

# COMP110: Principles of Computing

## 7: Algorithm Strategies

# Recursion and induction



# A boolean identity

$$\neg(X_1 \vee X_2 \vee \cdots \vee X_n) = \neg X_1 \wedge \neg X_2 \wedge \cdots \wedge \neg X_n$$

# Proving the identity

# Proving the identity

- ▶ We can verify the formula for individual values of  $n$

# Proving the identity

- ▶ We can verify the formula for individual values of  $n$
- ▶ (e.g. by drawing a truth table with all  $2^n$  possible values of  $X_1, \dots, X_n$ )

# Proving the identity

- ▶ We can verify the formula for individual values of  $n$
- ▶ (e.g. by drawing a truth table with all  $2^n$  possible values of  $X_1, \dots, X_n$ )
- ▶ How do we **prove** it for **all**  $n$ ?

# Proving the identity

- ▶ We can verify the formula for individual values of  $n$
- ▶ (e.g. by drawing a truth table with all  $2^n$  possible values of  $X_1, \dots, X_n$ )
- ▶ How do we **prove** it for **all**  $n$ ?
- ▶ We can use **proof by induction**



# Case $n = 1$

$$\neg(X_1) = \neg X_1$$

# Case $n = 2$

$$\neg(X_1 \vee X_2) = \neg X_1 \wedge \neg X_2$$

# Case $n = 2$

$$\neg(X_1 \vee X_2) = \neg X_1 \wedge \neg X_2$$

Exercise Sheet ii, question 3(a)

Case  $n = k, k > 2$

# Case $n = k, k > 2$

- Suppose we have already proved the formula for all  $n < k$

# Case $n = k, k > 2$

- ▶ Suppose we have already proved the formula for all  $n < k$
- ▶ Use this to show that the formula holds for  $n = k$

# Case $n = k, k > 2$

- ▶ Suppose we have already proved the formula for all  $n < k$
- ▶ Use this to show that the formula holds for  $n = k$

$$\begin{aligned}\neg(X_1 \vee X_2 \vee \cdots \vee X_k) &= \neg(X_1 \vee (X_2 \vee \cdots \vee X_k)) \\ &= \neg X_1 \wedge \neg(X_2 \vee \cdots \vee X_k) \text{ (} n = 2 \text{ case)} \\ &= \neg X_1 \wedge (\neg X_2 \wedge \cdots \wedge \neg X_k) \text{ (} n = k - 1 \text{ case)}\end{aligned}$$

# Completing the proof



# Completing the proof

- ▶ We know:

# Completing the proof

- ▶ We know:
  - ▶ The formula works for  $n = 1$  and  $n = 2$

# Completing the proof

- ▶ We know:
  - ▶ The formula works for  $n = 1$  and  $n = 2$
  - ▶ If the formula works for  $n = k - 1$ , then it works for  $n = k$

# Completing the proof

- ▶ We know:
  - ▶ The formula works for  $n = 1$  and  $n = 2$
  - ▶ If the formula works for  $n = k - 1$ , then it works for  $n = k$
- ▶ The formula works for  $n = 1$  and  $n = 2$

# Completing the proof

- ▶ We know:
  - ▶ The formula works for  $n = 1$  and  $n = 2$
  - ▶ If the formula works for  $n = k - 1$ , then it works for  $n = k$
- ▶ The formula works for  $n = 1$  and  $n = 2$
- ▶ Therefore the formula works for  $n = 2 + 1 = 3$

# Completing the proof

- ▶ We know:
  - ▶ The formula works for  $n = 1$  and  $n = 2$
  - ▶ If the formula works for  $n = k - 1$ , then it works for  $n = k$
- ▶ The formula works for  $n = 1$  and  $n = 2$
- ▶ Therefore the formula works for  $n = 2 + 1 = 3$
- ▶ Therefore the formula works for  $n = 3 + 1 = 4$

# Completing the proof

- ▶ We know:
  - ▶ The formula works for  $n = 1$  and  $n = 2$
  - ▶ If the formula works for  $n = k - 1$ , then it works for  $n = k$
- ▶ The formula works for  $n = 1$  and  $n = 2$
- ▶ Therefore the formula works for  $n = 2 + 1 = 3$
- ▶ Therefore the formula works for  $n = 3 + 1 = 4$
- ▶ ...

# Completing the proof

- ▶ We know:
  - ▶ The formula works for  $n = 1$  and  $n = 2$
  - ▶ If the formula works for  $n = k - 1$ , then it works for  $n = k$
- ▶ The formula works for  $n = 1$  and  $n = 2$
- ▶ Therefore the formula works for  $n = 2 + 1 = 3$
- ▶ Therefore the formula works for  $n = 3 + 1 = 4$
- ▶ ...
- ▶ Therefore the formula works for all positive integers  $n$



# A formula for summation

$$\sum_{i=1}^n i = \frac{1}{2}n(n+1)$$

# A formula for summation

$$\sum_{i=1}^n i = \frac{1}{2}n(n+1)$$

# A formula for summation

$$\sum_{i=1}^n i = \frac{1}{2}n(n+1)$$

►  $n = 1: 1 = \frac{1}{2} \times 1 \times 2$

# A formula for summation

$$\sum_{i=1}^n i = \frac{1}{2}n(n+1)$$

- ▶  $n = 1: 1 = \frac{1}{2} \times 1 \times 2$
- ▶  $n = 2: 1 + 2 = \frac{1}{2} \times 2 \times 3 = 3$

# A formula for summation

$$\sum_{i=1}^n i = \frac{1}{2}n(n+1)$$

- ▶  $n = 1: 1 = \frac{1}{2} \times 1 \times 2$
- ▶  $n = 2: 1 + 2 = \frac{1}{2} \times 2 \times 3 = 3$
- ▶  $n = 3: 1 + 2 + 3 = \frac{1}{2} \times 3 \times 4 = 6$

# A formula for summation

$$\sum_{i=1}^n i = \frac{1}{2}n(n+1)$$

- ▶  $n = 1: 1 = \frac{1}{2} \times 1 \times 2$
- ▶  $n = 2: 1 + 2 = \frac{1}{2} \times 2 \times 3 = 3$
- ▶  $n = 3: 1 + 2 + 3 = \frac{1}{2} \times 3 \times 4 = 6$
- ▶ ...

# Proving the formula

# Proving the formula

- ▶ We can verify the formula for individual values of  $n$



# Proving the formula

- ▶ We can verify the formula for individual values of  $n$
- ▶ How do we **prove** it for **all**  $n$ ?

# Proving the formula

- ▶ We can verify the formula for individual values of  $n$
- ▶ How do we **prove** it for **all**  $n$ ?
- ▶ We can use **proof by induction**

# Proving the formula

# Proving the formula

## Base case

# Proving the formula

## Base case

►  $n = 1: 1 = \frac{1}{2} \times 1 \times 2$

# Proving the formula

## Base case

►  $n = 1: 1 = \frac{1}{2} \times 1 \times 2$

## Inductive assumption

# Proving the formula

## Base case

►  $n = 1: 1 = \frac{1}{2} \times 1 \times 2$

## Inductive assumption

►  $\sum_{i=1}^{k-1} i = \frac{1}{2}(k-1)k$

# Proving the formula

## Base case

►  $n = 1: 1 = \frac{1}{2} \times 1 \times 2$

## Inductive assumption

►  $\sum_{i=1}^{k-1} i = \frac{1}{2}(k-1)k$

## Therefore



# Proving the formula

## Base case

▶  $n = 1: 1 = \frac{1}{2} \times 1 \times 2$

## Inductive assumption

▶  $\sum_{i=1}^{k-1} i = \frac{1}{2}(k-1)k$

## Therefore

▶  $\sum_{i=1}^k i = \left( \sum_{i=1}^{k-1} i \right) + k$

# Proving the formula

## Base case

▶  $n = 1: 1 = \frac{1}{2} \times 1 \times 2$

## Inductive assumption

▶  $\sum_{i=1}^{k-1} i = \frac{1}{2}(k-1)k$

## Therefore

▶  $\sum_{i=1}^k i = \left( \sum_{i=1}^{k-1} i \right) + k$

▶  $= \frac{1}{2}(k-1)k + k$  (by inductive assumption)

# Proving the formula

## Base case

►  $n = 1: 1 = \frac{1}{2} \times 1 \times 2$

## Inductive assumption

►  $\sum_{i=1}^{k-1} i = \frac{1}{2}(k-1)k$

## Therefore

►  $\sum_{i=1}^k i = \left( \sum_{i=1}^{k-1} i \right) + k$

►  $= \frac{1}{2}(k-1)k + k$  (by inductive assumption)

►  $= \frac{1}{2}k^2 - \frac{1}{2}k + k$

# Proving the formula

## Base case

▶  $n = 1: 1 = \frac{1}{2} \times 1 \times 2$

## Inductive assumption

▶  $\sum_{i=1}^{k-1} i = \frac{1}{2}(k-1)k$

## Therefore

▶  $\sum_{i=1}^k i = \left( \sum_{i=1}^{k-1} i \right) + k$

▶  $= \frac{1}{2}(k-1)k + k$  (by inductive assumption)

▶  $= \frac{1}{2}k^2 - \frac{1}{2}k + k$

▶  $= \frac{1}{2}k^2 + \frac{1}{2}k$

# Proving the formula

## Base case

▶  $n = 1: 1 = \frac{1}{2} \times 1 \times 2$

## Inductive assumption

▶  $\sum_{i=1}^{k-1} i = \frac{1}{2}(k-1)k$

## Therefore

▶  $\sum_{i=1}^k i = \left( \sum_{i=1}^{k-1} i \right) + k$

▶  $= \frac{1}{2}(k-1)k + k$  (by inductive assumption)

▶  $= \frac{1}{2}k^2 - \frac{1}{2}k + k$

▶  $= \frac{1}{2}k^2 + \frac{1}{2}k$

▶  $= \frac{1}{2}k(k+1)$

# Proving the formula

## Base case

►  $n = 1: 1 = \frac{1}{2} \times 1 \times 2$

## Inductive assumption

►  $\sum_{i=1}^{k-1} i = \frac{1}{2}(k-1)k$

## Therefore

►  $\sum_{i=1}^k i = \left(\sum_{i=1}^{k-1} i\right) + k$

►  $= \frac{1}{2}(k-1)k + k$  (by inductive assumption)

►  $= \frac{1}{2}k^2 - \frac{1}{2}k + k$

►  $= \frac{1}{2}k^2 + \frac{1}{2}k$

►  $= \frac{1}{2}k(k+1)$

So **if** the formula works for  $n = k - 1$ , **then** it works for  $n = k$

# Completing the proof

# Completing the proof

- ▶ We know:



# Completing the proof

- ▶ We know:
  - ▶ The formula works for  $n = 1$

# Completing the proof

- ▶ We know:
  - ▶ The formula works for  $n = 1$
  - ▶ If the formula works for  $n = k - 1$ , then it works for  $n = k$

# Completing the proof

- ▶ We know:
  - ▶ The formula works for  $n = 1$
  - ▶ If the formula works for  $n = k - 1$ , then it works for  $n = k$
- ▶ The formula works for  $n = 1$

# Completing the proof

- ▶ We know:
  - ▶ The formula works for  $n = 1$
  - ▶ If the formula works for  $n = k - 1$ , then it works for  $n = k$
- ▶ The formula works for  $n = 1$
- ▶ Therefore the formula works for  $n = 1 + 1 = 2$

# Completing the proof

- ▶ We know:
  - ▶ The formula works for  $n = 1$
  - ▶ If the formula works for  $n = k - 1$ , then it works for  $n = k$
- ▶ The formula works for  $n = 1$
- ▶ Therefore the formula works for  $n = 1 + 1 = 2$
- ▶ Therefore the formula works for  $n = 2 + 1 = 3$

# Completing the proof

- ▶ We know:
  - ▶ The formula works for  $n = 1$
  - ▶ If the formula works for  $n = k - 1$ , then it works for  $n = k$
- ▶ The formula works for  $n = 1$
- ▶ Therefore the formula works for  $n = 1 + 1 = 2$
- ▶ Therefore the formula works for  $n = 2 + 1 = 3$
- ▶ Therefore the formula works for  $n = 3 + 1 = 4$

# Completing the proof

- ▶ We know:
  - ▶ The formula works for  $n = 1$
  - ▶ If the formula works for  $n = k - 1$ , then it works for  $n = k$
- ▶ The formula works for  $n = 1$
- ▶ Therefore the formula works for  $n = 1 + 1 = 2$
- ▶ Therefore the formula works for  $n = 2 + 1 = 3$
- ▶ Therefore the formula works for  $n = 3 + 1 = 4$
- ▶ ...

# Completing the proof

- ▶ We know:
  - ▶ The formula works for  $n = 1$
  - ▶ If the formula works for  $n = k - 1$ , then it works for  $n = k$
- ▶ The formula works for  $n = 1$
- ▶ Therefore the formula works for  $n = 1 + 1 = 2$
- ▶ Therefore the formula works for  $n = 2 + 1 = 3$
- ▶ Therefore the formula works for  $n = 3 + 1 = 4$
- ▶ ...
- ▶ Therefore the formula works for all positive integers  $n$



# Exercise

Prove

$$\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$$

# Thinking inductively

# Thinking inductively

- ▶ I want to prove something for all  $n$

# Thinking inductively

- ▶ I want to prove something for all  $n$
- ▶ Given  $k$ , if I had already proved  $n = k - 1$  then I could prove  $n = k$

# Thinking inductively

- ▶ I want to prove something for all  $n$
- ▶ Given  $k$ , if I had already proved  $n = k - 1$  then I could prove  $n = k$
- ▶ I can also prove  $n = 1$

# Thinking inductively

- ▶ I want to prove something for all  $n$
- ▶ Given  $k$ , if I had already proved  $n = k - 1$  then I could prove  $n = k$
- ▶ I can also prove  $n = 1$
- ▶ Therefore by induction I can prove the result for all  $n$

# Recursion

# Recursion

- ▶ A **recursive** function is a function that **calls itself**



# Recursion

- ▶ A **recursive** function is a function that **calls itself**

```
def factorial(n):  
    if n <= 1:  
        return 1  
    else:  
        return n * factorial(n-1)
```

# Thinking recursively

# Thinking recursively

- ▶ I want to solve a problem

# Thinking recursively

- ▶ I want to solve a problem
- ▶ If I already had a function to solve smaller instances of the problem, I could use it to write my function

# Thinking recursively

- ▶ I want to solve a problem
- ▶ If I already had a function to solve smaller instances of the problem, I could use it to write my function
- ▶ I can solve the smallest possible problem

# Thinking recursively

- ▶ I want to solve a problem
- ▶ If I already had a function to solve smaller instances of the problem, I could use it to write my function
- ▶ I can solve the smallest possible problem
- ▶ Therefore I can write a recursive function

# Exercise

- ▶ **Write** a pseudocode function to calculate the total size of all files in a directory and its subdirectories
- ▶ You may use the following functions in your pseudocode:
  - ▶ LISTDIR(directory): return a list of names of all files and folders in the given directory
  - ▶ GETSIZE(filename): return the size, in bytes, of the given file
  - ▶ ISDIR(name), ISFILE(name): determine whether the given name refers to a file or a directory

**procedure** CALCDIRSIZE(directory)

...

▶ return total size in bytes

**end procedure**

# Algorithm strategies





# The knapsack problem

# The knapsack problem

- ▶ There is a set  $X$  of **items**

# The knapsack problem

- ▶ There is a set  $X$  of **items**
- ▶ Each item  $x$  has a weight  $\text{weight}(x)$  and a value  $\text{value}(x)$

# The knapsack problem

- ▶ There is a set  $X$  of **items**
- ▶ Each item  $x$  has a weight  $\text{weight}(x)$  and a value  $\text{value}(x)$
- ▶ There is a maximum weight  $W$

# The knapsack problem

- ▶ There is a set  $X$  of **items**
- ▶ Each item  $x$  has a weight  $\text{weight}(x)$  and a value  $\text{value}(x)$
- ▶ There is a maximum weight  $W$
- ▶ What subset  $S \subseteq X$  maximises the total value, whilst not exceeding the maximum weight?

# The knapsack problem

- ▶ There is a set  $X$  of **items**
- ▶ Each item  $x$  has a weight  $\text{weight}(x)$  and a value  $\text{value}(x)$
- ▶ There is a maximum weight  $W$
- ▶ What subset  $S \subseteq X$  maximises the total value, whilst not exceeding the maximum weight?
- ▶ In other words: find  $S \subseteq X$  to maximise

$$\sum_{x \in S} \text{value}(x)$$

subject to

$$\sum_{x \in S} \text{weight}(x) \leq W$$

# Algorithm strategies

# Algorithm strategies

- ▶ Brute force



# Algorithm strategies

- ▶ Brute force
- ▶ Greedy

# Algorithm strategies

- ▶ Brute force
- ▶ Greedy
- ▶ Divide-and-conquer

# Algorithm strategies

- ▶ Brute force
- ▶ Greedy
- ▶ Divide-and-conquer
- ▶ Dynamic programming

# Brute force

# Brute force

- ▶ Try **every possible** solution and decide which is best

# Brute force

- ▶ Try **every possible** solution and decide which is best
- procedure** KNAPSACK( $X, W$ )

# Brute force

- Try **every possible** solution and decide which is best

**procedure** KNAPSACK( $X, W$ )

$S_{\text{best}} \leftarrow \{\}$

# Brute force

- Try **every possible** solution and decide which is best

**procedure** KNAPSACK( $X, W$ )

$S_{\text{best}} \leftarrow \{\}$

$V_{\text{best}} \leftarrow 0$



# Brute force

- Try **every possible** solution and decide which is best

**procedure** KNAPSACK( $X, W$ )

$S_{\text{best}} \leftarrow \{\}$

$V_{\text{best}} \leftarrow 0$

**for** every subset  $S \subseteq X$  **do**

# Brute force

- Try **every possible** solution and decide which is best

**procedure** KNAPSACK( $X, W$ )

$S_{\text{best}} \leftarrow \{\}$

$V_{\text{best}} \leftarrow 0$

**for** every subset  $S \subseteq X$  **do**

**if**  $\text{weight}(S) \leq W$  and  $\text{value}(S) > V_{\text{best}}$  **then**

# Brute force

- Try **every possible** solution and decide which is best

**procedure** KNAPSACK( $X, W$ )

$S_{\text{best}} \leftarrow \{\}$

$v_{\text{best}} \leftarrow 0$

**for** every subset  $S \subseteq X$  **do**

**if**  $\text{weight}(S) \leq W$  and  $\text{value}(S) > v_{\text{best}}$  **then**

$S_{\text{best}} \leftarrow S$

# Brute force

- Try **every possible** solution and decide which is best

**procedure** KNAPSACK( $X, W$ )

$S_{\text{best}} \leftarrow \{\}$

$V_{\text{best}} \leftarrow 0$

**for** every subset  $S \subseteq X$  **do**

**if**  $\text{weight}(S) \leq W$  and  $\text{value}(S) > V_{\text{best}}$  **then**

$S_{\text{best}} \leftarrow S$

$V_{\text{best}} \leftarrow \text{value}(S)$

# Brute force

- Try **every possible** solution and decide which is best

**procedure** KNAPSACK( $X, W$ )

$S_{\text{best}} \leftarrow \{\}$

$V_{\text{best}} \leftarrow 0$

**for** every subset  $S \subseteq X$  **do**

**if**  $\text{weight}(S) \leq W$  and  $\text{value}(S) > V_{\text{best}}$  **then**

$S_{\text{best}} \leftarrow S$

$V_{\text{best}} \leftarrow \text{value}(S)$

**end if**

# Brute force

- Try **every possible** solution and decide which is best

**procedure** KNAPSACK( $X, W$ )

$S_{\text{best}} \leftarrow \{\}$

$V_{\text{best}} \leftarrow 0$

**for** every subset  $S \subseteq X$  **do**

**if**  $\text{weight}(S) \leq W$  and  $\text{value}(S) > V_{\text{best}}$  **then**

$S_{\text{best}} \leftarrow S$

$V_{\text{best}} \leftarrow \text{value}(S)$

**end if**

**end for**

# Brute force

- Try **every possible** solution and decide which is best

**procedure** KNAPSACK( $X, W$ )

$S_{\text{best}} \leftarrow \{\}$

$V_{\text{best}} \leftarrow 0$

**for** every subset  $S \subseteq X$  **do**

**if**  $\text{weight}(S) \leq W$  and  $\text{value}(S) > V_{\text{best}}$  **then**

$S_{\text{best}} \leftarrow S$

$V_{\text{best}} \leftarrow \text{value}(S)$

**end if**

**end for**

**return**  $S_{\text{best}}$

# Brute force

- Try **every possible** solution and decide which is best

**procedure** KNAPSACK( $X, W$ )

$S_{\text{best}} \leftarrow \{\}$

$V_{\text{best}} \leftarrow 0$

**for** every subset  $S \subseteq X$  **do**

**if**  $\text{weight}(S) \leq W$  and  $\text{value}(S) > V_{\text{best}}$  **then**

$S_{\text{best}} \leftarrow S$

$V_{\text{best}} \leftarrow \text{value}(S)$

**end if**

**end for**

**return**  $S_{\text{best}}$

**end procedure**



# Socratic FALCOMPED

# Socratic FALCOMPED

- ▶ If  $X$  contains  $n$  elements, how many subsets of  $X$  are there?

# Socratic FALCOMPED

- ▶ If  $X$  contains  $n$  elements, how many subsets of  $X$  are there?
- ▶ Therefore what is the time complexity of the brute force algorithm?

# Socratic FALCOMPED

- ▶ If  $X$  contains  $n$  elements, how many subsets of  $X$  are there?
- ▶ Therefore what is the time complexity of the brute force algorithm?
- ▶ If we add one element to  $X$ , what happens to the running time of the algorithm?

# Greedy algorithm

# Greedy algorithm

- At each stage of building a solution, take the **best** available option

# Greedy algorithm

- At each stage of building a solution, take the **best** available option

**procedure** KNAPSACK( $X, W$ )

# Greedy algorithm

- At each stage of building a solution, take the **best** available option

**procedure** KNAPSACK( $X, W$ )

$S \leftarrow \{\}$



# Greedy algorithm

- At each stage of building a solution, take the **best** available option

**procedure** KNAPSACK( $X, W$ )

$S \leftarrow \{\}$

**for** each  $x \in X$ , in descending order of  $\text{value}(x)$  **do**

# Greedy algorithm

- At each stage of building a solution, take the **best** available option

**procedure** KNAPSACK( $X, W$ )

$S \leftarrow \{\}$

**for** each  $x \in X$ , in descending order of  $\text{value}(x)$  **do**  
    **if**  $\text{weight}(S) + \text{weight}(x) \leq W$  **then**

# Greedy algorithm

- At each stage of building a solution, take the **best** available option

**procedure** KNAPSACK( $X, W$ )

$S \leftarrow \{\}$

**for** each  $x \in X$ , in descending order of  $\text{value}(x)$  **do**  
    **if**  $\text{weight}(S) + \text{weight}(x) \leq W$  **then**  
        add  $x$  to  $S$

# Greedy algorithm

- At each stage of building a solution, take the **best** available option

**procedure** KNAPSACK( $X, W$ )

$S \leftarrow \{\}$

**for** each  $x \in X$ , in descending order of  $\text{value}(x)$  **do**  
    **if**  $\text{weight}(S) + \text{weight}(x) \leq W$  **then**  
        add  $x$  to  $S$   
    **end if**

# Greedy algorithm

- At each stage of building a solution, take the **best** available option

**procedure** KNAPSACK( $X, W$ )

$S \leftarrow \{\}$

**for** each  $x \in X$ , in descending order of  $\text{value}(x)$  **do**  
    **if**  $\text{weight}(S) + \text{weight}(x) \leq W$  **then**  
        add  $x$  to  $S$   
    **end if**  
**end for**

# Greedy algorithm

- At each stage of building a solution, take the **best** available option

**procedure** KNAPSACK( $X, W$ )

$S \leftarrow \{\}$

**for** each  $x \in X$ , in descending order of  $\text{value}(x)$  **do**

**if**  $\text{weight}(S) + \text{weight}(x) \leq W$  **then**

        add  $x$  to  $S$

**end if**

**end for**

**return**  $S$

**end procedure**

# Greedy algorithm

# Greedy algorithm

- ▶ Time complexity is dominated by sorting  $X$  by value



# Greedy algorithm

- ▶ Time complexity is dominated by sorting  $X$  by value
- ▶ The rest of the algorithm runs in linear time

# Greedy algorithm

- ▶ Time complexity is dominated by sorting  $X$  by value
- ▶ The rest of the algorithm runs in linear time
- ▶ In some problems an appropriately chosen greedy solution is **optimal**

# Greedy algorithm

- ▶ Time complexity is dominated by sorting  $X$  by value
- ▶ The rest of the algorithm runs in linear time
- ▶ In some problems an appropriately chosen greedy solution is **optimal**
  - ▶ A\* pathfinding

# Greedy algorithm

- ▶ Time complexity is dominated by sorting  $X$  by value
- ▶ The rest of the algorithm runs in linear time
- ▶ In some problems an appropriately chosen greedy solution is **optimal**
  - ▶ A\* pathfinding
  - ▶ Huffman coding

# Greedy algorithm

- ▶ Time complexity is dominated by sorting  $X$  by value
- ▶ The rest of the algorithm runs in linear time
- ▶ In some problems an appropriately chosen greedy solution is **optimal**
  - ▶ A\* pathfinding
  - ▶ Huffman coding
- ▶ **However** the greedy solution to the knapsack problem may not be optimal!

# Divide and conquer

# Divide and conquer

- ▶ Break the problem into smaller, easier to solve **subproblems**

# Divide and conquer

- ▶ Break the problem into smaller, easier to solve **subproblems**
- ▶ Requires that the solution to the original problem is composed of the solutions to the smaller problem



# Divide and conquer

- ▶ Break the problem into smaller, easier to solve **subproblems**
- ▶ Requires that the solution to the original problem is composed of the solutions to the smaller problem
- ▶ Example from last time: **binary search**

# Divide and conquer

- ▶ Break the problem into smaller, easier to solve **subproblems**
- ▶ Requires that the solution to the original problem is composed of the solutions to the smaller problem
- ▶ Example from last time: **binary search**
  - ▶ Problem: find an element in a list

# Divide and conquer

- ▶ Break the problem into smaller, easier to solve **subproblems**
- ▶ Requires that the solution to the original problem is composed of the solutions to the smaller problem
- ▶ Example from last time: **binary search**
  - ▶ Problem: find an element in a list
  - ▶ Subproblem: find the element in a list of half the size

# Divide and conquer for the knapsack problem

# Divide and conquer for the knapsack problem

- ▶ Consider an element  $x \in X$  with  $\text{weight}(x) \leq W$

# Divide and conquer for the knapsack problem

- ▶ Consider an element  $x \in X$  with  $\text{weight}(x) \leq W$
- ▶ Let  $X'$  be  $X$  with  $x$  removed

# Divide and conquer for the knapsack problem

- ▶ Consider an element  $x \in X$  with  $\text{weight}(x) \leq W$
- ▶ Let  $X'$  be  $X$  with  $x$  removed
- ▶ The solution to the knapsack problem either includes  $x$  or it doesn't

# Divide and conquer for the knapsack problem

- ▶ Consider an element  $x \in X$  with  $\text{weight}(x) \leq W$
- ▶ Let  $X'$  be  $X$  with  $x$  removed
- ▶ The solution to the knapsack problem either includes  $x$  or it doesn't
- ▶ The solution is **either**:



# Divide and conquer for the knapsack problem

- ▶ Consider an element  $x \in X$  with  $\text{weight}(x) \leq W$
- ▶ Let  $X'$  be  $X$  with  $x$  removed
- ▶ The solution to the knapsack problem either includes  $x$  or it doesn't
- ▶ The solution is **either**:
  - ▶ The solution to the knapsack problem on  $X'$  with maximum weight  $W$ , **or**

# Divide and conquer for the knapsack problem

- ▶ Consider an element  $x \in X$  with  $\text{weight}(x) \leq W$
- ▶ Let  $X'$  be  $X$  with  $x$  removed
- ▶ The solution to the knapsack problem either includes  $x$  or it doesn't
- ▶ The solution is **either**:
  - ▶ The solution to the knapsack problem on  $X'$  with maximum weight  $W$ , **or**
  - ▶ The solution to the knapsack problem on  $X'$  with maximum weight  $W - \text{weight}(x)$ , plus  $x$

# Divide and conquer for the knapsack problem

- ▶ Consider an element  $x \in X$  with  $\text{weight}(x) \leq W$
- ▶ Let  $X'$  be  $X$  with  $x$  removed
- ▶ The solution to the knapsack problem either includes  $x$  or it doesn't
- ▶ The solution is **either**:
  - ▶ The solution to the knapsack problem on  $X'$  with maximum weight  $W$ , **or**
  - ▶ The solution to the knapsack problem on  $X'$  with maximum weight  $W - \text{weight}(x)$ , plus  $x$
- ▶ ... whichever has the greater value

# Divide and conquer for the knapsack problem

- ▶ Consider an element  $x \in X$  with  $\text{weight}(x) \leq W$
- ▶ Let  $X'$  be  $X$  with  $x$  removed
- ▶ The solution to the knapsack problem either includes  $x$  or it doesn't
- ▶ The solution is **either**:
  - ▶ The solution to the knapsack problem on  $X'$  with maximum weight  $W$ , **or**
  - ▶ The solution to the knapsack problem on  $X'$  with maximum weight  $W - \text{weight}(x)$ , plus  $x$
- ▶ ... whichever has the greater value
- ▶ Base case: the solution to the knapsack problem on the empty set **is** the empty set

# Divide and conquer for the knapsack problem

# Divide and conquer for the knapsack problem

**procedure** KNAPSACK( $X, W, k$ )

# Divide and conquer for the knapsack problem

```
procedure KNAPSACK( $X, W, k$ )  
  if  $k < 0$  then
```

# Divide and conquer for the knapsack problem

```
procedure KNAPSACK( $X, W, k$ )  
  if  $k < 0$  then  
    return  $\{\}$ 
```



# Divide and conquer for the knapsack problem

```
procedure KNAPSACK( $X, W, k$ )  
  if  $k < 0$  then  
    return {}  
  end if
```

# Divide and conquer for the knapsack problem

```
procedure KNAPSACK( $X, W, k$ )  
  if  $k < 0$  then  
    return {}  
  end if  
   $S \leftarrow \text{KNAPSACK}(X, W, k - 1)$ 
```

# Divide and conquer for the knapsack problem

```
procedure KNAPSACK( $X, W, k$ )  
  if  $k < 0$  then  
    return  $\{\}$   
  end if  
   $S \leftarrow \text{KNAPSACK}(X, W, k - 1)$   
  if  $\text{weight}(x_k) \leq W$  then
```

# Divide and conquer for the knapsack problem

```
procedure KNAPSACK( $X, W, k$ )  
  if  $k < 0$  then  
    return  $\{\}$   
  end if  
   $S \leftarrow \text{KNAPSACK}(X, W, k - 1)$   
  if  $\text{weight}(x_k) \leq W$  then  
     $S' \leftarrow \text{KNAPSACK}(X, W - \text{weight}(x_k), k - 1) \cup \{x_k\}$ 
```

# Divide and conquer for the knapsack problem

**procedure** KNAPSACK( $X, W, k$ )

**if**  $k < 0$  **then**

**return**  $\{\}$

**end if**

$S \leftarrow \text{KNAPSACK}(X, W, k - 1)$

**if**  $\text{weight}(x_k) \leq W$  **then**

$S' \leftarrow \text{KNAPSACK}(X, W - \text{weight}(x_k), k - 1) \cup \{x_k\}$

**return** whichever of  $S, S'$  has the larger value

# Divide and conquer for the knapsack problem

```
procedure KNAPSACK( $X, W, k$ )  
  if  $k < 0$  then  
    return  $\{\}$   
  end if  
   $S \leftarrow \text{KNAPSACK}(X, W, k - 1)$   
  if  $\text{weight}(x_k) \leq W$  then  
     $S' \leftarrow \text{KNAPSACK}(X, W - \text{weight}(x_k), k - 1) \cup \{x_k\}$   
    return whichever of  $S, S'$  has the larger value  
  else
```

# Divide and conquer for the knapsack problem

```
procedure KNAPSACK( $X, W, k$ )  
  if  $k < 0$  then  
    return  $\{\}$   
  end if  
   $S \leftarrow \text{KNAPSACK}(X, W, k - 1)$   
  if  $\text{weight}(x_k) \leq W$  then  
     $S' \leftarrow \text{KNAPSACK}(X, W - \text{weight}(x_k), k - 1) \cup \{x_k\}$   
    return whichever of  $S, S'$  has the larger value  
  else  
    return  $S$ 
```

# Divide and conquer for the knapsack problem

```
procedure KNAPSACK( $X, W, k$ )  
  if  $k < 0$  then  
    return  $\{\}$   
  end if  
   $S \leftarrow \text{KNAPSACK}(X, W, k - 1)$   
  if  $\text{weight}(x_k) \leq W$  then  
     $S' \leftarrow \text{KNAPSACK}(X, W - \text{weight}(x_k), k - 1) \cup \{x_k\}$   
    return whichever of  $S, S'$  has the larger value  
  else  
    return  $S$   
  end if
```



# Divide and conquer for the knapsack problem

```
procedure KNAPSACK( $X, W, k$ )  
  if  $k < 0$  then  
    return  $\{\}$   
  end if  
   $S \leftarrow \text{KNAPSACK}(X, W, k - 1)$   
  if  $\text{weight}(x_k) \leq W$  then  
     $S' \leftarrow \text{KNAPSACK}(X, W - \text{weight}(x_k), k - 1) \cup \{x_k\}$   
    return whichever of  $S, S'$  has the larger value  
  else  
    return  $S$   
  end if  
end procedure
```

# Time complexity

# Time complexity

- ▶ Each call to KNAPSACK has, in the worst case, **two** recursive calls to KNAPSACK

# Time complexity

- ▶ Each call to KNAPSACK has, in the worst case, **two** recursive calls to KNAPSACK
- ▶ Number of calls is

$$\underbrace{1 + 2 + 4 + 8 + \dots + 2^i + \dots}_{n \text{ terms}}$$

# Time complexity

- ▶ Each call to KNAPSACK has, in the worst case, **two** recursive calls to KNAPSACK
- ▶ Number of calls is

$$\underbrace{1 + 2 + 4 + 8 + \dots + 2^i + \dots}_{n \text{ terms}}$$

- ▶ Thus the worst case time complexity is  $O(2^n)$  — still exponential!

# Time complexity

- ▶ Each call to KNAPSACK has, in the worst case, **two** recursive calls to KNAPSACK
- ▶ Number of calls is

$$\underbrace{1 + 2 + 4 + 8 + \dots + 2^i + \dots}_{n \text{ terms}}$$

- ▶ Thus the worst case time complexity is  $O(2^n)$  — still exponential!
- ▶ However in the **average** case many of the calls have only a single recursive call, so this is still more efficient than brute force

# Overlapping subproblems

# Overlapping subproblems

- ▶ Here we end up solving the **same subproblem multiple times**



# Overlapping subproblems

- ▶ Here we end up solving the **same subproblem multiple times**
- ▶ Can save time by **caching** (remembering) these sub-solutions

# Overlapping subproblems

- ▶ Here we end up solving the **same subproblem multiple times**
- ▶ Can save time by **caching** (remembering) these sub-solutions
- ▶ This is called **memoization**

# Overlapping subproblems

- ▶ Here we end up solving the **same subproblem multiple times**
- ▶ Can save time by **caching** (remembering) these sub-solutions
- ▶ This is called **memoization**
- ▶ One of several techniques in the category of **dynamic programming**

# Dynamic programming for the knapsack problem

# Dynamic programming for the knapsack problem

**procedure** KNAPSACK( $X, W, k$ )

# Dynamic programming for the knapsack problem

**procedure** KNAPSACK( $X, W, k$ )

**if** KNAPSACK( $X, W, k$ ) has already been computed **then**

# Dynamic programming for the knapsack problem

**procedure** KNAPSACK( $X, W, k$ )

**if** KNAPSACK( $X, W, k$ ) has already been computed **then**

**return** previously computed result

# Dynamic programming for the knapsack problem

**procedure** KNAPSACK( $X, W, k$ )

**if** KNAPSACK( $X, W, k$ ) has already been computed **then**

**return** previously computed result

**end if**



# Dynamic programming for the knapsack problem

```
procedure KNAPSACK( $X, W, k$ )  
  if KNAPSACK( $X, W, k$ ) has already been computed then  
    return previously computed result  
  end if  
  if  $k < 0$  then  
    cache and return  $\{\}$   
  end if  
   $S \leftarrow \text{KNAPSACK}(X, W, k - 1)$   
  if  $\text{weight}(x_k) \leq W$  then  
     $S' \leftarrow \text{KNAPSACK}(X, W - \text{weight}(x_k), k - 1) \cup \{x_k\}$   
    cache and return whichever of  $S, S'$  has the larger  
value  
  else  
    cache and return  $S$   
  end if  
end procedure
```

# Socratic FALCOMPED

# Socratic FALCOMPED

- ▶ What is the maximum possible number of entries in the table of intermediate results?

# Socratic FALCOMPED

- ▶ What is the maximum possible number of entries in the table of intermediate results?
- ▶ Therefore what is the time complexity of the dynamic programming algorithm?

# Summary of algorithm strategies

# Summary of algorithm strategies

- ▶ Brute force

# Summary of algorithm strategies

- ▶ Brute force
  - ▶ Good enough for small/simple problems

# Summary of algorithm strategies

- ▶ Brute force
  - ▶ Good enough for small/simple problems
- ▶ Greedy



# Summary of algorithm strategies

- ▶ Brute force
  - ▶ Good enough for small/simple problems
- ▶ Greedy
  - ▶ Efficient for certain problems, but doesn't always give optimal solutions

# Summary of algorithm strategies

- ▶ Brute force
  - ▶ Good enough for small/simple problems
- ▶ Greedy
  - ▶ Efficient for certain problems, but doesn't always give optimal solutions
- ▶ Divide-and-conquer

# Summary of algorithm strategies

- ▶ Brute force
  - ▶ Good enough for small/simple problems
- ▶ Greedy
  - ▶ Efficient for certain problems, but doesn't always give optimal solutions
- ▶ Divide-and-conquer
  - ▶ Good if the problem can be broken down into simpler subproblems

# Summary of algorithm strategies

- ▶ Brute force
  - ▶ Good enough for small/simple problems
- ▶ Greedy
  - ▶ Efficient for certain problems, but doesn't always give optimal solutions
- ▶ Divide-and-conquer
  - ▶ Good if the problem can be broken down into simpler subproblems
- ▶ Dynamic programming

# Summary of algorithm strategies

- ▶ Brute force
  - ▶ Good enough for small/simple problems
- ▶ Greedy
  - ▶ Efficient for certain problems, but doesn't always give optimal solutions
- ▶ Divide-and-conquer
  - ▶ Good if the problem can be broken down into simpler subproblems
- ▶ Dynamic programming
  - ▶ Makes divide-and-conquer more efficient if subproblems often reoccur

# Exercise Sheet iii

- ▶ Recursion and induction
- ▶ Due in class on **Tuesday 12th November** (next week)

# Worksheet C

