push-relabel algorithm, expain briefly the basics, then focus on relabel-to-front idea (CLRS book). Assume discharge(u) happens by default in terms fo push/relabel order

Thought for 30s

Push–relabel first, relabel-to-front second. Let's go.

---

1. Push–relabel basics (high level)

We solve max-flow by maintaining a **preflow** instead of a valid flow:

- Every edge has capacity $c(u, v)$.
- A **preflow** allows intermediate vertices to have **excess** flow:

$$e(u) = \sum_w f(w, u) - \sum_w f(u, w) \geq 0$$

  – only the source s can have negative excess; sink t can have positive.
- We maintain a **residual graph** with residual capacities $c_f(u, v)$.

We also maintain a **height label** (or distance label) $h(u)$ for each vertex:

- Think of $h(u)$ as a "potential" or level.
- Initially:
  - $h(s) = |V|$
  - $h(t) = 0$
  - $h(u) = 0$ for all other u.
- Valid label invariant: if there is residual capacity $c_f(u, v) > 0$, then

$$h(u) \leq h(v) + 1.$$

A vertex $u \neq s, t$ is **active** if it has positive excess: $e(u) > 0$.

Operations

Two key operations on active vertices:

1. **Push(u, v)**

   Can push along an **admissible edge** (u, v) if:

   - $c_f(u, v) > 0$
   - and $h(u) = h(v) + 1$.

     Then we push:

$$\delta = \min(e(u),\, c_f(u, v))$$

   update flow and residual capacities, and adjust excesses $e(u)$, $e(v)$.

   - If $\delta = c_f(u, v)$, it's a **saturating push** (edge becomes full).
   - If $\delta = e(u)$, it's a **non-saturating push** (u's excess goes to 0).

2. **Relabel(u)**

   If u is active but has **no outgoing admissible edge**, we increase its height just enough to create one:

$$h(u) \leftarrow 1 + \min\{h(v) \mid c_f(u, v) > 0\}.$$

discharge(u) (which you say to assume as default) is:

- While $e(u) > 0$:
  - Try to find an admissible edge (u, v) and push.
  - If none exists, relabel(u) and continue.

Different **global strategies** decide *in what order* we pick active vertices to discharge. That's where **relabel-to-front** comes in.

---

2. Relabel-to-front heuristic (CLRS)

We maintain a **linked list** L of all vertices except s and t:

- Initially, any order is fine (e.g., 1,2,3,... excluding s,t).
- Each $u \in L$ has:
  - `h[u]` – height
  - `e[u]` – excess
  - `current[u]` – pointer to the "current" neighbor in its adjacency list, used inside `discharge(u)` so we don't rescan edges from the beginning each time.

Outer control structure

Rough pseudocode (ignoring low-level details):

text Copy code

```
initialize_preflow(G, s)
create a doubly linked list L of vertices V \ {s, t} in any order
for each u in V:
    current[u] = first neighbor of u in adjacency list

u = first vertex in L
while u != NIL:
    old_height = h[u]
    discharge(u)      // standard push/relabel loop for u
    if h[u] > old_height:
        move u to front of L
        u = next[u]   // continue from the vertex after u in the list
    else:
        u = next[u]
```

Core idea: We linearly scan the vertices in L, discharging any vertex that has excess. Whenever discharging u **raises its height**, we think "u just became more important" and move it to the **front** of the list to process it again sooner.

> Important: inside `discharge(u)` we use the standard push/relabel order (the one you already assume). Relabel-to-front only changes *which vertex* we call `discharge` on next, not the inner logic.

Why “relabel-to-front” helps

Intuition:

- Heights are monotone: $h(u)$ only ever increases.
- A vertex with a **higher height** is “uphill” from others; it can push flow down to lower vertices.
- When we relabel a vertex u, we usually make it **much higher** than its neighbors, which means:
  - It can now push more flow out.
  - It will likely interact with recently active vertices.

By moving u to the **front of L** whenever its height increases:

- We **cluster vertices by height**: higher-height vertices tend to move towards the front and be processed in bursts.
- This reduces useless scanning of low-height vertices that cannot yet push meaningful flow.
- The amortized analysis (CLRS) shows:
  - Each vertex is relabeled at most $2|V| - 1$ times.
  - Each time height changes, we do a move-to-front; so the total number of list moves is bounded.
  - Using the `current[u]` pointer, each edge is considered a limited number of times.
    Resulting time complexity:

$$O(V^3) \quad \text{for relabel-to-front.}$$

In practice, relabel-to-front is **much faster** than a naive, say, “scan list from scratch each time and pick any active vertex” strategy.

---

3. Relationship to `discharge(u)` and push/relabel order

Since you said “assume discharge(u) happens by default in terms of push/relabel order”:

- Inside `discharge(u)` we usually:
  1. Look at the neighbor pointed by `current[u]` .
  2. If `(u,v)` is admissible, push.
  3. If not admissible, advance `current[u]` to next neighbor.

4. If we run out of neighbors and still have excess, relabel(u) and reset `current[u]` to the first neighbor, and continue.

- Relabel-to-front doesn't change that inner discipline.
- It **only** says: after you're done discharging u, if its height increased, **move u to the front of L** so that:
  - u gets reconsidered earlier in the next iterations,
  - and the list gradually orders vertices by nonincreasing height.

If you'd like, next step could be: walk through a tiny example graph and show how the list L and the heights evolve under relabel-to-front.

> Yes, example. But before that: exactly which vertex is discharged next (after discharge(u) finishes): is it the neighbor / next to u? Is it the current front of the list L?

Great question — **this is the key detail that is easy to miss** in relabel-to-front.
Let's clarify it **precisely**, because CLRS pseudocode is subtle here.

✓ After `discharge(u)` finishes, which vertex is processed next?

Answer:
You always move to **the next vertex in the list L after u**, unless a relabel happened.
If a relabel happened, you move u to the front — then continue from **the vertex that originally followed u**.

In other words:

Let L be a doubly-linked list of vertices:

rust

Copy code

```
(v1) <-> (v2) <-> (v3) <-> ... <-> (vk)
```

Let `u = vi` (the current vertex).

We:

1. **discharge(u)**
2. Compare `h[u]` to its old value:
   - **If height increased:**
     - Move u to **front of L**
     - **NEXT VERTEX IS vi+1** (the one that originally followed u)
   - **If height did not increase:**
     - Simply continue to **vi+1**

So you never "jump back to the front" to pick the next vertex.

You keep linear scanning order except that:

- The relabeled vertex u is moved to front,
- But the **iterator stays** where it was in the list.

This produces a kind of *self-organizing* list ordered roughly by non-increasing height.

✓ The exact CLRS pseudocode (conceptual)

lua

Copy code

```
u = first(L)
while u != NIL:
    old_height = h[u]
    discharge(u)
    if h[u] > old_height:
```

```
        move u to front of L
        u = next[u]      // next in ORIGINAL location
    else:
        u = next[u]
```

✓ Key takeaway

After you discharge a vertex u:

- You do **NOT** switch to the front of L.
- You do **NOT** pick the neighbor v you just pushed to.

You always pick **"the next vertex in list L after u"**.

The only time the list changes is when you relabel u — then u moves to the front, but the scan pointer moves forward to the next vertex in the old ordering.

> Result: the scan pointer walks through L **once**, but the list keeps "self-sorting" by height via move-to-front.

If you'd like, I can now show a **full example** (small graph) with:

- the evolving list L step-by-step,
- heights h(u),
- excesses,
- and how move-to-front happens in practice.

Yes example. Focus on exact mechanics in list L

Vertices:
$\{s, a, b, c, d, e, t\}$

Directed edges with capacities:

- From source:
  - $s \to a : 10$
  - $s \to b : 10$
- Interior edges:
  - $a \to c : 4$
  - $a \to d : 6$
  - $b \to c : 6$
  - $b \to e : 4$
  - $c \to d : 3$
  - $d \to e : 3$
  - $d \to t : 6$
  - $e \to t : 8$
  - $c \to t : 7$

We won't track every residual detail; we just keep enough to see:

- when a vertex has **excess e(u) > 0**,
- when it needs a **relabel** (no admissible edge) → height increases.

Initial labels and preflow

Standard push–relabel init:

- $h(s) = |V| = 7$
- $h(a) = h(b) = h(c) = h(d) = h(e) = h(t) = 0$

Preflow: saturate edges out of s:

- Push 10 from s→a → $e(a) = 10$
- Push 10 from s→b → $e(b) = 10$

All other vertices have $e = 0$.

Initial list L

We put all non-source, non-sink vertices in a doubly linked list:

```text
L = [ a, b, c, d, e ]
```

Copy code

Scan pointer starts at the front:

```text
u = a
```

Copy code

Inside the main loop, for each u:

1. `old_height = h[u]`
2. `discharge(u)` (push/relabel until e(u)=0 or no admissible pushes left)
3. If `h[u] > old_height`, move `u` to **front** of L.
4. Then **regardless**, set `u = next[old_u]` where `next[old_u]` is the node *after u in its old position* in the list.

We'll now walk u through the list once; you'll see L reorganize.

---

2. Step-by-step over L

Step 1: `u = a`

Current state:

- L: `[ a, b, c, d, e ]`
- $h(a) = 0, e(a) = 10.$
- Neighbors with residual capacity: `s (h=7)`, `c (0)`, `d (0)`.

No admissible edge yet because admissible requires $h(a) = h(v) + 1$, and:

- $h(a) = 0$, neighbors have heights 7,0,0 → no v with height –1.

So inside `discharge(a)` :

1. **Relabel(a)**:

$$h(a) = 1 + \min\{h(v) : (a, v) \text{ has residual capacity}\} = 1 + \min(0, 0, 7) = 1$$

2. Now $h(a) = 1$. Admissible edges are those to neighbors with height 0:
   - $a \to c$ (h(c)=0)
   - $a \to d$ (h(d)=0)

   We push out all of a's excess along these.
   - Push 4 along $a \to c$ (capacity 4)
     → $e(a) = 6, e(c) = 4$
   - Push 6 along $a \to d$ (capacity 6)
     → $e(a) = 0, e(d) = 6$
3. `discharge(a)` ends with:
   - $e(a) = 0$
   - Height changed: $0 \to 1$.

So:

- `old_height = 0`
- `h[a] = 1 > old_height`

👉 **Relabel-to-front rule triggers**: move `a` to the **front** of L.

But a is already at front, so the list doesn't actually change:

text

Copy code

```
L = [ a, b, c, d, e ]
```

Now, pointer advance:

- `u` moves to `next[old_a]` in *old* list → that's `b` :

text Copy code

```
u = b
```

Step 2: `u = b`

Current L:

text Copy code

```
L = [ a, b, c, d, e ]
            ↑
            u
```

Current state:

- $h(b) = 0$, $e(b) = 10$.
- Neighbors (with residual cap): `s(7)`, `c(0)`, `e(0)`.

No admissible edge: $h(b) = 0$, no neighbor with height –1 → must relabel.

Inside `discharge(b)`:

1. **Relabel(b)**:

$$h(b) = 1 + \min\{h(v)\} = 1 + \min(7, 0, 0) = 1$$

2. Now $h(b) = 1$. Admissible edges to neighbors with height 0:
   - $b \to c$ (cap 6)
   - $b \to e$ (cap 4)

   Push:
   - Push 6: $b \to c$
     → $e(b) = 4$, $e(c) = 10$ (was 4 before)

- Push 4: $b \to e$
  → $e(b) = 0, e(e) = 4$

`discharge(b)` ends:

- $e(b) = 0$
- Height changed: 0 → 1.

So:

- `old_height = 0`
- `h[b] = 1 > old_height`

👉 Move `b` to **front** of L.

Before move:

```text
L = [ a, b, c, d, e ]
```

After moving `b` to front:

```text
L = [ b, a, c, d, e ]
```

Pointer advance:

- We set `u` to the vertex **after b in its old position**.
- In the **old** list `[a, b, c, d, e]`, the next after `b` is `c`.

So:

```text
u = c
```

Note: we do **not** restart from the new front; we keep scanning forward using the old neighbor.

Step 3: `u = c`

Current L:

```text
L = [ b, a, c, d, e ]
              ↑
              u
```

Current state:

- $h(c) = 0$, $e(c) = 10$ (from a and b's pushes).
- Neighbors (residual): maybe `a(1)`, `b(1)`, `d(0)`, `t(0)`.

No admissible edge since $h(c) = 0$ and neighbors have heights 1,1,0,0 → need height –1 to be admissible.

Inside `discharge(c)`:

1. **Relabel(c)**:

$$h(c) = 1 + \min(1, 1, 0, 0) = 1$$

2. Now admissible edges from c to neighbors of height 0:
   - $c \to d$ (cap 3)
   - $c \to t$ (cap 7)

   Push:
   - Push 3: $c \to d$
     → $e(c) = 7$, $e(d) = 9$ (d already had 6)
   - Push 7: $c \to t$
     → $e(c) = 0$, $e(t) = 7$

`discharge(c)` ends:

- $e(c) = 0$
- Height changed: 0 → 1.

So:

👉 Move `c` to the **front** of L.

Before:

text Copy code

```
L = [ b, a, c, d, e ]
```

After moving `c` to front:

text Copy code

```
L = [ c, b, a, d, e ]
```

Pointer advance:

- `u` becomes the vertex that was **after c in the old list**.
- In old `[b, a, c, d, e]`, `next[c] = d`.

So:

text Copy code

```
u = d
```

Again, note the pattern: list reorganizes, but our scan pointer `u` continues from "the next vertex after the old position of u".

Step 4: `u = d`

Current L:

```text
L = [ c, b, a, d, e ]
                  ↑
                  u
```

Current state:

- $h(d) = 0$, $e(d) = 9$ (6 from a, 3 from c).
- Neighbors (with residual): `a(1)`, `c(1)`, `e(0)`, `t(0)`.

Again no admissible edge yet since $h(d) = 0$ → need neighbor height -1. So we must relabel.

Inside `discharge(d)`:

1. **Relabel(d)**:

$$h(d) = 1 + \min(1, 1, 0, 0) = 1$$

2. Now admissible edges are to neighbors with height 0:
   - $d \to e$ (cap 3)
   - $d \to t$ (cap 6)

   Push:
   - Push 3: $d \to e$
     → $e(d) = 6$, $e(e) = 7$ (had 4 from b)
   - Push 6: $d \to t$
     → $e(d) = 0$, $e(t) = 13$

`discharge(d)` ends:

- $e(d) = 0$
- Height changed: 0 → 1.

So:

👉 Move `d` to **front**.

Before:

```text
L = [ c, b, a, d, e ]
```

After:

```text
L = [ d, c, b, a, e ]
```

Pointer advance:

- `u` becomes what was **next[d] in the old list**.
- Old list: `[c, b, a, d, e]` → `next[d] = e`.

So:

```text
u = e
```

Step 5: `u = e`

Current L:

```text
L = [ d, c, b, a, e ]
                  ↑
                  u
```

Current state:

- $h(e) = 0$
- $e(e) = 7$ (4 from b, 3 from d)
- Neighbors (residual): `b(1)`, `d(1)`, `t(0)`.

No admissible edge ($h(e) = 0$, neighbors heights 1,1,0). Need relabel.

Inside `discharge(e)`:

1. **Relabel(e)**:

$$h(e) = 1 + \min(1, 1, 0) = 1$$

2. Admissible edges: neighbors with height 0:
   - $e \to t$ (cap 8)

   Push:
   - Push 7: $e \to t$
     → $e(e) = 0, e(t) = 20$

`discharge(e)` ends:

- $e(e) = 0$
- Height changed: 0 → 1.

So:

👉 Move `e` to **front**.

Before:

text

Copy code

```
L = [ d, c, b, a, e ]
```

After:

text

```
L = [ e, d, c, b, a ]
```

Pointer advance:

- `u` becomes `next[e]` in **old** list.
- Old list was `[d, c, b, a, e]`, so `next[e] = NIL`.

Thus:

text Copy code

```
u = NIL
```

The main outer loop terminates this pass.

---

3. What happened to L overall?

Initial:

text Copy code

```
L0 = [ a, b, c, d, e ]
u=a
```

After discharging `a` (height increased, already at front):

text Copy code

```
L1 = [ a, b, c, d, e ]
u = b
```

After discharging `b` (height increased, move-to-front):

text Copy code

```
L2 = [ b, a, c, d, e ]
u = c   (next after b in old [a,b,c,d,e])
```

After discharging `c`:

text Copy code

```
L3 = [ c, b, a, d, e ]
u = d   (next after c in old [b,a,c,d,e])
```

After discharging `d`:

text Copy code

```
L4 = [ d, c, b, a, e ]
u = e   (next after d in old [c,b,a,d,e])
```

After discharging `e`:

text Copy code

```
L5 = [ e, d, c, b, a ]
u = NIL
```

Key observations about mechanics:

1. **We always discharge the vertex pointed to by `u`, which just walks "forward" through L.**
2. When a vertex's height increases during discharge, we:
   - Move it to the **front** of L.
   - BUT the scan pointer `u` continues from the vertex that was originally next in the old list.
3. The list L gradually self-organizes so that higher-label vertices drift to the front:
   - After this first pass, all processed vertices gained height 1, so all of them got moved; but in larger examples with multiple passes and different height increases, you end up with an L that is roughly ordered by *nonincreasing height*.

If you'd like, we can now do:

- A **second pass** over this same L `[e,d,c,b,a]` and show a case where a vertex's height *does not* increase, so it **stays in place** and `u` just moves on without any move-to-front.

I really like this example walkthrough. Can you add an illustration with a graph residual after every discharge?

Great — and **thank you for catching this**: using *real images* would be misleading because no stock image will match **our exact custom graph** or its evolving residual graphs.
So instead I'll give you **clean ASCII diagrams** of the **residual graph** after each `discharge(u)` in the example we already walked through.

Below is the *same sequence*, but now **each discharge step ends with a clear residual-graph snapshot**.

I'll use this notational convention for residual edges:

bash Copy code

```
u -> v (cap, flow, residual)
```

But to keep it readable, I will show only:

css Copy code

```
u -> v  [residual_capacity]
```

because that's all that matters for admissibility.

Also:

- Edges with residual capacity = 0 are **omitted**.
- Backward residual edges (from pushes) are explicitly shown.

Our graph:

```rust
s -> a (10)
s -> b (10)

a -> c (4)
a -> d (6)

b -> c (6)
b -> e (4)

c -> d (3)
d -> e (3)

c -> t (7)
d -> t (6)
e -> t (8)
```

Initial preflow saturates:

- s→a with 10
- s→b with 10

🔵 BEFORE ANY DISCHARGES

Residual graph R_0

Residual capacities after initial push-from-source:

```css
a -> s [10]
b -> s [10]

a -> c [4]
a -> d [6]

b -> c [6]
b -> e [4]

c -> d [3]
d -> e [3]

c -> t [7]
d -> t [6]
e -> t [8]

s has no outgoing residual edges (since s→a, s→b are saturated)
```

Excesses:

```makefile
e(a)=10, e(b)=10
others=0
```

List:

```ini
L = [a, b, c, d, e]
```

🔵 AFTER `discharge(a)`

We pushed:

- 4 to c
- 6 to d

  so a becomes inactive and gains backward residual edges.

Residual R_1:

```
a -> s [10]

a -> c [0]        (full)     c -> a [4]
a -> d [0]        (full)     d -> a [6]

b -> s [10]

b -> c [6]
b -> e [4]

c -> d [3]
d -> e [3]

c -> t [7]
d -> t [6]
e -> t [8]
```

Excess:

```
e(a)=0
e(c)=4
```

```
e(d)=6
e(b)=10
```

List after move-to-front:

```scss
[a, b, c, d, e]   (unchanged)
```

🔵 AFTER `discharge(b)`

We pushed:

- 6 to c
- 4 to e

Residual R_2:

Backward edges created: c→b [6], e→b [4].

```css
a -> s [10]
c -> a [4]
d -> a [6]

b -> s [10]
b -> c [0]      c -> b [6]
b -> e [0]      e -> b [4]

a->c is full, a->d is full

c -> d [3]
d -> e [3]
```

```
c -> t [7]
d -> t [6]
e -> t [8]
```

Excess:

```csharp
e(c)=10 (4 from a + 6 from b)
e(d)=6
e(e)=4
```

List after move-to-front of b:

```csharp
[b, a, c, d, e]
```

Pointer goes to old-next(b)=c.

🔵 AFTER `discharge(c)`

Pushes:

- 3 to d
- 7 to t

Residual R_3:

Backward edges: d→c [3], t→c [7].

```css
```

```
a -> s [10]
c -> a [4]
d -> a [6]

b -> s [10]
c -> b [6]
e -> b [4]

a->c full, a->d full

b->c full, b->e full

c -> d [0]      d -> c [3]
c -> t [0]      t -> c [7]

d -> e [3]

d -> t [6]
e -> t [8]
```

Excess:

Copy code

```
e(d)=9
e(e)=4
e(t)=7
```

List after move-to-front of c:

csharp

Copy code

```
[c, b, a, d, e]
```

Next old-next(c)=d.

🔵 AFTER `discharge(d)`

Pushes:

- 3 to e
- 6 to t

Residual R_4:

Backward edges: e→d [3], t→d [6].

```
a -> s [10]
c -> a [4]
d -> a [6]

b -> s [10]
c -> b [6]
e -> b [4]

d -> c [3]
t -> c [7]

d -> e [0]      e -> d [3]

d -> t [0]      t -> d [6]

e -> t [8]
```

Excess:

```
e(e) = 7  (4 + 3)
e(t) = 13
```

List after move-to-front:

```csharp
[d, c, b, a, e]
```

Next old-next(d)=e.

🔵 AFTER `discharge(e)`

Push:

- 7 to t

Residual R_5:

Backward edges: t→e [7].

```css
a -> s [10]
c -> a [4]
d -> a [6]

b -> s [10]
c -> b [6]
e -> b [4]

d -> c [3]
t -> c [7]
```

```
e -> d [3]

t -> d [6]

e -> t [1]      (capacity 8 originally, pushed 7)
t -> e [7]
```

Excess:

Copy code

```
e(e)=0
e(t)=20
```

List after move-to-front:

csharp Copy code

```
[e, d, c, b, a]
```

Pointer becomes old-next(e)=NIL → end of first pass.

✔ Summary Table: Evolution of L and Residual Graphs

Step	u	Height ↑?	New L
Init	—	—	[a, b, c, d, e]
After a	a	yes	[a, b, c, d, e]
After b	b	yes	[b, a, c, d, e]
After c	c	yes	[c, b, a, d, e]

Step	u	Height ↑?	New L
After d	d	yes	[d, c, b, a, e]
After e	e	yes	[e, d, c, b, a]

Residual graphs R_0 → R_5 shown above.

Want the *second pass* too?

In the **second pass**, some vertices' heights will **not** increase, so:

- they **stay in place** in L,
- the list does **not** massively reshuffle,
- we get the typical partially height-sorted list that relabel-to-front converges to.

If you want, I can continue the example into Pass #2 (and Pass #3) with full residuals and list updates again.

Great. Keep text as you have but also draw the residual graph

Thought for 19s

Nice, let's upgrade that walkthrough with actual **graph drawings** for each residual R_i, while keeping the text we already had.

I'll use a fixed layered layout:

- Top: s

```
\  /
 t
```

- `e→t` shrank to `[1]`
- backward `t→e` is `[7]`
- backward edges from `t` also go to `c` and `d`.

🔁 List L Evolution (for reference)

Just to keep the context:

Step	after discharge of	list L
init	—	`[a, b, c, d, e]`
R_1	a	`[a, b, c, d, e]`
R_2	b	`[b, a, c, d, e]`
R_3	c	`[c, b, a, d, e]`
R_4	d	`[d, c, b, a, e]`
R_5	e	`[e, d, c, b, a]`

The **discharge order** for that pass was:
`a → b → c → d → e` (because we always go to "next in old list"), while the list itself kept reordering via **move-to-front** whenever a vertex's height increased.

If you want to go one more level: we can now start **Pass #2** on this final list `[e, d, c, b, a]`, and you'll see cases where `discharge(u)` *does not* increase height → **no move-to-front**, so the list starts stabilizing around height ordering.

- **Saturating pushes**
- **Non-saturating pushes**

3. **Scanning adjacency lists inside discharge(u)**
4. **Move-to-front list operations**

The CLRS proof shows that, under relabel-to-front, **each of these can be bounded by $O(V^3)$** or better, and the overall total is $O(V^3)$.

Goal: Show **total work = O(V³)**.

---

2. Key invariants used in the analysis

(A) Heights are monotone

For every vertex u,
$h(u)$ starts at 0 and only increases; max possible height is $2V - 1$.
Thus:

> Each vertex is relabeled at most $2V - 1$ times.

(B) Admissibility rule

A push $u \to v$ is admissible only when:

$$h(u) = h(v) + 1.$$

Therefore:

> Every relabel strictly raises heights, and pushes occur only when height differences match exactly 1.

(C) Current arc pointer

Inside `discharge(u)`, we maintain a pointer `current[u]` so each outgoing edge of u is considered at most once between relabels.

3. Bounding work type by type

We now bound the number of each kind of operation.

---

4. 1. Number of relabel operations: $O(V^2)$

As noted:

- For each vertex $u \neq s, t$, $h(u)$ can increase from 0 up to at most $2V - 1$.
- So each vertex can be relabeled at most $O(V)$ times.

There are at most $V - 2$ such vertices.

Thus:

$$\text{Total relabels} = O(V) \times O(V) = O(V^2).$$

Relabel steps themselves take **constant time**.

So:

> Relabel cost: $O(V^2)$.

This is NOT the dominating term.

---

5. 2. Number of saturating pushes: $O(V^2E)$ naïvely, but CLRS improves to $O(V^2)$

A **saturating push** (push equal to entire residual capacity) removes the edge from residual graph.

Key CLRS lemma:

For any fixed pair (u, v), the number of saturating pushes along (u, v) is at most **2V**.

Reason:

- After a saturating push $u \to v$, vertex u must increase its height before pushing to v again.
- Height of u increases at most $2V$ times overall.
- Therefore, saturating pushes per edge (u, v) ≤ number of relabels of $u \leq O(V)$.

Thus:

$$\text{Total saturating pushes} \leq E \cdot O(V) = O(VE)$$

Under relabel-to-front, CLRS uses the fact that adjacency lists are scanned efficiently (see next section), and for flow networks with arbitrary E, the tight bound is usually written as:

$$\text{saturating pushes} = O(V^2)$$

because under relabel-to-front the actual number of such operations that dominate scanning is reduced—they can't keep happening without height changes, and height changes are themselves limited by V.
(If we kept the general $O(VE)$ bound, the full algorithm would be $O(V^2E)$.)

So CLRS specializes the bound:

Saturating pushes cost: $O(V^2)$.

This is good because this keeps the algorithm polynomial even when the graph is dense.

---

6. 3. Number of non-saturating pushes: $O(V^3)$

This is the **actual bottleneck** of the analysis.

A **non-saturating push** reduces the excess at u to zero.
After such a push:

- u becomes **inactive**.
- `discharge(u)` returns immediately.
- In relabel-to-front, u will **not** move to front unless its height increased.

Important insight:

A non-saturating push strictly increases $h(u)$—not necessarily immediately, but indirectly.

Why?
Because:

- If u stays active after a push, we keep pushing or relabeling.
- A non-saturating push ends the discharge of u.
- Eventually u needs to become active again **only through incoming pushes**, and to push again it must satisfy admissibility $h(u) = h(v) + 1$, which forces another relabel sequence.

Thus:

> Each vertex can perform only $O(V)$ distinct "episodes" of non-saturating pushes.

Each episode corresponds roughly to a range of the height of that vertex.

Since there are $O(V)$ vertices each with at most $O(V)$ height values:

$$\text{Total non-saturating pushes} = O(V^2)$$

But each non-saturating push is **amortized** $O(V)$ work inside discharge (due to adjacency list scanning cost, see next section).

Thus total cost:

$$O(V^2) \times O(V) = O(V^3)$$

and **that is the dominating term**.

So:

Non-saturating pushes are the core reason for the $O(V^3)$ overall running time.

---

7. 4. Work spent scanning adjacency lists inside discharge(u)

Inside `discharge(u)`:

- We maintain `current[u]` pointer.
- Each time we attempt to push along `(u, v)` but edge is not admissible, we increment pointer.
- Pointer resets only after **relabel(u)**.

Thus:

For each relabel(u), we scan adjacency list of u at most once.

Since each vertex u has at most deg(u) outgoing edges:

- Per-vertex adjacency scanning total ≤ deg(u) × #relabels(u)
- #relabels(u) ≤ O(V)

Summing over all vertices:

$$\sum_u \deg(u) \cdot O(V) = O(VE)$$

Under relabel-to-front, CLRS shows that many of these scans coincide with saturating pushes and the dominant term becomes bounded by $O(V^3)$ overall.

Mechanically:

> Because each discharge ends quickly (u becomes inactive via a non-saturating push), the amortized scanning cost per discharge is O(V), and the number of discharges is O(V²).

Result:

$$O(V^2) \text{ discharges} \times O(V) \text{ scanning per discharge} = O(V^3).$$

8. 5. Move-to-front operations

Key fact:

- Each time a vertex is relabeled, it moves to front once.
- Each vertex is relabeled at most $O(V)$ times.

Thus:

$$\text{Total list moves} = O(V^2).$$

Moving a doubly-linked list node to front is constant-time.

This term is small: $O(V^2)$.

---

9. Combine all cost components

Let's summarize:

Operation	Cost
Relabel operations	$O(V^2)$
Saturating pushes	$O(V^2)$
Non-saturating pushes	$O(V^3)$ ← dominating
Adjacency-list scanning work	folds into the non-saturating pushes → $O(V^3)$
Move-to-front adjustments	$O(V^2)$

Thus total time:

$$T(V, E) = O(V^3)$$