COMP110: Principles of Computing

10: Algorithm Strategies

Worksheets

- Worksheet 6: due **today**
- Worksheet 7: due **next Monday**

FALMOUTH UNIVERSITY

Recursion

Recursion

Recursion

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

Recursion

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

```
int factorial(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * factorial(n-1);
}
```

Recursion

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

```
int factorial(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * factorial(n-1);
}
```

- Recursive functions need a **base case** where they stop recursing, otherwise they will go **forever**

Recursion

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

```
int factorial(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * factorial(n-1);
}
```

- Recursive functions need a **base case** where they stop recursing, otherwise they will go **forever**
- (Or rather, until a **stack overflow**)

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

The call stack

The call stack

- Recall: nested function calls are handled using a **stack**

The call stack

- Recall: nested function calls are handled using a **stack**
- Recursive functions are no different

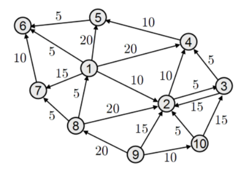
The call stack

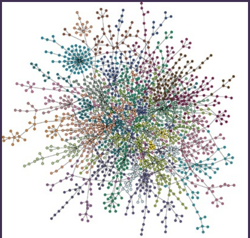
- Recall: nested function calls are handled using a **stack**
- Recursive functions are no different
- This means if a recursive function contains **local variables**, they are **independent** between instances of the function

Graphs and trees

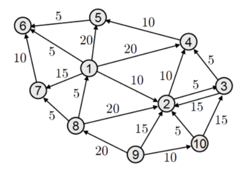
Graphs

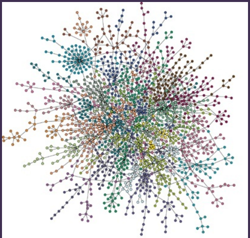
Graphs





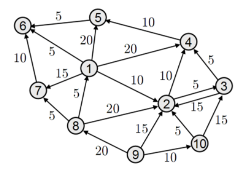
Graphs

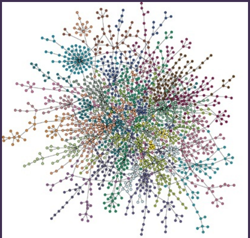




- A **graph** is defined by:

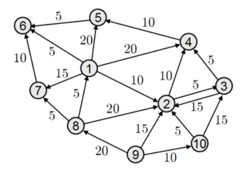
Graphs

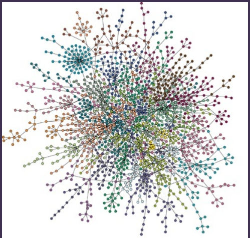




- A **graph** is defined by:
  - A collection of **nodes** or **vertices** (points)

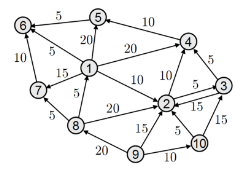
Graphs

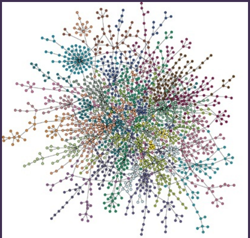




- A **graph** is defined by:
  - A collection of **nodes** or **vertices** (points)
  - A collection of **edges** or **arcs** (lines or arrows between points)

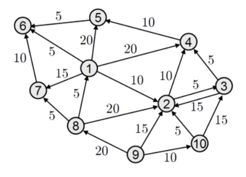
Graphs

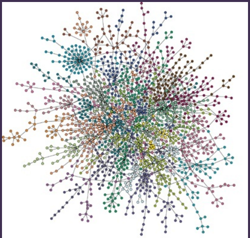




- A **graph** is defined by:
  - A collection of **nodes** or **vertices** (points)
  - A collection of **edges** or **arcs** (lines or arrows between points)
- Often used to model **networks** (e.g. social networks, transport networks, game levels, automata, ...)

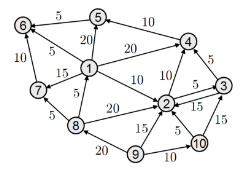
Graphs

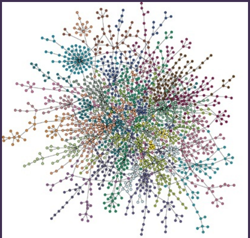




- A **graph** is defined by:
  - A collection of **nodes** or **vertices** (points)
  - A collection of **edges** or **arcs** (lines or arrows between points)
- Often used to model **networks** (e.g. social networks, transport networks, game levels, automata, ...)
- **Directed** graph: edges are arrows

Graphs





- A **graph** is defined by:
  - A collection of **nodes** or **vertices** (points)
  - A collection of **edges** or **arcs** (lines or arrows between points)
- Often used to model **networks** (e.g. social networks, transport networks, game levels, automata, ...)
- **Directed** graph: edges are arrows
- **Undirected** graph: edges are lines

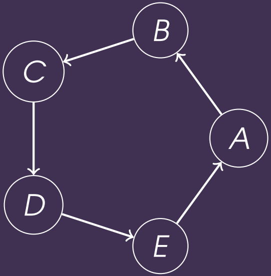
Drawing graphs

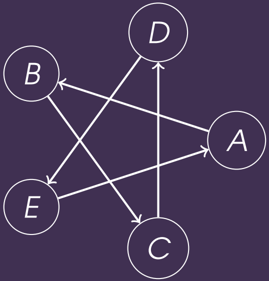
Drawing graphs

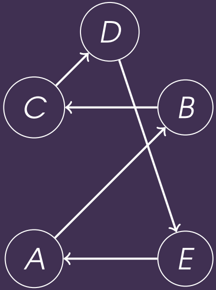
- A graph does not necessarily specify the physical **positions** of its nodes

Drawing graphs

- A graph does not necessarily specify the physical **positions** of its nodes
- E.g. these are technically the same graph:

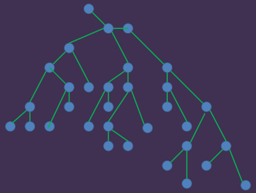


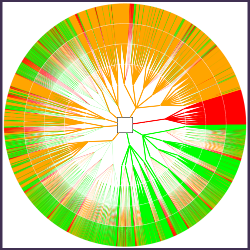




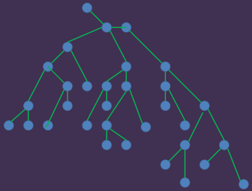
Trees

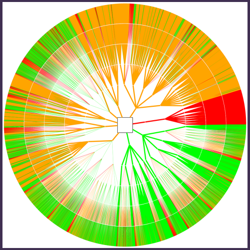
Trees





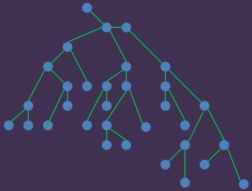
Trees

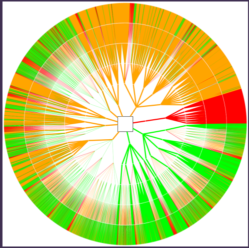




- A **tree** is a special type of directed graph where:

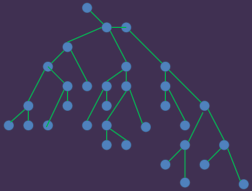
Trees

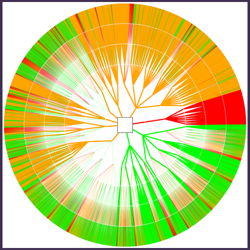




- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges

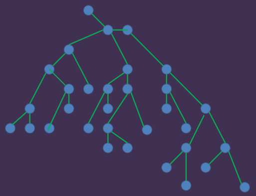
Trees

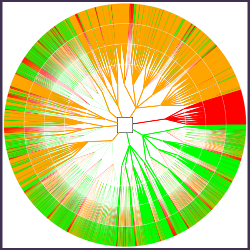




- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges
  - All other nodes have exactly 1 incoming edge

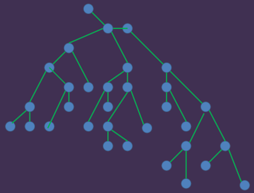
Trees

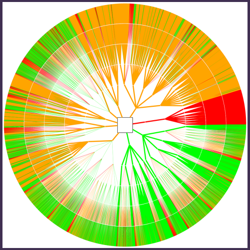




- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges
  - All other nodes have exactly 1 incoming edge
- Edges go from **parent** to **child**

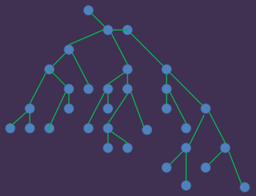
Trees

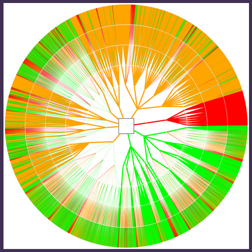




- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges
  - All other nodes have exactly 1 incoming edge
- Edges go from **parent** to **child**
  - All nodes except the root have exactly one parent

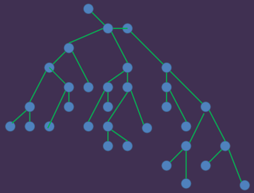
Trees

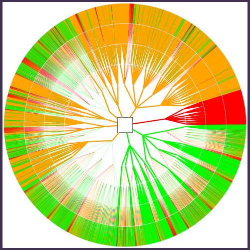




- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges
  - All other nodes have exactly 1 incoming edge
- Edges go from **parent** to **child**
  - All nodes except the root have exactly one parent
  - Nodes can have 0, 1 or many children

Trees





- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges
  - All other nodes have exactly 1 incoming edge
- Edges go from **parent** to **child**
  - All nodes except the root have exactly one parent
  - Nodes can have 0, 1 or many children
- Used to model **hierarchies** (e.g. file systems, object inheritance, scene graphs, state-action trees, behaviour trees, ...)

Tree traversal

Tree traversal

Tree traversal

- **Traversal**: visiting all the nodes of the tree

Tree traversal

- **Traversal**: visiting all the nodes of the tree
- Two main types

Tree traversal

- **Traversal**: visiting all the nodes of the tree
- Two main types
  - Depth first

Tree traversal

- **Traversal**: visiting all the nodes of the tree
- Two main types
  - Depth first
  - Breadth first

Tree traversal

Tree traversal

procedure DepthFirstSearch

Tree traversal

procedure DEPTHFIRSTSEARCH
    let S be a stack

Tree traversal

procedure DEPTHFIRSTSEARCH
    let S be a stack
    push root node onto S

Tree traversal

procedure DEPTHFIRSTSEARCH
    let S be a stack
    push root node onto S
    while S is not empty **do**

Tree traversal

procedure DepthFirstSearch
    let S be a stack
    push root node onto S
    while S is not empty **do**
        pop n from S

Tree traversal

procedure DEPTHFIRSTSEARCH
  let S be a stack
  push root node onto S
  while S is not empty **do**
    pop n from S
    print n

Tree traversal

procedure DepthFirstSearch
  let *S* be a stack
  push root node onto *S*
  while *S* is not empty **do**
    pop *n* from *S*
    print *n*
    push children of *n* onto *S*

Tree traversal

procedure DepthFirstSearch
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*
        print *n*
        push children of *n* onto *S*
    end while
end procedure

Tree traversal

procedure DEPTHFIRSTSEARCH
  let *S* be a stack
  push root node onto *S*
  while *S* is not empty **do**
    pop *n* from *S*
    print *n*
    push children of *n* onto *S*
  end while
end procedure

procedure BREADTHFIRSTSEARCH

Tree traversal

```
procedure DepthFirstSearch
    let S be a stack
    push root node onto S
    while S is not empty do
        pop n from S
        print n
        push children of n onto S
    end while
end procedure

procedure BreadthFirstSearch
    let Q be a queue
```

Tree traversal

procedure DEPTHFIRSTSEARCH
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*
        print *n*
        push children of *n* onto *S*
    end while
end procedure

procedure BREADTHFIRSTSEARCH
    let *Q* be a queue
    enqueue root node into *Q*

Tree traversal

procedure DepthFirstSearch
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*
        print *n*
        push children of *n* onto *S*
    end while
end procedure

procedure BreadthFirstSearch
    let *Q* be a queue
    enqueue root node into *Q*
    while *Q* is not empty **do**

Tree traversal

```
procedure DepthFirstSearch
    let S be a stack
    push root node onto S
    while S is not empty do
        pop n from S
        print n
        push children of n onto S
    end while
end procedure

procedure BreadthFirstSearch
    let Q be a queue
    enqueue root node into Q
    while Q is not empty do
        dequeue n from Q
```

Tree traversal

procedure DEPTHFIRSTSEARCH
    let S be a stack
    push root node onto S
    while S is not empty **do**
        pop n from S
        print n
        push children of n onto S
    end while
end procedure

procedure BREADTHFIRSTSEARCH
    let Q be a queue
    enqueue root node into Q
    while Q is not empty **do**
        dequeue n from Q
        print n

Tree traversal

procedure DepthFirstSearch
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*
        print *n*
        push children of *n* onto *S*
    end while
end procedure

procedure BreadthFirstSearch
    let *Q* be a queue
    enqueue root node into *Q*
    while *Q* is not empty **do**
        dequeue *n* from *Q*
        print *n*
        enqueue children of *n* into *Q*

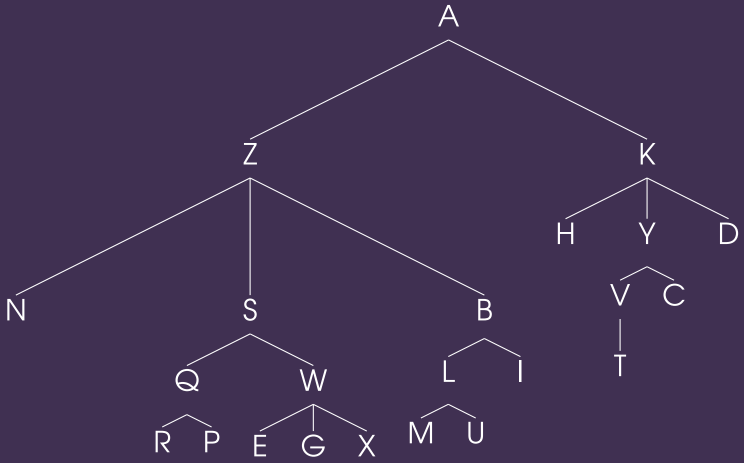
Tree traversal

procedure DepthFirstSearch
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*
        print *n*
        push children of *n* onto *S*
    end while
end procedure

procedure BreadthFirstSearch
    let *Q* be a queue
    enqueue root node into *Q*
    while *Q* is not empty **do**
        dequeue *n* from *Q*
        print *n*
        enqueue children of *n* into *Q*
    end while
end procedure

Tree traversal example

Socrative `FALCOMPED`



Recursive depth first search

Recursive depth first search

procedure DepthFirstSearch(n)

Recursive depth first search

```
procedure DepthFirstSearch(n)
    print n
```

Recursive depth first search

procedure DEPTHFIRSTSEARCH(n)
    print n
    for each child c of n **do**

Recursive depth first search

```
procedure DepthFirstSearch(n)
    print n
    for each child c of n do
        DepthFirstSearch(c)
```

Recursive depth first search

```
procedure DepthFirstSearch(n)
    print n
    for each child c of n do
        DepthFirstSearch(c)
    end for
end procedure
```

Recursive depth first search

```
procedure DepthFirstSearch(n)
    print n
    for each child c of n do
        DepthFirstSearch(c)
    end for
end procedure
```

- Compare to the pseudocode on the previous slide. Where is the stack?

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 $\mathrm{weight}(x)$ and a value $\mathrm{value}(x)$

The knapsack problem

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

The knapsack problem

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

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

subject to

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

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 weight$(S) \leq W$ and 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 weight$(S) \leq W$ and value$(S) > v_{\text{best}}$ **then**
      $S_{\text{best}} \leftarrow S$
      $v_{\text{best}} \leftarrow$ 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 weight$(S) \leq W$ and value$(S) > v_{\text{best}}$ **then**
            $S_{\text{best}} \leftarrow S$
            $v_{\text{best}} \leftarrow$ 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 weight(S) $\leq W$ and value(S) $> v_{\text{best}}$ **then**
            $S_{\text{best}} \leftarrow S$
            $v_{\text{best}} \leftarrow$ 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 weight$(S) \leq W$ and value$(S) > v_{\text{best}}$ **then**
      $S_{\text{best}} \leftarrow S$
      $v_{\text{best}} \leftarrow$ value(S)
    end if
  end for
  return S_{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 weight$(S) \leq W$ and value$(S) > v_{\text{best}}$ **then**
            $S_{\text{best}} \leftarrow S$
            $v_{\text{best}} \leftarrow$ value(S)
        end if
    end for
    return S_{best}
end procedure

Socrative `FALCOMPED`

Socrative `FALCOMPED`

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

Socrative `FALCOMPED`

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

Socrative `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 $\mathrm{value}(x)$ **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 $\mathsf{value}(x)$ **do**
        if $\mathsf{weight}(S) + \mathsf{weight}(x) \leq W$ **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 $\mathrm{value}(x)$ **do**
    if $\mathrm{weight}(S) + \mathrm{weight}(x) \leq W$ **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 $\mathrm{value}(x)$ **do**
    if $\mathrm{weight}(S) + \mathrm{weight}(x) \leq W$ **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 $\mathrm{value}(x)$ **do**
        if $\mathrm{weight}(S) + \mathrm{weight}(x) \leq W$ **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 **subprogrems**

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 $\mathrm{weight}(x) \leq W$

Divide and conquer for the knapsack problem

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

Divide and conquer for the knapsack problem

- Consider an element $x \in X$ with $\mathtt{weight}(x) \leq W$
- 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 $\mathtt{weight}(x) \leq W$
- 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 $\mathtt{weight}(x) \leq W$
- 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 $\mathtt{weight}(x) \leq W$
- 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 $W - \mathtt{weight}(x)$, 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 $\mathrm{weight}(x) \leq W$
- 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 $W - \mathrm{weight}(x)$, 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
    $S \leftarrow \textsc{Knapsack}(X, W, k-1)$
    if $\mathrm{weight}(x_k) \leq W$ **then**

Divide and conquer for the knapsack problem

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

Divide and conquer for the knapsack problem

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

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

Time complexity

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

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

- 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

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

- 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**
  - **Not** memorization!

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**
  - **Not** memorization!
- 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
  $S \leftarrow \textsc{Knapsack}(X, W, k-1)$
  if $\text{weight}(x_k) \leq W$ **then**
    $S' \leftarrow \textsc{Knapsack}(X, W - \text{weight}(x_k), k-1) \cup \{x_k\}$
    cache and return whichever of S, S' has the larger value
  else
    cache and return S
  end if
end procedure

Socrative `FALCOMPED`

Socrative FALCOMPED

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

Socrative `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