Greedy Algorithms

Pedro Ribeiro

DCC/FCUP

2018/2019

Greedy Algorithms

- A **greedy algorithm** is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

Greedy Algorithm

- At each step choose the **"best"** local choice
- Never look "behind" or change any decisions already made
- Never look to the "future" to check if our decision has negative consequences

Coin Change Problem (Cashier's Problem)

Input: a set of coins S and a quantity K we want to create with the coins

Output: the minimum number of coins to make the quantity K
(we can repeat coins)

Input/Output Example

Input: $S = \{1, 2, 5, 10, 20, 50, 100, 200\}$
(we have an infinite supply of each coin)
$K = 42$

Output: 3 coins $(20 + 20 + 2)$

Coin Change Problem

A greedy algorithm for the coin change problem

In each step choose the largest coin that we will not take us past quantity K

Examples (with $S = \{1, 2, 5, 10, 20, 50, 100, 200\}$):

- $K = \mathbf{35}$
  - **20** (total: 20) + **10** (total: 30) + **5** (total: 35)[3 coins]
- $K = \mathbf{38}$
  - 20 + 10 + 5 + 2 + 1 [5 coins]
- $K = \mathbf{144}$
  - 100 + 20 + 20 + 2 + 2 [5 coins]
- $K = \mathbf{211}$
  - 200 + 10 + 1 [3 coins]

Coin Change Problem

- Does this algorithm always give the **minimum** amount of coins?

- For the common money systems (ex: euro, dollar)... yes!
- For a general coin set... no!

Examples:

- $S = \{1, 2, 5, 10, 20, 25\}$, $K = 40$
  - Greedy gives 3 coins $(25 + 10 + 5)$, but it is possible to use 2 $(20 + 20)$
- $S = \{1, 5, 8, 10\}$, $K = 13$
  - Greedy gives 4 coins $(10 + 1 + 1 + 1)$, but it is possible to use 2 $(5 + 8)$

  (Will it be enough that a single coin is larger than the double of the previous coin?)
- $S = \{1, 10, 25\}$, $K = 40$
  - Greedy gives 7 coins $(25 + 10 + 1 + 1 + 1 + 1 + 1)$,
    but is is possible to only use 4 coins $(10 + 10 + 10 + 10)$

Greedy Algorithms

- **"Simple"** idea, but it does not always work
  - Depending on the problem, it may or may note give an optimal answer
- Normally, the **running time is very low** (ex: linear or linearithmic)
- The **hard part** is to prove optimality
- Typically is it applied in **optimization problems**
  - Find the "best" solution among all possible solutions, according to a given criteria (goal function)
  - Generally it involves finding a minimum or a maximum
- A very common pre-processing step is... **sorting**!

Properties needed for a greedy approach to work

Optimal Substructure

When the optimal solution of a problem contains in itself solutions for subproblems of the same type

Example

Let $min(K)$ be the minimum amount of coins to make quantity K. If that solution uses a coin of value v, then the remaining coins to use are given precisely by $min(K - v)$.

- If a problem presents this characteristic, we say it respects the **optimality principle**.

Properties needed for a greedy approach to work

Greedy Choice Property

An optimal solution is consistent with the greedy choice of the algorithm.

Example

In the case of euro coins, there is an optimal solution using the largest coin which is still smaller or equal than the quantity we need to make.

- **Proving** this property is normally the *"hardest"* part

Cashier's Problem: Proof

- Let $H = \{h_1, h_2, h_5, h_{20}, h_{50}, h_{100}, h_{200}\}$ be an optimal solution with h_v coins of each value v
- If $h_{100} > 1$, H would not be optimal (we could just substitute two 100 coins by one of 200). Therefore, $h_{100} \leq 1$
- Using the same reasoning, $h_{50} \leq 1$, $h_{10} \leq 1$, $h_5 \leq 1$ and $h_1 \leq 1$
- If $h_{20} > 2$, H would not be optimal (we could just substitute three 20 coins by one of 50 and one of 10). Therefore, $h_{20} \leq 2$ (and $h_2 \leq 2$)
- $h_2 = 2$ and $h1 = 1$ can't happen at the same time (we could just use a 5 coin instead). Therefore, $2h_2 + h_1 \leq 4$ (and $20h_{20} + 10h_{10} \leq 40$)

Cashier's Problem: Proof

- We have:
  - $h_1 \leq 1$
  - $h_2 \leq 2$ (and $2h_2 + h_1 \leq 4$)
  - $h_5 \leq 1$
  - $h_{10} \leq 1$
  - $h_{20} \leq 2$ (and $20h_{20} + 10h_{10} \leq 40$)
  - $h_{50} \leq 1$
  - $h_{100} \leq 1$
- Combining what was said before:
  - $5h_5 + 2h_2 + h_1 \leq 9$
  - $10h_{10} + 5h_5 + 2h_2 + h_1 \leq 19$
  - $20h_{20} + 10h_{10} + 5h_5 + 2h_2 + h_1 \leq 49$
  - $50h_{50} + 20h_{20} + 10h_{10} + 5h_5 + 2h_2 + h_1 \leq 99$
  - $100h_{100} + 50h_{50} + 20h_{20} + 10h_{10} + 5h_5 + 2h_2 + h_1 \leq 199$
- Let $V = \{1, 2, 5, 10, 20, 50, 100\}$.
- We have that $\sum_{i=1}^{k} v_i h_i < v_{k+1}$. Therefore, H has the same number of coins as our greedy solution! $\square$

Fractional Knapsack

Fractional Knapsack Problem

Input: A backpack of capacity C
A set of n materials, each one with weight w_i and value v_i

Output: The allocation of materials to the backpack that maximizes the transported value.

The materials can be "broken" in smaller pieces, that is, we can decide to take only quantity x_i of object i, with $0 \leq x_i \leq 1$.

What we want is therefore to obey the following constraints

- The materials fit in the backpack ($\sum\limits_i x_i w_i \leq C$)
- The value transported is the maximum possible (maximize $\sum\limits_i x_i v_i$)

Fractional Knapsack

Input Example

Input: 5 objects and $C = 100$

i	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60

What is the optimal answer in this case?

- Always choose the material with the largest value:

i	1	2	3	4	5
x_i	0	0	1	0.5	1

This would give a total weight of 100 and a total value of 146.

Fractional Knapsack

Input Example

Input: 5 objects e $C = 100$

i	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60

What is the optimal answer in this case?

- Always choose the material with the smallest weight:

i	1	2	3	4	5
x_i	1	1	1	1	0

This would give a total weight of 100 and a total value of 156.

Fractional Knapsack

nput Example

Input: 5 objects e $C = 100$

i	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60

What is the optimal answer in this case?

- Always choose the material with the largest *value/weigth* ratio:

i	1	2	3	4	5
v_i/w_i	2	1.5	2.2	1.0	1.2
x_i	1	1	1	0	0.8

This would give a total weight of 100 and a total value of 164.

Fractional Knapsack

Theorem

Always choosing the largest possible quantity of the material with the largest *value*/*weigth* ratio is a strategy leading to an optimal total value.

1) Optimal Substructure

Consider an optimal solution and its material m with the best ratio.

If we remove it from the backpack, then the remaining objects must contain the optimal solution for the materials othen than m and for a backpack with capacity $C - x_m w_m$

If that is not the case, then the initial solution was also not optimal!

Fractional Knapsack

Theorem

Always choosing the largest possible quantity of the material with best *value/weigth* ratio is a strategy that gives an optimal value

2) Greedy Choice Property

We want to prove that the largest possible quantity of the material m with the best ratio (v_m/w_m) should be included in the backpack.

The value of the backpack: $value = \sum_i x_i v_i$.

Let $q_i = x_i w_i$ be the quantity of material i: $value = \sum_i q_i v_i / w_i$

If we still have some material m available, then swapping any other material i with m will give origin to a better total value:

$q_m v_m / w_m \geq q_i v_i / w_i$ (by definition of m) $\square$

Fractional Knapsack

Greedy Algorithm for Fractional Knapsack

- Sort the materials by decreasing order of *value*/*weigth* ratio
- Process the next material in the sorted list:
  - If it fits entirely on the backpack, include it all and continue to the next material
  - If it does not fit entirely, include the largest possible quantity and terminate

Temporal Complexity:

- Sorting: $\mathcal{O}(n \log n)$
- Processing: $\mathcal{O}(n)$
- Total: $\mathcal{O}(n \log n)$

Interval Scheduling

Interval Scheduling Problem

Input: A set of n activities, each one starting on time s_i and finishing on time f_i.

Output: Largest possible quantity of activities without overlapping

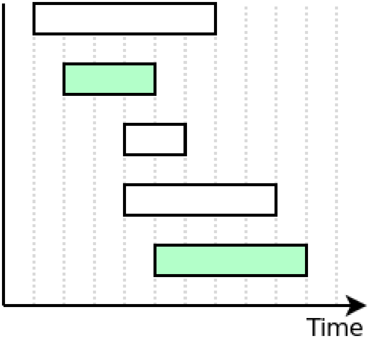
Two intervals i and j overlap if there is a time k where both are active.

Interval Scheduling

Input Example

Input: 5 activities:

i	1	2	3	4	5
s_i	1	2	4	4	5
f_i	7	5	6	9	10



Interval Scheduling

Greedy "pattern": Establish an order according to a certain criteria and then choose activities that do not overlap with activities already chosen

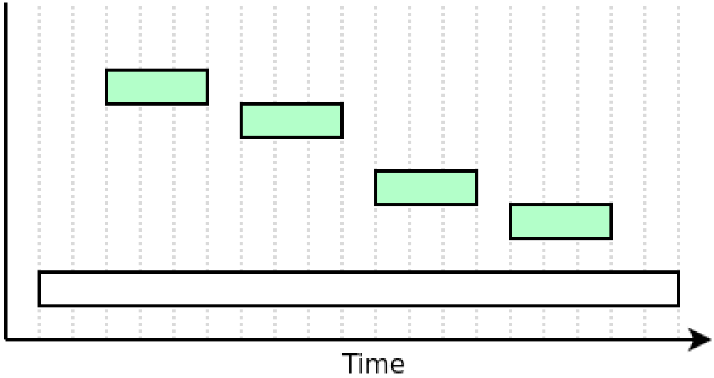
Some possible ideas:

- [Earliest start] Allocate by increasing order of s_i
- [Earliest finish] Allocate by increasing order of f_i
- [Smallest interval] Allocate by increasing order of $f_i - s_i$
- [Smallest number of conflicts] Allocate by increasing order of the number of activities that overlap with it

Interval Scheduling

[Earliest start] Allocate by increasing order of s_i

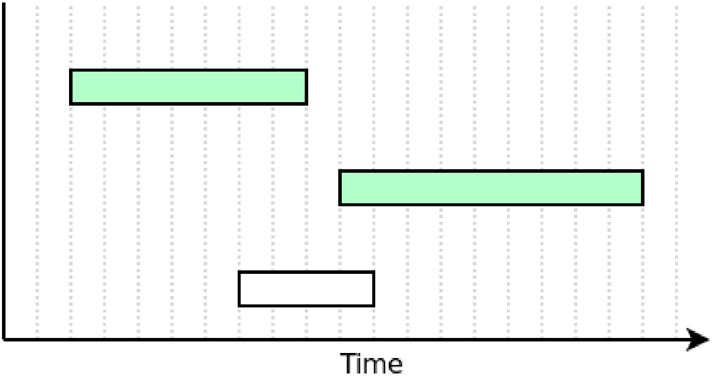
Counter-Example:



Interval Scheduling

[Smallest interval] Allocate by increasing order of $f_i - s_i$

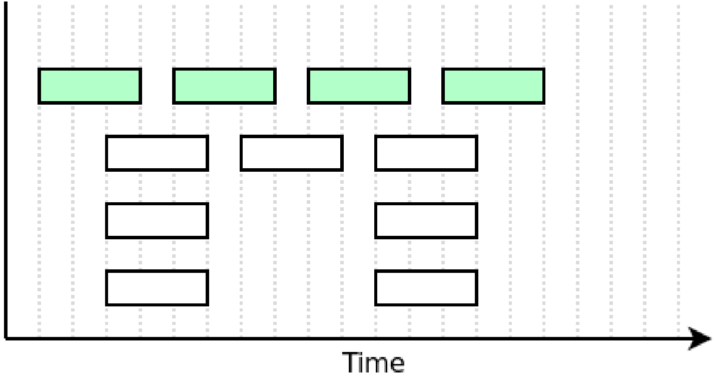
Counter-Example:



Interval Scheduling

[Smallest number of conflicts] Allocate by increasing order of the number of activities that overlap with it

Counter-Example:



Interval Scheduling

[Earliest finish] Allocate by increasing order of f_i

Counter-Example: Does not exist!
In fact, this greedy strategy produces an optimal solution!

Theorem

Always choose the non-overlapping activity with the smallest possible finish time will produce an optimal solution

1) Optimal substructure

Consider an optimal solution and the activity m with the smallest f_m.

If we remove that activity, then the remaining activities must contain an optimal solution for all activities starting after f_m.

If that is not the case, then the initial solution would not be optimal!

Interval Scheduling

Theorem

Always choose the non-overlapping activity with the smallest possible finish time will produce an optimal solution

2) Greedy Choice Property

Let's assume that the activities are sorted by increasing order of finish times.

Let $G = \{g_1, g_2, \ldots, g_n\}$ be the solution created by the greedy algorithm.

Let's show by **induction** that given any other optimal solution H, we can modify the first k activities of H so that they match the first k activities of G, without introducing any overlap.

When $k = n$, the solution H corresponds to G and therefore $|G| = |H|$.

Interval Scheduling

Base Case: $k = 1$

- Let the other optimal solution be $H = \{h_1, h_2, \ldots, h_n\}$
- We need to show that g_1 could substitute h_1
- By definition, we have that $f_{g_1} \leq f_{h_1}$
- Therefore, g_1 could stay in h_1 place without creating any overlap
- This proves that g_1 can be the beginning of any optimal solution!

Inductive Step (assuming it's true until k)

- We assume another optimal solution is $H = \{g_1, \ldots, g_k, h_{k+1}, \ldots h_m\}$
- We have to show that g_{k+1} could substitute h_{k+1}
- $s_{g_{k+1}} \geq f_{g_k}$ (there is no overlap)
- Therefore, $f_{g_{k+1}} \leq f_{h_{k+1}}$ (that is the way the greedy algorithm chooses)
- Given that, g_{k+1} could stay in h_{k+1}'s place without creating overlaps
- This proves that g_{k+1} could be chosen to extend our optimal (greedy) solution! $\square$

Interval Scheduling

Greedy Algorithm for Interval Scheduling

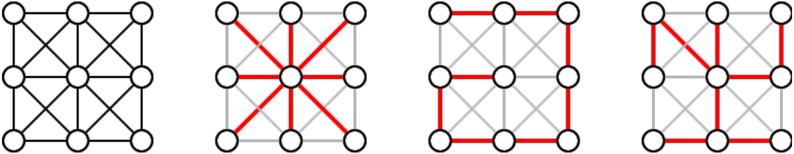
- Sort the activities by increasing order of finish time
- Start by initializing $G = \emptyset$
- Keep adding to G the next activity (that is, with the smaller f_i) that does not overlap with any activity of G

Temporal Complexity:

- Sort: $\mathcal{O}(n \log n)$
- Process: $\mathcal{O}(n)$
- Total: $\mathcal{O}(n \log n)$

Spannning Tree

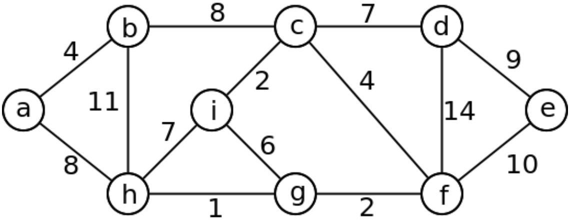
- A **spanning tree** is a subset of edges of a non directed graph that forms a tree connecting all nodes
- The following figure shows a graph and three possible spanning trees:



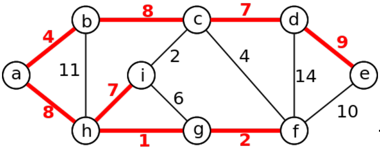
- Multiple spanning trees for the same graph may exist
- A spanning tree for a graph $G = (V, E)$ has $|V| - 1$ edges
    - If it has less edges, it does not connect the graph
    - If it has more edges, it forms a cycle

Minimum Spanning Trees

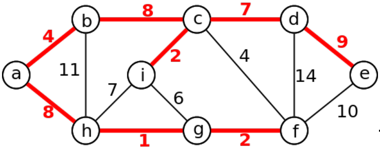
- If the graph is weighted (weights associated with each edge, we can have the notion of a **minimum spanning tree** - **MST**), which is the spanning tree that minimizes the sum of its edges.
- The following figure shows a non directed weighted graph. What is its minimum spanning tree?



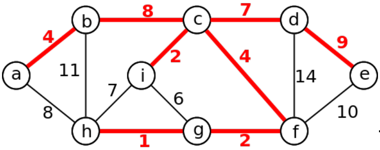
Minimum Spanning Trees



Total cost: 46 = 4+8+7+9+8+7+1+2



Total cost: 41 = 4+8+7+9+8+2+1+2



Total cost: 37 = 4+8+7+9+1+2+4+2

In fact this last tree is a **minimum spanning tree**!

Minimum Spanning Trees

- There might be **more than one MST**.
  - For example, if the weights are all equal, all spanning trees are MSTs!
- In terms of **applications**, an MST is really useful. For instance:
  - When we want to connect computers in a networks using the minimum amount of cable
  - When we want to connect houses to the electrical network using the minimal amount of wire
- How to **find an MST** for a given graph?
  - There is an exponential number of spanning trees
  - Finding all possible spanning trees and choosing the best is not efficient!
  - How to do better?

Algorithms for computing a MST

- We will discuss a classical algorithm: **Prim**
- This and other MST algorithms (ex: Kruskal) are **greedy**: in each step we add a new edge, guaranteeing that the newly added edges are part of an MST

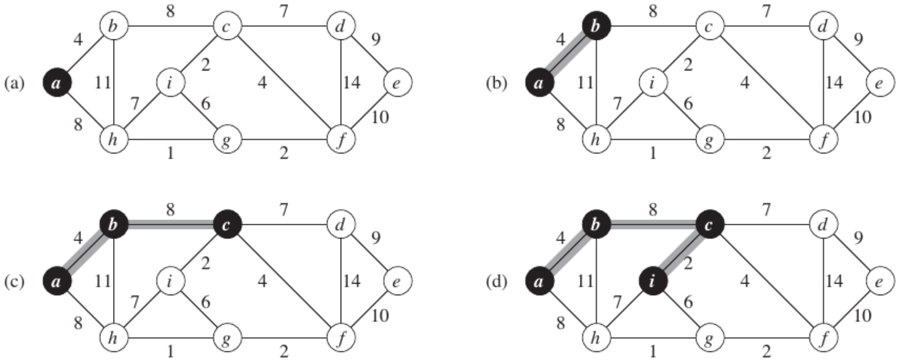
Generic Algorithm for MST

$T \leftarrow \emptyset$
While T is not an MST **do**
  Find an edge (u, v) that is "safe" to add
  $T \leftarrow T \cup (u, v)$
return(T)

Prim Algorithm

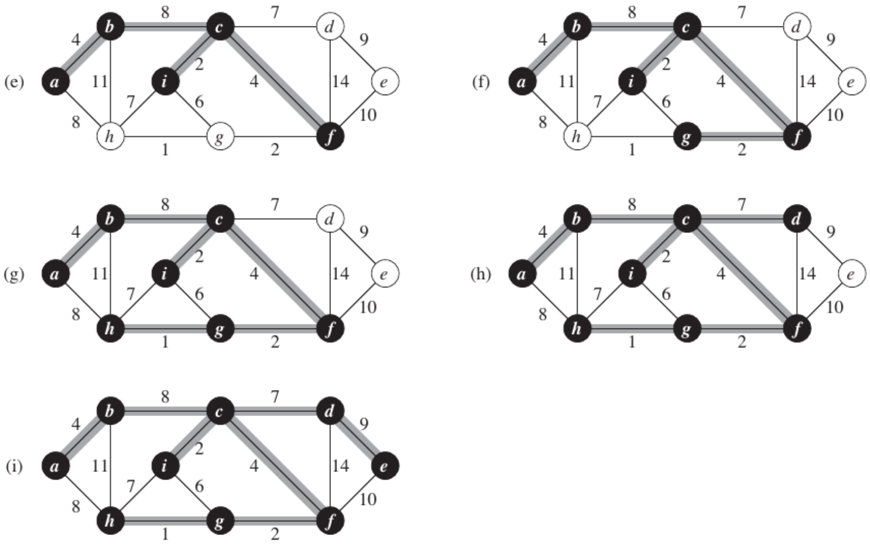
- Start in any node
- At each step **add to the tree the node whose cost is smaller**, that is, the one with the minimum weight that connects to any node already in the tree. In case of a tie, any choice works.
- Let's see step by step for the example graph.

Prim Algorithm



(image from Introduction to Algorithms, 3rd Edition)

Prim Algorithm



(image from Introduction to Algorithms, 3rd Edition)

Prim Algorithm - Proof of correctness

- Let's prove that Prim produces an MST for a non-weighted undirected **connected** graph G with n nodes:
  - Let $T = \{T_1, \ldots, T_{n-1}\}$ be the spanning tree produced by Prim
  - Let $H = \{H_1, \ldots, H_{n-1}\}$ be any minimum spanning tree
- If $T = H$ then we are done, obviously. If not, we will show that we could transform H into T without changing the cost.
- Suppose edge e is the first edge added in the construction of T that is not in H:
  - Let V be the nodes connected the moment before e is added
  - Suppose that e connects nodes u_e and w_e where u_e is the parent of w_e ($u_e \in V$ and $w_e \notin V$)
  - Because H is a spanning tree, there is a path P between u_e and w_e
  - There must be an edge f in this path P with one node in V and another not in V
  - When we were constructing T, we could have added f, but we didn't. Therefore, $weight(f) \geq weight(e)$

Prim Algorithm - Proof of correctness

- Let's build a new MST H_2 from H, replacing edge f with edge e:
  - H_2 is still connected because all paths that required f can now use e
  - H_2 is acyclic as H_2 still has $n - 1$ edges
  - The cost of H_2 is smaller or equal than H since $weight(e) \leq weight(f)$
- If $H_2 \neq T$ we continue as above, substituting edge by edge until we transform H into T
- Therefore, T is a minimum spanning tree! $\square$

Prim Agorithm

Let's put the idea of Prim into "code":

Prim algorithm for computing the MST of G (starting in node r)

Prim(G, r):
  For all nodes v of G **do**:
    $v.dist \leftarrow \infty$
    $v.parent \leftarrow NULL$
  $r.dist \leftarrow 0$
  $Q \leftarrow G.V$ /* All vertices of G */
  While $Q \neq \emptyset$ **do**
    $u \leftarrow$ GET-MIN(Q) /* Node with smaller *dist* */
    For all nodes v adjacent to u **do**
      If $v \in Q$ and $weight(u, v) < v.dist$ **then** /* Update distances */
        $v.parent \leftarrow u$
        $v.dist \leftarrow weight(u, v)$

Prim's Complexity

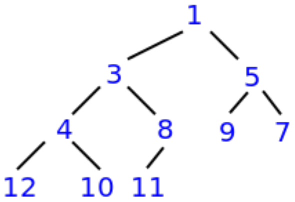
- Consider we are computing the MST of a graph $G = \{V, E\}$
- The complexity of Prim depends on GET-MIN:
  - GET-MIN will be called $|V|$ times
  - Each edge will be considered two times (one for each if its endpoints) in the cycle that updates *dist*
  - Therefore the complexity is $\mathcal{O}(|E| + |V| \times cost(\text{GET-MIN}))$
- A "naive" implementation where the next node is discovered using linear search with a cycle would be $\mathcal{O}(|E| + |V|^2)$
- We can reduce this to **linearithmic** running time if we use a data structure that supports GET-MIN in logarithmic time
- A data structure for this (returning the minimum or maximum element) is known as a **priority queue**

Heap: an implementation of a priority queue

- An **heap** is a data structure organized as a balanced binary tree, implementing a **priority queue**
- There are two basic heap types:
  - **max-heaps:** the priority element is the maximum
  - **min-heaps:** the priority element is the minimum
- For a binary tree to be considered a heap, it has to respect the following condition: **the parent of a node has always an higher priority when compared to that node** (ex: in a max-heap, the children of a node must have smaller values that the node).
- An heap must be a **a complete binary tree until the second to last level and the last level must be completely filled up from left to right**.
  - This guarantees that the maximum height of the tree with n nodes is proportional to $\log_2 n$

- An **heap is usually implemented as an array**, where:
  - The **children** of a node (i) are the nodes in positions $(i * 2)$ and $(i * 2 + 1)$
  - The **parent** of a node (i) is the node in position $(i/2)$.

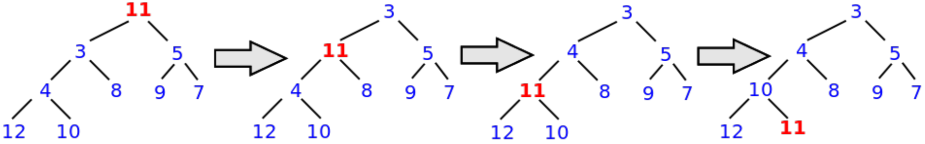
The following figure illustrates a **min-heap** and the corresponding array:



1	2	3	4	5	6	7	8	9	10
1	3	5	4	8	9	7	12	10	11

Heap: an implementation of a priority queue

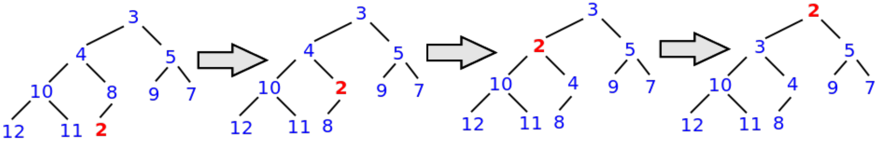
- There are two important heap operations: **removing** and **inserting**
- **Removing** an element is to remove the root
  - In a min-heap the root is the globally smallest element
  - In a max-heap the root is the globally largest element
- After removing the root, we need to reestablish the heap conditions. We can do the following:
  - We take the last element and we put it at the root position
  - That element "goes down" (**down-heap**), swapping with the hight priority child, until the heap condition is again valid
  - At most we do $\mathcal{O}(\log n)$ swaps, because the tree is balanced!



Heap: an implementation of a priority queue

- For **Inserting** an element, we could:
  - Start by putting it on the last position
  - The element "goes up" (**up-heap**), swapping with the parent, until the heap condition is reestablished
  - At most we do $\mathcal{O}(\log n)$ swaps, because the tree is balanced, because the tree is balanced!

Example for inserting 2



HeapSort

- An example application of heaps is... **sorting**! Assume we have n elements. Then, **HeapSort** is essentially the following:
  - Insert each element in the heap in $n \times \mathcal{O}(\log n)$
  - Call the remove operation n times. The element will be removed in sorted order! This step will also take $n \times \mathcal{O}(\log n)$
  - The entire process will therefore take $\mathcal{O}(n \log n)$
- Although it does not change the complexity of the whole sorting procedure, the $\mathcal{O}(n \log n)$ bound for the initial part of building the heap (inserting all elements into the heap) is not tight:
  - Assume we just put all elements an the array in the beginning
  - We could then just call **down-heap** on all elements from positions $n/2$ down to 1
  - We can show that this would have a total cost of $\mathcal{O}(n)$: **we can build a max or min-heap from an unordered array in linear time** *(we will not have time to show a proof here - have a look at the CLRS book for a formal proof - see section "5.3 - Building a heap")*

Prim algorithm and priority queues

- Recall that Prim's complexity is $\mathcal{O}(|E| + |V| \times \textit{cost}(\text{GET-MIN}))$
- Supposing we use a specialized data structure for GET-MIN, we need to take into account the time to update (lower) the values of node distances: $\mathcal{O}(|E| \times \textit{cost}(\text{UPDATE}) + |V| \times \textit{cost}(\text{GET-MIN}))$
- With a **min-heap**:
  - Each GET-MIN will cost $\mathcal{O}(\log |V|)$ (just call the remove operation of the heap)
  - Each update will also cost $\mathcal{O}(\log |V|)$ (because an update can only decrease the value, we can call up-heap on that node)
- The final complexity of Prim with heaps is $\mathcal{O}(|E| \log |V| + |V| \log |V|$, which is the same as $\mathcal{O}(|E| \log |V|)$, assuming $|E| \geq |V| - 1$ (or else, no spanning tree would even be possible).

Greedy Algorithms

- A powerful and flexible idea
- The hard part is usually to prove that it gives an optimal result
  - Optimality is not guaranteed because we are not exploring completely all our search space
  - Generally it is easier to prove non-optimality (counter-example)
  - A simple way of analysing is to think about a case where there is a tie in the greedy condition: what would the algorithm choose in that case? Does it matter?
  - We have shown examples of some possible proof techniques for greedy algorithms:
    - *"Exchange argument"*: show that you can iteratively transform any optimal solution into the solution produced by the algorithm without changing the cost
    - *"Stay-Ahead"*: find a measure by which your greedy algorithm stays ahead of the other (optimal) solution you choose to compare to
- When greedy works, usually it is efficient (low complexity)
- There is no "magic recipe" for all *greedy* algorithms: you need experience!