COMP2123 Review Notes

Important: ADT, Time Complexity, Space Complexity, Correctness

Correctness:

- data structures → invariant (the invariant holds before an operation is executed, the operation has the desired result and the invariant still holds after the operation is executed)
- greedy → exchange argument (see 9-Greedy)
- divide and conquer → induction (recursion)

1 - Time Complexity Analysis

We say that $T(n)$ is ...	if ...
constant	$T(n) = \Theta(1)$
logarithmic	$T(n) = \Theta(\log n)$
linear	$T(n) = \Theta(n)$
quasi-linear	$T(n) = \Theta(n \log n)$
quadratic	$T(n) = \Theta(n^2)$
cubic	$T(n) = \Theta(n^3)$
exponential	$T(n) = \Theta(c^n)$

2 - Lists

Data Structure	Operation	Time
Array	`get(i)` , `set(i, e)`	O(1)
	`insert(0, e)` , `remove(0)` from the start	O(n)
	`insert(i, e)` , `remove(i)` in the middle	O(n)
	`append(e)` , `pop()` from the end	O(1)
	`replace(i, e)` , `size()` , `isEmpty()`	O(1)
Singly Linked	`get(i)`	O(n)
/ **Doubly Linked**	`addFirst(e)` , `removeFirst()` from start	O(1)
	`add(i, e)` , `remove(i)` in middle	O(n)
	`addLast(e)` , `removeLast()` from end	O(n) / O(1)
	`replace(i, e)`	O(n)
	`size()`	O(n)
Stack	push(e), pop()	O(1)
Queue	enqueue(e), dequeue()	O(1)

3 - Trees

Tree ADT

Method	Time
`size()` , `isEmpty()`	O(1)
`root()` , `parent(p`	O(1)
`children(p)`	O(c)
`numChildren(p)`	O(1)
`isInternal(p)` , `isExternal(p)` , `isRoot(p)`	O(1)

- Tree Traversals: Preorder, Inorder, Postorder → O(n)

4 - Binary Search Trees

Binary Search Tree (BST)

1. **Properties**:
   - For any node v, keys in the left subtree are less than v, and keys in the right subtree are greater than v.
   - In-order traversal of a BST results in keys in increasing order.
2. **Operations**:
   - **Search**: `search(k, v)` O(h) (h = height)
   - **Insertion**: `put(k, o)` O(h)
     - If the key exists, replace the value.
     - Otherwise, insert at the reached external node.
   - **Deletion**: `remove(k)` O(h)
     - **Case 1**: Node has one external child — remove and promote the other child.
     - **Case 2**: Node has two internal children — find the in-order successor, copy its value, and delete it.
3. **Complexity**:
   - **Space**: O(n)
   - **Best Case Height**: O(log n)
   - **Worst Case Height**: O(n)

Range Queries in BST

- **Purpose**: Find all keys k in range $k1 \leq k \leq k2$.
- **Algorithm**: in-order (ascending order)
  - Traverse left or right subtree based on key comparisons.
  - $|\text{inside nodes}| \leq |\text{output}|$; $|\text{boundary node}| \leq 2 * \text{tree height}$
  - **Time Complexity**: $O(|\text{output}| + \log n)$

AVL Tree (Self-Balancing BST)

nodes usually contains a key/a pointer

1. **Properties**:
   - Balanced BST where the heights of left and right subtrees of every node differ by at most 1.
   - **Height**: O(log n)
2. **Operations**:
   - **Search:**
     - **Time Complexity**: O(log n)
   - **Insertion**:
     - Standard BST insertion, followed by rebalancing.
     - If balance property is violated, perform rotations
     - **Time Complexity**: O(log n)
   - **Deletion**:
     - Standard BST deletion, followed by rebalancing using rotations if necessary.
     - **Time Complexity**: O(log n)
3. **Trinode Restructuring**:
   - Helps restore balance by making the middle node of three unbalanced nodes the root of that subtree.
   - **Time Complexity**: O(1) for rotation operations.

Map ADT (Key-Value Store)

Method	Description
`size()`	Returns the number of entries
`isEmpty()`	Checks if the map is empty
`entrySet()`	Returns an iterable of all entries
`keySet()`	Returns an iterable of all keys

`values()`	Returns an iterable of all values

5 - Priority Queues

Priority queue implementations

Method	Unsorted List	Sorted List	Heap
size, isEmpty	$O(1)$	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$	$O(\log n)$
min	$O(n)$	$O(1)$	$O(1)$
removeMin	$O(n)$	$O(1)$	$O(\log n)$
remove	$O(1)$	$O(1)$	$O(\log n)$
replaceKey	$O(1)$	$O(n)$	$O(\log n)$
replaceValue	$O(1)$	$O(1)$	$O(1)$

Priority Queue Sorting Algorithms

1. **Selection Sort (Unsorted Sequence)**:
   - **Time Complexity**: `insert`: O(n), `remove_min`: O(n^2)
2. **Insertion Sort (Sorted Sequence)**:
   - **Time Complexity**: `insert`: O(n^2), `remove_min`: O(n)
3. **Heap-Sort** (Uses a min-heap for efficient sorting)
   - **Time Complexity**: `insert`, `remove_min`: O(logn)

i	A	s
0	7, 4, 8, 2, 5, 3, 9	3
1	2, 4, 8, 7, 5, 3, 9	5
2	2, 3, 8, 7, 5, 4, 9	5
3	2, 3, 4, 7, 5, 8, 9	4
4	2, 3, 4, 5, 7, 8, 9	4
5	2, 3, 4, 5, 7, 8, 9	5
6	2, 3, 4, 5, 7, 8, 9	6

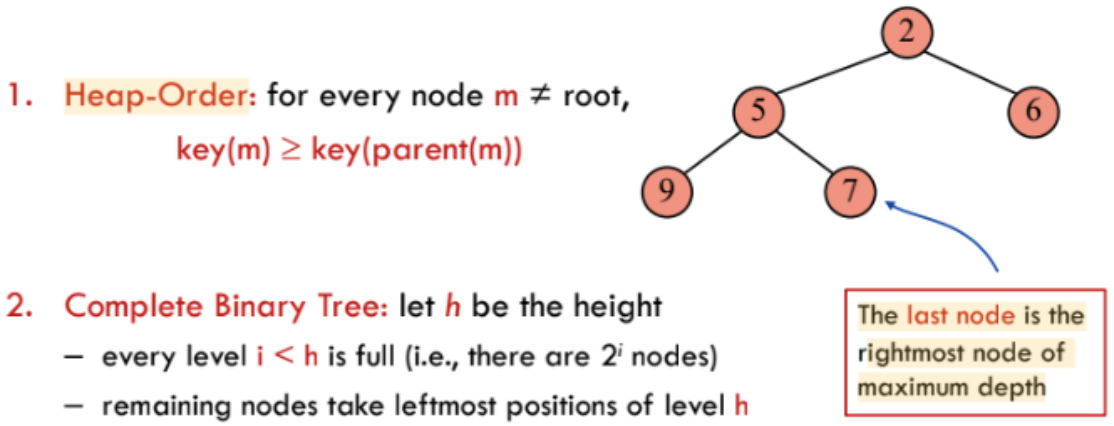
```
def selection_sort(A):
    n ← size(A)
    for i in [0:n] do
        # find s ⩾ i minimizing A[s]
        s ← i
        for j in [i:n] do
            if A[j] < A[s] then
                s ← j
        # swap A[i] and A[s]
        A[i], A[s] ← A[s], A[i]
```

i	A	j
1	7, 4, 8, 2, 5, 3, 9	0
2	4, 7, 8, 2, 5, 3, 9	2
3	4, 7, 8, 2, 5, 3, 9	0
4	2, 4, 7, 8, 5, 3, 9	2
5	2, 4, 5, 7, 8, 3, 9	1
6	2, 3, 4, 5, 7, 8, 9	6

```
def insertion_sort(A):
    n ← size(A)
    for i in [1:n] do
        x ← A[i]
        # move forward entries > x
        j ← i
        while j > 0 and x < A[j-1] do
            A[j] ← A[j-1]
            j ← j - 1
        # if j>0 ⇒ x ≥ A[j-1]
        # if j<i ⇒ x < A[j+1]
        A[j] ← x
```

Heap Data Structure (Min-Heap)

1. **Properties**: (remember to explain both nodes and the root of the heap)

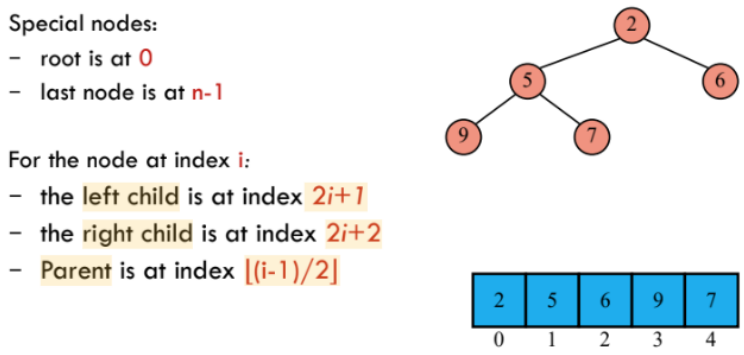


2. **Height**: O(logn).
3. **Min-Heap Operations**:

Operation	Description	Time
Build Heap	Builds a heap from an unsorted array	O(n)
`insert`	Insert at the last position and perform **upheap**	O(logn)
`remove_min`	Replace the root with the last node, remove the last node, and perform **downheap**	O(logn)
Access root `min`	Returns the min/max element without removal	O(1)
Decrease/Increase Key	Adjusts the value of a specific key	O(logn)
Find Next Last Node After Deletion	Start from the old last node, go up to find a right child or root, go to sibling, then down to leaf	O(logn)

- **Counting Inversions with Selection Sort**:
  - **Inversions** are pairs (i,j) in an array A where i<j but A[i]>A[j].
  - By performing **Selection Sort** on an array, the **number of swaps** needed is equal to the **number of inversions** in the array. This is because each swap corrects one inversion, moving a larger element to its proper position relative to a smaller element that was previously after it.
- **Finding the k-th Value in Sorted Order Using Heaps**:
  - Given an unsorted array A, you can use a **min-heap** or **max-heap** to efficiently find the k-th smallest or largest element without fully sorting the array.
  - **Using a Min-Heap**:
    - A **min-heap** helps to track the smallest elements in the array.
    - The **root** of the min-heap is the smallest element.
    - To find the k-th largest element, keep only the k largest elements in the min-heap by replacing the root if a new element is larger. After inserting k elements, the root will be the k-th largest element.
  - **Using a Max-Heap**:
    - A **max-heap** works similarly but tracks the largest elements.
    - To find the k-th smallest element, maintain a max-heap of size k, where the root is the largest among the k smallest elements seen so far.
    - If a new element is smaller than the root, replace the root, ensuring the heap only keeps the smallest k elements. After processing all elements, the root will be the k-th smallest element.
- **Merging k Sorted Lists of Length m Using a Min-Heap**:
  - Given k sorted lists, each of length m, the goal is to merge them into a single sorted list efficiently.
  - **Method**:
    - Use a **min-heap** where each element in the heap represents the smallest unprocessed element from each of the k lists.
    - Inserting an element into the min-heap takes O(logk) time.
    - For k lists, inserting and removing elements from the heap will take O(klogk) time per element added to the result list.
  - **Overall Complexity**:
    - Since there are m elements in each of the k lists, the total number of elements to merge is km.
    - The time complexity for merging all elements is O(kmlogk), as each insertion and deletion operation on the min-heap takes O(logk) time.

Heap Array Representation



- **Counting Inversions with Selection Sort**:
  - the **number of swaps** needed is equal to the **number of inversions** in the array.
- **Finding the k-th Value in Sorted Order Using Heaps**:
  - keep only the k largest elements in the min-heap by replacing the root if a new element is larger. After inserting k elements, the root will be the k-th largest element.
- **Merging k Sorted Lists of Length m Using a Min-Heap**:
  - For k lists, inserting and removing elements from the heap will take O(klogk) time per element added to the result list.
    Overall Complexity: O(kmlogk)

6 - Hashing

Key Concepts in Hash Tables

- **Load Factor (α)**: Defined as α=n/N (number of elements over table size). Ideally, α should be below 0.75 for optimal performance.

Implementations of Map ADT

Implementation	Description	Insert (put)	Search (get)	Delete (remove)	Space
Simple List-Based Map	Stores items in an unsorted list	O(1)(append)	O(n)	O(n)	O(n)
Array-Based Map	Uses an array for keys in a fixed range (e.g., [0, N-1])	O(1)	O(1)	O(1)	O(N) (for fixed range N)
Hash Table	Maps keys to indices using a hash function	O(1) (average)	O(1)(average)	O(1)(average)	O(n)

Collision Handling

Method	Description	Average Time Complexity	Suitable Scenarios
Separate Chaining	Each table cell points to a linked list that holds all items hashed to that index	O(1+α), worst-case O(n)	Allows higher load factors; requires extra memory for linked lists
Linear Probing	Places colliding items in the next available position (circularly)	O(1) with low load factor, worst-case O(n) (`get` , `put` , `remove`)	Efficient with careful load factor control, but may cause clustering
Cuckoo Hashing	Uses two hash tables and two hash functions	worse-case O(1) for `get` , `remove` , expected O(1) for `put`	Offers worst-case guarantees for lookups; more complex implementation

Eviction Cycle Detection:

1. **Use a Counter**: to keep track of the number of evictions. If we evict enough times we are guaranteed to have a cycle.
2. **Flag Entries**: Keep an additional flag for each entry. Every time we evict an entry, we flag it. After a successful put, we need to unflag the entries flagged.

Set Implementation Using a Map

- **Map-Based Set**: Use a map to store elements as keys, ignoring values.
- **Performance**: Using a hash-based map, primary set operations (add, remove, contains) typically have O(1) time complexity.

- **Using Hash Table with Doubly Linked List for `get` , `put` , `delete` in O(1) Time**:
  - By using a hash table in combination with a doubly linked list, we can perform `get` , `put` , and `delete` operations in constant time, O(1).
  - The doubly linked list maintains a pointer to the elements in the hash table, allowing for efficient insertion and deletion.
  - This approach is commonly used in data structures like **LRU (Least Recently Used) Cache**, where the hash table provides O(1) access to elements, and the doubly linked list maintains the order of elements for quick updates.
- **Finding the Most Frequent Value in O(n) Time**:
  - To find the most frequent element in a collection, a hash table can be used to count occurrences.
  - **Collision Handling**: To handle hash collisions, linear probing or chaining can be used.

- After constructing the hash table, which stores the frequency of each element, traverse the hash table to identify the element with the highest frequency.
- This approach runs in O(n) time because inserting and updating counts in the hash table takes O(1) on average per element, and scanning the hash table at the end also takes O(n).

- **Using `get()` in a Multimap with O(1+s) Time Complexity**:
  - A **multimap** allows multiple values to be associated with the same key.
  - By implementing a multimap with a hash table, we can retrieve all values associated with a key.
  - Since multiple values may be stored at the same hash address, using `get()` requires retrieving all values associated with that key.
  - The time complexity of `get()` is O(1+s), where s is the number of values associated with the key (i.e., the size of the list at that address). This includes the O(1) time to access the hash bucket and O(s) to retrieve all values.

7 - Graphs

Graph Properties

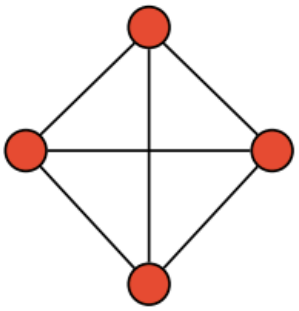
Tree: A connected acyclic graph.

Fact: $\sum_{v \text{ in } V} \deg(v) = 2m$

Fact: In a simple undirected graph $m \leq n\,(n - 1)/2$

Notation

n	number of vertices
m	number of edges
Δ	maximum degree



Example: K_4

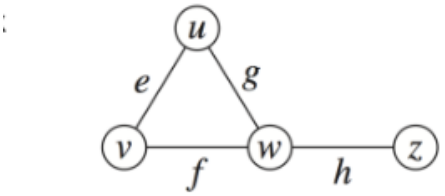
n = 4

m = 6

max deg = 3

▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$O(n + m)$	$O(n + m)$	$O(n^2)$
incidentEdges(**v**)	$O(m)$	O(deg(**v**))	$O(n)$
getEdge(**u**, **v**)	$O(m)$	O(min(deg(**u**), deg(**v**)))	O(1)
insertVertex(**x**)	O(1)	O(1)	$O(n^2)$
insertEdge(**u**, **v**, **x**)	O(1)	O(1)	O(1)
removeVertex(**v**)	$O(m)$	O(deg(**v**))	$O(n^2)$
removeEdge(**e**)	O(1)	O(1)	O(1)

if Adjacency List need traverse, `removeVertex`: O(n+m) `removeEdge`: O(m)



	0	1	2	3
u → 0		e	g	
v → 1	e		f	
w → 2	g	f		h
z → 3			h	

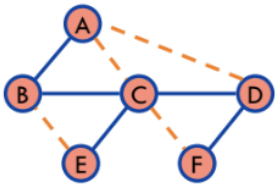
Algorithm	Complexity (Adjacency List)	Edges
DFS	O(n+m)	back edges
BFS	O(n+m)	cross edges

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Cut edges	√	

Non-tree DFS edge (v, w)

w is an ancestor of v in the DFS tree

Called back edges

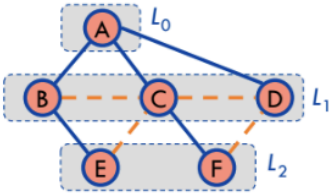


DFS

Non-tree BFS edge (v, w)

w is in the same level as v or in the next level

Called cross edges



BFS

Identifying cut edges in O(n+m) time

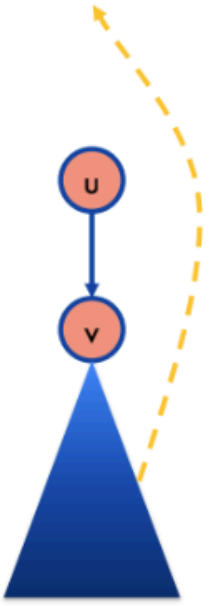
Compute a DFS tree of the input graph G=(V, E)

For every u in V, compute level[u], its level in the DFS tree

For every vertex v compute the highest level that we can reach by taking DFS edges down the tree and then one back edge up. Call this down_and_up[v]

Fact: A DFS edge (u, v) where u = parent[v] is not a cut edge if and only if down_and_up[v] ≤ level[u]

Basis of an O(n+m) time algorithm for finding cut edges



The University of Sydney

Page 32

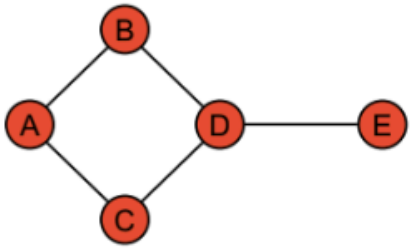
Identifying cut edges in O(n+m) time

Compute a DFS tree of the input graph G=(V, E)

For every u in V, compute level[u], its level in the DFS tree

For every vertex v compute the highest level that we can reach by taking DFS edges down the tree and then one back edge up. Call this down_and_up[v]

返祖边（Back edge）：连接当前节点和它祖先节点的边（但不包括直接的父节点）



	level	d&u
A	0	0
B	1	0
C	3	0
D	2	0
E	3	3

如果 down_and_up[v] ≤ level[u]，则边 (u,v)(u, v)(u,v) 不是割边。

如果 down_and_up[v] > level[u]，则边 (u,v)(u, v)(u,v) 是割边。

The University of Sydney

Page 33

2. Bipartite Check: **Using BFS to** color each node per layer.

- **Bipartite**: Adjacent nodes in different layers.

3. Cycle Detection

- **Using Adjacency List**: O(n)
  - Perform DFS to mark each node.
  - If you revisit a marked node, a cycle exists.
- **Using Adjacency Matrix**: O(n)
  - Perform DFS to mark nodes by rows and columns.
  - Check in-degrees and out-degrees to avoid getting stuck.

4. Identifying Cut Vertices and Cut Edges: O(n+m)

- **Cut Edges (Bridges)**:
  - In a DFS tree, for an edge (u,v) where u is the parent and v is the child:
    - If `down-and-up[v] > level(u)`, (u,v) is a cut edge.
    - If `down-and-up[v] ≤ level(u)`, (u,v) is not a cut edge.

- **Cut Vertices (Articulation Points)**:
  - The root of the DFS tree is a cut vertex if it has two or more children.
  - For any internal node u with two or more children, u is a cut vertex if and only if `down-and-up[v] ≥ level(u)` for any child v.

8 - Graph Algorithms

1. Dijkstra's Algorithm

-

---

2. Shortest Path Problem with Vertex Costs

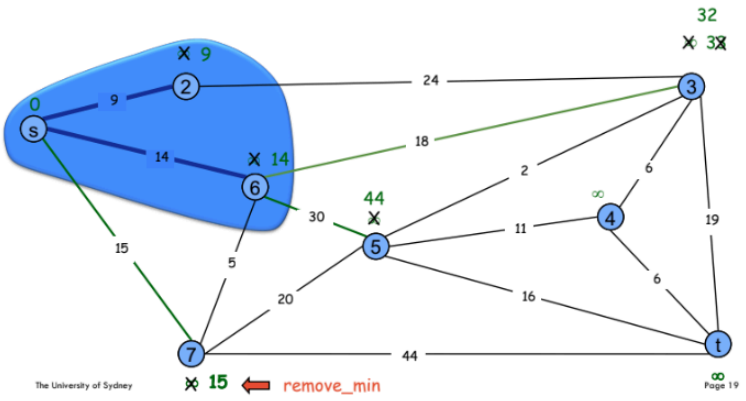
- Construct a new graph G′ based on the original graph G:
  - For each edge (u,v) in G, add an edge (u′,v′) in G′ with weight weight(u,v)+cost(v).
- **Binary Heap**:
  - Each insertion is O(logn).
- **Fibonacci Heap**:
  - Supports O(1) decrease key operations when edge weights are relaxed.

Dijkstra’s Algorithm

Standard Heap: O(mlogn); Fibonacci Heap: O(m+nlogn); Binary Heap: O((m+n)logn)

- **Algorithm Overview**:
  - Initialize the starting node distance as 0, and all other nodes as infinity.
  - Update distances of adjacent nodes accordingly while expanding the shortest path.

S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }



The University of Sydney Page 19

Dijkstra's Algorithm Correctness

Invariant: For each $u \in S = V \setminus Q$, we have $D[u] = dist_w(s, u)$

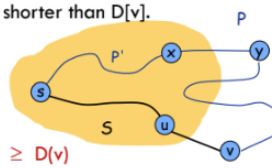
对于每个已经在集合 S 中的顶点 u，我们已经找到了从起点 s 到该顶点的最短路径距离 D[u]

Proof: (by induction on $|S|$) 假设对于已经加入 S 的顶点，这个不变量成立，接着证明加入新的顶点后依然保持成立

Base case: $|S| = 1$ is trivial since $D[s] = 0$

Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be next node added to S and u=parent[v]
- The shortest s-u path plus (u, v) is an s-v path of length D[v]
- Consider any s-v path P. We'll see that it's no shorter than D[v].
- Let x-y be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it leaves S:



$$w(P) \geq w(P') + w(x,y) \geq D(x) + w(x,y) \geq D(y) \geq D(v)$$

inductive hypothesis — Def of D(y) — Dijkstra chose v instead of y

The University of Sydney Page 32

Dijkstra's Algorithm Correctness

Invariant: The set { (u, parent[u]) : u in S \ { s } } forms a shortest path tree from s to every node in S

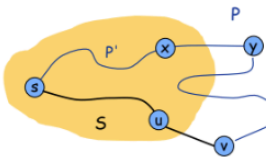
对于已经确定最短路径的节点（在集合 S 中），每个节点 u 和它的父节点 parent[u] 形成了一棵从起点 s 出发的最短路径树

Proof: (by induction on $|S|$)

Base case: $|S| = 1$ is trivial since tree is empty

Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be next node added to S and u=parent[v]
- $dist_w(s, v) = dist_w(s, u) + w[u, v]$
- Adding v to S we still have the invariants



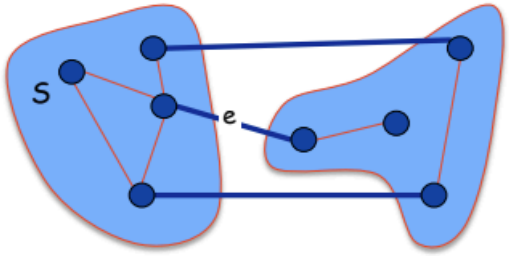
The University of Sydney Page 33

Minimum Spanning Tree

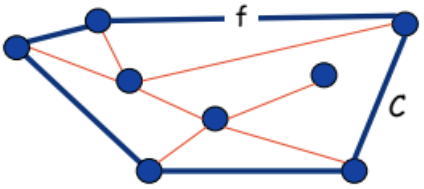
Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST contains e.

Cycle property. Let C be any cycle, and let f be the max cost edge belonging to C. Then the MST does not contain f.



e is in the MST



f is not in the MST

The University of Sydney

Page 38

Cycle-Cut Intersection

Claim. A cycle and a cutset intersect in an even number of edges.

Prim's Algorithm

- **Steps**:
  1. Initialize the MST with an arbitrary vertex.
  2. Add the minimum-weight edge connecting the current tree to any vertex not in the tree.
  3. Repeat until all vertices are included in the MST.
- **Time Complexity**:
  - Binary heap: O(mlogn); Fibonacci heap: O(m+nlogn).

Kruskal's Algorithm

- **Steps**:
  1. Sort all edges by weight.
  2. For each edge, check if it forms a cycle with the current MST using Union-Find.
  3. If no cycle is formed, add the edge to the MST.
  4. Stop once n−1 edges are added.
- **Time Complexity**:
  - Sorting edges: O(mlogm).

- Using Union-Find to check cycles: O(n^2)
- `make_sets(A)` : O(n) `find(u)` : O(1) `union(u, v)` : O(n)

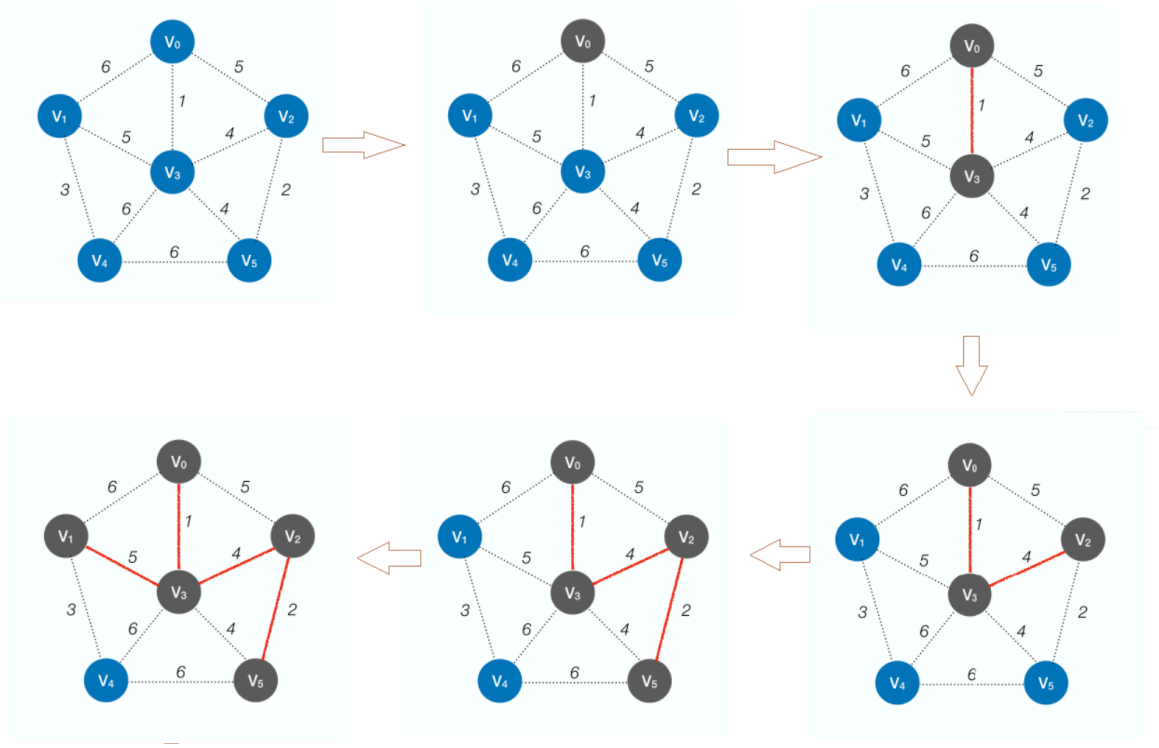
Union-Find (in Kruskal's Algorithm to detect cycles)

- **Operations**:
  1. **make_sets**: Initializes singleton sets for each element.
  2. **find**: Finds the root representative of the set containing an element, with path compression to speed up future queries.
  3. **union**: Merges two sets, using union by rank to keep the tree shallow.
- **Time Complexity**:
  - **make_sets**: O(n).
  - **find**: O(α(n)) with path compression.
  - **union**: O(α(n)) with union by rank, where α(n) is the inverse Ackermann function (almost constant in practice).

Prim's Algorithm

Time Complexity

- **O(E log V)** using a heap
- **O(E + V log V)** using Fibonacci heap



```
def prim(G, c):
  u ← arbitrary vertex in V
  S ← { u }
  T ← ∅
  while |S| < |V| do
    (u, v) ← min cost edge s.t. u in S and v not in S
    add (u, v) to T
    add v to S
  return T
```

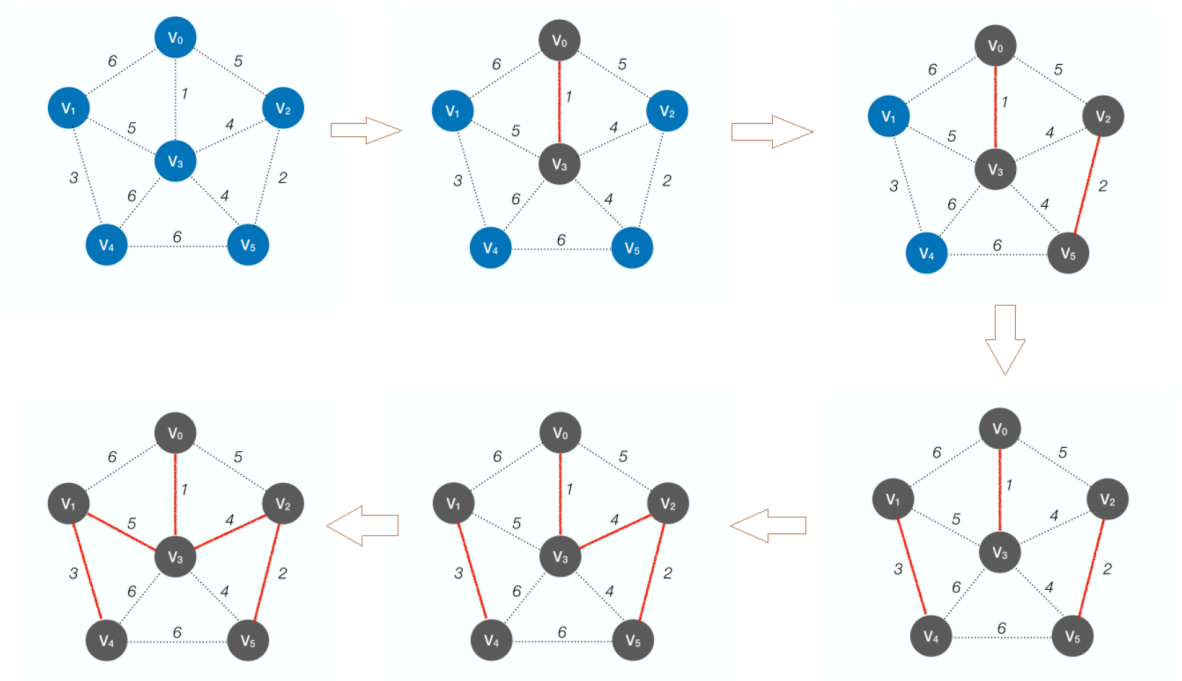
Kruskal's Algorithm

Kruskal's algorithm with a union find implementation

Time Complexity

O(ElogV) → **O(ElogE)**（因为在连通图中，E 最多为V^2，

所以 logE 和 logV 的数量级是相同的）。



```
def Kruskal(G,c):

  sort E in increasing c-value
  answer ← [ ]
  comp ← make_sets(V)
  for (u,v) in E do
    if comp.find(u) ≠ comp.find(v) then
      answer.append( (u,v) )
      comp.union(u, v)
  return answer
```

Union find operations:
- make_sets(A) : one call with |A| = |V|
- find(a) : 2m calls
- union(a,b) : n-1 calls

Shortest Path Problem with Edge Weights and Vertex Costs

- Create a new graph G′ by modifying each edge (u,v) in G.
- In G′, set the weight of (u′,v′) to **edge weight + cost of the destination vertex**.
- **Run Dijkstra's Algorithm** on G′

Fibonacci Heap:

- **Decrease Key Operation**: Triggered when an edge weight is relaxed, with complexity O(m)×O(1)=O(m).
- **Insertion on Vertices**: O(nlogn).
- **Total Complexity with Fibonacci Heap**: O(m+nlogn).

9 -Greedy Algorithms

```
def generic_greedy(input):

  # initialization
  initialize result

  determine order in which to consider input

  # iteratively make greedy choice
  for each element i of the input (in above order) do
    if element i improves result then
      update result with element i

  return result
```

Key Problems and Algorithms

Problem	Greedy Choice	Complexity	Correctness
Fractional Knapsack	Maximize benefit within a weight constraint by taking fractions of items	O(nlogn) for sorting, O(n) for	Optimal (exchange

		processing	argument)
Interval Partitioning	Minimize the number of classrooms needed for lectures, ensuring no overlap in the same room.	O(nlogn) using a priority queue	Optimal (uses priority queue)
Huffman Encoding	Combine lowest frequency characters	O(n+dlogd), where d is the number of distinct characters	Optimal (builds minimum encoding tree)

Theorem: The greedy strategy of picking item with highest benefit to weight ratio computes an optimal solution.

Proof (sketch):

- Use an exchange argument
- Assume for simplicity that all ratios are different $b_i/w_i \neq b_k/w_k$
- Consider some feasible solution x different than the greedy one
- There must be items i and k s.t. $x_i<w_i$, $x_k>0$ and $b_i/w_i > b_k/w_k$
- If we replace some k with some of i, we get a better solution
- How much? $\min\{w_i-x_i, x_k\}$
- Thus, there is no better solution than the greedy one

Thm: Huffman's algorithm computes a minimum length encoding tree of (C, f)

Proof (by induction):

- If $|C| = 1$ then the encoding is trivially optimal
- If $|C| > 1$ then let (C', f') be the contracted instance
- By inductive hypothesis, the encoding tree T' constructed for (C', f') is optimal
- Recall that

$$\sum_{c \text{ in } C} f(c) * depth_T(c) = \sum_{c \text{ in } C'} f'(c) * depth_{T'}(c) + f(i) + f(k)$$

thus, the tree T is optimal for (C, f)

Summary of Concepts

1. **Optimal Solution in an Algorithm**:
    - The algorithm always returns an **optimal solution**.

- It does not always find the minimum difference between two values; instead, it finds the **minimum for each element**.
- This ensures the overall minimum in the difference as well.

2. **Tree Construction with Frequencies f=2if = 2^if=2i**:
   - When given frequencies in the form f=2i, the algorithm constructs a tree where **every internal node has an external node**.

     f=2if = 2^i
   - The algorithm creates a new node and picks the smallest frequency, corresponding to the leaf (external node).
   - Every internal node will always have an external frequency 2i.

     2i2^i
3. **Determining Interval Length in a Set**:
   - To find the interval covering all points in a set, **sort** the points first.
   - For each point, check if it has been covered.
   - If not covered, start from that point and expand the interval to cover other points.
   - **Time Complexity**: O(nlogn) due to sorting.
4. **Optimal Job Scheduling**:
   - To get the optimal schedule, **sort jobs** by the ratio of **job weight to time**.
   - Compute the **ratio of job weight to time** for prioritization.
   - This returns the optimal schedule that maximizes the total weight.
   - **Time Complexity**: O(nlogn), as sorting is required.

10 - Divide and Conquer I

- **Divide**: If it's a base case (like a problem of size 1), solve directly; otherwise, split the problem into sub-problems.
- **Recur/Delegate**: Recursively solve each sub-problem.
- **Conquer**: Combine the solutions of the sub-problems to form the overall solution.

Complexity Analysis via Recurrence Relations

Example Algorithm	Recurrence	Solution	Correctness
Array Binary Search	T(n)=T(n/2)+O(1)	O(logn)	Invariant (x is in A)
Merge Sort	T(n)=2T(n/2)+O(n)	O(nlogn)	Induction
Quick Sort	T(n)=T(nL)+T(nR)+O(n)	O(nlogn) on average	

Binary search correctness

Invariant: If x is in A before the divide step, then x is in A after the divide step

- if A[⌊n/2⌋]>x, then x must be A[0 to ⌊n/2⌋-1]
- if A[⌊n/2⌋]<x, then x must be in A[⌊n/2⌋+1 to n-1]

Every divide step leads to a smaller array.

Thus, if x in A, we will eventually inspect its position due to the invariant and return "Yes".

Thus, if x in not in A, then eventually we reach the empty array and return "No".

Merge: Correctness

Induction hypothesis:

- After the i-th iteration, our result contains the i smallest elements in sorted order

Base case:

- After 0 iterations, our result is empty, so it contains the 0 smallest elements in sorted order

Induction:

- Assume IH after iteration k, to prove it after iteration k+1
- Since both halves are sorted and we add the smallest element not already in result, result now contains the k+1 smallest elements
- Sorted order follows from the fact that both halves are sorted, thus adding the smallest element implies sorted order of result

Merge-Sort: Correctness

Induction hypothesis:

- Merge-Sort correctly sorts an array of size i

Base case:

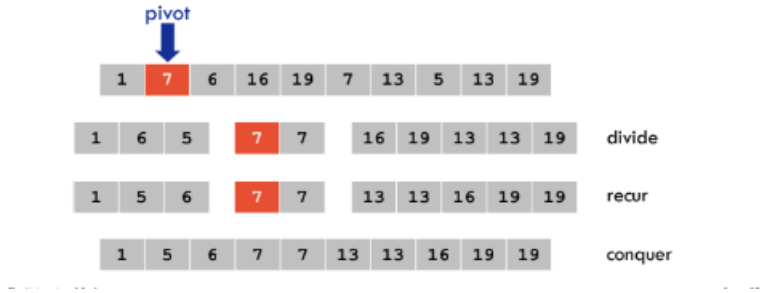
- If our array has size 0 or 1, it's already sorted

Induction:

- Assume IH for all arrays up to size k, to prove it for array of size k+1
- Splitting the array in half gives us two array of size at most k, so by IH those are sorted correctly
- We proved that given two sorted arrays, Merge returns a correctly sorted array containing the elements of both arrays
- Hence, by running Merge on the two sorted halves, we sort the original array

Quick sort

1. Divide Choose a random element from the list as the pivot
   Partition the elements into 3 lists:
   (i) less than, (ii) equal to and (iii) greater than the pivot
2. Recur Recursively sort the less than and greater than lists
3. Conquer Join the sorted 3 lists together



Proof by Unrolling - T(n)=2T(n/2) + O(n)

Hence, the recursion is:

$$T(n) = \begin{cases} T(n) = 2T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

which solves to

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n) = c \cdot n + 2 \cdot c \cdot \frac{n}{2} + 4 \cdot c \cdot \frac{n}{4} + \ldots = O(n \log n)$$

Simplifying the Series: The recursive term $T\left(\frac{n}{2^k}\right)$ reaches the base case when $\frac{n}{2^k} = 1$, which implies $k = \log_2(n)$. Substituting $k = \log_2(n)$ gives:

$$T(n) = n \cdot T(1) + c \cdot n \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{n}\right)$$

Some recurrence formulas with solutions

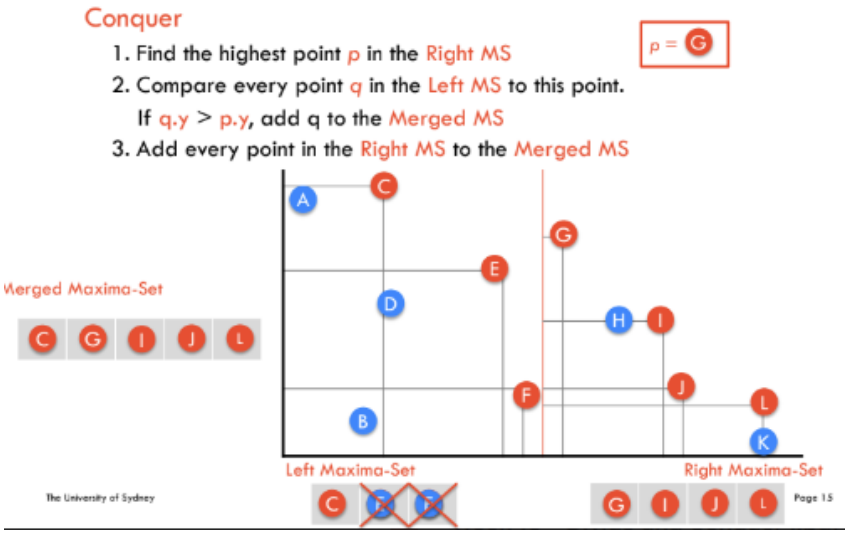
Recurrence	Solution
T(n) = 2 T(n/2) + O(n)	T(n) = O(n log n)
T(n) = 2 T(n/2) + O(log n)	T(n) = O(n)
T(n) = 2 T(n/2) + O(1)	T(n) = O(n)
T(n) = T(n/2) + O(n)	T(n) = O(n)
T(n) = T(n/2) + O(1)	T(n) = O(log n)
T(n) = T(n-1) + O(n)	T(n) = $O(n^2)$
T(n) = T(n-1) + O(1)	T(n) = O(n)

11 - Divide and Conquer II

Maxima-Set (Pareto Frontier)

1. **Preprocessing**: Sort points by x-coordinate (and y-coordinate to break ties).
2. **Divide**: Split into two halves.
3. **Recur**: Find maxima for each half.
4. **Conquer**: Combine results by comparing points on the left to the highest point on the right.

- **Time Complexity**: O(nlogn) due to sorting and merging.



Integer Multiplication

Let $x = x_1\ 2^{n/2} + x_0$ and $y = y_1\ 2^{n/2} + y_0$

$$x\ y = x_1\ y_1\ 2^n + (x_1\ y_0 + x_0\ y_1)\ 2^{n/2} + x_0\ y_0$$
$$(x_1 + x_0)\ (y_1 + y_0) = x_1\ y_1 + x_1\ y_0 + x_0\ y_1 + x_0\ y_0$$

We can compute the product of two n-digit numbers by making 3 recursive calls on n/2-digit numbers and then combining the solutions to the subproblems.

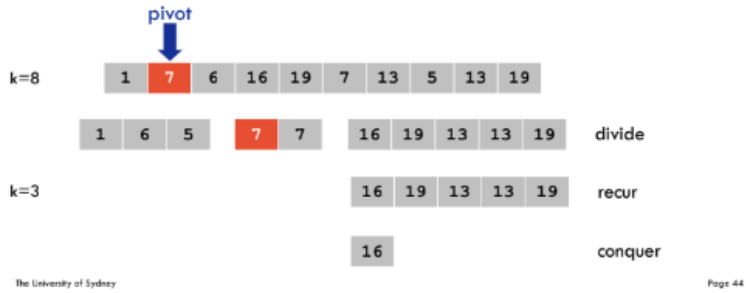
Divide step (produce halves) takes O(n)
Recur step (solve subproblems) takes 3 T(n/2)
Conquer step (add up results) takes O(n)

$$T(n) = \begin{cases} 3\,T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $T(n) = O(n^{\log_2 3})$, where $\log_2 3 \approx 1.6$
Better than naïve!!!

Selection (Median of Medians)

1. Divide Choose a random element from the list as the pivot
   Partition the elements into 3 lists:
   (i) less than, (ii) equal to and (iii) greater than the pivot
2. Recur Recursively select right element from correct list
3. Conquer Return solution to recursive problem



Master Theorem

Master Theorem

先看主定理的内容：
设某个递归算法的时间复杂度递归公式为：$T(n) = a * T(\frac{n}{b}) + n^d$,其中$a > 1, b > 1, d > 0$。则有：

- 当$a < b^d$,也就是$log_b\, a < d$时，$T(n) = O(n^d)$
- 当$a = b^d$,也就是$log_b\, a = d$时，$T(n) = O(n^{log_b\, a} * logn)$
- 当$a > b^d$,也就是$log_b\, a > d$时，$T(n) = O(n^{log_b\, a})$

Let f(n) and T(n) be defined as follows:

$$T(n) = \begin{cases} a\,T(n/b) + f(n) & \text{for } n \geq d \\ c & \text{for } n < d \end{cases}$$

Depending on a, b and f(n) the recurrence solves to:

1. if $f(n) = O(n^{\log_b a - \varepsilon})$ for $\varepsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$,
2. if $f(n) = \Theta(n^{\log_b a} \log^k n)$ for $k \geq 0$ then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$,
3. if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ and $a\,f(n/b) \leq \delta\, f(n)$ for $\varepsilon > 0$ and $\delta < 1$ then $T(n) = \Theta(f(n))$,

Note: If f(n) is given as big-O, you can only conclude T(n) as big-O (not Θ).

Note: You should be able to solve all recurrences in this class using unrolling, but if you are comfortable using the Master Theorem, go for it.