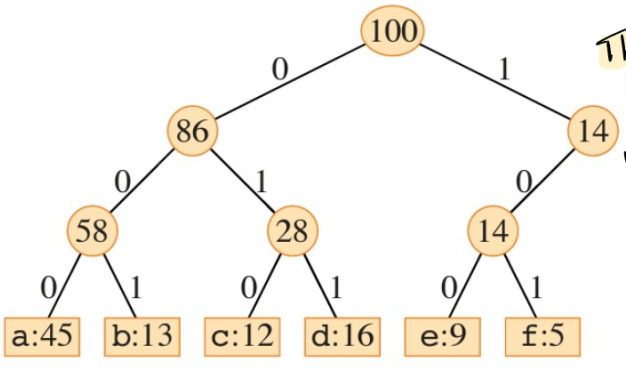
15.3-2

Prove that a non-full binary tree cannot correspond to an optimal prefix-free code.

Def: Optimal prefix-free code is a type of binary encoding used primarly for compressing data efficiently. A prefix-code is a type of code in which no code word is a prefix of any other code word. It allows for efficient an unambiguous decoding.

Example of non-full binary tree:



There is no codeword that starts with '1'. ↳ which makes not the optimal cost

↳ It has an inner node that has only a child, the average codeword lenght for a prefix-free code is proportional to the weighted sum of the root to leaf distances. Since, we can see from the tree of the example, that a node with only one child leads to an unnecessarily long path for at least one codeword, and thus is not optimal.

15.3-3

What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

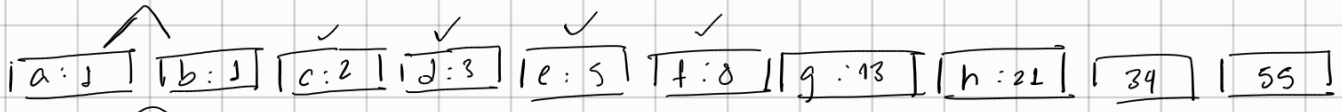
a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

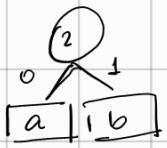
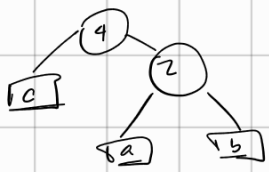
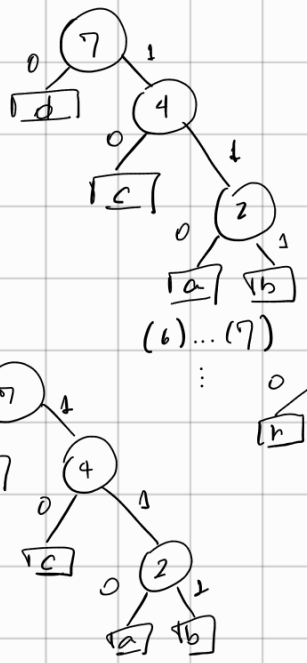
Can you generalize your answer to find the optimal code when the frequencies are the first n Fibonacci numbers?

```
HUFFMAN(C)
1  n = |C|        // length of a set of elements
2  Q = C          // min-priority queue
3  for i = 1 to n − 1
4      allocate a new node z
5      x = EXTRACT-MIN(Q)
6      y = EXTRACT-MIN(Q)
7      z.left = x
8      z.right = y
9      z.freq = x.freq + y.freq
10     INSERT(Q, z)
11 return EXTRACT-MIN(Q)      // the root of the tree is the only node left
```

A step-by-step solution:

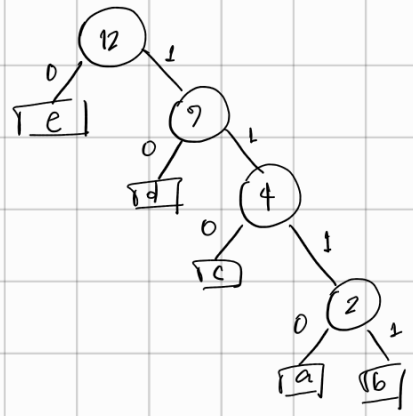
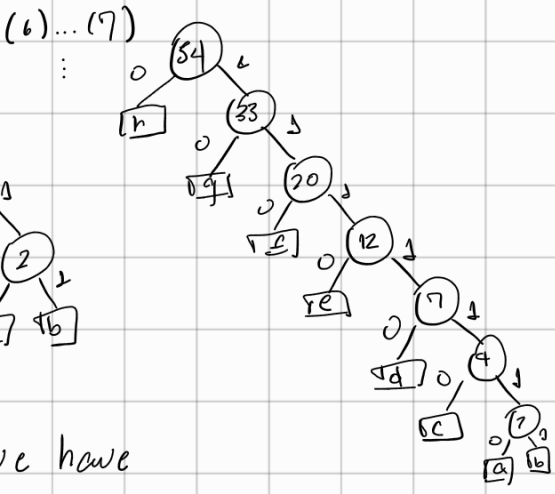
(1)



(2) (1)  ... (2)  (3) 

expanding the problem

8

(4)  (5) (6)...(7) 

(2) : 1 1 1 1 1 1 0

b : 1 1 1 1 1 1 1

c : 1 1 1 1 1 1

d :

⋮

From the drawn example we have the intuition that the sum at step k always equals to $F_{k+2} - 1$, and now we have that:

$$\sum_{i=1}^{k} F_i = F_{k+2} - 1$$

→ By mathematical induction:

Base case: LHS → $\sum_{i=1}^{1} F_i = F_1 = 1$ ✓

RHS → $k = 1 = F_3 - 1 = 2 - 1 = 1$ ✓

Inductive Hypothesis:

→ Check that the formula holds for $k = n$

$$\sum_{i=1}^{n} F_i = F_{n+2} - 1$$

Inductive Step:

→ Prove that the formula holds for $k = n + 1$

LHS: $\sum_{i=1}^{n+1} F_i = F_{(n+1)+2} - 1 = F_{n+3} - 1$

$$= \sum_{i=1}^{n} F_i + F_{n+1}$$

$$= (F_{n+2} + F_{n+1}) - 1$$

From the Fibonacci recurrence relation $F_{n+3} = F_{n+2} + F_{n+1}$,

then,

$$\sum_{i=3}^{n+1} F_i = F_{n+3} - 1 \rightarrow \text{match the above formula.}$$

15.3-5

Given an optimal prefix-free code on a set C of n characters, you wish to transmit the code itself using as few bits as possible. Show how to represent any optimal prefix-free code on C using only $2n - 1 + n \lceil \lg n \rceil$ bits. (*Hint:* Use $2n - 1$ bits to specify the structure of the tree, as discovered by a walk of the tree.)

- Take as example the above constructed tree, to represent its structure we need n leafs to represent the characters and there is n-1 internal nodes in a full binary tree, thus the total number of nodes is $n + (n-1) = 2n-1$.
- We can represent each character using $\lceil \lg n \rceil$ bits, thus we will need $n\lceil \lg n \rceil$ bits for labeling the characters.

Example for character labeling:

Step 2: Character Labeling

Assign a binary representation to each leaf node (A, B, C, D):

1. There are $n = 4$ characters, so each character requires $\lceil \log_2 4 \rceil = 2$ bits.
2. Assign labels:
   - $A : 00$
   - $B : 01$
   - $C : 10$
   - $D : 11$

Character labeling bits:

$n \cdot \lceil \log_2 n \rceil = 4 \cdot 2 = 8$ bits

16.1-1

If the set of stack operations includes a MULTIPUSH operation, which pushes k items onto the stack, does the $O(1)$ bound on the amortized cost of stack operations continue to hold?

Amortized Analysis: typically used for data structures like stacks, heaps or hash tables. It is a technique used to determine the average performance of an algorithm over a seq. of operations, ensuring that while some

operations may be costly, the average cost per operations remains low.

From what we know in an stack push and pop operations have an amortized cost of $O(1)$.
The MULTIPUSH operations will push k elements to the stack. Assuming we have n operations and we call MULTIPUSH n times then by aggregate method $O(nk)$, and finding an amortized cost per operation $\frac{O(nk)}{n} = O(k) \neq O(1)$, and we can conclude that the time is not constant.

Aggregate Method

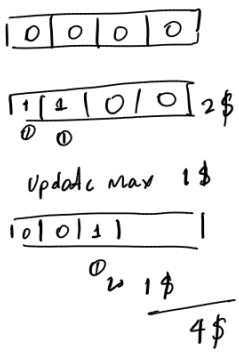
16.1-2

Show that if a DECREMENT operation is included in the k-bit counter example, n operations can cost as much as $\Theta(nk)$ time.

Example: $k = 3$

[Underlined bits flip. Show costs later.]

counter value	A 2 1 0	cost
0	0 0 0	0
1	0 0 1	1
2	0 1 0	3
3	0 1 1	4
4	1 0 0	7
5	1 0 1	8
6	1 1 0	10
7	1 1 1	11
0	0 0 0	14
⋮	⋮	15



Cost of INCREMENT = Θ(# of bits flipped) .

Analysis

Each call could flip k bits, so n INCREMENTs takes $O(nk)$ time.

Observation

Not every bit flips every time.

[Show costs from above.]

bit	flips how often	times in n INCREMENTs
0	every time	n
1	1/2 the time	$\lfloor n/2 \rfloor$
2	1/4 the time	$\lfloor n/4 \rfloor$
⋮		
i	$1/2^i$ the time	$\lfloor n/2^i \rfloor$
⋮		
$i \geq k$	never	0

$$\text{Therefore, total \# of flips} = \sum_{i=0}^{k-1} \lfloor n/2^i \rfloor < n \sum_{i=0}^{\infty} 1/2^i = n\left(\frac{1}{1-1/2}\right) = 2n .$$

Formula of an infinite Geometric series:
$S = \frac{a}{1-r}$, a is the first term of the series. r is the common ratio $|r| < 1$

↳ In here first term is 1, since $i=0$. $1/2^0 = 1$, the common ratio is $1/2$ since each term is in the form $1/2^i$.

Similar to the INCREMENT example each call could flip k bits.

But let's see DECREMENT first:

value	
7	1 1 1
6	1 1 0
5	1 0 1
4	1 0 0
3	0 1 1

Now we see that since either INCREMENT or DECREMENT can happen n times, and the worst case is when we need to flip k bits then n $O(k)$ operations $\Rightarrow O(nk)$.

Amortized cost = $\frac{O(nk)}{n} = O(k)$.

Note: When both operations are mixed, you lose the cummulative benefit because each operation can undo the work of the other.

16.1-3

Use aggregate analysis to determine the amortized cost per operation for a sequence of n operations on a data structure in which the ith operation costs i if i is an exact power of 2, and 1 otherwise.

We have n operations,

$\begin{cases} i & \text{if } 2^k \quad (k \le n) \quad 2, 4, 8, 16 \\ 1 & \text{otherwise} \quad 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 \ldots n \end{cases}$

→ There is $\lfloor \log(n) \rfloor$ with cost i in a sequence of n elements, and $n - \lfloor \log(n) \rfloor$ constant cost operations

The aggregate cost is:

$$\sum_{i=1}^{\lfloor \log n \rfloor} 2^i + \sum_{i=1}^{n - \lfloor \log(n) \rfloor} (1)$$

$S = \frac{a \cdot (r^m - 1)}{r - 1}$ ← Geometric series

a: first term
r: common ratio
m: number of terms

$$\frac{2(2^{\lfloor \log n \rfloor} - 1)}{2 - 1} + n - \lfloor \log(n) \rfloor$$

$$2(2^{\lfloor \log n \rfloor}) - 2 + n - \lfloor \log(n) \rfloor$$

$$2n + n - \lfloor \log(n) \rfloor - 2,$$

→ for n sufficient large $\approx 3n = O(n)$

Thus amortized cost $= \frac{O(n)}{n} = O(1)$

Accounting Method

16.2-1

You perform a sequence of PUSH and POP operations on a stack whose size never exceeds k. After every k operations, a copy of the entire stack is made automatically, for backup purposes. Show that the cost of n stack operations, including copying the stack, is $O(n)$ by assigning suitable amortized costs to the various stack operations.

→ First we assigned a value for each operation (Push, Pop, Copy)

In accounting method we assign a larger amortized cost to the operation.

→ Push operation costs $O(1)$, then we gave an amortized cost of 2$. Where, 1 unit for operation, 1 unit for credit.

Stack

$ (units)

→ Pop operation costs $O(1)$ then we can give it a cost of 1 $

→ Copy costs $O(k)$ where k is the current size of the array. we gave 0$ since it would be amortized with the other operations.

We know that: Total amortized cost ≥ Total actual cost

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

→ There might be n Push and Pop operations, which makes the total cost $O(n)$.

→ Copy happens every k operations, each with a cost of $O(k)$ and then for n operations the Stack is copy $\lfloor n/k \rfloor$ times. Total cost: $O(k)\lfloor n/k \rfloor = O(n)$

Then total cost $O(n) + O(n) = O(n)$

Accounting method: ↻ Aggregate method?

→ We can then see, that credit is never negative and the inequality above is satisfied.

→ Each operation has an amortized cost of $O(1)$, and for n operations, this is $O(1) \cdot n = O(n)$

16.2-3

You wish not only to increment a counter but also to reset it to 0 (i.e., make all bits in it 0). Counting the time to examine or modify a bit as $\Theta(1)$, show how to implement a counter as an array of bits so that any sequence of n INCREMENT and RESET operations takes $O(n)$ time on an initially zero counter. (*Hint:* Keep a pointer to the high-order 1.)

↳ For this solution we will make use of a pointer, pointing to the highest-order 1.

Algorithm:

INITIALIZATION
- Initialize an array bits[]
- Set the pointer high = -1

INCREMENT:
- Starting with the least significant bit, flip it.
- Update the value of high to the index with the highes-order 1

RESET:
- Flip all the bits before the high pointer

Visualization: high = -1

bits (array): 0, 0, 0, 0 ;

INCREMENT: high = 0 (index)
1, 0, 0, 0

↓ : high = 1
0, 1, 0, 0

↓ : high = 1
1, 1, 0, 0

RESET : high = -1
0, 0, 0, 0 (reset k bits)

Pseudo code:

```
bits = [0] * size      // define a size for the array
high = -1              O(1) no need
INCREMENT(bits, high)
    i = 0
    while i < bits.size and bits[i] == 1:  // Flip 1 to 0
        bits[i] = 0
        i = i + 1
    if i < bits.size:  // Flip 0 to 1
        bits[i] = 1                2$
        high = max(high, i)        1$
RESET(bits, high)
    for i = 0 to high
        bits[i] = 0
    high = -1          → 1$ Reset
```

Amortized Analysis:

As for the counter in the book, we assume that it costs \$1 to flip a bit. In addition, we assume it costs \$1 to update $A.max$.

Setting and resetting of bits by INCREMENT will work exactly as for the original counter in the book: \$1 will pay to set one bit to 1; \$1 will be placed on the bit that is set to 1 as credit; the credit on each 1 bit will pay to reset the bit during incrementing.

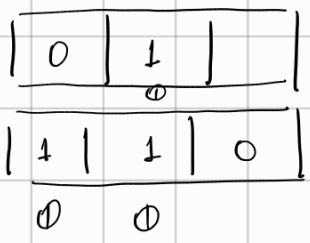
In addition, we'll use \$1 to pay to update *max*, and if *max* increases, we'll place an additional \$1 of credit on the new high-order 1. (If *max* doesn't increase, we can just waste that \$1—it won't be needed.) Since RESET manipulates bits at positions only up to $A.max$, and since each bit up to there must have become the high-order 1

at some time before the high-order 1 got up to $A.max$, every bit seen by RESET has \$1 of credit on it. So the zeroing of bits of A by RESET can be completely paid for by the credit stored on the bits. We just need \$1 to pay for resetting max.

Thus charging \$4 for each INCREMENT and \$1 for each RESET is sufficient, so the sequence of n INCREMENT and RESET operations takes $O(n)$ time.

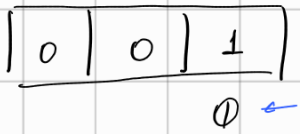
b What it means, summarized

(1) Flipping bits cost 2\$, but we only use when flipping 0 to 1 and the 1\$ is used as credit. Ex.



(2) To update high we need 1\$.

(3) Might be an overflow, then you need 1\$ more for it



Thus this will give us a total of 4\$.

→ Reset expenses are covered already because when it flips a 1 back to zero, there is a credit to use. However, we will need a 1\$ for setting high back to -1

→ Since there is non negative credit at all times the running time for a n sequence of INCREMENT and RESET is $O(n)$

Potential Method

16.3-1

Suppose you have a potential function Φ such that $\Phi(D_i) \geq \Phi(D_0)$ for all i, but $\Phi(D_0) \neq 0$. Show that there exists a potential function Φ' such that $\Phi'(D_0) = 0$, $\Phi'(D_i) \geq 0$ for all $i \geq 1$, and the amortized costs using Φ' are the same as the amortized costs using Φ.

The amortized cost of an operation is defined using the potential function Φ as:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

c_i is the actual cost of the i-th operation

$\Phi(D_i)$ is the potential after the i-th operation

$\Phi(D_{i-1})$ is the potential before the i-th operation.

First step in potential method: define potential function $\Phi : D \to R_0^+$ with $\Phi(D_i) \geq \Phi(D_0)$ for all i. (and typically $\Phi(D_0) = 0$)

Now for our problem:

We define a new potential function $\Phi'(D_i)$ as:

$$\Phi'(D_i) = \Phi(D_i) - \Phi(D_0) \quad (1)$$

At the initial state: $\Phi'(D_0) = \Phi(D_0) - \Phi(D_0) = 0$

For any state D_i: since $\Phi(D_i) \geq \Phi(D_0)$, it follows that

$$\Phi'(D_i) = \Phi(D_i) - \Phi(D_0) \geq 0$$

Thus the new potential function in (1) satisfies the two required conditions: $\Phi'(D_0) = 0$

$$\Phi'(D_i) \geq 0 \text{ for all } i \geq 1$$

Amortized cost using Φ':

$$\begin{aligned} \hat{c}_i' &= c_i + \Phi'(D_i) - \Phi'(D_{i-1}) \\ &= c_i + \left(\Phi(D_i) - \cancel{\Phi(D_0)}\right) - \left(\Phi(D_{i-1}) - \cancel{\Phi(D_0)}\right) \\ &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \end{aligned}$$

$\hat{c}_i' = \hat{c}_i$, $\therefore$ The amortized cost using Φ' are the same as using Φ.

16.3-3

Consider an ordinary binary min-heap data structure supporting the instructions INSERT and EXTRACT-MIN that, when there are n items in the heap, implements each operation in $O(\lg n)$ worst-case time. Give a potential function Φ such that the amortized cost of INSERT is $O(\lg n)$ and the amortized cost of EXTRACT-MIN is $O(1)$, and show that your potential function yields these amortized time bounds. Note that in the analysis, n is the number of items currently in the heap, and you do not know a bound on the maximum number of items that can ever be stored in the heap.

Answer:

Let D_i be the heap after the ith operation, and let D_i consist of n_i elements. Also, let k be a constant such that each INSERT or EXTRACT-MIN operation takes at most $k \ln n$ time, where $n = \max(n_{i-1}, n_i)$. (We don't want to worry about taking the log of 0, and at least one of n_{i-1} and n_i is at least 1. We'll see later why we use the natural log.)

Define

$$\Phi(D_i) = \begin{cases} 0 & \text{if } n_i = 0\,, \\ k n_i \ln n_i & \text{if } n_i > 0\,. \end{cases}$$

This function exhibits the characteristics we like in a potential function: if we start with an empty heap, then $\Phi(D_0) = 0$, and we always maintain that $\Phi(D_i) \geq 0$.

Before proving that we achieve the desired amortized times, we show that if $n \geq 2$, then $n \ln \frac{n}{n-1} \leq 2$. We have

$$\begin{aligned}
n \ln \frac{n}{n-1} &= n \ln \left(1 + \frac{1}{n-1}\right) \\
&= \ln \left(1 + \frac{1}{n-1}\right)^n \\
&\leq \ln \left(e^{\frac{1}{n-1}}\right)^n \qquad \text{(since } 1 + x \leq e^x \text{ for all real } x\text{)} \\
&= \ln e^{\frac{n}{n-1}} \\
&= \frac{n}{n-1} \\
&\leq 2 ,
\end{aligned}$$

assuming that $n \geq 2$. (The equation $\ln e^{\frac{n}{n-1}} = \frac{n}{n-1}$ is why we use the natural log.)

If the ith operation is an INSERT, then $n_i = n_{i-1} + 1$. If the ith operation inserts into an empty heap, then $n_i = 1$, $n_{i-1} = 0$, and the amortized cost is

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&\leq k \ln 1 + k \cdot 1 \ln 1 - 0 \\
&= 0 .
\end{aligned}$$

If the ith operation inserts into a nonempty heap, then $n_i = n_{i-1} + 1$, and the amortized cost is

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&\leq k \ln n_i + k n_i \ln n_i - k n_{i-1} \ln n_{i-1} \\
&= k \ln n_i + k n_i \ln n_i - k(n_i - 1) \ln(n_i - 1) \\
&= k \ln n_i + k n_i \ln n_i - k n_i \ln(n_i - 1) + k \ln(n_i - 1) \\
&< 2k \ln n_i + k n_i \ln \frac{n_i}{n_i - 1} \\
&\leq 2k \ln n_i + 2k \\
&= O(\lg n_i) .
\end{aligned}$$

If the ith operation is an EXTRACT-MIN, then $n_i = n_{i-1} - 1$. If the ith operation extracts the one and only heap item, then $n_i = 0$, $n_{i-1} = 1$, and the amortized cost is

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&\leq k \ln 1 + 0 - k \cdot 1 \ln 1 \\
&= 0 .
\end{aligned}$$

If the ith operation extracts from a heap with more than 1 item, then $n_i = n_{i-1} - 1$ and $n_{i-1} \geq 2$, and the amortized cost is

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&\leq k \ln n_{i-1} + k n_i \ln n_i - k n_{i-1} \ln n_{i-1} \\
&= k \ln n_{i-1} + k(n_{i-1} - 1) \ln(n_{i-1} - 1) - k n_{i-1} \ln n_{i-1} \\
&= k \ln n_{i-1} + k n_{i-1} \ln(n_{i-1} - 1) - k \ln(n_{i-1} - 1) - k n_{i-1} \ln n_{i-1} \\
&= k \ln \frac{n_{i-1}}{n_{i-1} - 1} + k n_{i-1} \ln \frac{n_{i-1} - 1}{n_{i-1}} \\
&< k \ln \frac{n_{i-1}}{n_{i-1} - 1} + k n_{i-1} \ln 1 \\
&= k \ln \frac{n_{i-1}}{n_{i-1} - 1} \\
&\leq k \ln 2 \qquad \text{(since } n_{i-1} \geq 2\text{)} \\
&= O(1) .
\end{aligned}$$

A slightly different potential function—which may be easier to work with—is as follows. For each node x in the heap, let $d_i(x)$ be the depth of x in D_i. Define

$$\begin{aligned}\Phi(D_i) &= \sum_{x \in D_i} k(d_i(x) + 1) \\ &= k\left(n_i + \sum_{x \in D_i} d_i(x)\right),\end{aligned}$$

where k is defined as before.

Initially, the heap has no items, which means that the sum is over an empty set, and so $\Phi(D_0) = 0$. We always have $\Phi(D_i) \geq 0$, as required.

Observe that after an INSERT, the sum changes only by an amount equal to the depth of the new last node of the heap, which is $\lfloor \lg n_i \rfloor$. Thus, the change in potential due to an INSERT is $k(1 + \lfloor \lg n_i \rfloor)$, and so the amortized cost is $O(\lg n_i) + O(\lg n_i) = O(\lg n_i) = O(\lg n)$.

After an EXTRACT-MIN, the sum changes by the negative of the depth of the old last node in the heap, and so the potential *decreases* by $k(1 + \lfloor \lg n_{i-1} \rfloor)$. The amortized cost is at most $k \lg n_{i-1} - k(1 + \lfloor \lg n_{i-1} \rfloor) = O(1)$.

16.3-4

What is the total cost of executing n of the stack operations PUSH, POP, and MULTIPOP, assuming that the stack begins with s_0 objects and finishes with s_n objects?

PUSH: Actual cost

- $c_i = O(1)$ to add one element
- Change in potential: $\Phi(D_i) - \Phi(D_{i-1}) = 1$

Amortized cost for push $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

$\hat{c}_i = 1 + 1 = 2$

Pop: Actual cost → $c_i = O(1)$ remove one element

Change in potential: since it reduces by 1 then

$\Phi(D_i) - \Phi(D_{i-1}) = -1$

Amortized cost for pop $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

$= 1 + (-1) = 0$

MULTIPOP: Actual cost → $c_i = O(k)$, the number of objects removed

Change in potential: since it reduces by k elements, then

$\Phi(D_i) - \Phi(D_{i-1}) = -k$

Amortized cost: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

$= k + (-k) = 0$

For a sequence of n operations (PUSH, POP, MULTIPOP):

1. Each PUSH contributes an amortized cost of 2
2. Each POP and MULTIPOP contributeds a cost of 0

Thus the total amortized cost is proportional to the number of PUSH operations: $O(p)$

→ The total potential change over all n operations

We have the formula: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

Def: For the total amortized cost: $\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \sum_{i=0}^{n} (\Phi(D_i) - \Phi(D_{i-1}))$

$$\sum \hat{c}_i = \sum c_i + \Phi(D_1) - \Phi(D_0) + \Phi(D_2) - \Phi(D_1) + \dots + \Phi(D_n) - \Phi(D_{n-1})$$

↳ telescopic series

$$\sum_{i=0}^{n} \hat{c}_i = \sum_{i=0}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

Therefore, with $\Phi(D_0) = s_0$, and $\Phi(D_n) = s_n$, over n operations

$$\Phi(D_n) - \Phi(D_0) = s_n - s_0$$

The the total amortized cost for n operations is:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} (\Phi(D_i) + \Phi(D_{i-1}))$$

↳ $$= \sum_{i=1}^{n} c_i + (s_n - s_0)$$

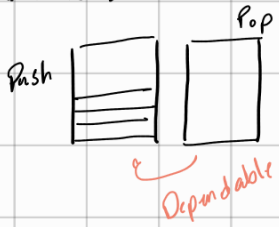
16.3-5

Show how to implement a queue with two ordinary stacks (Exercise 10.1-7) so that the amortized cost of each ENQUEUE and each DEQUEUE operation is $O(1)$.

Let's set the approach. Since stack work as LIFO style, two stacks are necessary to implement a queue. Then lets called them stack_push, stack_pop.

→ Every push will be send to stack_push and every pop will be made in stack_pop (ENQUEUE)

Visualization:



→ Every Push cost is always $O(1)$

→ Whe stack_pop is empty we need to move all the elements from stack_push to stack_pop and thus its cost is $O(n)$, when n elements need to be moved. (DEQUEUE)

ENQUEUE:

The potential method in this case, depends only on the size of stack_push.

cost is $O(1)$ (pushing),

$\Phi(D_i) - \Phi(D_{i-1}) = 1$,

$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

$= 1 + 1 = 2$

DEQUEUE:

CASE 1, there is elements in stack_pop:

Cost is $O(1)$ (popping),

$\Phi(D_i) - \Phi(D_{i-1}) = 0$,

$\hat{c}_i = 1$

CASE 2, we move n elements to it:

Cost is $O(k)$ (pushing the new elements) + $O(1)$ (popping)

$\Phi(D_i) - \Phi(D_{i-1}) = -k$

$\hat{c}_i = c_i + \Delta\Phi$

$\hat{c}_i = (k+1) + (-k) = 1$

Thus for ENQUEUE and DEQUEUE we have an amortized of $O(1)$.

16.3-6

Design a data structure to support the following two operations for a dynamic multiset S of integers, which allows duplicate values:

INSERT(S, x) inserts x into S.

DELETE-LARGER-HALF(S) deletes the largest $\lceil |S|/2 \rceil$ elements from S.

Explain how to implement this data structure so that any sequence of m INSERT and DELETE-LARGER-HALF operations runs in $O(m)$ time. Your implementation should also include a way to output the elements of S in $O(|S|)$ time.

Let's use a dynamic array and a max-heap structure

Operations:

INSERT (S, x)

Append x to the dynamic array O(1)

Insert x into the max heap O(1)

DELETE-LARGER-HALF (S):

Number of elements to be removed $k = \lceil |S|/2 \rceil$ O(1)

Perform k extract-max $O(k \log |S|)$

Removing elements from the array $O(|S|)$

Amortized Analysis:

INSERT takes O(1) for m operations then O(m)

DELETE-LARGER-HALF: after each deletion the size of S is halved.

→ If the initial size of S is n_0, then we can observe that for every half:

$$n_0, \frac{n_0}{2}, \frac{n_0}{4}, \ldots 1$$

The total cost over all DELETE-LARGER-HALF:

$$\text{Total cost} = \sum_{k=0}^{\log n_0} \frac{n_0}{2^k}$$

$$= n_0 \sum_{k=0}^{\log n_0} \frac{1}{2^k}$$

$$\frac{1}{2^{\log n_0 + 1}} = \frac{1}{2 \cdot 2^{\log n_0}} = \frac{1}{2 n_0}$$

⇒ Initial term $a = 1$

⇒ Common ratio $1/2$

For finite series: $\frac{a(1 - r^N)}{1 - r}$ ⇒ $\frac{(1 - (1/2)^{\log n_0 + 1})}{1 - 1/2}$

$= n_0 (2 - 1/n_0)$ ← Initial size of S

$= 2n_0 - 1 = O(n_0)$

⇒ $2(1 - 1/2n_0)$

⇒ $2 - 1/n_0$ → For large n n_0 becomes negligible

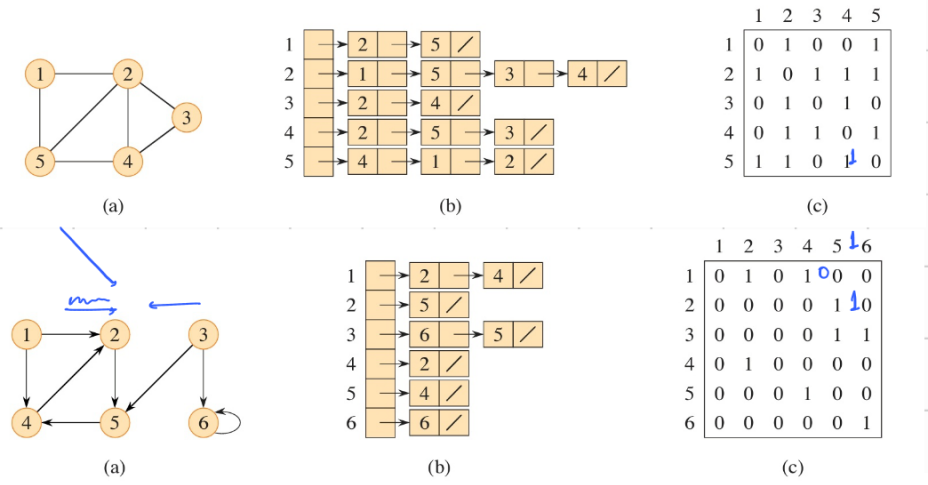
Thus, the total number of elements inserted into S across all operations ⇒ 2 is proportional to m.

Graphs

20.1-6

Most graph algorithms that take an adjacency-matrix representation as input require $\Omega(V^2)$ time, but there are some exceptions. Show how to determine whether a directed graph G contains a ***universal sink***—a vertex with in-degree $|V| - 1$ and out-degree 0—in $O(V)$ time, given an adjacency matrix for G.

Answer



We start by observing that if $a_{ij} = 1$, so that $(i, j) \in E$, then vertex i cannot be a universal sink, for it has an outgoing edge. Thus, if row i contains a 1, then vertex i cannot be a universal sink. This observation also means that if there is a self-loop (i, i), then vertex i is not a universal sink. Now suppose that $a_{ij} = 0$, so that $(i, j) \notin E$, and also that $i \neq j$. Then vertex j cannot be a universal sink, for either its in-degree must be strictly less than $|V| - 1$ or it has a self-loop. Thus if column j contains a 0 in any position other than the diagonal entry (j, j), then vertex j cannot be a universal sink.

Using the above observations, the following procedure returns TRUE if vertex k is a universal sink, and FALSE otherwise. It takes as input a $|V| \times |V|$ adjacency matrix $A = (a_{ij})$.

```
IS-SINK(A, k)
  let A be |V| × |V|
  for j = 1 to |V|                  // check for a 1 in row k
      if a_kj == 1
          return FALSE
  for i = 1 to |V|                  // check for an off-diagonal 0 in column k
      if a_ik == 0 and i ≠ k
          return FALSE
  return TRUE
```

Because this procedure runs in $O(V)$ time, we may call it only $O(1)$ times in order to achieve our $O(V)$-time bound for determining whether directed graph G contains a universal sink.

Observe also that a directed graph can have at most one universal sink. This property holds because if vertex j is a universal sink, then we would have $(i, j) \in E$ for all $i \neq j$ and so no other vertex i could be a universal sink.

The following procedure takes an adjacency matrix A as input and returns either a message that there is no universal sink or a message containing the identity of the universal sink. It works by eliminating all but one vertex as a potential universal sink and then checking the remaining candidate vertex by a single call to IS-SINK.

Final procedure

```
UNIVERSAL-SINK(A)
  let A be |V| × |V|
  i = j = 1
  while i ≤ |V| and j ≤ |V|
      if a_ij == 1
          i = i + 1
      else j = j + 1
  if i > |V|
      return "there is no universal sink"
  elseif IS-SINK(A, i) == FALSE
      return "there is no universal sink"
  else return i "is a universal sink"
```

20.1-7

The ***incidence matrix*** of a directed graph $G = (V, E)$ with no self-loops is a $|V| \times |E|$ matrix $B = (b_{ij})$ such that

$$b_{ij} = \begin{cases} -1 & \text{if edge } j \text{ leaves vertex } i\,, \\ 1 & \text{if edge } j \text{ enters vertex } i\,, \\ 0 & \text{otherwise}\,. \end{cases}$$

Describe what the entries of the matrix product BB^{T} represent, where B^{T} is the transpose of B.

Answer

$$BB^T(i,j) = \sum_{e \in E} b_{ie} b_{ej}^T = \sum_{e \in E} b_{ie} b_{je}$$

- If $i = j$, then $b_{ie}b_{je} = 1$ (it is $1 \cdot 1$ or $(-1) \cdot (-1)$) whenever e enters or leaves vertex i, and 0 otherwise.
- If $i \neq j$, then $b_{ie}b_{je} = -1$ when $e = (i, j)$ or $e = (j, i)$, and 0 otherwise.

Thus,

$$BB^T(i,j) = \begin{cases} \text{degree of } i = \text{in-degree} + \text{out-degree} & \text{if } i = j\,, \\ -(\#\text{ of edges connecting } i \text{ and } j) & \text{if } i \neq j\,. \end{cases}$$

20.1-8

Suppose that instead of a linked list, each array entry $Adj[u]$ is a hash table containing the vertices v for which $(u, v) \in E$, with collisions resolved by chaining. Under the assumption of uniform independent hashing, if all edge lookups are equally likely, what is the expected time to determine whether an edge is in the graph? What disadvantages does this scheme have? Suggest an alternate data structure for each edge list that solves these problems. Does your alternative have disadvantages compared with the hash table?

Expected time to determine if an Edge is in the Graph:
→ Under the assumption of uniform independent hashing, the expected time for a hash table lookup is $O(1)$.

Disadvantage:
→ In the worst case (due the hash collisions), the lookup time can degrade to $O(V)$, where V is the number elements

Alternative:

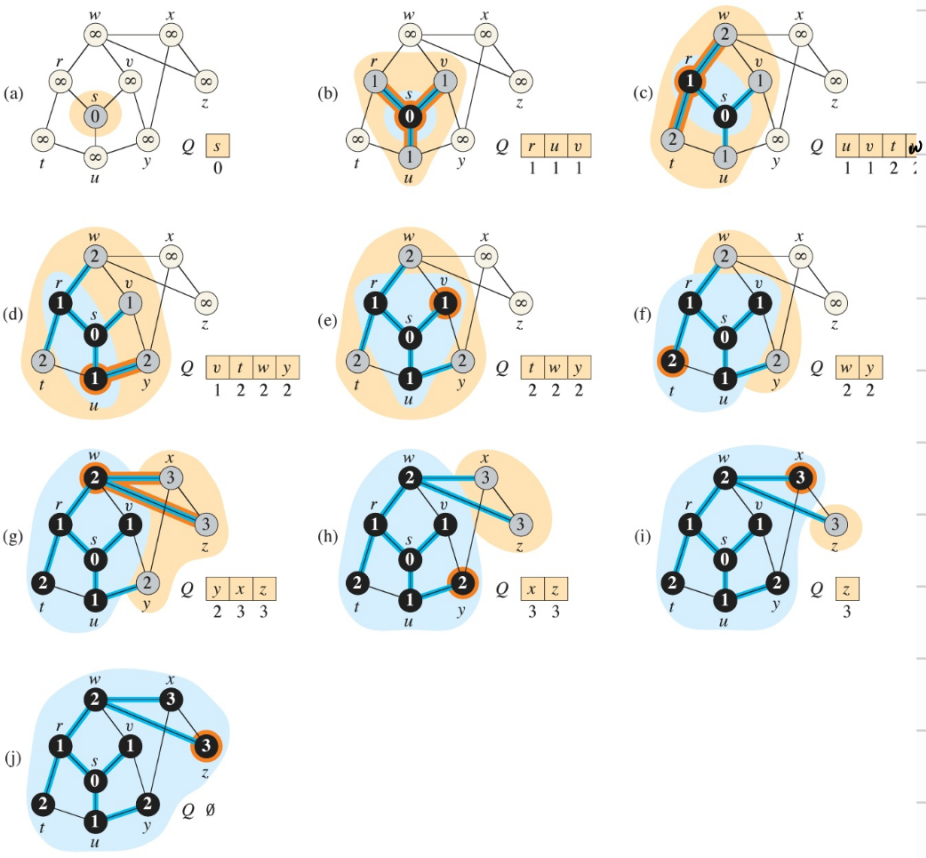
Ordered arrays, Binary Search trees. Improve worst case to $O(\log|V|)$, but set an average of the same running time.

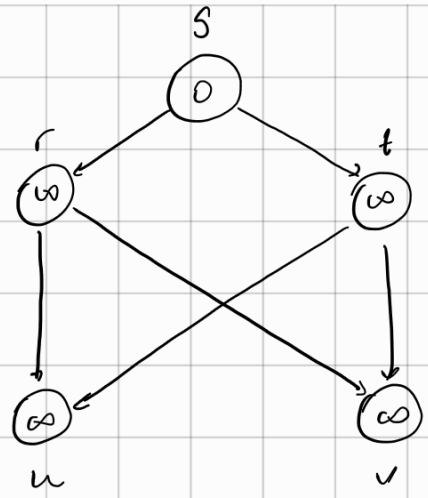
BFS

20.2-6

Give an example of a directed graph $G = (V, E)$, a source vertex $s \in V$, and a set of tree edges $E_\pi \subseteq E$ such that for each vertex $v \in V$, the unique simple path in the graph (V, E_π) from s to v is a shortest path in G, yet the set of edges E_π cannot be produced by running BFS on G, no matter how the vertices are ordered in each adjacency list.

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each vertex v in G.Adj[u]    // search the neighbors of u
13          if v.color == WHITE          // is v being discovered now?
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)            // v is now on the frontier
18      u.color = BLACK                  // u is now behind the frontier
```



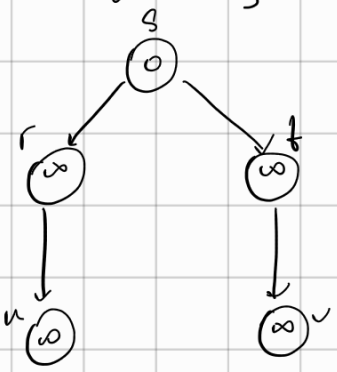


Edges = { sr, st, ru, rv, tu, tv }

After running BFS in this graph

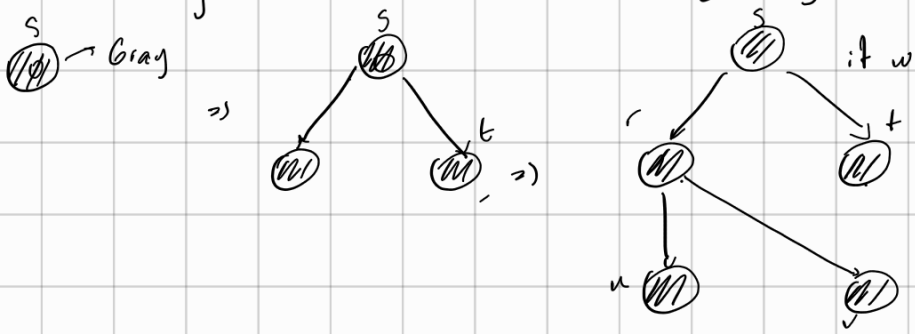
E_π = { sr, st, ru, tv }

Then G_a = { V, E_π }



From the drawing above a unique simple path in $G_a = (V, E_\pi)$ is also a shortest path $G = (V, E)$.

Now let's imagine we run BFS in $G = \{V, E\}$ (illustrating the process):



it will dequeue either r or t, depending of the running, in this case it'll go with vertex r. And the vertex tv is never discover

→ And if t is chosen the ru will not be discovered, and

in conclusion BFS cannot discover $G_\pi = \{V, E_\pi\}$.

20.2-7

There are two types of professional wrestlers: "faces" (short for "babyfaces," i.e., "good guys") and "heels" ("bad guys"). Between any pair of professional wrestlers, there may or may not be a rivalry. You are given the names of n professional wrestlers and a list of r pairs of wrestlers for which there are rivalries. Give an $O(n + r)$-time algorithm that determines whether it is possible to designate some of the wrestlers as faces and the remainder as heels such that each rivalry is between a face and a heel. If it is possible to perform such a designation, your algorithm should produce it.

Answer

Create a graph G where each vertex represents a wrestler and each edge represents a rivalry. The graph will contain n vertices and r edges.

Perform as many BFS's as needed to visit all vertices. Assign all wrestlers whose distance is even to be babyfaces and all wrestlers whose distance is odd to be heels. Then check each edge to verify that it goes between a babyface and a heel. This solution would take $O(n + r)$ time for the BFS, $O(n)$ time to designate each wrestler as a babyface or heel, and $O(r)$ time to check edges, which is $O(n + r)$ time overall.

DFS

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
1  time = time + 1                  // white vertex u has just been discovered
2  u.d = time
3  u.color = GRAY
4  for each vertex v in G.Adj[u]    // explore each edge (u, v)
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)
8  time = time + 1
9  u.f = time
10 u.color = BLACK                  // blacken u; it is finished
```

1. ***Tree edges*** are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
2. ***Back edges*** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.
3. ***Forward edges*** are those nontree edges (u, v) connecting a vertex u to a proper descendant v in a depth-first tree.
4. ***Cross edges*** are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

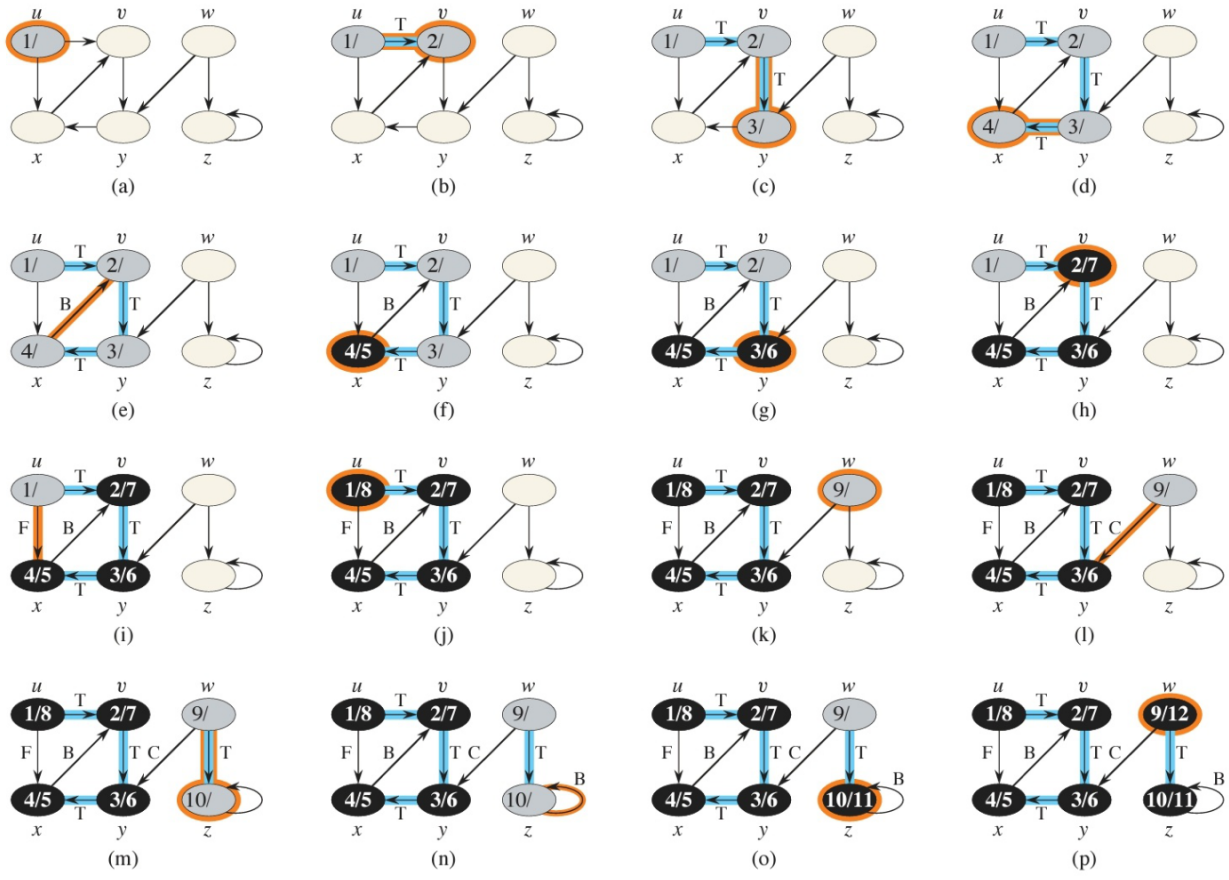


Figure 20.4 The progress of the depth-first-search algorithm DFS on a directed graph. Edges are classified as they are explored: tree edges are labeled T, back edges B, forward edges F, and cross edges C. Timestamps within vertices indicate discovery time/finish times. Tree edges are highlighted in blue. Orange highlights indicate vertices whose discovery or finish times change and edges that are explored in each step.

20.3-5

Show that in a directed graph, edge (u, v) is

a. a tree edge or forward edge if and only if $u.d < v.d < v.f < u.f$,

b. a back edge if and only if $v.d \leq u.d < u.f \leq v.f$, and

c. a cross edge if and only if $v.d < v.f < u.d < u.f$.

(a) Tree Edge or Forward Edge

We need to prove:

$u.d < v.d < v.f < u.f$

(b) and (c) Examples

- $u.d < v.d$: Vertex v is discovered while exploring u, making v a descendant of u.
- $v.f < u.f$: v finishes processing before u finishes, as v is a descendant of u.
- $v.d < v.f$: By definition, every vertex is finished after it is discovered.

Thus, $u.d < v.d < v.f < u.f$ holds for both tree edges and forward edges.

(b) Back Edge

We need to prove:

$v.d \leq u.d < u.f \leq v.f$

- $v.d \leq u.d$: v is an ancestor of u, so v is discovered before u.
- $u.f \leq v.f$: u finishes before v finishes, since u is a descendant of v.
- $u.d < u.f$: By definition of DFS.

Thus, $v.d \leq u.d < u.f \leq v.f$ holds for back edges.

(c) Cross Edge

We need to prove:

$v.f < u.d < u.f$

- $v.f < u.d$: v finishes processing before u is discovered, so v is neither an ancestor nor a descendant of u.
- $u.d < u.f$: By definition of DFS.

Thus, $v.f < u.d < u.f$ holds for cross edges.

20.3-6

Rewrite the procedure DFS, using a stack to eliminate recursion.

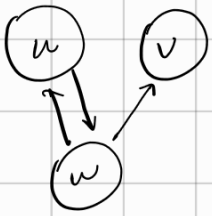
```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NULL                 ← parents
time = 0
for each vertex u ∈ G.V
    if u.color == white
        DFS-VISIT(G, u)        ← change
DFS-VISIT(G, u)                ← Recursive
time = time + 1                ⟶
u.d = time                     ⟶
u.color = GRAY                 ⟶
for each vertex v in G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
time += 1
u.f = time
u.color = BLACK
```

time = DFS-VISIT(G, u, time)

```
DFS-VISIT(G, u, time)
Let V_stack the stack to initialize
time += 1
u.d = time
u.color == GRAY
V_stack.push(u)

while V_stack is not empty:
    u = V_stack.top()
    all_neighbors_visited = TRUE

    for vertex v in G.Adj[u]: // Find an unvisited adjacent vertex
        if v.color == WHITE"
            v.color == GRAY
            v.pi = u
            time += 1
            v.d = time
            V_stack.push(v)
            all_neighbors_visited = False
            break       // Take only one adjacent vertex if found unvisited

    if all_neighbors_visited == TRUE:
        v.color = BLACK
        time += 1
        v.f = time
        V_stack.pop()

return time
```

20.3-7

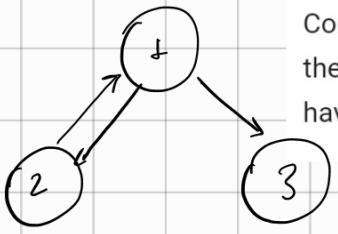
Give a counterexample to the conjecture that if a directed graph G contains a path from u to v, and if $u.d < v.d$ in a depth-first search of G, then v is a descendant of u in the depth-first forest produced.



Consider a graph with 3 vertices u, v, and w, and with edges (w, u), (u, w), and (w, v). Suppose that DFS first explores w, and that w's adjacency list has u before v. We next discover u. The only adjacent vertex is w, but w is already grey, so u finishes. Since v is not yet a descendant of u and u is finished, v can never be a descendant of u.

20.3-8

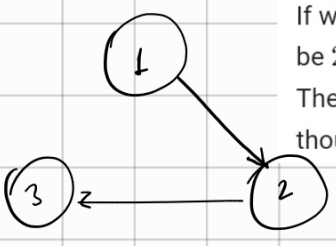
Give a counterexample to the conjecture that if a directed graph G contains a path from u to v, then any depth-first search must result in $v.d \leq u.f$.



Consider the directed graph on the vertices $\{1, 2, 3\}$, and having the edges $(1, 2)$, $(1, 3)$, $(2, 1)$ then there is a path from 2 to 3. However, if we start a DFS at 1 and process 2 before 3, we will have $2.f = 3 < 4 = 3.d$ which provides a counterexample to the given conjecture.

20.3-10

Explain how a vertex u of a directed graph can end up in a depth-first tree containing only u, even though u has both incoming and outgoing edges in G.



If we pick our first root to be 3, that will be in its own DFS tree. Then, we pick our second root to be 2, since the only thing it points to has already been marked BLACK, we won't be exploring it. Then, picking the last root to be 1, we don't screw up the fact that 2 is along in a DFS tree even though it has both an incoming and outgoing edge in G.

20.3-11

Let $G = (V, E)$ be a connected, undirected graph. Give an $O(V + E)$-time algorithm to compute a path in G that traverses each edge in E exactly once in each direction. Describe how you can find your way out of a maze if you are given a large supply of pennies.

Let's make use of the Hierholzer's Algorithm

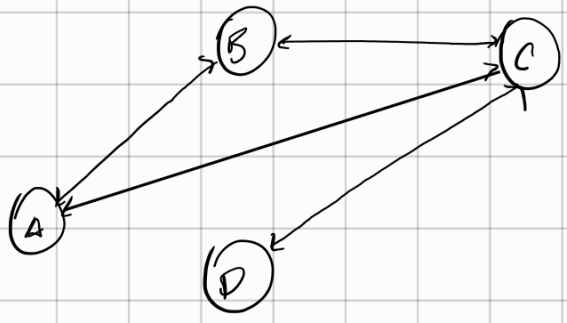
→ Each undirected edge is treated as two directed edges thus $|E| \to 2|E|$

→ For undirected graphs an Eulerian circuit exist if all vertices have even degrees

PSEUDOCODE:

```
EULERIAN_PATH(G): // G as adjacency list
Let V_stack be an initialized stack
path = []
current = start_vertex
```

```
while V_stack is not empty or G[current] exists:
    if not G[current] // No unused edges
        path.append(current)
        current = V_stack.top()
        V_stack.pop()
    else
        V_stack.push(current)
        next_vertex = G[current]
        Remove that edge
        Remove current from G[next_vertex]
        current = next_vertex
path.append(current)
```



Topological Sort

TOPOLOGICAL-SORT(G)

1 call DFS(G) to compute finish times $v.f$ for each vertex v
2 as each vertex is finished, insert it onto the front of a linked list
3 **return** the linked list of vertices

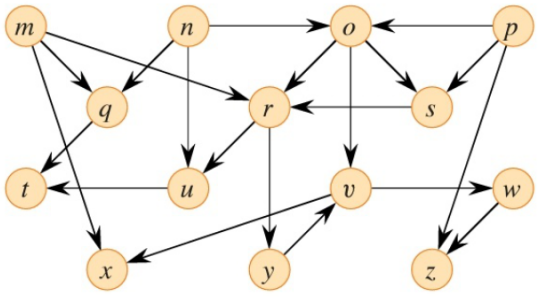
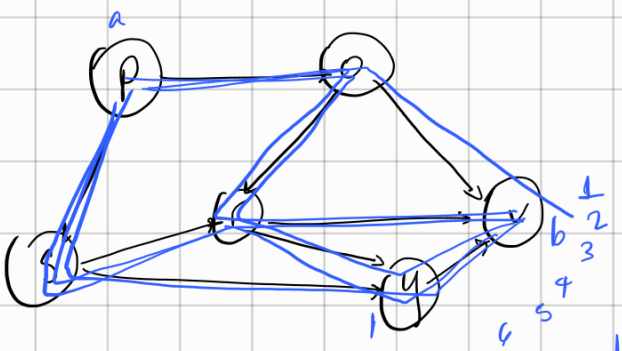


Figure 20.8 A dag for topological sorting.

20.4-2

Give a linear-time algorithm that, given a directed acyclic graph $G = (V, E)$ and two vertices $a, b \in V$, returns the number of simple paths from a to b in G. For example, the directed acyclic graph of Figure 20.8 contains exactly four simple paths from vertex p to vertex v: $\langle p, o, v \rangle$, $\langle p, o, r, y, v \rangle$, $\langle p, o, s, r, y, v \rangle$, and $\langle p, s, r, y, v \rangle$. Your algorithm needs only to count the simple paths, not list them.




```
COUNT SIMPLE PATHS (G, a, b)
let topo_order be a list
topo_order = TOPOLOGICAL SORT (G)
let count be array of size |V|
count[a] = 1
for each vertex u in topo_order
    for each neighbor v of u:
        count[v] = count[u] + count[v]
return count[b]
```

Def: Topological Sort takes $O(V+E)$

→ In the solution traversing the edges to compute the paths also takes $O(V+E)$

1. **Topological Sort**:
   - Perform a topological sort on G.
   - One valid order: $[p, o, s, r, y, v]$.
2. **Dynamic Programming**:
   - Initialize count:

Plaintext — Copy code

```
count[p] = 1, count[o] = 0, count[s] = 0, count[r] = 0, count[y] = 0, count[v] = 0
```

   - Process vertices in topological order:
     - p: $count[o] += count[p]$, $count[s] += count[p]$

Plaintext — Copy code

```
count[o] = 1, count[s] = 1
```

     - o: $count[v] += count[o]$, $count[r] += count[o]$

Plaintext — Copy code

```
count[v] = 1, count[r] = 1
```

     - s: $count[r] += count[s]$, $count[y] += count[s]$

Plaintext — Copy code

```
count[r] = 2, count[y] = 1
```

     - r: $count[v] += count[r]$, $count[y] += count[r]$

Plaintext — Copy code

```
count[v] = 3, count[y] = 3
```

     - y: $count[v] += count[y]$

Plaintext — Copy code

```
count[v] = 6
```

20.4-3

Give an algorithm that determines whether an undirected graph $G = (V, E)$ contains a simple cycle. Your algorithm should run in $O(V)$ time, independent of $|E|$.

Answer

An undirected graph is acyclic (i.e., a forest) if and only if a DFS yields no back edges.

- If there's a back edge, there's a cycle.
- If there's no back edge, then by Theorem 22.10, there are only tree edges. Hence, the graph is acyclic.

Thus, we can run DFS: if we find a back edge, there's a cycle.

- Time: $O(V)$. (Not $O(V + E)$!)
  If we ever see $|V|$ distinct edges, we must have seen a back edge because (by Theorem B.2 on p. 1174) in an acyclic (undirected) forest, $|E| \leq |V| - 1$.

```
FUNCTION ContainsSimpleCycle(graph, V):
    visited ← array of size V, initialized to FALSE

    FOR each vertex v IN V:
        IF visited[v] is FALSE:
            IF DFS(graph, v, -1, visited) is TRUE:
                RETURN TRUE  # Cycle found

    RETURN FALSE  # No cycles found
```

```
FUNCTION DFS(graph, current, parent, visited):
    visited[current] ← TRUE

    FOR each neighbor IN graph[current]:
        IF visited[neighbor] is FALSE:
            IF DFS(graph, neighbor, current, visited) is TRUE:
                RETURN TRUE  # Cycle detected
        ELSE IF neighbor ≠ parent:
            RETURN TRUE  # Back edge detected (cycle)

    RETURN FALSE  # No cycle found
```

20.4-5

Another way to topologically sort a directed acyclic graph $G = (V, E)$ is to repeatedly find a vertex of in-degree 0, output it, and remove it and all of its outgoing edges from the graph. Explain how to implement this idea so that it runs in time $O(V + E)$. What happens to this algorithm if G has cycles?

```
TOPOLOGICAL-SORT(G)
 // Initialize in-degree, Θ(V) time.
 for each vertex u ∈ G.V
     u.in-degree = 0
 // Compute in-degree, Θ(V + E) time.
 for each vertex u ∈ G.V
     for each v ∈ G.Adj[u]
         v.in-degree = v.in-degree + 1
 // Initialize Queue, Θ(V) time.
 Q = ∅
 for each vertex u ∈ G.V
     if u.in-degree == 0
         ENQUEUE(Q, u)
 // while loop takes O(V + E) time.
 while Q ≠ ∅
     u = DEQUEUE(Q)
     output u
     // for loop executes O(E) times total.
     for each v ∈ G.Adj[u]
         v.in-degree = v.in-degree − 1
         if v.in-degree == 0
             ENQUEUE(Q, v)
 // Check for cycles, O(V) time.
 for each vertex u ∈ G.V
     if u.in-degree ≠ 0
         report that there's a cycle
 // Another way to check for cycles would be to count the vertices
 // that are output and report a cycle if that number is < |V|.
```

To find and output vertices of in-degree 0, we first compute all vertices' in-degrees by making a pass through all the edges (by scanning the adjacency lists of all the vertices) and incrementing the in-degree of each vertex an edge enters.

- Computing all in-degrees takes $\Theta(V + E)$ time ($|V|$ adjacency lists accessed, $|E|$ edges total found in those lists, $\Theta(1)$ work for each edge).

We keep the vertices with in-degree 0 in a FIFO queue, so that they can be enqueued and dequeued in $O(1)$ time. (The order in which vertices in the queue are processed doesn't matter, so any kind of FIFO queue works.)

- Initializing the queue takes one pass over the vertices doing $\Theta(1)$ work, for total time $\Theta(V)$.

As we process each vertex from the queue, we effectively remove its outgoing edges from the graph by decrementing the in-degree of each vertex one of those edges enters, and we enqueue any vertex whose in-degree goes to 0. We do not need to actually remove the edges from the adjacency list, because that adjacency list will never be processed again by the algorithm: Each vertex is enqueued/dequeued at most once because it is enqueued only if it starts out with in-degree 0 or if its in-degree becomes 0 after being decremented (and never incremented) some number of times.

- The processing of a vertex from the queue happens $O(V)$ times because no vertex can be enqueued more than once. The per-vertex work (dequeue and output) takes $O(1)$ time, for a total of $O(V)$ time.
- Because the adjacency list of each vertex is scanned only when the vertex is dequeued, the adjacency list of each vertex is scanned at most once. Since the sum of the lengths of all the adjacency lists is $\Theta(E)$, at most $O(E)$ time is spent in total scanning adjacency lists. For each edge in an adjacency list, $\Theta(1)$ work is done, for a total of $O(E)$ time.

Thus the total time taken by the algorithm is $O(V + E)$.

The algorithm outputs vertices in the right order (u before v for every edge (u, v)) because v will not be output until its in-degree becomes 0, which happens only when every edge (u, v) leading into v has been "removed" due to the processing (including output) of u.

If there are no cycles, all vertices are output.

- Proof: Assume that some vertex v_0 is not output. Vertex v_0 cannot start out with in-degree 0 (or it would be output), so there are edges into v_0. Since v_0's in-degree never becomes 0, at least one edge (v_1, v_0) is never removed, which means that at least one other vertex v_1 was not output. Similarly, v_1 not output means that some vertex v_2 such that $(v_2, v_1) \in E$ was not output, and so on. Since the number of vertices is finite, this path $(\cdots \to v_2 \to v_1 \to v_0)$ is finite, so we must have $v_i = v_j$ for some i and j in this sequence, which means there is a cycle.

If there are cycles, not all vertices will be output, because some in-degrees never become 0.

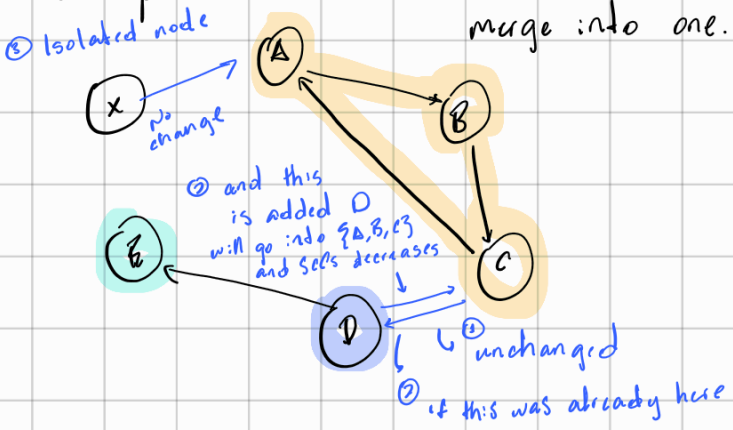
- Proof: Assume that a vertex in a cycle is output (its in-degree becomes 0). Let v be the first vertex in its cycle to be output, and let u be v's predecessor in the cycle. In order for v's in-degree to become 0, the edge (u, v) must have been "removed," which happens only when u is processed. But this cannot have happened, because v is the first vertex in its cycle to be processed. Thus no vertices in cycles are output.

20.5-1
How can the number of strongly connected components of a graph change if a new edge is added?

→ Two Possibilities:

1. Stays the same: The new edge connects two vertices that are already in the same SCC or does not affect the reachability between SCCs.
2. SCCs decreases: If the new edge connects two different SCCs and creates a path between them such that the two SCCs merge into one.

Example

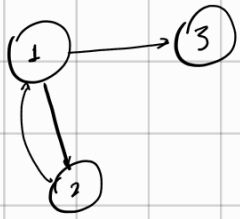


20.5-3
Professor Bacon rewrites the algorithm for strongly connected components to use the original (instead of the transpose) graph in the second depth-first search and scan the vertices in order of *increasing* finish times. Does this modified algorithm always produce correct results?

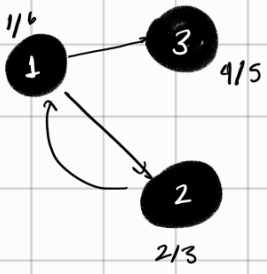
The modified algorithm doesn't produce correct results.

Take as consideration the following graph:

①



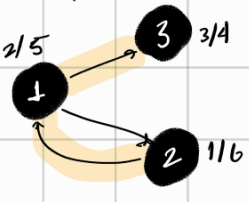
After running the first DFS graph becomes



Sorting it in increasing finishing times:

{2, 3, 1}

Now repeat the DFS in order of increasing finishing times, that is we start from vertex 2. Then DFS result in:



The tree edges are highlighted and from there we can see that we only have one SCC {1,2,3}, which is wrong! From the graph directly we see that node 3 can't get to any of the other nodes, and that the correct SCCs should be {1,2} {3}.

20.5-4

Prove that for any directed graph G, the transpose of the component graph of G^{T} is the same as the component graph of G. That is, $((G^{\mathrm{T}})^{\mathrm{SCC}})^{\mathrm{T}} = G^{\mathrm{SCC}}$.

(2%)

Strongly connected component: every vertex in the strongly connected component is reachable from every other vertex in the same strongly connected component.

(4%)

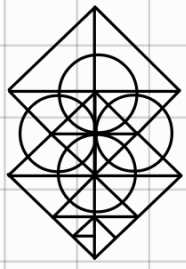
Assume that there are $node_a$ and $node_b$ in the same strongly connected component in G, as the definition of the strongly connected component, there is the path from $node_a$ to $node_b$ and another path from $node_b$ to $node_a$. Hence, $(G^T)^{SCC}$ and G^{SCC} have the same vertex sets (denoted v).

(4%)

Suppose that there is an edge (v_i, v_j) in $(G^T)^{SCC}$, then there will be an edge (v_j, v_i) in $((G^T)^{SCC})^T$. As a result, there exists the edge $(node_u, node_v)$ which $node_u \in v_i$ and $node_v \in v_j$ in G^T, and implies that $(node_v, node_u)$ is the edge in G which means that (v_j, v_i) is an edge in G^{SCC}. Therefore, $(G^T)^{SCC} = (G^{SCC})^T$. And $(G^T)^T = G$ for any graph, so $((G^T)^{SCC})^T = G^{SCC}$.

20.5-5

Give an $O(V + E)$-time algorithm to compute the component graph of a directed graph $G = (V, E)$. Make sure that there is at most one edge between two vertices in the component graph your algorithm produces.



To compute the SCCs of a graph, two of the algorithms that can be used are: (1) Kosaraju's and (2) Tarjan's algorithm and both runs on $O(V+E)$ of their complexity time.

→ Now what the problem is asking, that after finding the SCCs grap there is at least one edge that connects them:

Then let's demonstrate by an example:

Input Graph

Vertices:

$V = \{A, B, C, D, E, F\}$

Edges:

$E = \{A \to B, B \to C, C \to A, B \to D, D \to E, E \to F, F \to D\}$

Step 1: Compute SCCs (Assumption)

Using an $O(V + E)$ -time algorithm like Tarjan's or Kosaraju's, the SCCs are identified as follows:

- **SCC 1**: A, B, C
- **SCC 2**: D, E, F

SCC Mapping $u.scc$:

$u.scc = \{A : 1, B : 1, C : 1, D : 2, E : 2, F : 2\}$

Step 2: Post-Processing (Construct Component Graph)

2.1: Construct the Multiset T

$T = \{u.scc : u \in V\} = \{1, 1, 1, 2, 2, 2\}$

Sort T using **counting sort** (since $u.scc$ values are integers in $1, 2$):

$T_{\text{sorted}} = \{1, 2\}$

Extract the distinct SCCs:

$V^{SCC} = \{1, 2\}$

2.2: Construct the Edge Set S

Traverse all edges $(u, v) \in E$ and map them to SCCs using $u.scc$ and $v.scc$:

Edge (u, v)	$u.scc$	$v.scc$	Add to S?
$A \to B$	1	1	No (self-loop)
$B \to C$	1	1	No (self-loop)
$C \to A$	1	1	No (self-loop)
$B \to D$	1	2	Yes
$D \to E$	2	2	No (self-loop)
$E \to F$	2	2	No (self-loop)
$F \to D$	2	2	No (self-loop)

Resulting Edge Set S:

$S = \{(1, 2)\}$

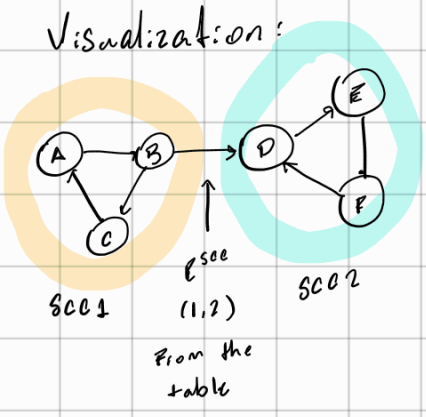
2.3: Sort S

Since $S = (1, 2)$ has only one edge, sorting does not change the result. $S_{\text{sorted}} = (1, 2)$.

2.4: Remove Duplicates

Remove duplicate edges in S_{sorted} (though there are none in this case). The resulting edge set:

$E^{SCC} = \{(1, 2)\}$



Final Component Graph

The **component graph** G^{SCC} is:

- **Vertices**: $V^{SCC} = 1, 2$
- **Edges**: $E^{SCC} = (1, 2)$

20.5-6

Give an $O(V + E)$-time algorithm that, given a directed graph $G = (V, E)$, constructs another graph $G' = (V, E')$ such that G and G' have the same strongly connected components, G' has the same component graph as G, and $|E'|$ is as small as possible.

Answer

The basic idea is to replace the edges within each SCC by one simple, directed cycle and then remove redundant edges between SCC's. Since there must be at least k edges within an SCC that has k vertices, a single directed cycle of k edges gives the k-vertex SCC with the fewest possible edges.

The algorithm works as follows:

1. Identify all SCC's of G. Time: $\Theta(V + E)$, using the SCC algorithm in Section 22.5.
2. Form the component graph G^{SCC}. Time: $O(V + E)$, by Exercise 22.5-5.
3. Start with $E' = \emptyset$. Time: $O(1)$.
4. For each SCC of G, let the vertices in the SCC be $v_1, v_2, \ldots, v_k$, and add to E' the directed edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k), (v_k, v_1)$. These edges form a simple, directed cycle that includes all vertices of the SCC. Time for all SCC's: $O(V)$.
5. For each edge (u, v) in the component graph G^{SCC}, select any vertex x in u's SCC and any vertex y in v's SCC, and add the directed edge (x, y) to E'. Time: $O(E)$.

Thus, the total time is $\Theta(V + E)$.