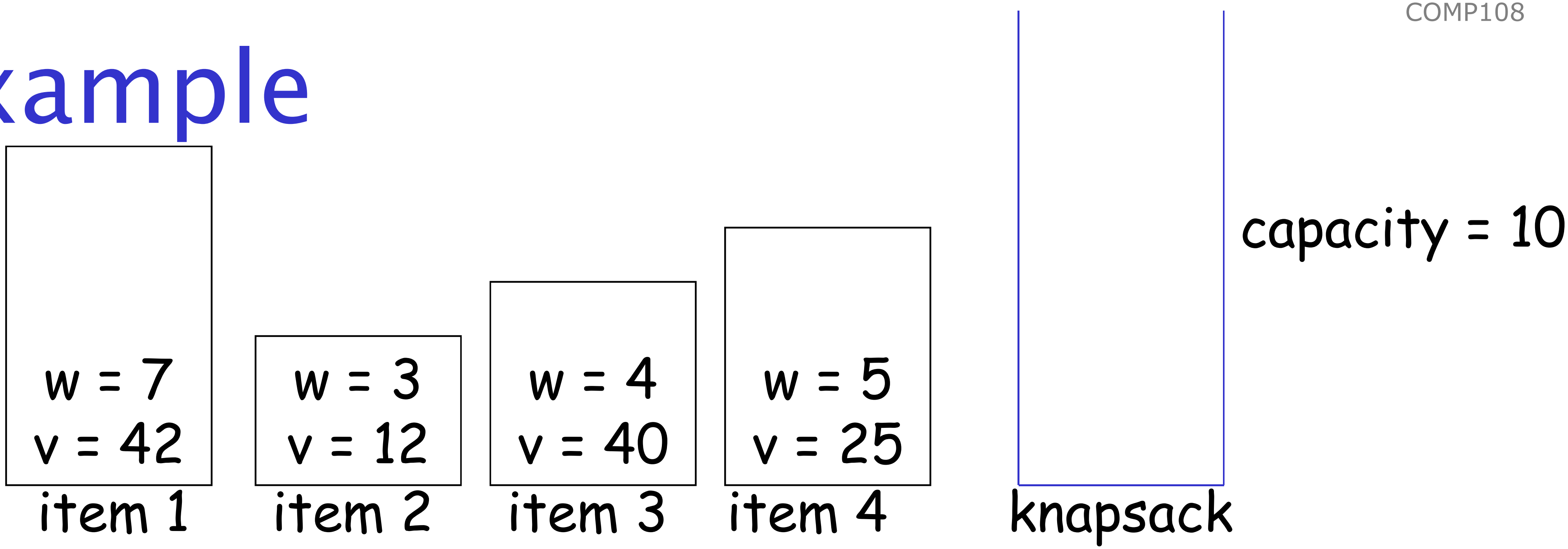
Knapsack Problem

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



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?

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 letters?

Exercises (2)

Suppose the password still has 5 characters

- the characters may **NOT** be **distinct**
- each character can be any of the 26 upper case letter

How many combinations are there?

Exercises (3)

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

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

How many combinations are there?