Recurrence Relations
1. Unroll to find pattern
2. Write formula for kth unroll
3. Solve for # of rolls to reach base case
4. Plug into general formula

- $T(n) = 2T\left(\frac{n}{2}\right) + cn$
  $= 2\left(2T\left(\frac{n}{4}\right) + \frac{cn}{2}\right) + cn = 4T\left(\frac{n}{4}\right) + 2cn$
  $= 4\left(2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right) + 2cn = 8T\left(\frac{n}{8}\right) + 3cn$
  ... $= 2^k T\left(\frac{n}{2^k}\right) + kcn$

Terminates when $\frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow k = \log_2 n$

Thus: $T(n) = 2^k T\left(\frac{n}{2^k}\right) + kcn = n\,T(1) + cn\log_2 n$
$= \Theta(n \lg n)$

Properties

- Assume all functions we consider are positive functions
- $f(n) + g(n) = \Theta(\max(f(n), g(n))$
- $f(n) + O(f(n)) = \Theta(f(n))$
- If $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$, then
  - $f_1(n) + f_2(n) = \Theta(g_1(n) + g_2(n)) = \Theta(\max(g_1(n), g_2(n))$

- If $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$, then
  - $f_1(n) \times f_2(n) = \Theta(g_1(n) \times g_2(n))$

Some useful relations

- For any two constant $a, b > 1$
  - $\log_a n = \Theta(\log_b n) = \Theta(\lg n)$
- $1 + 2 + 3 + \cdots + n = \sum_{i=1}^{n} i = \Theta(n^2)$ (Arithmetic sum)
- $1 + 2^2 + 3^2 + \cdots + n^2 = \sum_{i=1}^{n} i^2 = \Theta(n^3)$
- $1 + 2^d + 3^d + \cdots + n^d = \sum_{i=1}^{n} i^d = \Theta(n^{d+1})$
- $\lg 1 + \lg 2 + \cdots + \lg n = \lg n! = \Theta(n \lg n)$
- $1 + \frac{1}{2} + \left(\frac{1}{2}\right)^2 + \cdots + \left(\frac{1}{2}\right)^m = \Theta(1)$ (Geometric sum)
- For any $0 < r < 1$, $1 + r + r^2 + \cdots r^m = \frac{1 - r^{m+1}}{1 - r} = \Theta(1)$
- For any $r > 1$, $1 + r + r^2 + \cdots r^m = \frac{r^{m+1} - 1}{r - 1} = \Theta(r^m)$

Binary Search:
- Array must be sorted
- O(1) best case (find element immediately)
- Θ(log n) worse, average case

BST:
- h = O(n) and h = Ω(lg n)
- balanced BST will be Θ(log n) for all operations, where h = log n
- Θ(h) for all operations

T(n) denote the worst case time complexity
- each recursive call will take us one level down
- total amount of levels we can go is bounded by the height of the tree, h = O(n

T(n)=4T(n/4) + n
T(1)=1

Solution:

$$\begin{aligned} T(n) &= 4 \cdot T(n/4) + n \\ &= 4\,[4 \cdot T(n/16) + n/4] + n \\ &= 16 \cdot T(n/16) + 2n \\ &= 16\,[4 \cdot T(n/64) + n/16] + 2n \\ &= 64 \cdot T(n/64) + 3n \end{aligned}$$

We can infer that in the k-th step, we have:

$$= 4^k \cdot T(n/4^k) + k \cdot n$$

The base case will be reached when $n/4^k = 1$, that is, when $k = \log_4 n$. Substituting this value of k into the general expression:

$$\begin{aligned} T(n) &= 4^{\log_4 n} \cdot T(n/4^{\log_4 n}) + n \cdot \log_4 n \\ &= n \cdot T(n/n) + n \cdot \log_4 n \\ &= n \cdot T(1) + n \cdot \log_4 n \end{aligned}$$

Since $T(1) = 1$, we have:

$$\begin{aligned} &= n + n \cdot \log_4 n \\ &= \Theta(n \log_4 n) \end{aligned}$$

Since logarithms of different bases differ only by a constant factor, we typically omit the base when using asymptotic notation:

$$= \Theta(n \log n)$$

)

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= [T(n-2) + n - 1] + n \\ &= T(n-2) + 2n - 1 \\ &= [T(n-3) + n - 2] + 2n - 1 \\ &= T(n-3) + 3n - (1+2) \\ &= [T(n-4) + n - 3] + 3n - (1+2) \\ &= T(n-4) + 4n - (1+2+3) \end{aligned}$$

```
def tree_search(node, k):
        # if is null or has it is equal to the key, k
        if node = Null or k = x.key
                return x
        # if the key is less than the node's key, go left
        if k < x.key
                return tree_search(x.left, k)
        # otherwise go right
        else:
                return tree_search(x.right, k)
def insert(node, k):
        # initialize pointers
        y = Null
        x = node.root
        # initialize new leaf node z
        z.key = k
        z.left, z.right = Null

        # iterative tree search for position of new node, z
        # locate parent y of z
        while x is not Null: # total iterations <= height
                y = x
                if z.key < x.key:
                        x = x.left
                else
                        x = x.right
```

```
        # setup z as appropriate child of y
        z.parent = y
        if y is Null:
                node.root = z
        elif z.key < y.key
                y.left = z
        else
                y.right = z
def augment_size(treenode x):
        # post order traversal
        if x is not null:
                # recursively set up size info for all left sub tree nodes
                left_size = augment_size(x.left)
                # same for right
                right_size = augment_size(x.right)
                x.size = left + right_size + 1
                return x.size
        return 0
```

Selection Sort

- Repeatedly search for the minimum in the non-sorted part of the array
- When you repeatedly search the minimum it takes n+(n-1)+...+1 because you iterate until the end of the unsorted part each time so you get (n(n+1))/2 = (n^2+n)/2 which is in O(n^2)
- Θ(n^2) best, worse, and average case

```
def selection_sort(A):
        n = len(A)
        if n <= 1:
                return
        for barrier_id in range(n-1):
                # find index of min in A[start:]
                min_id = find_minimum(A, start=barrier_id)
                #swap
                A[barrier_id], A[min_id] = (
                        A[min_id], A[barrier_id]
                )
```

Merge Sort

- find the middle element and index of the input array
- divide and conquer by halfing the subarrays recursively until they are of size 1 Θ(log n)
- merge in a sorted order in linear time Θ(n)
- Θ(n log n) best, worse, average case
- T(n) = 2T(n/2) + cn

```
def MergeSort( A, l, r )
        if (l ≥ r)
                return;
        mid = ⌊(l + r) / 2 ⌋;
        LeftA = MergeSort( A, l, mid );
        RightA = MergeSort( A, mid+1, r );
        B = Merge(LeftA, RightA);
        return B
```

Quick Sort

- Faster than merge sort (usually) and uses less memory
- same partitioning as quick select
- we can do random partitioning too to avoid the worst case because we pick more balanced partitions (sub arrays are roughly same size)
- Worst Case: Θ(n^2) we pick smallest / largest as our pivot
- Best, Average Case: Θ(n logn)

Quick Select

- find kth smallest / largest element
- Select a pivot (randomly or first/last element)
- rearrange so that elements less / greater are on left / right
- Worst Case: Θ(n^2) we pick smallest / largest as our pivot
- Best, Average Case: Θ(n) partition into two halves and “eliminate” one

---

Solution: $\theta(n^2)$

The recurrence relation for the modified mergesort is $T(n) = n^2 + 2T(n/2)$. Let's try to solve the recurrence by unrolling the loop.

$$
\begin{aligned}
T(n) &= 2T(n/2) + n^2 \\
&= 2[2T(n/4) + (\frac{n}{2})^2] + n^2 \\
&= 4T(n/4) + 2(\frac{n^2}{2^2}) + n^2 \\
&= 4T(n/4) + n^2(1 + \frac{1}{2}) \\
&= 4[2T(n/8) + (\frac{n}{4})^2] + n^2(1 + \frac{1}{2}) \\
&= 8T(n/8) + 4(\frac{n^2}{4^2}) + n^2(1 + \frac{1}{2}) \\
&= 8T(n/8) + n^2(1 + \frac{1}{2} + \frac{1}{4})
\end{aligned}
$$

At k-th step, we have $T(n) = 2^k T(n/2^k) + n^2(1 + \frac{1}{2} + \frac{1}{4} + ... + \frac{1}{2^{k-1}})$

Base case is T(1).

$$
\begin{aligned}
n &= 2^k \\
=> logn &= k
\end{aligned}
$$

Substituting the value of k, we get

$$
\begin{aligned}
T(n) &= 2^{logn} T(1) + n^2(1 + 1/2 + ...) \\
&= \theta(n^2)
\end{aligned}
$$

Solution: No slicing is being done, so a constant amount of time is required outside of the recursive calls. Therefore the recurrence relation is:

$$T(n) = 2T(n/2) + \Theta(1)$$

We have not seen this recurrence before, so we must solve it by unrolling. For simplicity, assume $T(n) = 2T(n/2) + 1$. If we perform unrolling k times, we will get the following:

$$T(n) = 2^k T(n/2^k) + 1 + 2 + 4 + \cdots + 2^{k-1}.$$

This unrolling stops when we have $n/2^k = 1$, meaning $2^k = n$ and $k = \log_2 n$. Furthermore, note that $1 + 2 + 4 + 8 + \cdots + 2^{k-1} = 2^k - 1$ (geometric sum). We thus have

$$T(n) = nT(1) + 2^k - 1 = n \cdot \Theta(1) + n - 1 = \Theta(n).$$

It will have $\Theta(n)$ as its solution.

Notice that this recurrence is similar to the recurrence for binary search, which was $T(n/2)+\Theta(1)$, and whose solution was $\Theta(\log n)$. The difference between the two is the factor of 2 on the $T(n/2)$; binary search does not have this because it only makes one recursive call. This difference has a big effect: it reduces the time complexity from $\Theta(n)$ all the way down to $\Theta(\log n)$.

What makes quicksort O(n log n) and quickselect is O(n) is that we always need to explore all branches of the recursive tree for quicksort and only a single branch for quickselect. Let's compare the time complexity recurrence relations of quicksort and quickselect to prove my point.

Quicksort:

```
T(n) = n + 2T(n/2)
     = n + 2(n/2 + 2T(n/4))
     = n + 2(n/2) + 4T(n/4)
     = n + 2(n/2) + 4(n/4) + ... + n(n/n)
     = 2^0(n/2^0) + 2^1(n/2^1) + ... + 2^log2(n)(n/2^log2(n))
     = n (log2(n) + 1)      (since we are adding n to itself log2 + 1 times)
```

Quickselect:

```
T(n) = n + T(n/2)
     = n + n/2 + T(n/4)
     = n + n/2 + n/4 + ... n/n
     = n(1 + 1/2 + 1/4 + ... + 1/2^log2(n))
     = n (1/(1-(1/2))) = 2n                          (by geometric series)
```

Hashmaps

Intuitively, should spread elements uniformly; if there are n elements and m slots, each slot will have n / m elements

Complexity

Search: ET = Theta(1 + n/m), worst case **Theta(n)**

Insert: **Theta(1)**

Delete: **Theta(1 + n/m)**, worst case Theta(n)

Querying is done using 'in' **Theta(1)** average complexity for dict / set, **O(n)** for list

Conceptual ideas

Load Factor: **a = n / m** (avg num elements per linked list)

Cons

Cannot be used to query a closest value, perform a range query (list of nums in [a, b])

Only support membership, insert, delete

No locality: similar items may map to different bins

BFS

Complexity

*Each node can be **added to the queue exactly once** (maximum of number of nodes)*

*Each node will be **pending exactly once***

Every node in graph can only be ***processed (change from undiscovered to pending) once***

For each node, u we process, we need time proportional to the number of (outgoing) edges incident on u

Range over all nodes (all while loops), ***each edge will be visited exactly once for directed graphs, and twice for undirected graphs***

Thus, total cost in the for loop is O(|E|)

Initializing status takes O(|V|) time in the beginning

Total time complexity takes **O(|V| + |E|)** time

Conceptual ideas

BFS will visit exactly the set of nodes that are reachable from s:

Start at s → look at all reachable nodes → look at their neighbors and who is reachable from each neighbor

→ Therefore, we reach each node neighbor through a bunch of edges that all trail from s

→ This implies that if not all nodes are reachable from s, then some node may not be visited (if disconnected)

How to explore all nodes?

→ restart from an undiscovered node until all nodes are discovered

→ call BFS multiple times, maintain status of the nodes the whole way

→ If a node is undiscovered, perform BFS from that node as the source, continue this for all nodes in graph.nodes

(If the input is an undirected graph with k components, BFS will be called k times)

Can be used to answer questions such as:

Is this input of an undirected graph connected? Is there a path from u to v? BFS(G, U)

Python code

```
def bfs(graph, source):
    Status = {node: 'undiscovered' for node in graph.nodes}
    Status[source] = 'pending'
    # while there are still pending nodes
    While pending:
        U = pending.popleft()
        For v in graph.neighbors(u) # adjust adj. List of u
            If status[v] == 'undiscovered': # explore edge (u, v)
                Status[v] = 'pending'
                pending.append(v)
        Status[u] = 'visited'
```

Shortest Path

Complexity

O(|V| + |E|)

Conceptual ideas

Starting at s, find all nodes distance 1 from s → use those to find all nodes distance 2, 3 … Until found all reachable nodes

To get a node at distance k from s, you have to first reach a node at distance k - 1 from s and extend it via an edge

For any k ≥ 1:

All nodes distance k from s are added to the queue before any node of distance > k

Thus nodes are added to queue in increasing order of distance to the source

BFS Trees:

The path from s to any node u in the BFS Tree is a shortest path

in general we run the full BFS algorithm, then we obtain a collection of BFS-Trees called a BFS Forest

In **BFS, explores nodes in order of their first discovery time**

Because we explore as broad as possible before going deeper, we can easily compute the path to a source node

DFS

Complexity

Each node will be explored exactly once

Each edge will be explored exactly once in directed graphs, twice in undirected graphs

Initialization takes **O(|V|)** time

Total Time Complexity takes **O(|V| + |E|)** time

Conceptual ideas

Always explore the new node first, thus we use a stack

DFS visits all reachable nodes from the source; we can exhaust it to find all nodes like BFS

If G is connected, it will visit all nodes in the component

We cannot retrieve the shortest path since we always go deeper and deeper instead of looking at each existing neighbor first

Nesting Claim A:

take any two nodes u and v, assume start[u] ≤ start[v]. If while exploring u we reached v, then exploring of v has to be done before we finish exploring u

Either start[u] ≤ start[v] ≤ finish[v] ≤ finish[u] **or** start[u] ≤ finish[u] ≤ start[v] ≤ finish[v]

Nesting Claim B:

If a graph $G = (V, E)$ does not have cycles, and node v is reachable from u, then finish[v] ≤ finish[u]

Python code

```
Def full_dfs(graph):
    Status = {'node': 'undiscovered' for node in graph.nodes}
    For node in graph.nodes: # to visit all nodes in a graph, we restart from undiscovered nodes
```

```
        If status[v] == 'undiscovered':
            dfs(graph, v, status)

    Def dfs(graph, u, status = None):
        # start dfs at node u, initialize status
        If status is not None:
            Status = {'node': 'undiscovered' for node in graph.nodes}
        # set u to be pending
        Status[u] = pending
        For v in graph.neighbors(u): # explore edge (u, v)
            If status[v] == 'undiscovered':
                dfs(graph, v, status)
        Status[u] = 'visited'
```

Topological Sort

Complexity

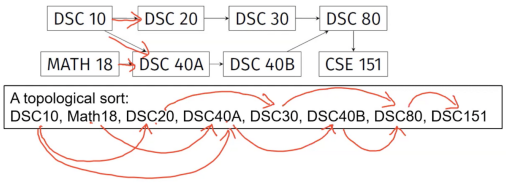
First perform DFS on graph → **O(|V| + |E|)**

output in decreasing order → O(|V|) (don't need to sort; we have an array from 0 to t), if a node takes time in [0, t - 1], we fill the corresponding slot

Conceptual ideas

Only works on DAG because if there was a cycle, then it eliminates valid ordering

If v is reachable from u, then u comes before v in the Topological Sort (nodes w/ later finish time come first



Bellman Ford

Complexity

Setup takes O(V) time; There are O(VE) updates; Each update is constant; Total Complexity is **O(VE)**

Conceptual ideas

If a graph doesn't have negative cycle, then est distances stop after V-1 iterations; If distances continue to decrease after V iterations, they exist

1. Initialize est as inf and predecessors as none
2. For each node, check if current Shortest Path est can be improved, repeat V - 1 nodes

Python code

```
Def bellman_ford(graph, weights, source):
    Def update(u, v, weights, est, predecessor):
        If est[u] + weights[(u, v)] < est:
            Est[v] = est[u] + weights[(u, v)]
            Predecessor[v] = u
    Est = {node: float('inf') for node in graph.nodes}
    Est[source] = 0
    predecessor = {node: None for node in graph.nodes}
    for i in range(len(graph.nodes) - 1):
        # loop through all edges, update if the distance is better
        for (u, v) in graph.edges:
            update(u, v, weights, est, predecessor)
    return est, predecessor
```

Dijkstra

Complexity

Creating priority queue, Number of extract_min: **O(V)**

Costs of extract_min: **O(V log V)**

Number of change_priority: **O(E)**

Costs of change priority: **O(E log V)**

Total: **O((V+E) log V), improved to Θ(E+ V lg V) w/ fibonacci heap**

Conceptual ideas

1. Initialize source w/ distance 0, est: inf, empty set C of correct nodes
2. Create a priority queue / min heap and insert source node
3. While not empty, extract node u w/ min distance → for each neighbor(u) calculate distance through u to neighbor(u)
4. If dist(neighbor(u)) is less than current est, update

Python code

```
Def dijkstra(graph, weights, source):
    # similar setup to BF
    Est = {node: float('inf') for node in graph.nodes}
    Est[source] = 0
    predecessor = {node: None for node in graph.nodes}
    # while priority queue isnt empty, initially insert source
    Priority_queue = PriorityQueue(est)
    While priority_queue:
        u = priority_queue.extract_min()
        for v in graph.neighbors(u):
            changed = update(u, v, weights, est, pred) # returns true and updates the dist if shorter
            if changed:
                priority_queue.change_priority(v, est[v])
     Return est, pred
```

Summary

Graph Traversal / Search: BFS / DFS

O(V + E), BFS can be used to compute SSSP for unweighted graphs, or for graphs where all edges have same edge weight

SSSP

BF for arbitrary graphs: O(VE)

Dijkstra for positively weighted graphs: O((V + E)log V) or imporved to O(E + V log V)