T Classes

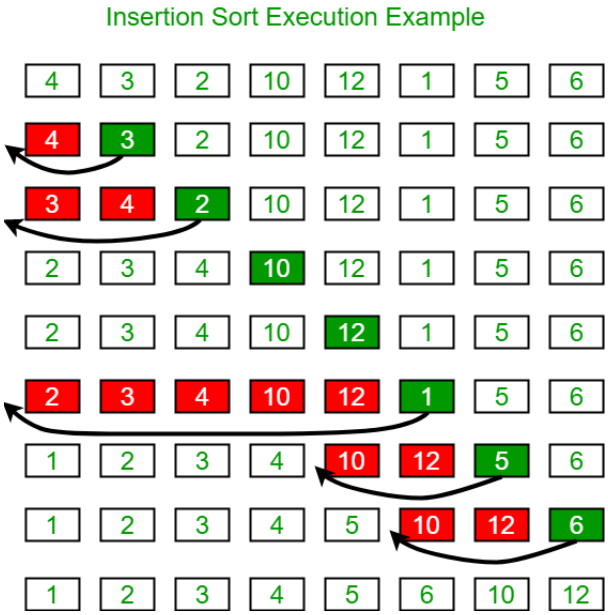
Index

1. Asymptotic Notation (Review)

1.1. What is an Algorithm?

- Computational Procedure
    - Well defined Input and Output behavior
    - Well defined (Sequence of) Operations
- Algorithm Description using Pseudo-Code
    - Allows the description of the basic algorithmic operations without the redundancy and ambiguity of natural languages

1.1.1. Example: Sorting - Insertion Sort




```
InsertionSort(A)
1. for j = 2 to length[A]
2.     key = A[j]
3.     i = j - 1
4.     while i > 0 and A[i] > key do
5.         A[i+1] = A[i]
6.         i = i - 1
7.     A[i+1] = key
```

- Execution time:
  - Best-Case Analysis: T(n) is a Linear Function of n
  - Worst-Case Analysis: T(n) is a Quadratic function of n

1.2. Analysis of Algorithms

- Complexity Measures
  - Execution Time
  - Memory Space for Data and Temporary Storage

1.2.1. Execution Time

Problem Size	**Execution Time**			
n	f(n) = n	f(n) = n^2	f(n) = 2^n	f(n) = n!
10	0,01 µs	0,10 µs	1 µs	3,63 ms
20	0,02 µs	0,40 µs	1 ms	77,1 years
30	0,03 µs	0,90 µs	1 s	8,4*10^{15} years
40	0,04 µs	1,60 µs	18,3 min	
50	0,05 µs	2,50 µs	13 days	

Problem Size	Execution Time		
100	0,10 µs	10 µs	$4*10^{13}$
1000	1,00 µs	1 µs	

1.3. Asymptotic Notation

- Characterizes Execution Times as a Function on Input Instance Sizes.
- Asymptotic Notation Allows to define Growth Rates as a Function of Input Instance Sizes.
- Constants are Somewhat Irrelevant to Growth Rate
    - Although 4n is worse than 2n asymptotically they grow at the same rate, in this case n.
- Notation:
    - O notation: Upper Asymptotic Limit
        - Code takes at most this long to run
    - Ω notation: Lower Asymptotic Limit
        - Code takes at least this long to run
    - Θ notation: Tight Asymptotic Limit
        - Code takes exactly* this long to run
        - *Except for constant factors and lower order terms
        - Only exists when $O = \Omega$!

1.4. Summations (...)

2. Data Structures (Review)

2.1. Sets and Bags

- Sets vs. Bags
    - Items distinguished by a key (typically an integer)
    - Sets do not contain replicated items
    - Unordered vs. Ordered
- Basic Operations:
    - insertItem(key, ref) and removeItem(key)
    - lookUpItem(key) or containsItem(key)
- Typical Implementations
    - Arrays:

        - Insertion at the end; remove by shifting elements to the end; linear search.
        - If ordered insertion more costly, lookup using binary search (O(log(n)))
    - Linked List:
        - Insertion either at tail or head; remove done by adjusting pointers; lookup still linear and ordering does not help much as pointers still need to be traversed

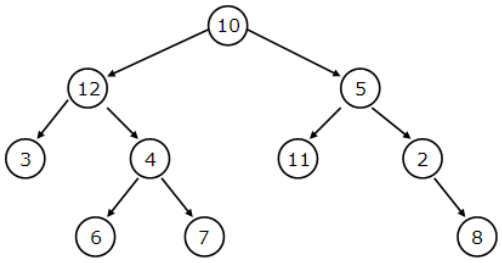
2.2. Queues and Stacks

- Queue
    - First-In First-Out (FIFO) policy
    - Insert Item at the end (tail)
    - Remove Item, if not empty, from the head
- Stack
    - Last-In First-Out (LIFO)
    - Insert (or Push) Item at the top of the stack
    - Remove (or Pop) Item from the top of the stack
- Typical Implementations
    - Array or Singly-Linked Lists

2.3. Trees

- Hierarchical Structure
    - Parent/Child and Sibling Relations
    - Internal and Leaf Nodes
    - Traversal Operations recursively defined
    - Balanced vs. Unbalanced key for efficient data organization
    - O(log(n)) search on balanced trees vs. O(n) for extremely unbalanced trees
- Basic Operations
    - insertItem(key), removeItem(key)
    - Traverse all nodes starting at the root (top or first node)
- Typical Implementations
    - Array
        - Integer indices 2*i and 2*i+1 defined the children of node at index i of the array for a binary tree embedded in an array
    - Linked Data Structures
        - Conceptually simpler for manipulation but much less compact
- Recursive Data Structure
- Ubiquitous N-way Trees
    - Commercial Databases

    - Compilers
- Binary case:
    - Left and Right Children
- Complete vs. Incomplete and Balanced vs. Unbalanced
    - Height and Number of Nodes in Tree
- Three Basic Recursively defined Visit Patterns:



Levelorder tree traversal
10, 12, 5, 3, 4, 11, 2, 6, 7, 8
Inorder tree traversal
3, 12, 6, 4, 7, 10, 11, 5, 2, 8
Preorder tree traversal
10, 12, 3, 4, 6, 7, 5, 11, 2, 8
Postorder tree traversal
3, 6, 7, 4, 12, 11, 8, 2, 5, 10

- **Preorder:** visit the root; then visit preorder the children left to right
- **Postorder:** visit postorder the children left to right, then the root
- **Inorder (binary trees):** visit inorder the left child, then the root and then the right child
- (ver último exercício da primeira ficha das TPs)

2.4. Graphs

- Arbitrary Relationships between Items
    - Nodes identified by a unique key
    - Edges define relationship between nodes
    - Directed vs. Undirected
    - Acyclic vs. Cyclic
    - Stacks, Queues, Tree are all special cases of Graphs
- Basic Operations
    - InsertNode(id), InsertEdge(id1, id2), RemoveNode(id), Succs(id), Pred(id)
    - Traversal Breath-First-Search/Depth-First-Search
- Typical Implementations
    - Arrays
        - Nodes denoted by integer indices, Edges represented by a matrix.
        - Insertion/Removal is complicated as this is a sparse representation.
    - Linked Data Structures
        - Conceptually simpler for manipulation and traversal.

2.5. Hash-Table

- Mapping between Domains

    - Typically maps an Enumerable Domain to Integers
    - Many-to-One Mapping leads to collisions in the implementation solved by closed-hashing or open-hashing data structures
    - Selecting an Hash function is key

- Basic Operations
    - insertItem(value) and lookUpItem(value)
- Typical Implementations
    - Array of Pointers to Linked Lists (open hashing)
    - Array of Structures with integer indices (close hashing)

2.6. Linear Search

- Search for an element (key) in an (dense)array – Comparison against a Unique Key
    - Returns index of element if found
- Array is Unsorted
    - Just scan the array from start to end and check at each location for key
    - Complexity: O(n)

```
LinearSearch(A, key)
  for i = 1 to length[A]
    if A[i] = key then
      return i
return 0
```

2.7. Binary Search

- Search for an element (key) in an (dense)array
    - Comparison against a Unique Key
    - Returns index of element if found
- Array is Sorted
    - Divide-and-Conquer using binary search
    - Complexity: O(log(n))
    - Harder on a linked-List

```
BinarySearch(A, left, right, key)
  if (left ≤ right) then
    m = [(left + right) / 2];
    if A[m] = key then
      return m;
    if (A[m] < key) then
```

```
        return BinarySearch(A, m+1, right, key);
    else if (A[m] > key) then
        return BinarySearch(A, left , m-1, key);
return 0
```

2.8. Min-Heap Data Structure

- Video Pedro Ribeiro: [EDados] Filas de Prioridade e Heaps
- Slides Pedro Ribeiro: [EDados] Filas de Prioridade e Heaps
- Heap Visualization
- Array of Values Interpreted as a Binary Tree
    - Root is A[1];
    - length(A) is Size of the Array
    - heapSize is the Number of Elements in the Heap
    - Relation between Nodes:
        - Parent(i) = [i / 2]
        - Left(i) = 2 * i
        - Right(i) = 2 * i + 1
- Min-Heap Property:
    - Value(A[Parent(i)]) ≤ Value(A[i])
- Basic Operations
    - ExtractMin:
        - Trivial, just return the top elements
        - But, need to adjust the Heap to have Min-Heap structure
        - Complexity: O(log(n))
    - SiftDown:
        - Internal operation to adjust the Min-Heap
        - Swap root with last value; swap parent with child that violates Min-Heap relationship
        - Complexity: O(log(n))
    - More Operations: SiftUp, Insert

3. Basic Graph Algorithms

3.1. Definitions

- Graph G = (V, E) – Defined by a set of Vertices V and Edges E
    - Edges denote connections between pairs of vertices
        - $E \subset V \times V$
        - In a Sparse Graph: $|E| \ll |V \times V|$
- Directed and Undirected Graphs
    - Notion of Direction in Edges or its Absense.

    - Undirected: e = (n1,n2) -> (n2,n1) but might not be explicitly represented
- Weigthed or Unweigthed
    - Weight associated with each Edge e ∈ E.
    - Function Cost: E -> R

3.2. Graph Paths

- A path from a vertex v to a vertex u is a sequence of vertices <v0, v1, v2, ..., vn> where:
    - v0 = v, vn = u
    - (vi,vi+1) ∈ E for i = 0, 1,..., n-1.
- The length of a path is defined as the number of edges in the path.
- A path forms a cycle if v = u. If no such cycle exists, the (directed or not) graph is called acyclic.

3.3. Graph Connectivity

- For an **undirected** graph G:
    - If every pair of vertices is connected by at least one path, then graph G is connected.
- For an **acyclic** graph G:
    - If G is connected, it is called a *tree*;
    - Otherwise, it is called a *forest*.

3.4. Breath First Seach (BFS) Traversal

- Video: Breadth First Traversal for a Graph | GeeksforGeeks

- Given G = (V, E) and source node s, BFS explores all nodes in G reachable from s

    - distance: the shortest distance from source s to each node
    - BF tree: organization of the paths from the source s to each node

- BFS Strategy:

    - Explores nodes expanded the frontier between visited and unvisted nodes uniformly
    - Nodes at distance k are visited before any node at distance k+1

- Implementation:

    - Uses a Worklist Algorithmic Strategy with a Queue Data Structures
    - Examines each node and place its successors in a queue to be examined later
    - Halts when all nodes have been visited.

- Implementation Details:

    - color[v]: status progress for node v
        - white: unprocessed
        - gray: in process
        - black: processed
    - pred[v]: predecessor of v in BF tree
    - dist[v]: visit distance for v
    - Queue Data Structure

        - Get node from Queue: Dequeue
        - Add (append) node to the end of a Queue: Enqueue
        - Empty Test

- Algorithm:

```
function BFS(Graph G, node s)
  for each node u ∈ V[G] - { s } do
    color[u] = white; dist[u] = ∞; pred[u] = NIL;
  color[s] = gray; dist[s] = 0; pred[s] = NIL;
  Q = { s };
  while (Q ≠ ∅) do
    u = Dequeue (Q);
    for each v ∈ Adj[u] do
      if color[v] = white then
        color[v] = gray;
        d[v] = d[u] + 1;
        pred[v] = u;
        Enqueue (Q, v);
    color[u] = black;
```

- Correctness:
    - For every edge (u,v):
        - If u is reachable then so is v
            - shortest path to v cannot be longer that shortest path to u plus edge (u,v)
            - Result is also correct if u in unreachable (independent of v's reachability)
    - At the end:
        - d(s,v): shortest distance (number of arcs) from s to v
        - dist[u] = d(s,u), for all nodes

- Time Complexity: O(V + E)
    - Initialization: O(V)
    - For each node:
        - Inserted in queue only once: O(V)
        - Adjacency list visited only once: O(E)

3.5. Depth First Search (DFS) Transversal

- Video: Depth First Traversal for a Graph | GeeksforGeeks

- Search Graph In Depth
    - Expand the most recently visited node

- Implementation:
    - color[u]: status of processing a node (white, gray, black)
    - dist[u]: entry sequence of a node

    - visit[u]: exit sequence of a node
- Algorithm:
    - Uses recursive call but can also be done using a stack (same principle)

```
function DFS-Visit(node u)
  color[u] = gray;
  dist[u] = time;
  time = time + 1;
  for each v ∈ Adj[u] do
    if color[v] = white then
      pred[v] = u;
      DFS-Visit(v);
  color[u] = black;
  visit[u] = time;
  time = time + 1;

function DFS(graph G)topolo
  for each node u ∈ V[G] do
    color[u] = white;
    pred[u] = NIL;
    visit[u] = 0;
  time = 1;
  for each node u ∈ V[G] do
    if color[u] = white then
      DFS-Visit(u);
```

- Time Complexity: O(V + E)
    - Initialization: O(V)
    - Invocation of DFS-Visit within DFS: O(V)
    - Edges examined in DFS-Visit: Q(E)
        - Invocation of DFS-Visit within DFS-Visit: O(V)

3.6. Topological Sorting

- Video: Topological Sorting (with Examples) | How to find all Topological Orderings of a Graph
- Topological Sorting of a DAG G=(V,E) is a sorting of the nodes (linearization) such that (u,v) ∈ E then u appears before v in the sorting
- Algorithm: By Elimation of Nodes and Edges

```
function Topological-Sort(Graph G)
  L = ∅; // List of nodes
  Q = ∅; // Queue of nodes
  for each v ∈ G do
    if v has no incoming edges (w,v) then
      Enqueue(Q,v);
  while Q ≠ ∅ do
```

```
    u = Head(Q);
    Eliminate edges (u,v);
    if v has no incoming edges (w,v) then
      Enqueue(Q,v);
    Dequeue(Q);
    Append u to end of list L;
```

- Algorithm:

```
function Topological-Sort(Graph G)
  Execute DFS(G) and compute visit[v] for each node v;
  During DFS-visit of v when complete visit for v
    prepend node v to list L;
  return list L;
```

- Execution Time Complexity
  - DFS: O(V+E)

3.7. Strongly Connected Components

- Video: Kosaraju's Algorithm - Strongly Connected Components | GeeksforGeeks
- Definition: A directed graph G = (V, E) has a strongly connected component (SCC) as the maximal set of nodes U ∈ V, such that u,v ∈ U, u is reachable from v, and v is reachable from u
  - Obs: a single node is a SCC
- Other definitions:
  - Reverse Graph of G = (V, E)
  - Key Observation:
    - G e G^T have the same SCCs
- Algorithm:

```
function SCCs(graph G)
  Generate reverse graph GR
  Execute DFS(GR) and compute post order traversal T
  Execute DFS(G) and visit nodes in reversed post order TR
  Each DFS tree corresponds to a new SCC
```

- Execution Time Complexity: O(V+E)

3.7.1. Sample Graph Problem

- Efficient algorithm to detect if a graph G = (V, E) is bipartite?

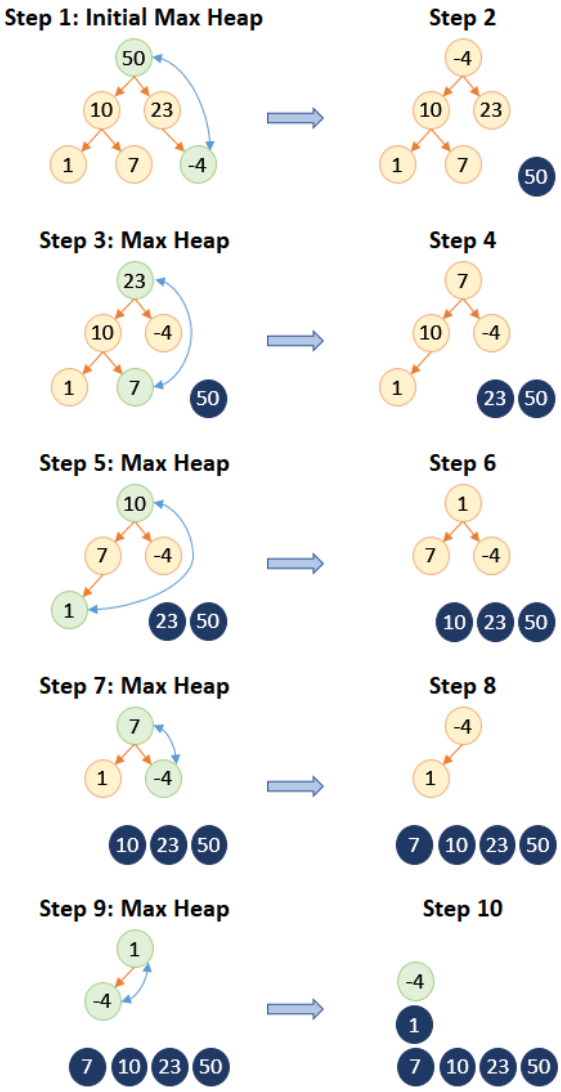
- G is bipartite if V can be divided into L and R, such that the edges (u,v) of G are such that u ∈ L and v ∈ R or vice-versa.

- Efficient algorithm to compute the diameter of tree T=(V,E) ?
  - Diameter: max δ(u,v), u,v∈V
  - Hint: Two BFSs

- Efficient algorithm to detect if G = (V, E) is semi-connected
  - A directed graph G = (V,E) is semi-connected if for any edge (u,v), u is reachable from v or v is reachable from u

4. Sorting (Review)

4.1. Heap Sorting



- Use a Max-Heap or a Min-Heap Data Structure
  - Partial Order (Incomplete) Binary Tree
  - Balanced with depth h or h-1 for all nodes

    - Top element (or Root) is Max or Min of the values in the Heap

- Basic Idea:
    - Select the top element of the Heap and rearrange the Heap
    - Insert element at the end of an array
    - When all elements have been extracted from the Heap, algorithm ends

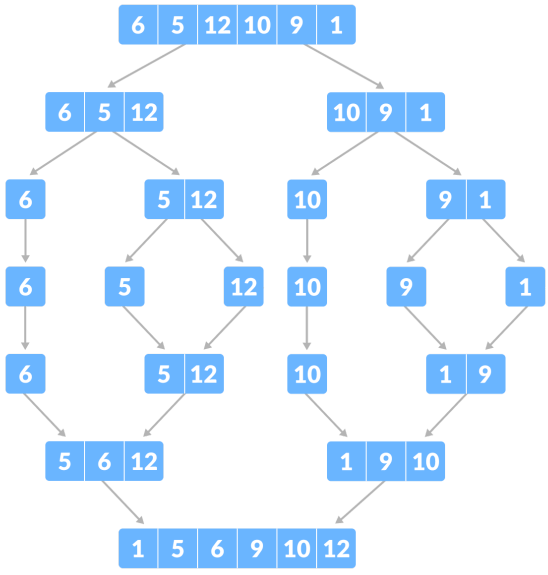
- Similar to Insertion Sorting
    - Data Structure, Heap makes the difference

```
buildHeap(A)
  heapSize[A] = length[A]
  for i = [length[A] / 2] downto 1
    siftDown(A, i)

HeapSort(A)
  buildHeap(A)
  for i = length[A] downto 2
    swap(A[1], A[i])
    heapSize[A]--
    siftDown(A, 1)
```

- Complexity:
    - Building the Heap is O(n log(n))
    - For each of the n elements of the array perform an O(h) operation
    - Overall worst-case complexity is O(n log n)
    - It is possible to prove a bound of O(n)

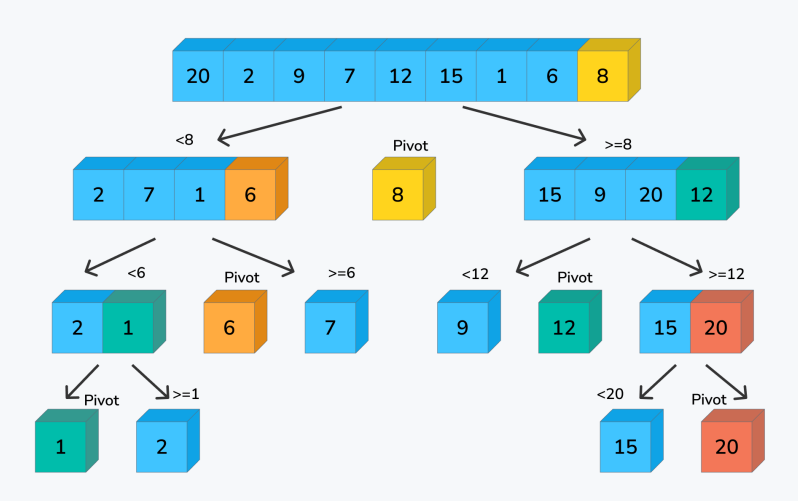
4.2. Merge Sort



```
MergeSort(A, p, r)
1.  if p < r then
2.     q = [(p+r) / 2]          // splitting criteria
3.     MergeSort(A, p, q)     // forward divide
4.     MergeSort(A, q+1, r)
5.     Merge(A, p, q, r)      // backward merge
```

- In the Worst-Case the execution time grows with n log n

4.3. Quick Sort



- Divide-and-Conquer Strategy
    - Divides the input into two sequences using a pivot element
    - Statistically, sequences are of equal length
    - Recursively sort each sequence in-place
- Key: Find a Good Pivot Element
    - Partitions the input sequence into equal-sized sequences
    - Simple partition: choose pivot as last/first element of sequence

```
Partition(A, p, r)
  x = A[r]
  i = p - 1
  for j = p to r-1 do
    if (A[j] ≤ x) then
      i = i + 1
      swap(A[i], A[j])
  swap(A[i+1], A[j])
  return i+1

QuickSort(A, p, r)
  if (p < r) then
    q = Partition(A, p, r)
```

```
    QuickSort(A, p, q-1)
    QuickSort(A, q+1, r)
```

- Worst-Case
    - Input array already sorted
    - Partition at each step is O(array length)
    - Uneven sequences, one of size 1 another of size n-1
    - Complexity: O(n2)
- Best-Case
    - Partition always finds the best pivot that splits current sequences into two equally-sized sequences
    - Recurrence relation: T(n)= 2 T(n/2) + cn
    - Complexity: O(nlog(n))

5. Brute-Force

5.1. What is Brute Force?

- A straightforward approach to problem solving
- Based on problem statement and definitions of the concepts involved
- Examples:
    - Numeric Calculations
    - Search by Enumeration of all possible domain points
    - Selection Sort
- Why the name "Brute Force"?
    - No cleverness on the implementation
    - Straightforward Implementation (very simple algorithm)
    - Uses computing power not cleverness…

5.2. Why Brute Force?

- Measure (or yard stick) for Algorithm Performance
    - Space and Time
- Correctness and Optimality
    - Algorithm is simple – it's correctness is trivially established
    - Optimality is usually ensured – all domain is explored…

5.3. Exhaustive Search

- A Brute Force Approach to Combinatorial Problems (which require generation of permutations, or subsets)
- Generate every element of Problem Domain
- Select Feasible ones (ones that satisfy constraints)
- Find the desired one (the one that optimizes some objective function)
- Works, but Only Feasible for Small Problem Sizes.

6. Greeedy Algorithms

- Strategy: At each Step of the Algorithm, select the option that is locally the best to find the overall optimal solution
- In many cases this strategy works.
- Examples:
    - Minimum-Cost Spanning Trees: Kruskal, Prim,
    - Single-Source Shortest Paths: Dijkstra.

6.1. Example: Selection of Activities

- Let S = {1, 2, ..., n} be a set of activities that share a common resource
    - Resource can only be used by one activity at a time
    - Activity i is characterized by:
        - start time: si
        - finish time: fi
        - activity execution interval: [si, fi)
    - Activities i and j are Compatible if [si, fi) and [sj, fj) are disjoint
- **Objective:** Find a/the Maximal set of Activities that are Mutually Compatible

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
si	1	3	0	5	3	5	6	8	8	2	12
fi	4	5	6	7	8	9	10	11	12	13	14

- It is possible to choose many mutually exclusive sets of tasks:
    - {a3, a9, a11}
    - {a1, a4, a8, a11}
    - {a2 a4, a9, a11}
- Assume Activities are sorted s.t. f1 ≤ f2 ≤ ... ≤ fn
- Greedy choice:
    - Select Activity with lowest finish time fk
        - Rationale? Maximize time for remaining activities

```
function selectActivitiesGreedy(set S, set F)
  n = length[s];
  A = {1};
  j = 1;
  for i = 2 to n do
   if (si ≥ fj) then
     A = A U {i};
     j = i;
return A
```

- Algorithm:
  - Select activity with lowest finishing time;
  - Check which other activities are compatible
  - Initialize activities by increasing finishing time
- { a1, a4, a8, a11 } optimal solution -> But not unique!
- Execution Time Complexity: O(n log n) + O(n) = O(n log n)

6.2. Example: Knapsack Problem

- Given n objects (1, ..., n) and one Knapsack of capacity W
- Each object has value vi and weight wi
- It is possible to transport a fraction xi of an object: 0 £ xi £ 1
- Transported weight cannot exceed W
- **Objective:** Maximize the transported value of objects while meeting the Knapsack's weight constraint
- Observations:
  - Sum of the considered objects cannot exeeced weight limit W
  - Optimal solution must fill up knapsack entirely,
    - Otherwise we could transport more fractional items, thus with larger aggregate value!

```
function fillUpKnapsackGreedy(v, w, W)
  weight = 0;
  while weight < W do
    select element i with maximal vi/wi
    if (wi + weight ≤ W) then
      xi = 1; weight += wi
    else
       xi = (W-weight)/ wi; weight = W
```

- Execution Time: O(n), or O(n log n)
- Algorithm finds the optimal solution!

6.3. Example: Minimize System Processing Time

i	1	2	3	4	5	6	7	8	9	10	11
si	4	3	2	5	3	5	6	8	8	2	7

- Strategy 1: Process longest jobs first!
  - Order of service = { 9, 8, 7, 11, 4, 6, 1, 2, 3, 10 }
  - Total Service Time = 10×8 + 9×8 + 8×7 + 7×6 + 6×5 + 5×4 + 4×3 + 3×3 + 2×2 + 1×2 = 315
- Strategy 2: Process shortest jobs first!
  - Order of service = { 10, 3, 2, 1, 6, 4, 11, 7, 8, 9 }
  - Total Service Time = 10×2 + 9×2 + 8×3 + 7×3 + 6×4 + 5×5 + 4×6 + 3×7 + 2×8 + 1×8 = 191
- Algorithm finds the optimal solution!

6.4. Example: Huffman Codes

Characters	a	b	c	d	e	f
Frequency (x1000)	45	13	12	16	9	5

- Variable-Length encoding:
  - Number of required bits: (45×1 + 13×3 + 12×3 + 16×3 + 9×4 + 5×4) ×1000 = 224000 bits
- Prefix-free codes:
  - No code is prefix of another code
  - 001011101 -> 0.0.101.1101
  - Codes represented by a complete binary tree
  - (ver imagem slide 29)

```
function Huffman(C)
  n = |C|;
  Q = C ; // Constructs priority queue
  for i = 1 to n - 1 do
    z = AllocateNode();
    x = left[z] = ExtractMin(Q);
    y = right[z] = ExtractMin(Q);
    f[z] = f[x] + f[y];
    Insert(Q, z);
return ExtractMin(Q);
```

- Execution Time: O(n log n)

7. Union-Find Operations on Disjoint Sets

(O ponto 7 não parece muito importante; Resumir melhor)

- Definitions
    - Data Structures for Disjoint Sets:
        - Supports dynamic sets of disjoint elements
        - Each set is represented by an element of the set;
        - Representative element is unchanged
            - Add or look-up operations
- Operations on Disjoint Sets
    - Representation:
        - Each element of each set is represented by a selected element x
    - Operations:
        - **MakeSet(x)**
            - Creates a new set represented by the element x
        - **Union(x,y)**
            - Merges the sets that have the elements x and y, Sx e Sy:
            - New set is: Sx È Sy
            - Sx end Sy are eliminated (disjoint sets)
            - Representative element of the new set is either x or y
        - **FindSet(x)**
            - Returns the set that contains the element x
- Examples of Application
    - Kruskal's MST Algorithm
        - MakeSet: Creates a subset of the nodes in the graph
        - FindSet: Identifies the set a given node belongs to.
        - Union: Merges the sets of nodes as a single newly created set.
    - Connected Components of a Graph G=(V, E)

7.1. Linked-List Implementation

- Organization:
    - Elements of each set in a (singly) linked list
    - First element of the list is the set's representative element
    - Each node of the list also has a pointer to the head of the list
- Operations:
    - CreateSet(x): Just creates a single node of the linked list with x;
    - Find(x): Scans the list for the x element and return Nil if not found;
    - Union(x,y): appends the elements of Find(y) list to the elements of Find(x) list.
- Time Complexity Analysis
    - Complexity is O(m2) for a sequence of m operations
    - Operations: (n = ém/2ù + 1; q = ëm/2û-1; m = n + q)

        - n MakeSet operations
        - Sequence of q Union(xi-1,xi) operations, for i = 2, ..., q
            - Each operation Union(xi-1,xi) updates i-1 elements
            - Cost of q operations: Q(q2)
        - Total cost of m operations is Q(n+q2), which represents Q(m2)

7.2. Heuristic: Weighted Union

- Associated with each set a weight
    - Simple heuristic: weigth is the number of elements
- Union operation:
    - Append the list with smaller number of elements to the list with **larger** number of elements
- Overall cost of m operations is improved
- Execution Time with Heuristic
    - Sequence of m operations (including n Union operations) is: O(m + n log n)

7.3. Tree-based Implementation

- Organization:
    - Each set os represented by a tree
    - Each element points to its parent in the tree
    - Set representative is the root of the tree
        - Root's parent is the root itself
- FindSet:
    - Scan parent until root is found (this.parent = this)
- Union:
    - Root's parent of one tree is made to point to the other tree's root
- Complexity: sequence of O(m) operations is O(m n)
    - Worst case when tree is a linked-list

7.4. Heuristic - Union by Rank

- **Union:** tree with less number of elements points to tree with larger number of elements
    - Use height of tree as an estimate of the number of nodes in the tree
        - **rank:** approximates the logarithm of size of sub-tree and is an upper bounds of the tree height
        - makes execution time analysis easier
- In union, root of tree with lower rank points to root of tree with higher rank

7.5 Heuristic - Path Compression

- On every FindSet operation, make each visited node along the path to the root to point directly to the root.
- Benefit will be reaped in subsequent FindSet operations

```
MakeSet(x)
  p[x] = x;
  rank[x] = 0;

FindSet(x)
  if x ≠ p[x] then
    p[x] = FindSet(p[x]);
return p[x]

Union(x,y)
  Link(FindSet(x), FindSet(y));

Link(x,y)
  if rank[x] > rank[y] then
    p[y] = x;
  else
    p[x] = y;
  if rank[x] = rank[y] then
    rank[y] = rank[y] + 1;
```

8. Minimum Cost Spanning Trees

8.1. Spanning Tree Definitions

- An undirected graph G = (V, E) is **connected** if for any pair of vertices (or nodes) there exists (at least) one path connecting the two vertices.
- Given an undirected, and **connected**, graph G = (V, E), a **spanning tree** is an acyclic, subset of the edges T ⊆ E connecting all the vertices (or nodes) in G.
- Given a spanning tree, its **cost** is the summation of the costs associated with its edges.

8.1.1 Minimum Cost Spanning Tree (MST)

- Given the graph G = (V, E), connected, undirected, with weight function w : E -> R, identify a spanning tree T, such that the summation of the edge weights of T is minimal

8.2. Brute-Force Algorithm

- Algorithm:

```
function MST-BruteForce(Graph G, function w)
  S = generateSpanningTrees(G);
  for each s ∈ S do
    sc= Cost(s,w);
  select s ∈ S with min cost;
  return s
```

- Probably Not a Good Idea...
  - The number of Trees can be Huge...
  - Matrix-Tree Theorem
  - Compute the Laplacian of the adjacency matrix
  - Number of distinct Trees is the product of all non-zero eigenvalues

8.3. Building an MST

- Greedy Approach:
  - Maintain a subset tree A of the graph G
  - Identify edge (u,v) added to A such that
    - A U {(u,v)} is still a subset tree A
    - Creates no ***cycles***, i.e., is a ***safe*** edge
- Algorithm:

```
function MST-Generic(Graph G, function w)
  A = {};
  while A is not a spanning tree do
    identify safe edge (u,v) for A;
    A = A U {(u,v)};
  return A
```

8.4. MST Definitions

- A **cut** (S, V-S) of an undirected graph G=(V,E) is a partion of V into disjoint sets of nodes.
- An edge (u,v) ∈ E **crosses** the cut (S, V-S) if one of its nodes is in S and the other node is in V-S.
- A cut **contains** a set of edges A if no edge ∈ A crosses the cut.
  - all edges connect nodes in either S or (V-S).
- An edge that crosses a cut with the lowest cost is designated as the **lightest edge**
- An edge is **safe** for A if its inclusion in A does not create any cycles in A

8.5. MST Construction Properties

- MSTs satisfy two very useful properties:

    - **Cycle Property:** The heaviest edge along a cycle is NEVER part of an MST.
    - **Cut Property:** Split the vertices of the graph any way you want into two sets A and B. The lightest edge with one endpoint in A and the other in B is ALWAYS part of an MST.

- **Observation:** If you add an edge to a tree and you create (exactly) one cycle, you can then remove any edge from that cycle and get another tree out.

- This observation, combined with the cycle and cut properties form the basis of all of the greedy algorithms for MSTs

8.6. Kruskal's Algorithm

- Video: Kruskal's Algorithm for Minimum Spanning Tree | GeeksforGeeks

- Algorithm Outline:

    - Start with each isolated node as its own cluster
    - Pick the lightest edge e ∈ E
        - if e connects two nodes in different clusters, then e is added to the MST and the clusters merged,
        - otherwise ignore it
    - Continue until |V|- 1 edges are added

- Implementation:

    - Algorithm maintains a forest of trees or subset A ⊂ E
    - Uses a disjoint-sets data structure for representing and merging clusters
    - Each set or cluster represents a sub-tree of the final MST

- Algorithm:

```
function MST-Kruskal(G,w)
  A = {};
  foreach v ∈ V[G] do
    MakeSet(v); // creates a cluster for v
  Sort edges ∈ E by non-decreasing order of weight;
  foreach (u,v) ∈ E[G] in sorted order do
    if FindSet(u) ≠ FindSet(v) then
      // (u,v) is the lightest and safe edge for A
      A = A U {(u,v)};
      Union(u,v); // merge clusters for u and v
return A;
```

- Time Complexity:
    - Depends on the implementation of disjoint sets
    - Possible to define O(E log E)
    - Given that $E < V^2$, we get O(E log V)

8.7. Borůvka's Algorithm

(Não é importante, o professor não deu na aula)

- Algorithm Outline:
  - Starts with N sets of nodes (N trees)
  - For each tree T
    - selected mininum weight edge incident in each tree T
      - an edge may be selected by multiple trees
  - Merge trees connected by selected edges
  - Terminates when there is a single tree
- Correct if weigths are distincts
  - Use lexicographic order to desambiguate between edges with the same weigth.
- Complexity: O(E log V)
  - Number of steps: O(log V)
  - Number of edges evaluated at each step: O(E)
  - Tree bookeeping: O(E)

8.8. Prim's Algorithm

- Video: Prim's Algorithm for MST(with Code Walkthrough) | GeeksforGeeks
- Builds MST from a Root node
  - Algorithm Starts with the Root node
  - Expands Tree one Edge at a time
  - **At each step the Algorithm choose the Lightest Safe Edge**
- Using Priority Queue Q
- key[v]:
  - lowest edge weight connecting v to a node in the Tree
- pred[v]:
  - predecessor of v in the Tree

```
function MST-Prim(G,w,r)
  Q = V[G]; // Priority queue Q
  foreach u ∈ Q do // Initialization
    key[u] = ∞;
  key[r] = 0;
  pred[r] = NIL; // Keep track of tree A
  while Q ≠ {} do /* O(V) */
    u = ExtractMin(Q); // Pick closest unprocessed node /* O(log V) */
    // ∃ (u,v) safe and light edge, for tree A
    foreach v ∈ Adj[u] do /* O(E) */
      if (v ∈ Q and w(u,v) < key[v]) then // Check if node is not in MST
```

```
            pred[v] = u;
            key[v] = w(u,v); // Min Heap Q is updated! /* O(log V) */
```

- Complexity: O(E log V)
    - Priority queue using a heap
    - For each edge (i.e., O(E)) exists in the worst case an update of Q with cost O(log V)

8.9. Comparison

- Although each of the above algorithms has the same worst-case running time, each one achieves this running time using different data structures and different approaches to build the MST
- There is no clear Winner among these 3 algorithms

9. Maximum Flow Algorithms

9.1. Maximum Flows in Graphs

- Given a Directed Graph G=(V, E):
    - Source node s and a Sink node t
    - Each edge (u,v) has a non-negative capacity c(u,v)
        - The edge capacity c(u,v) indicates the maximum value of flow that is possible to send from u to v through the edge (u,v)
        - Compute the Maximum Value of “flow” that can be
            - “Pushed” from the Source to the Sink
            - Subject to Edge Capacity Constraints

9.2. Multiple Sources and Sinks

- For Networks with Multiple Sources and/or Sinks:
    - Define a Super-Source connected to all Sources
    - Define a Super-Sink connected to all Sinks
    - Infinite capacity between super-source/sink and sources/sinks
- Augment the Graph to make it with one Source and one Sink!

9.3. Definitions

- A Flow Network G = (V, E) is a directed graph in which each edge (u,v) has a capacity $c(u, v) \geq 0$
    - If $(u,v) \notin E$, then $c(u,v) = 0$
- Two Special Nodes: Source s and Sink t
- All Nodes of G belong to a path from s to t
    - Connected graph, $|E| \geq |V| - 1$
- A Flow of G = (V, E) is a function f : V x V -> R such that:

    - $f(u, v) \leq c(u, v)$ for $u, v \in V$ (capacity constraint)
    - $f(u, v) = - f(v, u)$ for $u, v \in V$ (symmetry)
    - for $u \in V - \{s,t\}$: $\sum(u,v) = 0$, $v \in V$ (flow invariant/conservation)
- Flow Value: $|f| = \sum f(s,v)$, $v \in V$
- Maximum Flow Problem:
    - Given the flow network G with Source **s** and Sink **t**, compute the maximum flow value from **s** to **t**.

9.4. Ford-Fulkerson Method

- Definitions:
    - Residual Network
    - Augmenting Paths
    - Cuts on Flow Networks
        - Maximum-Flow / Minimum Cut Theorem
- Ford-Fulkerson Algorithm
    - Greedy Algorithm
    - Complexity
        - Convergency Problems

```
Ford-Fulkerson-Method(Graph G, node s, node t)
  initialize flow f to 0;
  while (exists an augmenting path P) do
    increase flow along P;
    update residual network;
  return f;
```

- (ver exemplo slides 13-25 MaxFlow-part1)

9.5. Residual Network

- Given G = (V, E), a flow f, and $u,v \in V$
    - **residual capacity** of (u,v) is the additional flow that is possible to send from u to v
        - $cf(u,v) = c(u,v) - f(u,v)$
    - **residual network** of G:
- $Gf = (V, Ef)$, where $Ef = \{ (u,v) \in V \times V : cf(u,v) > 0 \}$
    - Each edge (residual) of Gf only allows positive flow
- G = (V, E), f a flow, Gf residual network; f' a flow in Gf
- **Added Flow** f + f' defined for each pair $u,v \in V$:
    - $(f + f')(u,v) = f(u,v) + f'(u,v)$

- Value of the added flow |f + f'| = |f| + |f'|

9.6. Augmenting Paths

- Given G = (V, E) and the flow f
    - Augmenting Path p:
        - Simple Path from s to t in residual network Gf
    - Residual Capacity of p:
        - cf(p) = min { cf(u,v) : (u,v) in p }
    - cf(p) allows the definition of a flow fp in Gf, |fp| = cf(p) > 0
    - f' = f + fp is a flow in G, with value |f'|=|f| + |fp| > |f|

9.7. Cuts and Flows in a Flow Network

- A Cut (S, T) of G = (V, E) is a partition of V in S and T = (V – S), such that s ∈ S and t ∈ T – liquid flow of the cut (S, T): f(S,T) = ∑∑f(u,v), u∈S,v∈T – cut capacity (S, T): c(S,T) = ∑∑c(u,v), u∈S,v∈T – Obs: Cut includes only positive capacity edges; liquid flow also negative flows.
- If G = (V, E) has flow f, then the "liquid" flow throught the cut (S, T) is f(S,T) = |f|
    - T = (V-S); f(S,T U S) = f(S,T) + f(S,S); f(S,T) = f(S,V) - f(S,S)
    - f(S,T) = f(S,V) - f(S,S) = f(S,V) = f(s,V) + f(S - s,V) = f(s,V) = |f|
        - Obs: for u ∈ S - s, f(u, V) = 0

9.8. Max Flow/Min Cut Theorem

- Let G = (V, E), with source s and sink t, and flow f, then the following prepositions are equivalent:
    1. f is a maximum flow in G
    2. The residual network Gf has no augmenting paths
    3. |f| = c(S,T) for a cut (S,T) of G
- Proof: 1. => 2.
    - Assume f is a maximum flow in G and that Gf has an augmenting path
    - Then it is possible to define a new flow f + fp with value |f| + |fp| > |f|; a contradiction
- Proof 2. => 3.
    - Gf has no Augmenting Paths i.e., no path from s to t.
    - S = { v ∈ V : exists a path from s to v in Gf }; T = (V-S); s ∈ S and t ∈ T
    - With u ∈ S and v ∈ T, we have f(u,v) = c(u,v), as otherwise v would belong to S; and thus
        - |f| = f(S,T) = c(S,T)
- Proof 3. Þ 1.
    - Given that |f| £ c(S,T), for any cut (S,T) in G

- As |f| = c(S,T) (defined above), then f is a maximum flow

9.9. Ford-Fulkerson Basic Algorithm

```
Ford-Fulkerson(Graph G, node s, node t)
  foreach (u,v) ∈ E[G] do
    f[u,v] = 0;
    f[v,u] = 0;
  while exists an augmenting path p in residual network Gf do
    compute cf
    (p);
    foreach (u,v) ∈ p do
      f[u,v] = f[u,v] + cf(p) // Increase flow value
      f[v,u] = - f[u,v]
```

- (ver exemplo slides 35-42 MaxFlow-part1)
- For Rational Values of Capacities
  - Convert all capacities to integers by scaling
  - Number of augmenting paths limited by the maximum value of the flow |f*|
  - Complexity: O(E |f*|)
    - For example: DFS to find augmenting paths
- For Irrational Values of Capacities
  - Basic Algorithm might never terminate...
  - Basic Algorithm might even converge to incorrect value...

9.10. Edmonds-Karp Algorithm

- Choose Augmenting Paths using Shortest Path
  - Each Edge has a distance of 1
  - Use BFS in Gf to identify shortest path
  - Complexity: O(V E^2)
- (ver exemplo slide 46 MaxFlow-part1)

9.10.1 Analysis

- Definitions
  - δf(s,v): shortest distance from s to v in residual network Gf
  - δf'(s,v): shortest distance from s to v in residual network Gf'
  - Sequence of actions:
    - f -> Gf -> BFS -> p -> f' -> Gf' -> BFS -> p'
- Results:

    - δf(s,v) increases monotonically with each increase in flow
    - Number of flow increases is O(V E)
    - Execution Time is O(V E^2)
        - O(E) due to BFS and the increase of flow at each step

- δf(s,v) grows monotonically with each increase of flow
- Always choose an augmenting path with as few edges as possible
- Number of flow increases is O(V E)
- Distance from s to u increases at least 2 units each time the edge (u,v) is critical
- Complexity of Edmonds-Karp is O(V E^2)
    - Complexity of BFS is O(V+E) = O(E) (given V = O(E))
    - Increases of flow is O(E)

10. Shortest Path Algorithms

11. Divide and Conquer

12. Dynamic Programming

13. Backtracking

14. Linear Programming