Master Theorem

If \( T(n) = aT([n/b]) + O(n^d) \) for some constants \( a > 0, b > 1, \) and \( d \geq 0, \)

\[
T(n) = \begin{cases} 
O(n^d) & \text{if } d > \log_b a \\
O(n^d \log n) & \text{if } d = \log_b a \\
O(n^{\log_b a}) & \text{if } d < \log_b a.
\end{cases}
\]

✧ This single theorem will give you big-O answers to most recurrence relations for divide and conquer algorithms.
Proof of Master Theorem (simplified)

- Assume that $n$ is a power of $b$
  
  Let's us ignore the rounding in $\left\lfloor \frac{n}{b} \right\rfloor$

- Size of subproblem decrease by factor of $b$
  
  Reaches base case after $\log_b n$ levels

Branching factor is $a$, so $k^{th}$ level of tree has $a^k$

Sub-problems, each of size $n/b^k$
Master Theorem, continued

Each problem of size $n$ is divided into a subproblems of size $n/b$.

Size $n$

Size $n/b$

Size $n/b^2$

Size 1

Branching factor $a$

Depth $\log_b n$

Width $a^{\log_b n} = n^{\log_b a}$
Total work at level $k$ is

$$a^k \times O\left(\frac{n}{b^k}\right)^d = O(n^d) \times \left(\frac{a}{b^d}\right)^k$$

So total work (at all levels) is

$$O(n^d) \times \sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i$$

Big-O of sum of a geometric series?

- ratio < 1: first term $O(n^d)$
- ratio > 1: last term $O\left(n^d \times \left(\frac{a}{b^d}\right)^{\log_b n}\right) =
- ratio = 1: sum of $O(n^d)$ constants
Q2. (20 Points) Master Theorem

For each of the following recurrences, circle the case of the Master Theorem which must be applied in order to solve the recurrence. If case 1 applies, provide your choice of $\in$. If Case 3 applies, provide your choice of $\in$ and $c$. Finally, fill in the solution in the solution box for $T(n)$.

1. $T(n) = 4T\left(\frac{n}{2}\right) + n^2$  
   Case 1  Case 2  Case 3  $\in$  $c$
   
   $T(n) =$  

2. $T(n) = T\left(\frac{5n}{8}\right) + n$  
   Case 1  Case 2  Case 3  $\in$  $c$
   
   $T(n) =$  

3. $T(n) = 5T\left(\frac{n}{2}\right) + n^2$  
   Case 1  Case 2  Case 3  $\in$  $c$
   
   $T(n) =$
Warshall’s Algorithm

- Computes the transitive closure of a relation.
  - reachability in a graph is only an example of such a relation …
Warshall’s Algorithm

**Warshall Algorithm 1**
Warshall($M_R: n \times n$ 0-1 matrix)

$W := M_R$ ($W = [w_{ij}]$)
for($k=1$ to $n$) {
  for($i=1$ to $n$) {
    for($j=1$ to $n$) {
      $w_{ij} = w_{ij} \lor (w_{ik} \land w_{kj})$
    }
  }
}
return $W$

**Warshall Algorithm 2**
Warshall($M_R: n \times n$ 0-1 matrix)

$W := M_R$ ($W = [w_{ij}]$)
for($k=1$ to $n$) {
  for($i=1$ to $n$) {
    for($j=1$ to $n$) {
      if($w_{ik} = 1$) {
        for($j=1$ to $n$) {
          $w_{ij} = w_{ij} \lor w_{kj}$
        }
      }
    }
  }
}
return $W$
Example: $F$, the “Friend” Relation

$p_1$ F $p_2$
$p_3$ F $p_7$
$p_2$ F $p_7$
$p_5$ F $p_2$
$p_4$ F $p_5$
$p_5$ F $p_4$
$p_6$ F $p_1$

$M_R = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0
\end{bmatrix}$
Work the Algorithm:

**Warshall Algorithm 1**

Warshall($M_R$: $n \times n$ 0-1 matrix)

$W := M_R$ ($W = [w_{ij}]$)

for($k=1$ to $n$) {
  for($i=1$ to $n$) {
    for($j=1$ to $n$) {
      $w_{ij} = w_{ij} \lor (w_{ik} \land w_{kj})$
    }
  }
}

return $W$

\[
W = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]
Kruskal’s algorithm:

- Given a connected graph \( G=(V, E) \):
  
  \[
  E_T \leftarrow \text{empty set of edges}
  \]
  
  for each \( v \) in \( V \)
  
  make a singleton set \( \{v\} \)

  sort the edges of \( E \) by nondecreasing weight

  for each edge \((u,v)\) in \( E \)
  
  if \( S_u \neq S_v \), then
  
  replace \( S_u \) and \( S_v \) with \( S_u \cup S_v \)
  
  add \((u,v)\) to \( E_T \)

- Complexity is \( O(|E| \log |E|) \).
  - (With our simple union-find, more like \( O(|E|^2) \))
How does this work?

- Suppose that C and D are the two connected components in the forest $(V,E_T)$ that are connected by an edge $(u,v)$.

- Then $(u,v)$ must have the least weight of any edge between C and D (otherwise C and D would have already been connected).
Your turn!

Apply Kruskal’s algorithm to this graph:
Your turn!

Apply Kruskal’s algorithm to this graph:

Tree edges

<table>
<thead>
<tr>
<th>List of edges (sorted by weight)</th>
</tr>
</thead>
</table>
Your turn!

Apply Kruskal’s algorithm to this graph:

Tree edges | List of edges (sorted by weight)
--- | ---
bc (1) | 

Diagram of the graph with weights:

- Edge bc (weight 1)
- Edge bd (weight 6)
- Edge cd (weight 4)
- Edge ce (weight 2)
- Edge ad (weight 5)
- Edge de (weight 3)
Try it out!
Try it out!
Try it out!
Try it out!
Try it out!
Try it out!
Try it out!
Try it out!
Try it out!
Try it out!
Try it out!
Try it out!
Try it out!
Try it out!
Try it out!
Try it out!
Try it out!
Try it out!
Prim’s algorithm:

- In Kruskal’s algorithm, $E_T$ is a forest whose components are combined as the algorithm runs until just one component remains.
- Suppose instead that we start with an arbitrary vertex $r$, and then add edges while ensuring that $E_T$ is just a single tree at each stage.
- This is the essence of Prim’s algorithm:

let $V_T \leftarrow \{r\}$, $E_T \leftarrow$ empty set  
while $(V_T \neq V)$  
    find a light edge $(u,v)$ for some $u \in V_T$ and $v \notin V_T$  
    add $v$ to $V_T$; add $(u,v)$ to $E_T$  
$(V, E_T)$ is the required MST
Why does this work?

- $V_T$ cuts $V$ into two pieces: $V_T$ and $(V - V_T)$;

- The edges that we add to $E_T$ are light edges across the cut;

- Hence they are safe to add.
Choosing the edges:

- Store all vertices that are not in \((V_T, E_T)\) in a priority queue \(Q\) with an extractMin operation.

- If \(u\) is a vertex in \(Q\), what’s key\([u]\) — the value that determines \(u\)’s position in \(Q\)?
  - key\([u]\) = minimum weight of any edge from \(u\) into \(V_T\)
  - if no such edge exists, key\([u]\) = \(\infty\).

- We maintain information about the parent (in \((V_T, E_T)\)) of each vertex \(v\) in an array parent[].
  - \(E_T\) is kept implicitly as \(\{(v, \text{parent}[v]) \mid v \in V - Q - \{r\}\}\).
The input is the graph $G=(V, E)$, and a root $r \in V$.

```plaintext
for each $v$ in $V$
    key[v] ← \infty;
    parent[v] ← \text{null};
key[r] ← 0;
add all vertices in $V$ to the queue $Q$.

while (Q is nonempty) {
    u ← extractMin(Q);
    for each vertex $v$ that is adjacent to $u$ {
        if $v \in Q$ and weight($u,v$) < key[v] {
            parent[v] ← u;
            key[v] ← weight(u,v);
        }
    }
}
```
The input is the graph $G=(V, E)$, and a root $r \in V$.

```java
for each $v$ in $V$
    $key[v] \leftarrow \infty$;
    $parent[v] \leftarrow null$;
$key[r] \leftarrow 0$;
add all vertices in $V$ to the queue $Q$.

while (Q is nonempty) {
    $u \leftarrow extractMin(Q)$;
    for each vertex $v$ that is adjacent to $u$ {
        if $v \in Q$ and $weight(u,v) < key[v]$ {
            $parent[v] \leftarrow u$;
            $key[v] \leftarrow weight(u,v)$;
        }
    }
}
```

How can this test be implemented in $O(1)$?
The input is the graph $G=(V, E)$, and a root $r \in V$.

for each $v$ in $V$
  $key[v] \leftarrow \infty$;
  $parent[v] \leftarrow \text{null}$;
$key[r] \leftarrow 0$;
add all vertices in $V$ to the queue $Q$.

while (Q is nonempty) {
  $u \leftarrow \text{extractMin}(Q)$;
  for each vertex $v$ that is adjacent to $u$ {
    if $v \in Q$ and $\text{weight}(u,v) < key[v]$ {
      $parent[v] \leftarrow u$;
      $key[v] \leftarrow \text{weight}(u,v)$;
    }
  }
}

How can this test be implemented in $O(1)$?

When we change a key, we will also need to readjust the priority queue.
Complexity:

- Assuming a binary heap ...
  - Initialization takes $O(|V|)$ time.
  - Main loop is executed $|V|$ times, and each extractMin takes $O(\log |V|)$.
  - The body of the inner loop is executed a total of $O(|E|)$ times; each adjustment of the queue takes $O(\log |V|)$ time.

- Overall complexity: $O((|V|+|E|) \log |V|) = O(|E| \log |V|)$. 
Your turn!

Apply Prim’s algorithm to this graph:
Your turn!

✧ Apply Prim’s algorithm to this graph:

```
<table>
<thead>
<tr>
<th>Tree vertices</th>
<th>Priority Queue of remaining vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>
```

Diagram:
- Edge weights:
  - ab: 5
  - ac: 9
  - ad: 3
  - ae: 5
  - be: 6
  - bc: 1
  - bd: 2
  - cd: 3

- Initial state:
  - Tree vertices: a
  - Priority Queue of remaining vertices: {b, c, d, e}

- Next vertex to add to the tree: b
- Add b to the tree and update the priority queue:
  - Tree vertices: {a, b}
  - Priority Queue of remaining vertices: {c, d, e}

- Next vertex to add to the tree: c
- Add c to the tree and update the priority queue:
  - Tree vertices: {a, b, c}
  - Priority Queue of remaining vertices: {d, e}

- Next vertex to add to the tree: d
- Add d to the tree and update the priority queue:
  - Tree vertices: {a, b, c, d}
  - Priority Queue of remaining vertices: {e}

- Next vertex to add to the tree: e
- Add e to the tree and update the priority queue:
  - Tree vertices: {a, b, c, d, e}
  - Priority Queue of remaining vertices: {}
Your turn!

Apply Prim’s algorithm to this graph:

Tree vertices | Priority Queue of remaining vertices
---|---
a(−, −) |
Your turn!

Apply Prim’s algorithm to this graph:
Your turn!

Apply Prim’s algorithm to this graph:
Apply Prim’s Algorithm

Start here!
Try it out!
## P vs. NP

<table>
<thead>
<tr>
<th>P</th>
<th>NP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorting</td>
<td>Knapsack</td>
</tr>
<tr>
<td>Convex Hull</td>
<td>3SAT</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>TSP</td>
</tr>
</tbody>
</table>
Quicksort

Q U I C K S O R T
Q1. (20 Points) True/False Section (Circle the correct answer)

1. \( n! = o(n! \) )
2. \( c^{\lg n} = \Omega (n^{\lg c} ) \) where \( c > 1 \)
3. \( \frac{1}{2^{\lg n}} = O(1) \), for \( 0 < n \leq 1 \)
4. \( 2n^2 = o(n^2 \) )
5. Strassen's algorithm has complexity \( \Theta(n^{\log_2 7} ) \)
6. \( \text{NP} \) is the class of decision problems solvable in polynomial time
7. \( 5^{\log_5 n} = \Omega (\log_5 n) \)
8. \( 2^n = o(n! \) )
9. A trivial lower bound for any algorithm that solves \( n \times n \) matrix by matrix multiply is \( n^3 \)
10. NP is the class of decision problems solvable in polynomial time
7. \( 5^{\log_5 n} = \Omega(\log_5 n) \)  
   T  F

8. \( 2^n = o(n!) \)  
   T  F

9. A trivial lower bound for any algorithm that solves 
   \( n \times n \) matrix by matrix multiply is \( n^3 \)  
   T  F

10. Every weighted graph has a unique minimum spanning tree  
    T  F

11. Computing the determinant of a matrix by Gaussian Elimination 
    algorithm has efficiency \( \Theta(n^2) \)  
    T  F

12. The best case running time of bubble sort (with short circuit) is \( \Theta(n) \)  
    T  F

13. There is exactly one essentially complete binary tree with \( n \) nodes  
    T  F

14. \( C(n, k) = C(n-1, k-1) + C(n-1, k + 1) \), for \( n > k > 0 \)  
    T  F

15. A tight lower bound for the element uniqueness problem is \( \Omega(n \lg n) \)  
    T  F

16. \( \sum_{i=1}^{n} \lg i = \Theta(n^2 \lg n) \)  
    T  F
17. Euclid’s algorithm for finding the greatest common divisor of two positive integers is an example of decrease by a variable amount algorithm  \[ T \quad F \]

18. \[ \sum_{i=0}^{n-1} \frac{i(i + 1)}{2} = \frac{(n - 1)n(2n + 1)}{6} \]  \[ T \quad F \]

19. The problem of computing the least common multiple of two positive integers can be computed using Euclid’s algorithm for the greatest common divisor.  \[ T \quad F \]

20. An Euler circuit is a cycle in a graph that traverses every vertex of the graph exactly once  \[ T \quad F \]

21. \( \mathbf{P} \) is the class of optimization problems solvable in polynomial time  \[ T \quad F \]