# Project 13- CIDR Subnet & Supernet Project – Documentation

Name: Rifat

Reg No:21BRS1183

## Aim

The aim of this project is to design and implement an interactive CIDR (Classless Inter-Domain Routing) Subnet and Supernet simulator using Python and Streamlit. The tool helps users understand and plan IPv4 networks by supporting different subnetting strategies (equal split, VLSM, and hierarchical subnetting) and supernetting (CIDR aggregation), along with explanatory steps, tables, and visual aids.

## Code

The following Python code is used in Google Colab to set up the environment, launch the Streamlit-based CIDR Subnet & Supernet Project, and expose it via a Cloudflared tunnel:

```
# -*- coding: utf-8 -*-
"""Untitled6.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1FJgmv6znH6898dYOVInO0Y96dpMjZi1Y
"""

# ====== Install deps (no websockets conflicts) ======
!pip -q install streamlit==1.39.0 netaddr==1.3.0 pandas matplotlib

# ====== Get cloudflared binary ======
import os
if not os.path.exists("cloudflared"):
    !wget -q
https://github.com/cloudflare/cloudflared/releases/latest/download/cloudflared-
linux-amd64 -O cloudflared
    !chmod +x cloudflared

# ====== Ask Colab to upload guide image once, save as guide_image.png ======
from google.colab import files
```

```
if not os.path.exists("guide_image.png"):
    print("□ Please upload your guide image (any file name). It will be saved
as 'guide_image.png'.")
    uploaded = files.upload()
    if uploaded:
        fname = list(uploaded.keys())[0]
        os.rename(fname, "guide_image.png")
        print(f"✓ Saved '{fname}' as 'guide_image.png'.")

# ====== Write the Streamlit app with title & features ======
app_code = r'''
import streamlit as st
import pandas as pd
import math
import ipaddress as ipa
import matplotlib.pyplot as plt
import os

STUDENT_NAMES = "Ryan and Rifat"
GUIDE_NAME   = "Dr Swaminathan Annadurai"
GUIDE_IMAGE_PATH = "guide_image.png"  # Already uploaded via Colab

st.set_page_config(page_title="CIDR Subnet & Supernet Project", layout="wide")

# ---------- Basic CSS for colors / aesthetics ----------
st.markdown("""
<style>
body {
    background-color: #020617;
}
section[data-testid="stSidebar"] {
    background-color: #020617;
}
.header-banner {
    background-color: #f9fafb;
    border-radius: 16px;
    padding: 12px 18px;
    border: 1px solid #e5e7eb;
}
.big-title {
    font-size: 32px;
    font-weight: 800;
    color: #000000;  /* black title as requested */
}
.sub-title {
    font-size: 14px;
    color: #111827;
}
.card {
    background: linear-gradient(135deg, #020617 0%, #111827 50%, #020617 100%);
    border-radius: 16px;
    padding: 14px 18px;
    border: 1px solid #1f2937;
}
.section-title {
```

```
    font-size: 20px;
    font-weight: 700;
    color: #f9fafb;
}
</style>
""", unsafe_allow_html=True)

# ---------- Helper functions ----------
def ip_int_to_dotted(n: int) -> str:
    return ".".join(str((n >> (8*i)) & 255) for i in [3,2,1,0])

def prefix_to_mask(prefix: int) -> int:
    return (0xFFFFFFFF << (32 - prefix)) & 0xFFFFFFFF

def mask_to_dotted(mask_int: int) -> str:
    return ip_int_to_dotted(mask_int)

def parse_ip_decimal(ip_text: str) -> str:
    s = ip_text.strip()
    ipa.IPv4Address(s)  # validate
    return s

def usable_host_count(prefix: int) -> int:
    if prefix == 31:
        return 2
    if prefix == 32:
        return 1
    return max(0, (2**(32-prefix)) - 2)

def first_last_usable(nw: int, bc: int, prefix: int):
    # returns int addresses
    if prefix == 32:
        return (nw, nw)
    if prefix == 31:
        return (nw, bc)
    if bc - nw + 1 < 3:
        return (None, None)
    return (nw+1, bc-1)

def dotted_to_int(ip: str) -> int:
    parts = [int(x) for x in ip.split(".")]
    n = 0
    for p in parts:
        n = (n << 8) | p
    return n

def visualize_subnets(new_prefix: int, count: int):
    fig, ax = plt.subplots(figsize=(10,1.2))
    ax.set_facecolor("#020617")
    fig.patch.set_facecolor("#020617")
    ax.set_xlim(0,1); ax.set_ylim(0,1); ax.axis('off')
    if count <= 0: return fig
    width = 1.0 / count
    for i in range(count):
        ax.add_patch(plt.Rectangle((i*width, 0.1), width-0.002, 0.8,
```

```
edgecolor="#f97316", facecolor="#1f2937"))
        ax.text(i*width + width/2, 0.5, f"{i+1}", ha='center', va='center',
fontsize=9, color="#e5e7eb")
    ax.set_title(f"Subnet Visualization: /{new_prefix} (×{count})", pad=8,
color="#e5e7eb")
    return fig

# ---------- Sidebar: guide image + names ----------
with st.sidebar:
    st.markdown("### □ Project Guide", unsafe_allow_html=True)
    if os.path.exists(GUIDE_IMAGE_PATH):
        st.image(GUIDE_IMAGE_PATH, use_column_width=True)
    else:
        st.warning(f"Upload '{GUIDE_IMAGE_PATH}' in Colab Files to display the
guide image.")
    st.markdown(f"**{GUIDE_NAME}**")
    st.markdown(f"**Developed by {STUDENT_NAMES}**")
    st.markdown("---")
    st.markdown("**Reference Table: Prefix vs Hosts**")
    pref_data = []
    for p in range(24, 31):
        total = 2**(32-p)
        usable = usable_host_count(p)
        pref_data.append({"Prefix": f"/{p}", "Total addresses": total, "Usable
hosts": usable})
    st.dataframe(pd.DataFrame(pref_data), use_container_width=True)

# ---------- Header ----------
with st.container():
    st.markdown('<div class="header-banner">', unsafe_allow_html=True)
    c0, c1, c2 = st.columns([3.5, 1, 1])
    with c0:
        st.markdown('<div class="big-title">CIDR Subnet & Supernet
Project</div>', unsafe_allow_html=True)
        st.markdown('<div class="sub-title">Interactive tool for flexible
subnetting, supernetting, and network planning.</div>', unsafe_allow_html=True)
    with c1:
        if st.button("Learn"):
            st.info(
                "This project helps you design subnets (equal, VLSM,
hierarchical) and supernets. "
                "It uses IPv4 CIDR addressing with prefix lengths instead of
classful A/B/C."
            )
    with c2:
        if st.button("Help"):
            st.warning(
                "1) Enter base IP and prefix.\n"
                "2) Choose the number of networks and split mode for
subnetting.\n"
                "3) Use the Supernetting tab to aggregate multiple CIDR
blocks.\n"
                "4) Study the tables and step-by-step tips for explanation."
            )
    st.markdown('</div>', unsafe_allow_html=True)
```

```
st.markdown("<hr/>", unsafe_allow_html=True)

# ---------- Tabs ----------
tab1, tab2 = st.tabs(["□ Subnet Planner", "□ Supernetting"])

# =======================================================
# TAB 1: SUBNET PLANNER
# =======================================================
with tab1:
    st.markdown('<div class="section-title">User Input (Subnet Planner)</div>',
unsafe_allow_html=True)
    st.markdown("<div class='card'>1.1 IP address    1.2 Bits for
Network ID    2. Number of networks</div>", unsafe_allow_html=True)

    col1, col2, col3 = st.columns(3)
    with col1:
        ip_text = st.text_input("1.1 IP Address (dotted decimal)",
value="192.168.10.0")
    with col2:
        net_bits = st.number_input("1.2 Bits allocated for network ID (prefix
length)", min_value=0, max_value=32, value=24, step=1)
    with col3:
        num_networks = st.number_input("2. Number of networks", min_value=1,
value=4, step=1)

    st.markdown("**Split Mode**")
    split_mode = st.radio(
        "",
        ["Split equally", "Split by addresses (VLSM)", "Hierarchical split
(two-level)"],
        index=0,
    )

    host_list_text = ""
    level2_host_text = ""

    if split_mode == "Split by addresses (VLSM)":
        host_list_text = st.text_input(
            "Required usable addresses per network (comma-separated)",
            value="50, 20, 10"
        )
    elif split_mode == "Hierarchical split (two-level)":
        st.markdown(
            "<div class='card'>First, the base block is split equally into the
specified number of networks. "
            "Then <b>Network #1</b> is further split by another set of host
requirements.</div>",
            unsafe_allow_html=True,
        )
        level2_host_text = st.text_input(
            "Level-2: usable addresses for subnets inside Network #1 (comma-
separated)",
            value="50, 30, 10"
        )
```

```
    if st.button("Run Subnet Planner", type="primary"):
        try:
            # Base network
            base_ip = parse_ip_decimal(ip_text)
            base_prefix = int(net_bits)
            base_net = ipa.IPv4Network(f"{base_ip}/{base_prefix}",
strict=False)
            base_net_int = int(base_net.network_address)
            base_bc_int  = int(base_net.broadcast_address)
            base_block_size = base_bc_int - base_net_int + 1

            rows_main = []
            rows_level2 = []
            steps = []

            steps.append(f"1□ Base Network:
{base_net.network_address}/{base_prefix}")
            steps.append(f"   • Network range: {base_net.network_address} –
{base_net.broadcast_address}")
            steps.append(f"   • Total addresses in base block:
{base_block_size}")

            # ===== Split equally =====
            if split_mode == "Split equally":
                N = int(num_networks)
                if N <= 0:
                    st.error("Number of networks must be >= 1.")
                else:
                    bits_needed = math.ceil(math.log2(N))
                    new_prefix = base_prefix + bits_needed
                    if new_prefix > 32:
                        st.error("Cannot allocate that many equal networks from
this base block.")
                    else:
                        child_block_size = 2**(32 - new_prefix)
                        steps.append(f"2□ Equal Split Mode")
                        steps.append(f"   • Requested networks: {N}")
                        steps.append(f"   • Bits borrowed from host part:
{bits_needed}")
                        steps.append(f"   • New prefix per child:
/{new_prefix}")
                        steps.append(f"   • Child block size:
{child_block_size} addresses")

                        for i in range(N):
                            nw = base_net_int + i*child_block_size
                            bc = nw + child_block_size - 1
                            if bc > base_bc_int:
                                break
                            f, l = first_last_usable(nw, bc, new_prefix)
                            total_addr = child_block_size
                            usable_addr = usable_host_count(new_prefix)
                            allocated_addr = usable_addr
                            remaining_addr = total_addr - allocated_addr
```

```
                            lagging_addr = remaining_addr

                            rows_main.append({
                                "Network ID": ip_int_to_dotted(nw),
                                "First address": "-" if f is None else
ip_int_to_dotted(f),
                                "Last address": "-" if l is None else
ip_int_to_dotted(l),
                                "Broadcast address": ip_int_to_dotted(bc),
                                "Total address": total_addr,
                                "Usable address": usable_addr,
                                "Allocated address": allocated_addr,
                                "Remaining address": remaining_addr,
                                "Lagging address": lagging_addr,
                            })

            # ===== Split by addresses (VLSM) =====
            elif split_mode == "Split by addresses (VLSM)":
                if not host_list_text.strip():
                    st.error("Please enter at least one host requirement.")
                else:
                    host_reqs = [int(x.strip()) for x in
host_list_text.split(",") if x.strip()]
                    if len(host_reqs) == 0:
                        st.error("No valid numbers parsed.")
                    else:
                        steps.append("2□ VLSM Mode")
                        steps.append(f"   • Host requirements: {host_reqs}")
                        cur = base_net_int
                        for idx, h in enumerate(host_reqs, start=1):
                            if h <= 0:
                                st.error(f"Host requirement for network {idx}
must be positive.")
                                break

                            needed = h + 2 if h > 2 else h
                            host_bits = math.ceil(math.log2(needed))
                            pre = 32 - host_bits
                            block_size = 2**(32 - pre)
                            align_start = ((cur + block_size - 1) //
block_size) * block_size
                            nw = align_start
                            bc = nw + block_size - 1
                            if bc > base_bc_int:
                                st.error("Requirements do not fit inside the
base block.")
                                break

                            f, l = first_last_usable(nw, bc, pre)
                            total_addr = block_size
                            usable_addr = usable_host_count(pre)
                            allocated_addr = min(h, usable_addr)
                            remaining_addr = usable_addr - allocated_addr
                            lagging_addr = total_addr - h
```

```
                            rows_main.append({
                                "Network ID": ip_int_to_dotted(nw),
                                "First address": "-" if f is None else
ip_int_to_dotted(f),
                                "Last address": "-" if l is None else
ip_int_to_dotted(l),
                                "Broadcast address": ip_int_to_dotted(bc),
                                "Total address": total_addr,
                                "Usable address": usable_addr,
                                "Allocated address": allocated_addr,
                                "Remaining address": remaining_addr,
                                "Lagging address": lagging_addr,
                            })
                            cur = bc + 1

            # ===== Hierarchical split (two-level) =====
            else:
                N = int(num_networks)
                if N <= 0:
                    st.error("Number of networks must be >= 1.")
                else:
                    bits_needed = math.ceil(math.log2(N))
                    new_prefix = base_prefix + bits_needed
                    if new_prefix > 32:
                        st.error("Cannot allocate that many equal networks from
this base block.")
                    else:
                        child_block_size = 2**(32 - new_prefix)

                        steps.append("2⃣ Hierarchical Mode - Level 1")
                        steps.append(f"   • Level-1 equal split into {N}
networks of prefix /{new_prefix}")
                        for i in range(N):
                            nw = base_net_int + i*child_block_size
                            bc = nw + child_block_size - 1
                            if bc > base_bc_int:
                                break
                            f, l = first_last_usable(nw, bc, new_prefix)
                            total_addr = child_block_size
                            usable_addr = usable_host_count(new_prefix)
                            allocated_addr = usable_addr
                            remaining_addr = total_addr - allocated_addr
                            lagging_addr = remaining_addr

                            rows_main.append({
                                "Network ID": ip_int_to_dotted(nw),
                                "First address": "-" if f is None else
ip_int_to_dotted(f),
                                "Last address": "-" if l is None else
ip_int_to_dotted(l),
                                "Broadcast address": ip_int_to_dotted(bc),
                                "Total address": total_addr,
                                "Usable address": usable_addr,
                                "Allocated address": allocated_addr,
                                "Remaining address": remaining_addr,
```

```
                                "Lagging address": lagging_addr,
                            })

                        steps.append("3□ Hierarchical Mode - Level 2 inside
Network #1")
                        if level2_host_text.strip() and len(rows_main) > 0:
                            first_net_ip = rows_main[0]["Network ID"]
                            first_net =
ipa.IPv4Network(f"{first_net_ip}/{new_prefix}", strict=False)
                            lvl2_base_net_int = int(first_net.network_address)
                            lvl2_bc_int = int(first_net.broadcast_address)
                            lvl2_hosts = [int(x.strip()) for x in
level2_host_text.split(",") if x.strip()]
                            steps.append(f"   • Level-2 host requirements:
{lvl2_hosts}")

                            cur = lvl2_base_net_int
                            for idx, h in enumerate(lvl2_hosts, start=1):
                                if h <= 0:
                                    st.error(f"Level-2 host requirement #{idx}
must be positive.")
                                    break
                                needed = h + 2 if h > 2 else h
                                host_bits = math.ceil(math.log2(needed))
                                pre = 32 - host_bits
                                block_size = 2**(32 - pre)
                                align_start = ((cur + block_size - 1) //
block_size) * block_size
                                nw = align_start
                                bc = nw + block_size - 1
                                if bc > lvl2_bc_int:
                                    st.error("Level-2 requirements do not fit
inside Network #1.")
                                    break

                                f, l = first_last_usable(nw, bc, pre)
                                total_addr = block_size
                                usable_addr = usable_host_count(pre)
                                allocated_addr = min(h, usable_addr)
                                remaining_addr = usable_addr - allocated_addr
                                lagging_addr = total_addr - h

                                rows_level2.append({
                                    "Network ID": ip_int_to_dotted(nw),
                                    "First address": "-" if f is None else
ip_int_to_dotted(f),
                                    "Last address": "-" if l is None else
ip_int_to_dotted(l),
                                    "Broadcast address": ip_int_to_dotted(bc),
                                    "Total address": total_addr,
                                    "Usable address": usable_addr,
                                    "Allocated address": allocated_addr,
                                    "Remaining address": remaining_addr,
                                    "Lagging address": lagging_addr,
                                })
```

```
                                cur = bc + 1

            # ---------- Display results + steps & tips ----------
            tips = []
            tips.append("• Prefer VLSM when each subnet needs a different
number of hosts.")
            tips.append("• Hierarchical design is useful when one branch (like
Network #1) needs its own sub-subnets.")
            tips.append("• Watch the 'Remaining' and 'Lagging' addresses
columns to measure address wastage.")

            if len(rows_main) > 0:
                st.markdown('<div class="section-title">Primary Subnet
Plan</div>', unsafe_allow_html=True)
                df_main = pd.DataFrame(rows_main)
                cols_order = [
                    "Network ID",
                    "First address",
                    "Last address",
                    "Broadcast address",
                    "Total address",
                    "Usable address",
                    "Allocated address",
                    "Remaining address",
                    "Lagging address",
                ]
                df_main = df_main[cols_order]
                st.dataframe(df_main, use_container_width=True)

                st.markdown("**Step-by-step Explanation & Tips**")
                st.text_area("Steps", "\n".join(steps), height=250)
                st.info("\n".join(tips))

                # small visualization: only for equal/hierarchical when equal
blocks
                try:
                    if split_mode in ["Split equally", "Hierarchical split
(two-level)"]:
                        child_prefix = base_prefix +
math.ceil(math.log2(int(num_networks)))
                        fig = visualize_subnets(child_prefix, len(df_main))
                        st.pyplot(fig)
                except Exception:
                    pass
            else:
                st.info("No subnets were generated.")

            if len(rows_level2) > 0:
                st.markdown('<div class="section-title">Level-2 Split inside
Network #1</div>', unsafe_allow_html=True)
                df_lvl2 = pd.DataFrame(rows_level2)
                cols_order = [
                    "Network ID",
                    "First address",
                    "Last address",
```

```
                    "Broadcast address",
                    "Total address",
                    "Usable address",
                    "Allocated address",
                    "Remaining address",
                    "Lagging address",
                ]
                df_lvl2 = df_lvl2[cols_order]
                st.dataframe(df_lvl2, use_container_width=True)

        except Exception as e:
            st.error(f"Error: {e}")

# ======================================================
# TAB 2: SUPERNETTING
# ======================================================
with tab2:
    st.markdown('<div class="section-title">Supernetting (CIDR
Aggregation)</div>', unsafe_allow_html=True)
    st.markdown(
        "<div class='card'>Enter multiple CIDR blocks. The tool finds a common
supernet that aggregates them "
        "and shows how many routing entries are reduced.</div>",
        unsafe_allow_html=True,
    )

    nets_text = st.text_area(
        "Enter CIDR networks (comma or newline separated)",
        value="192.168.10.0/26, 192.168.10.64/26\n192.168.10.128/26"
    )

    if st.button("Run Supernetting", type="primary"):
        try:
            raw = [p.strip() for p in nets_text.replace("\\n",",").split(",")
if p.strip()]
            networks = []
            for item in raw:
                if "/" not in item:
                    st.error(f"Missing /prefix in '{item}'.")
                    st.stop()
                networks.append(ipa.IPv4Network(item.strip(), strict=False))

            networks.sort(key=lambda n: int(n.network_address))
            min_addr = int(networks[0].network_address)
            max_bc   = max(int(n.broadcast_address) for n in networks)

            def addr_to_bin(ipv4):
                return "".join(f"{int(o):08b}" for o in str(ipv4).split("."))

            bin_addrs = [addr_to_bin(n.network_address) for n in networks]
            common_len = 0
            for i in range(32):
                if len({b[i] for b in bin_addrs}) == 1:
                    common_len += 1
                else:
```

```
                    break

            # Expand to cover min..max
            def prefix_to_mask_local(prefix: int) -> int:
                return (0xFFFFFFFF << (32 - prefix)) & 0xFFFFFFFF

            agg_prefix = common_len
            while True:
                mask = prefix_to_mask_local(agg_prefix)
                agg_net = min_addr & mask
                agg_bc  = agg_net | (~mask & 0xFFFFFFFF)
                if agg_bc >= max_bc and agg_net <= min_addr:
                    break
                agg_prefix -= 1
                if agg_prefix < 0:
                    raise RuntimeError("Supernetting failed.")

            agg_net_ip = ip_int_to_dotted(agg_net)
            agg_bc_ip  = ip_int_to_dotted(agg_bc)

            rows = []
            for i, n in enumerate(networks, start=1):
                total_addr = int(n.broadcast_address) - int(n.network_address)
+ 1
                usable_addr = usable_host_count(n.prefixlen)
                rows.append({
                    "Input #": i,
                    "Network ID": str(n.network_address),
                    "Broadcast address": str(n.broadcast_address),
                    "Prefix": f"/{n.prefixlen}",
                    "Total address": total_addr,
                    "Usable address": usable_addr,
                })
            df_sup = pd.DataFrame(rows)

            st.markdown("**Original Networks**")
            st.dataframe(df_sup, use_container_width=True)

            st.markdown("**Aggregated Supernet Result**")
            st.markdown(
                f"- Supernet: **{agg_net_ip}/{agg_prefix}**  \n"
                f"- Range: **{agg_net_ip} - {agg_bc_ip}**  \n"
                f"- Original entries: **{len(networks)}**  → After
supernetting: **1**  \n"
                f"- Routing table reduction: **{len(networks)-1} entries**"
            )

            steps_super = []
            steps_super.append("1⃣ Normalize: parse all input CIDR blocks.")
            steps_super.append("2⃣ Convert network addresses to binary and find
the longest common prefix.")
            steps_super.append("3⃣ Expand the common prefix (if needed) so that
one block covers min..max range.")
            steps_super.append("4⃣ Form the aggregated supernet and compare
routing entries.")
```

```
            st.markdown("**Steps & Tips for Supernetting**")
            st.text_area("Steps (Supernetting)", "\n".join(steps_super),
height=200)
            st.info(
                "Tip: Supernetting is mainly used in backbone / ISP routers to
compress multiple contiguous routes "
                "into a single prefix, reducing routing table size."
            )

        except Exception as e:
            st.error(f"Error: {e}")
'''

with open("app.py","w") as f:
    f.write(app_code)

# ====== Start Streamlit in background ======
!nohup streamlit run app.py --server.port 8501 --server.address 0.0.0.0
>/dev/null 2>&1 &
print("✅ Streamlit started on http://localhost:8501")

import re, time, subprocess
from IPython.display import HTML, display

# Kill any previous tunnels (safe)
subprocess.run(["pkill","-f","cloudflared"], check=False,
stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)

def start_tunnel():
    return subprocess.Popen(
        ["./cloudflared","tunnel","--url","http://localhost:8501","--no-
autoupdate","--loglevel","info"],
        stdout=subprocess.PIPE, stderr=subprocess.STDOUT, text=True, bufsize=1
    )

url_pat = re.compile(r"https?://[a-z0-9\-]+\.trycloudflare\.com", re.I)

def get_url_from_proc(p, timeout=60):
    start = time.time()
    while time.time() - start < timeout:
        line = p.stdout.readline()
        if not line:
            time.sleep(0.1); continue
        m = url_pat.search(line)
        if m:
            return m.group(0)
    return None

print("□ Launching resilient Cloudflared tunnel (CIDR CN Project)…")
current_url = None
proc = start_tunnel()
url = get_url_from_proc(proc, timeout=90)
if url:
    current_url = url
```

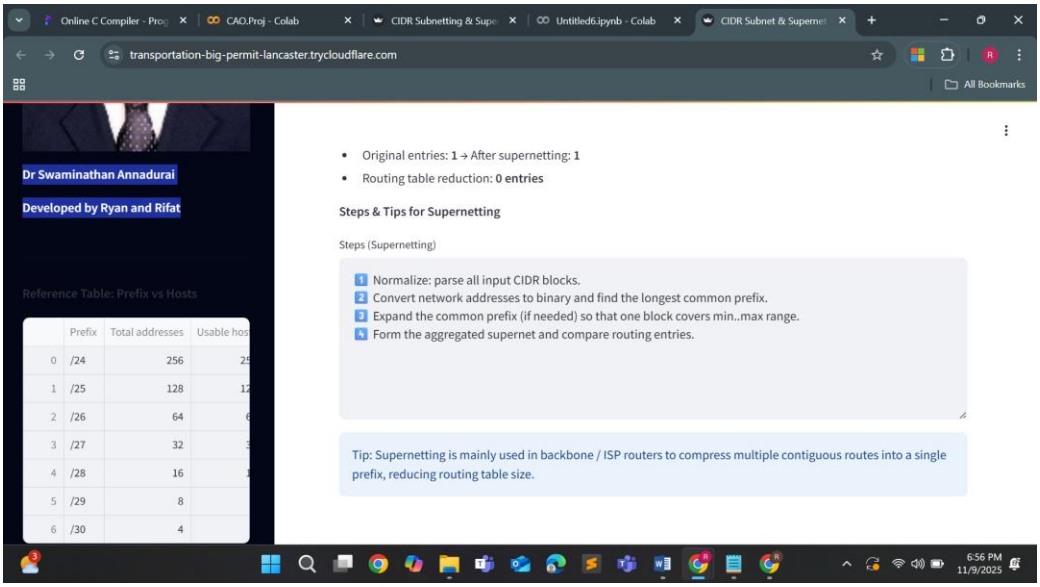
```
    print("\n✅ CIDR CN Project link:\n", current_url)
    display(HTML(f'<h3>✅ CIDR CN Project: <a href="{current_url}"
target="_blank">{current_url}</a></h3>'))
else:
    print("Waiting for URL… (check logs)")

# Keep the cell alive and auto-restart on crash/exit
try:
    while True:
        time.sleep(5)
        if proc.poll() is not None:
            print("\n Tunnel died. Restarting…")
            proc = start_tunnel()
            url = get_url_from_proc(proc, timeout=90)
            if url and url != current_url:
                current_url = url
                print("\n New CIDR CN Project link:\n", current_url)
                display(HTML(f'<h3> New CIDR CN Project link: <a
href="{current_url}" target="_blank">{current_url}</a></h3>'))
except KeyboardInterrupt:
    try:
        proc.terminate()
    except Exception:
        pass
    print("Tunnel stopped.")
```

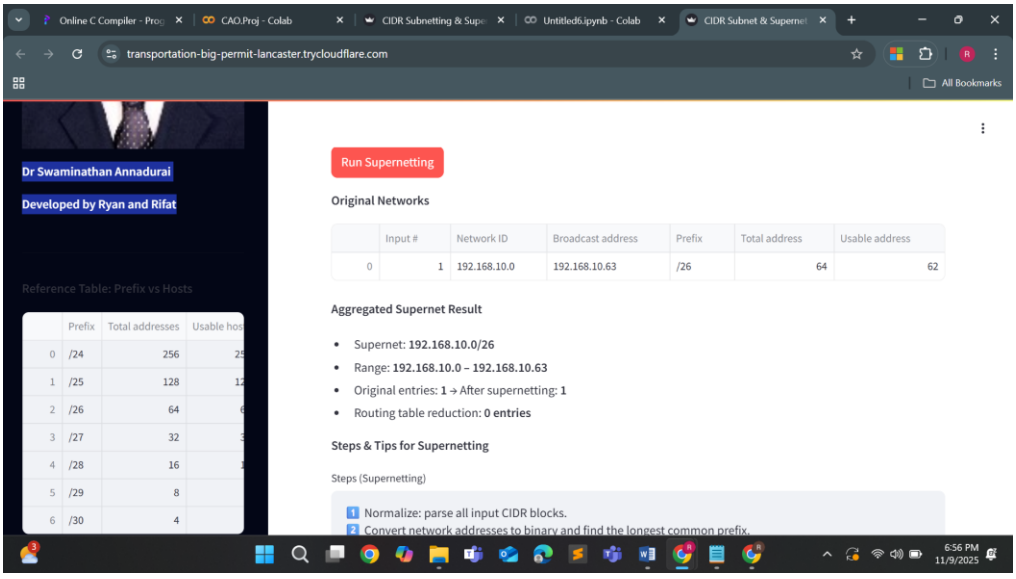
## Features of the Project

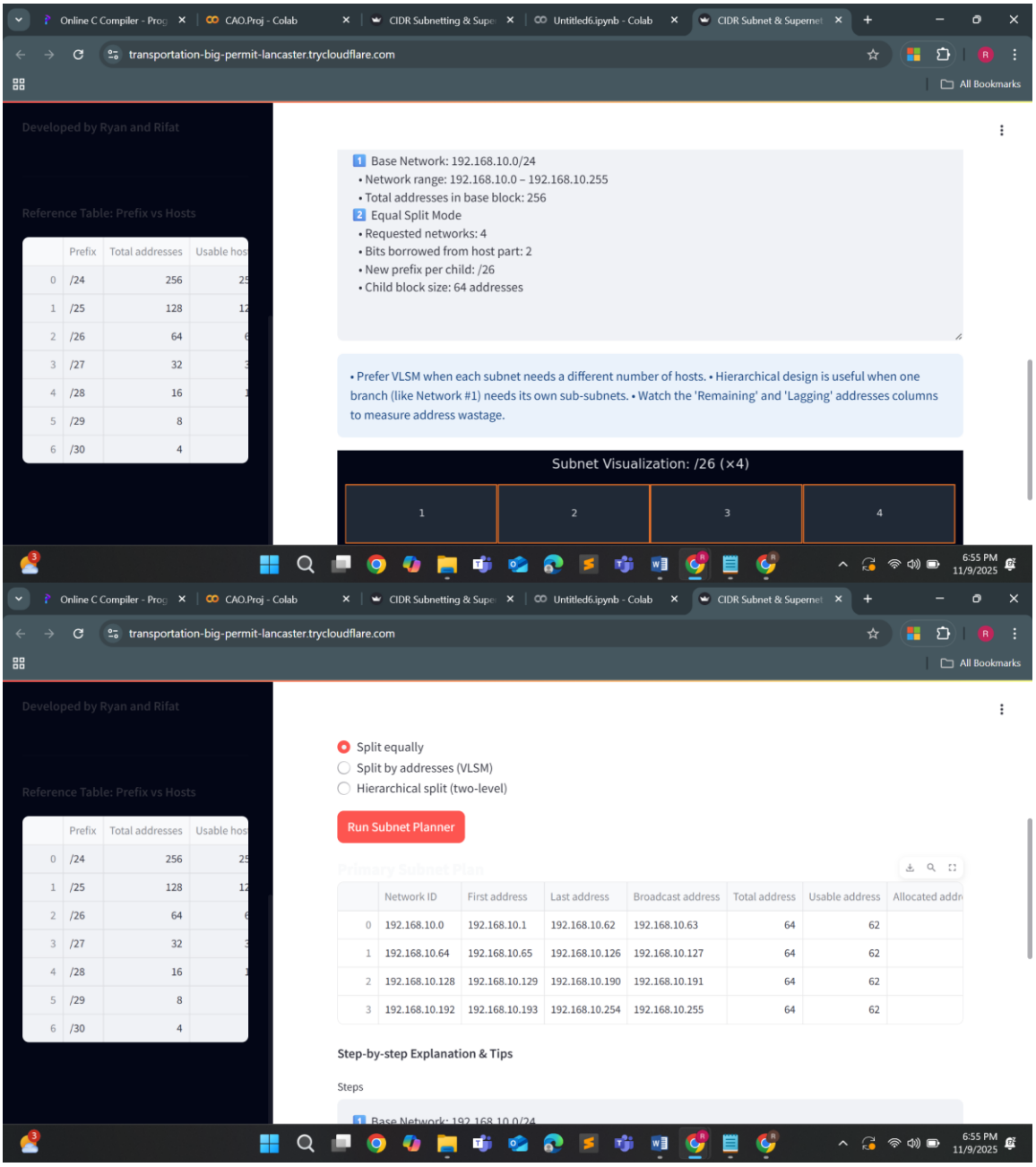
- Interactive CIDR Subnet & Supernet Project built with Python and Streamlit.
- User can input base IPv4 address and number of bits allocated for the network ID (prefix length).
- Supports three subnetting modes: split equally into N networks, split by usable host requirements (VLSM), and hierarchical two-level subnetting (equal split first, then further split of Network #1).
- Displays detailed results for each subnet, including Network ID, first and last usable address, broadcast address, total addresses, usable addresses, allocated addresses, remaining addresses, and lagging addresses.
- Provides step-by-step textual explanation and design tips to help understand the subnetting process.
- Visualizes subnet division using a simple colored bar representation for equal/hierarchical subnetting.
- Includes a Supernetting tab that accepts multiple CIDR blocks and computes an aggregated supernet, showing range, new prefix, and reduction in routing table entries.
- Sidebar section to display the guide's image automatically (uploaded once in Colab as guide_image.png), along with guide name and developer names (Ryan and Rifat).

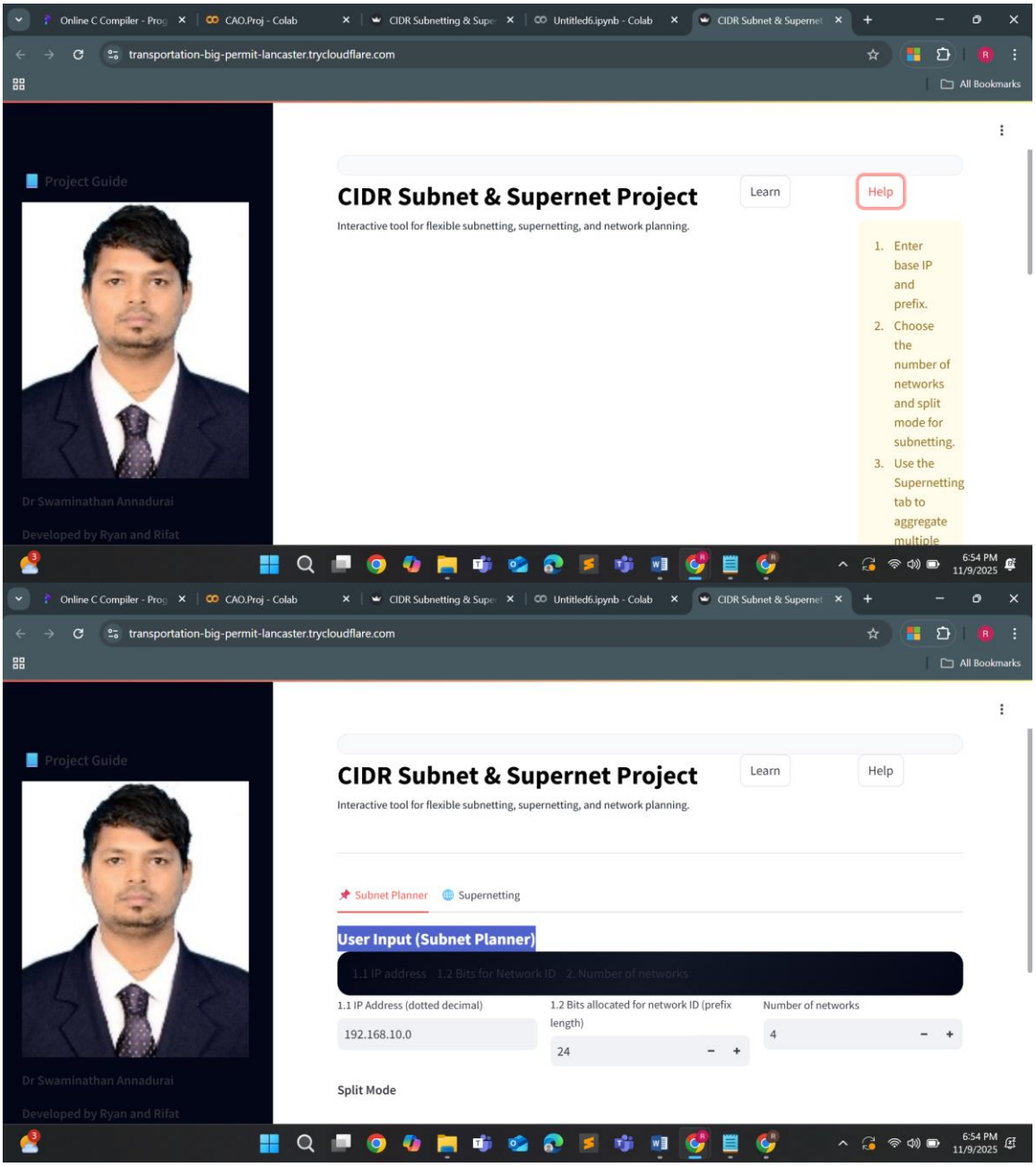
- Reference table in the sidebar showing prefix length versus total addresses and usable hosts for quick lookup.
- Aesthetic UI with custom colors, header banner, clear bold headings, and card-style information sections.
- Cloudflared tunnel integration from Colab to generate a public URL labelled as the 'CIDR CN Project' for demonstration.

## Output




Online C Compiler - Prog
CAO.Proj - Colab
CIDR Subnetting & Super
Untitled6.ipynb - Colab
CIDR Subnet & Supernet
transportation-big-permit-lancaster.trycloudflare.com
All Bookmarks
Developed by Ryan and Rifat
Reference Table: Prefix vs Hosts
Prefix
Total addresses
Usable hos
0 /24 256 25
1 /25 128 12
2 /26 64 6
3 /27 32 3
4 /28 16 1
5 /29 8
6 /30 4
1 Base Network: 192.168.10.0/24
• Network range: 192.168.10.0 – 192.168.10.255
• Total addresses in base block: 256
2 Equal Split Mode
• Requested networks: 4
• Bits borrowed from host part: 2
• New prefix per child: /26
• Child block size: 64 addresses
• Prefer VLSM when each subnet needs a different number of hosts. • Hierarchical design is useful when one branch (like Network #1) needs its own sub-subnets. • Watch the 'Remaining' and 'Lagging' addresses columns to measure address wastage.
Subnet Visualization: /26 (×4)
1
2
3
4
6:55 PM
11/9/2025
Split equally
Split by addresses (VLSM)
Hierarchical split (two-level)
Run Subnet Planner
Network ID
First address
Last address
Broadcast address
Total address
Usable address
0 192.168.10.0 192.168.10.1 192.168.10.62 192.168.10.63 64 62
1 192.168.10.64 192.168.10.65 192.168.10.126 192.168.10.127 64 62
2 192.168.10.128 192.168.10.129 192.168.10.190 192.168.10.191 64 62
3 192.168.10.192 192.168.10.193 192.168.10.254 192.168.10.255 64 62
Step-by-step Explanation & Tips
Steps


Online C Compiler - Prog
CAO.Proj - Colab
CIDR Subnetting & Super
Untitled6.ipynb - Colab
CIDR Subnet & Supernet
transportation-big-permit-lancaster.trycloudflare.com
All Bookmarks
Project Guide
Dr Swaminathan Annadurai
Developed by Ryan and Rifat
CIDR Subnet & Supernet Project
Learn
Help
Interactive tool for flexible subnetting, supernetting, and network planning.
1. Enter base IP and prefix.
2. Choose the number of networks and split mode for subnetting.
3. Use the Supernetting tab to aggregate multiple
6:54 PM
11/9/2025
Online C Compiler - Prog
CAO.Proj - Colab
CIDR Subnetting & Super
Untitled6.ipynb - Colab
CIDR Subnet & Supernet
transportation-big-permit-lancaster.trycloudflare.com
All Bookmarks
Project Guide
Dr Swaminathan Annadurai
Developed by Ryan and Rifat
CIDR Subnet & Supernet Project
Learn
Help
Interactive tool for flexible subnetting, supernetting, and network planning.
Subnet Planner
Supernetting
User Input (Subnet Planner)
1.1 IP address 1.2 Bits for Network ID 2. Number of networks
1.1 IP Address (dotted decimal)
192.168.10.0
1.2 Bits allocated for network ID (prefix length)
24
Number of networks
4
Split Mode
6:54 PM
11/9/2025