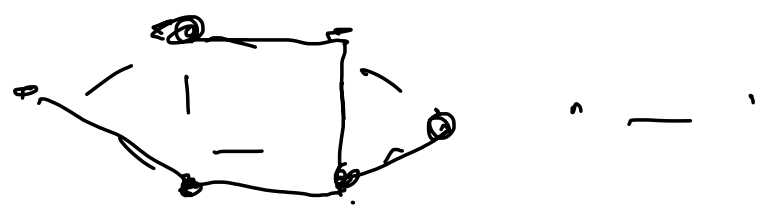
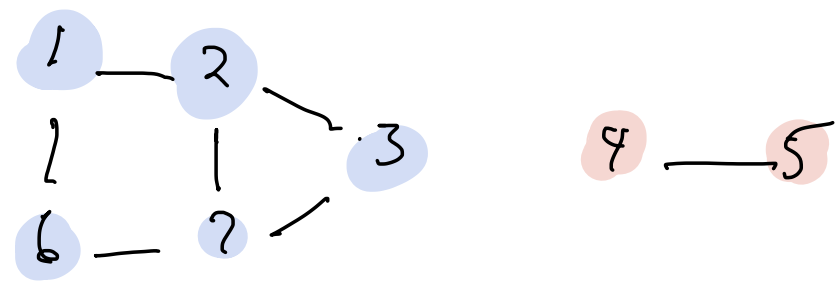
Data structures Packet 4

Depth-first search



```
void DFS(vertex here) {
    //assume visited[*] = false
    visited[here] = true;
    foreach neighbor (successors(here))
        if (!visited[neighbor]) DFS(neighbor);
}
```



Start at 2

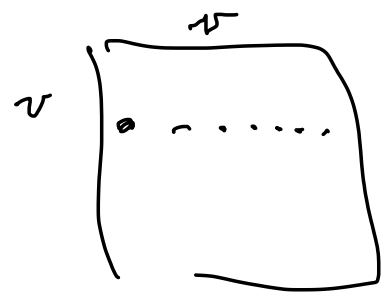
DFS(2) → DFS(1) → DFS(6) → DFS(7) → DFS(3)

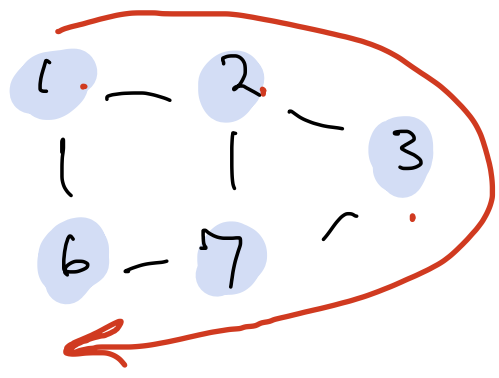
Adjacency list: $O(v' + e')$

↑ vertices in connected component

Adjacency matrix

$\Theta(v \cdot v)$

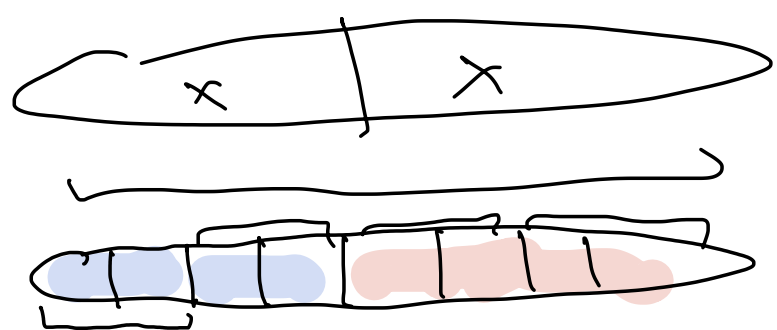




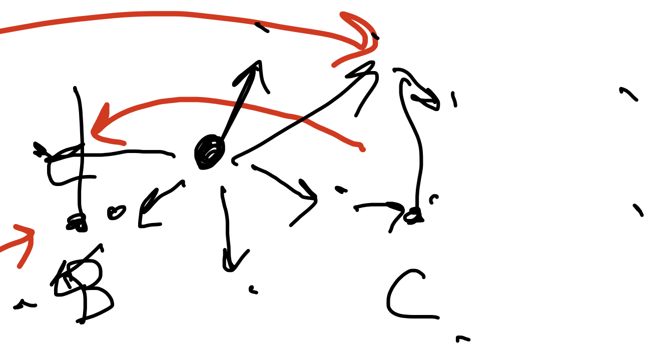

```
void DFS(vertex here) { //assume as before
    workStack = makeEmptyStack();
    push(workStack, here);
    while (!emptyStack(workStack)) {
        place = pop(workStack)
        if (visited[place]) continue;
        visited[place] = true;
        foreach neighbor (successors(place))
            if (!visited[neighbor])
                push(workStack, neighbor);
    } //while work to do
} // DFS()
```

Aside: iteration vs. recursion

mergeSort



Breadth first search.



applications

i) find how 2 people are related.

ii) find route through city streets.

iii) make fastest itinerary for plane travel.

odd: move anything else

unweighted graph (all edges are identical)

method: place unfinished work in a queue.

A B C

To tell if a graph is connected:

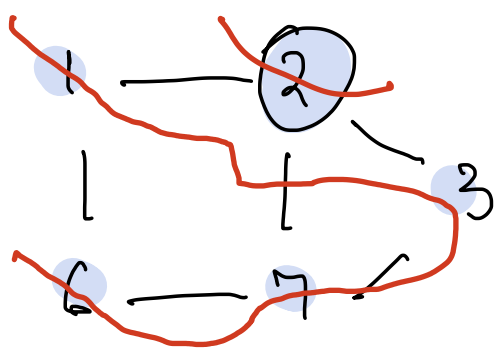
Do DFS from any vertex.

if any vertex is still unvisited, not connected.

otherwise: connected.

```
void BFS(vertex start) { // assume as before
    workQueue = makeQueue(); // Queue of vertices
    visited[start] = true;
    insertBack(workQueue, start);
    while (!emptyQueue(workQueue)) {
        place = deleteFromFront(workQueue);
        foreach neighbor (successors(place)) {
            if (!visited[neighbor]) {
                visited[neighbor] = true; place
                insertBack(workQueue, neighbor)
```

```
    } // not yet visited
  } // each neighbor
} // still work to do
} // BFS()
```

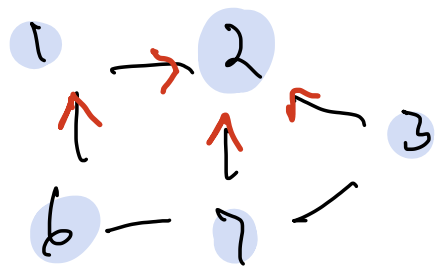


4 — 5

queue: ~~2~~ ~~1~~ ~~3~~ ~~7~~ ~~6~~

Enhancements to BFS

① to store path back to start:



4 5

② weighted edges: find closer vertices first.

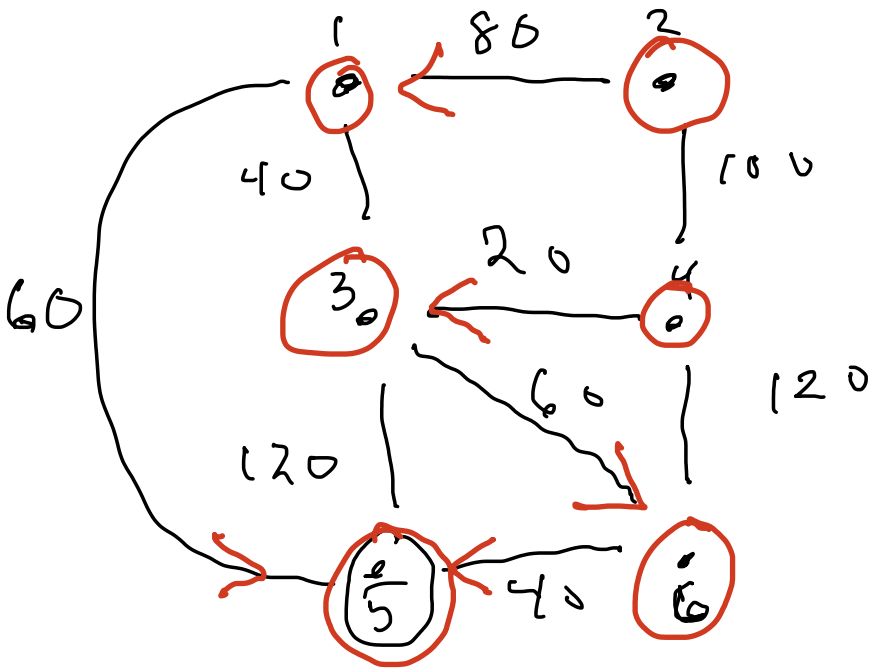
↑ distance

instead of a queue, use a priority queue (heap)

(top-light)

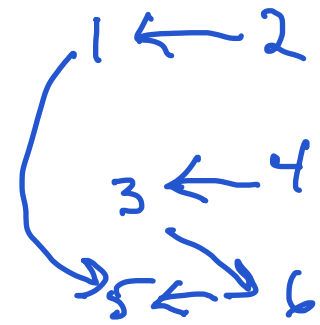
BFS for weighted graph

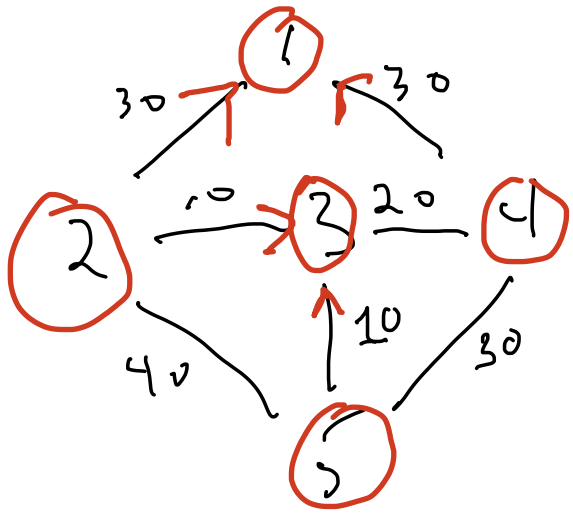
```
BFS(vertex start) {
    workHeap = makeHeap(); // top-light
    insertHeap(start, 0)
    while(!emptyHeap(workHeap)) {
        (place, distance) = delete(workHeap);
        visited[place] = (distance);
        foreach neighbor (...) {
            insertHeap(workHeap, neighbor, distance + edgeweight)
        }
    }
}
```

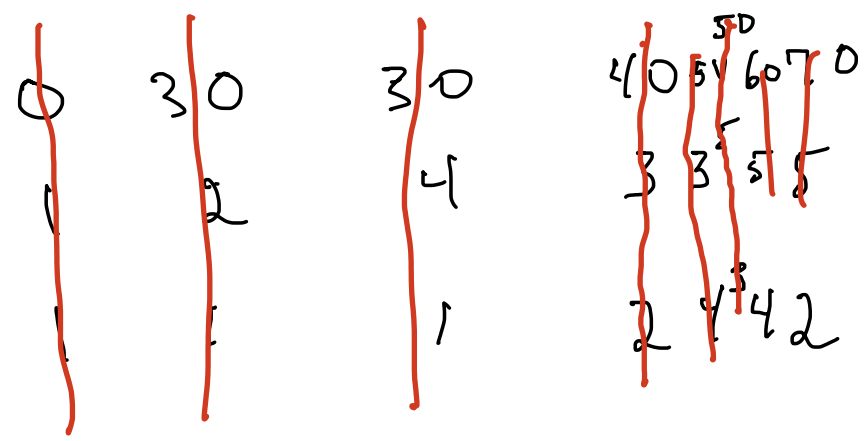


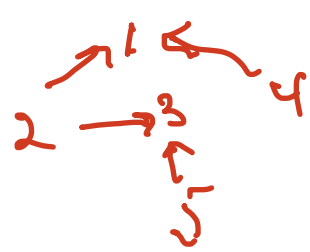
Heap

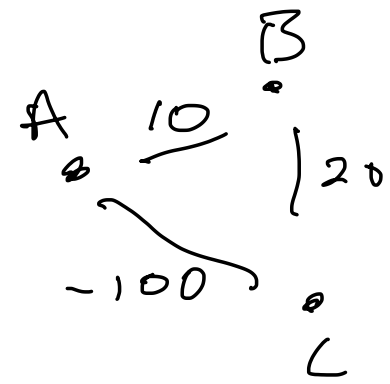
dist.	0	40		60	100	120	160	120	220
who	5	6		1	3	3	3	4	4	2
prd.	5	5		5	6	1	5	3	6	4









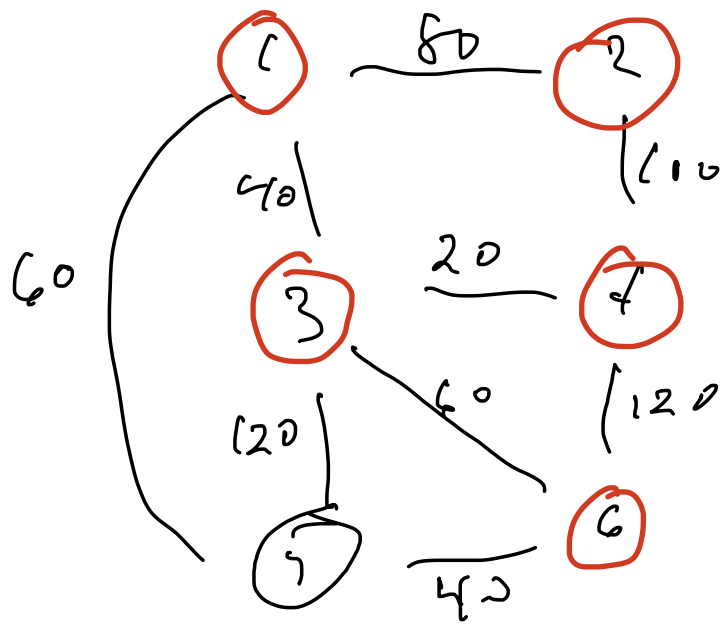


negative weights
breaks the algorithm
introduces ∞ cycles
therefore: renormalize to
positive weights by adding to
every weight.

Dijkstra's algorithm to find all shortest paths from a given vertex in a weighted undirected graph.

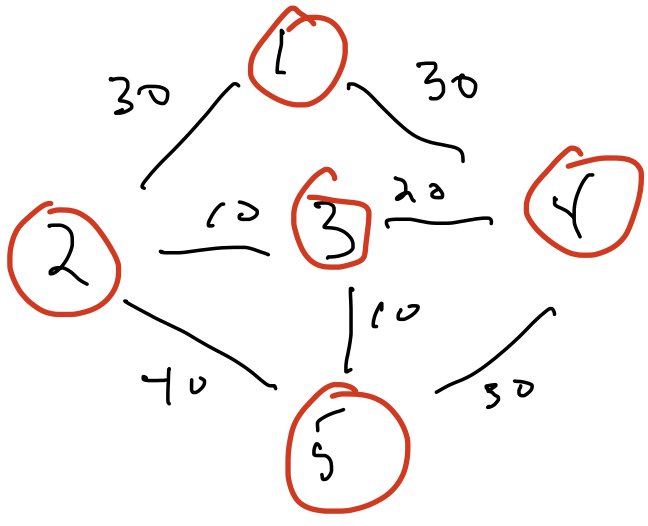
Rule: among all vertices that can extend a shortest path we have, choose one that results in shortest path.
Ties: going to same vertex, choose either.
going to different vertices, choose both.

"greedy" algorithm: at each step, improve the solution in the way that looks best.



path	length
5	0 ←
5 1	60
5 3	120
5 6	40 ←
5 1	60 ←
5 3	120
5 6 3	100
5 6 4	160
5 3	~~120~~
5 1 2	140
5 1 3	100 ←
5 6 3	~~100~~
5 6 4	160
5 1 2	140 ←
5 1 3 4	120 ←
~~5 6 4~~	~~160~~
5 1 3 4 2	220

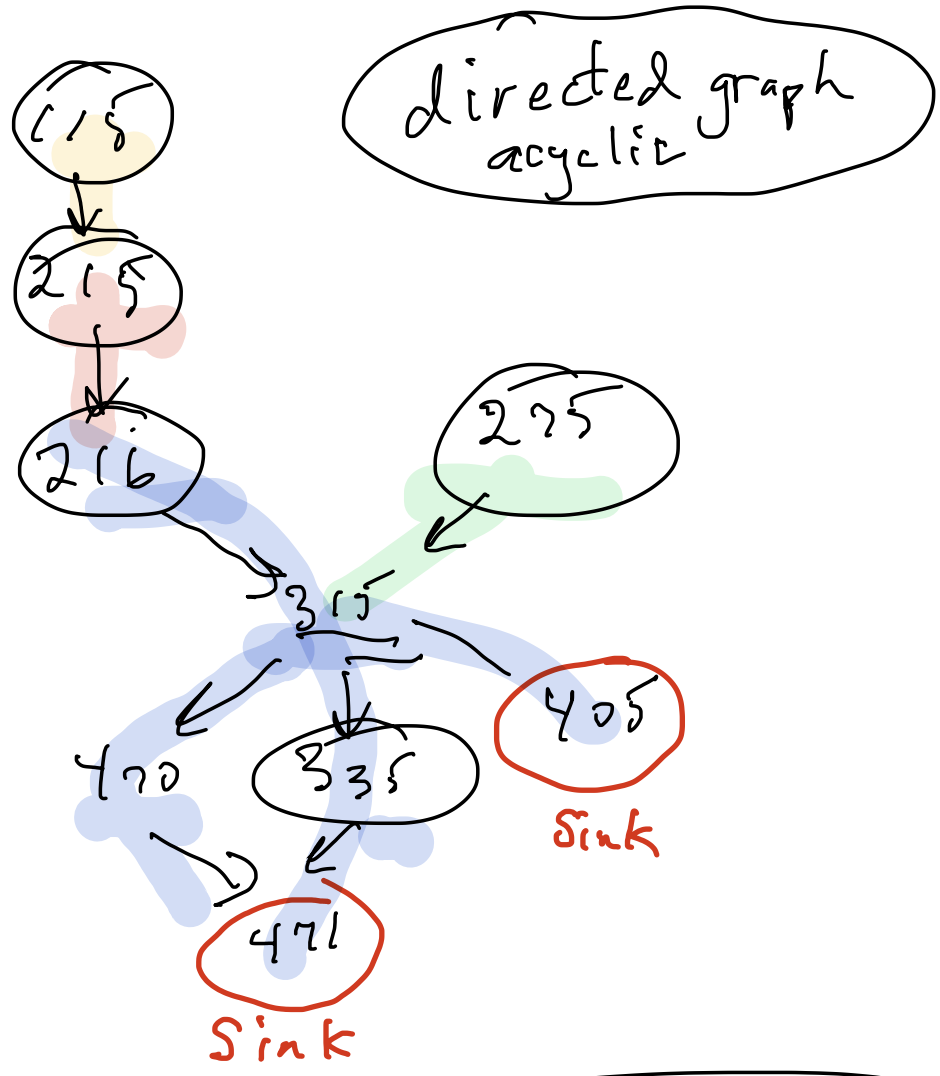
1 → 2
3 → 4
5 → 6



path	length
1	0 ←
1 2	30 ←
1 4	30 ←
1 2 3	40 ←
1 2 5	70
~~1 4 3~~	~~50~~
1 4 5	60
1 2 3 5	50 ←

1 → 2
1 → 4
2 → 3
3 → 5

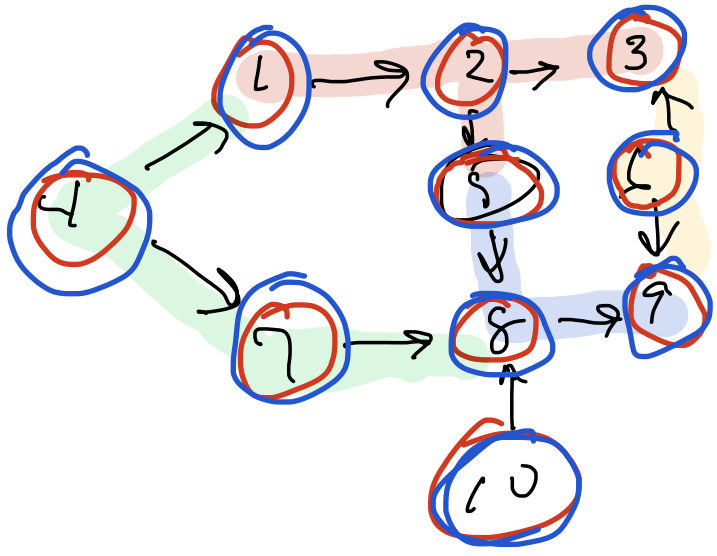
Topological Sort



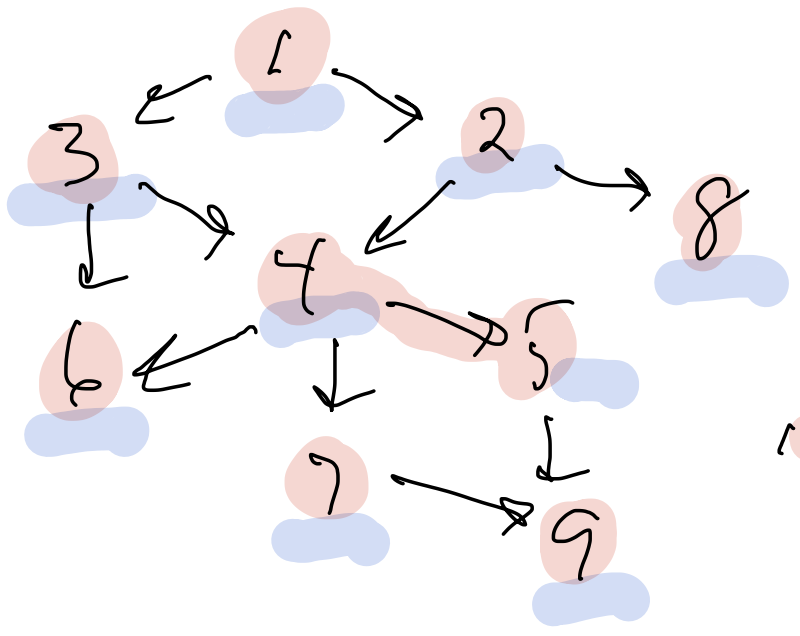
valid sequence.

115 275 215 216 315
405 335 470 471

115 275 215
216, 405 470 335 471
315



10 6 4 7 1 2 3 5 8 9

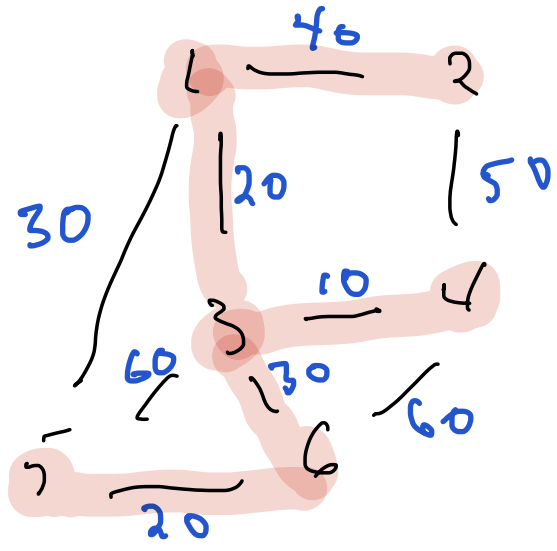


11

10

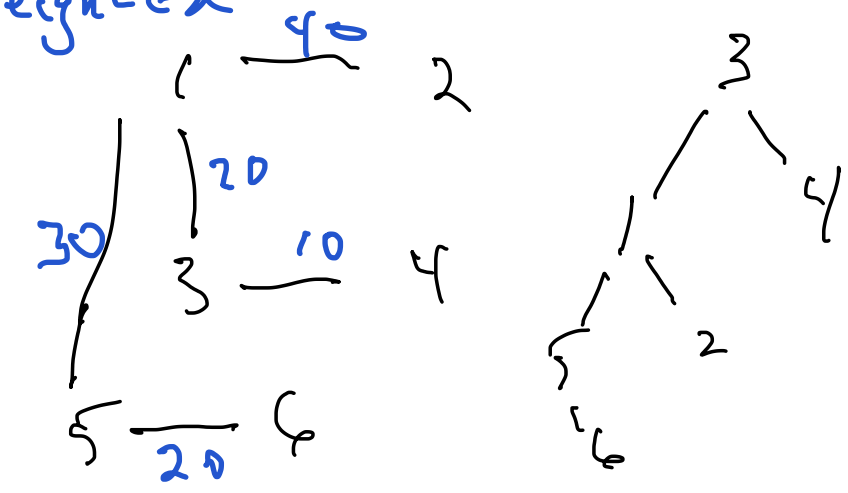
1 3 2 8 4 6 7 5 9

Spanning trees

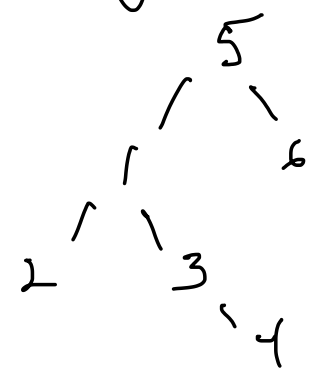


Connected, undirected graphs.

weighted



spanning tree: connected, acyclic, all vertices



minimum-weight

Prim's algorithm (add vertices)

tree = {a} where a is any vertex

do v-1 times:

tree ∪= {b} where b is the closest external vertex.

Implementation.

Heap (top-light) of all external vertices, using distance to current tree as the value for sorting.

Also store which vertex (edge) has that weight.

Initially: {a} = tree

heap = {neighbors of a, weights} ∪ {other vertices, ∞}

Loop: extract the first element b from heap.
for each of its neighbors,
recalculate entry in heap

↖ remove from heap (if new distance is better)

fix the weight, edge
put back in heap.

Complexity:

$\Theta(v + (\log v)e)$

↑ for each vertex

↖ readjusting external vertices.

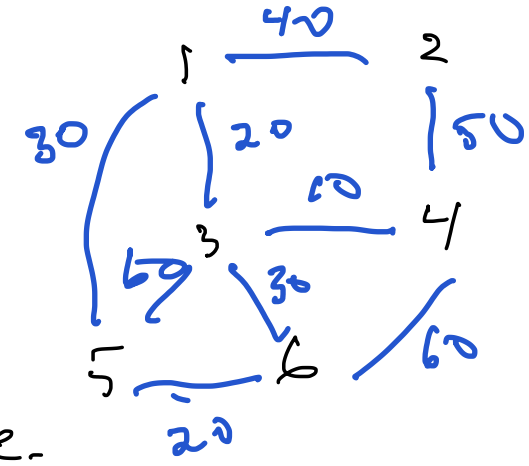
Kruskal's algorithm: (add edges)

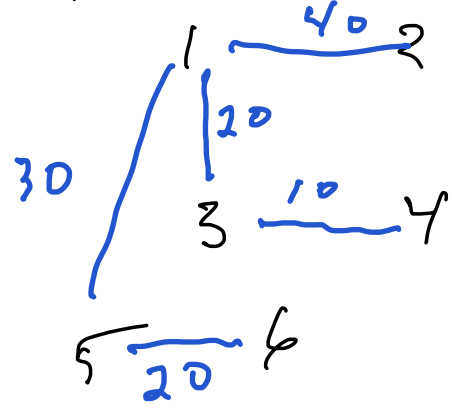
start with all vertices

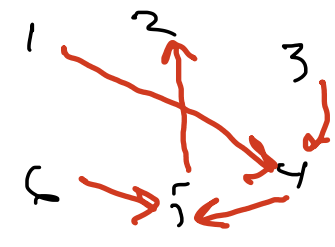
do v-1 times

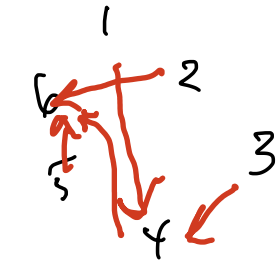
add best edge.

= edge with smallest weight that does not introduce a cycle.









Implementation: (1) sort all edges by weight.

(2) each next edge, see if adding it creates a cycle. If so, skip it.

Cycle detection (Union-find)

general idea: keep track for each vertex which component it belongs to.

operation: Connect two vertices (union)

operation: are two vertices already connected (find)

Data structure:

for each vertex, store its representative.

initially, the representative of a is a

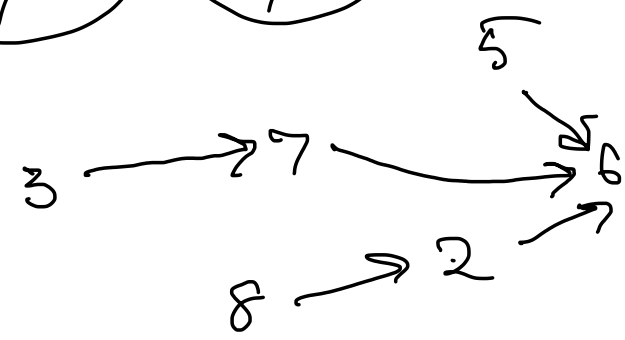
join a to b.

Find a's representative A. follow links

Find b's representative B.

make B the representative of A.

join (3,7) (8,2) (5,6)



find: (2,5) No.

join (2,5)

find: (3,2) No

union (3,2)

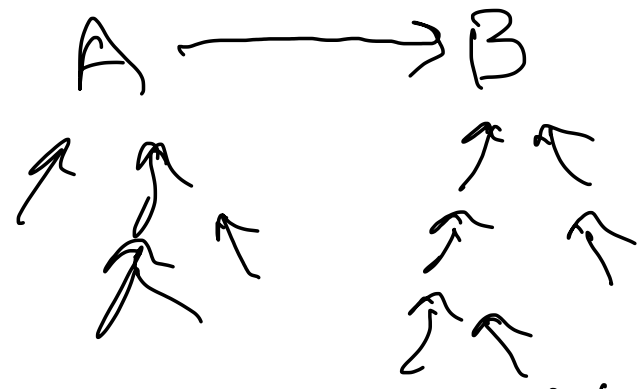
find (5,8) Yes. (in same component)

Improvements

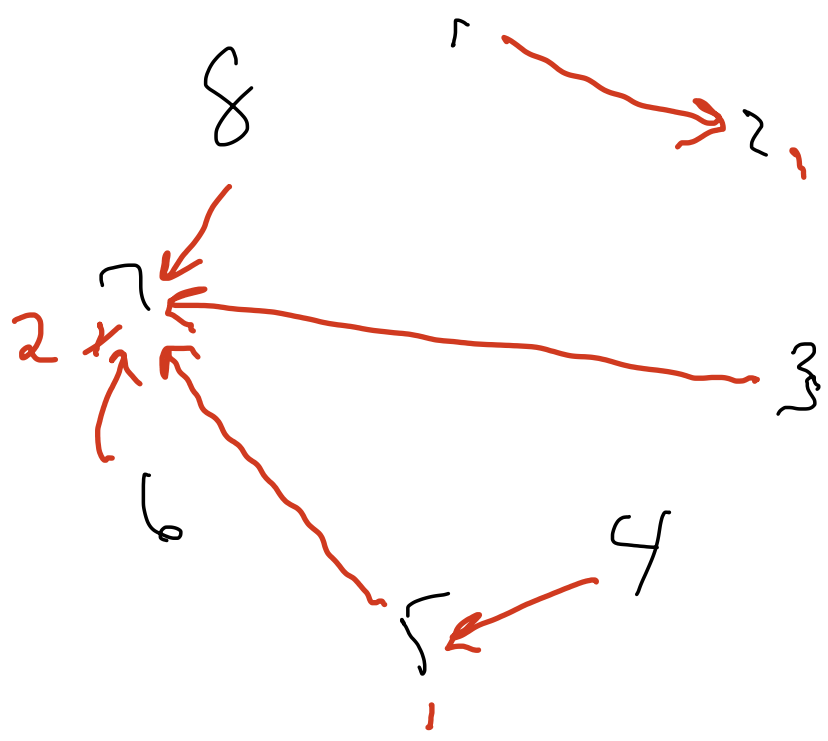
(1) Keep track of the height of each representative.

↑ length of the longest path to that representative.

to connect representatives A, B
arrow goes to the greater-height representative. Reason: avoid long paths.



(2) Compress paths as you follow them:
re-link vertices on path to its representative.
Reason: produces shorter paths.



	U	3	7	U	1	2
	U	4	5	U	6	4
✗	F	5	8	F	5	8 ✓
	U	4	3	F	1	4 ✗
✗	F	5	8			
	U	3	8			
✓	F	5	8			

Why Union-Find? Kruskal's algorithm for minimum weight spanning tree.

Complexity of Union-Find —

How expensive to do a Union?

⇒ How expensive to find representative?

$\Theta(\log^* n)$

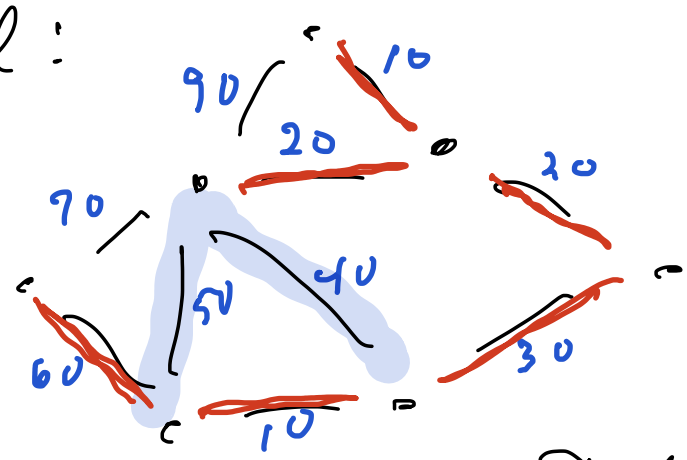
↑ # of times you take log until result is < 1.

$a = 31415926$

$\log_{10} a \simeq 8$

$\log_{10}(\log_{10} a) = .9$

Kruskal:



Data structure for Union-Find

```
typedef struct vertex_s {
    int name;
    struct vertex_s *representative;
        // null ⇒ me
    int depth; // 0 initially
}
```

Numerical algorithms

Not real numbers, not approximate solutions.
(consider CS 321)
Instead, integer algorithms.

Euclidean algorithm

to compute greatest common divisor of two integers.

gcd(·,·)

gcd(12,60) = 12
gcd(15,66) = 3

gcd(15,67) = 1

3·5 2·3·11

```
int gcd(int a, int b) {
    while (b != 0) {
        (a, b) = (b, a % b);
    }
    return a;
}
```

a	12	60	12
b	60	12	0

a	15	66	15	6	3
b	66	15	6	3	0

a	15	67	15	7	1
b	67	15	7	1	0

---

gcd(33, 279)

a	33	279	33	15	3
b	279	33	15	3	0

Fast exponentiation

$243^{745} \bmod 452$

$243 \cdot 243 \cdot \cdots \cdot 243$ (745)

why?

cryptography.

Example. $a^{64} = (((((a^2)^2)^2)^2)^2)^2$

a^4

a^8

a^{16}

a^{32}

a^{64}

$a^5 = a^4 \cdot a = (a^2)^2 \cdot a$

$a^{10} = ((a^2)^2 \cdot a)^2$

a^4, a^5, a^{10}

Pattern:

express exponent in base 2.

examine exponent left -> right.

accumulator, starts a 1.

see 0: square accumulator

see 1: square, multiply by a

a^{10} : $10_w = 1010_2$

$(((1^2 \cdot a)^2)^2 \cdot a)^2$

a^{24}

$24_{10} = 11000$

$((((1^2 \cdot a)^2 \cdot a)^2)^2)^2$

$((((a^2 \cdot a)^2)^2)^2$

a^5

If need to mod, do it repeatedly after every step.

$(a^2)^2 \bmod b$ $\quad ((a \bmod b)^2 \bmod b)^2 \bmod b$

Complexity: $a^p \bmod b$

# of steps = # of bits in p. $= \log_2 p$

# of multiplications $\leq$ 2 · # of steps

$= 2 \log_2 p$

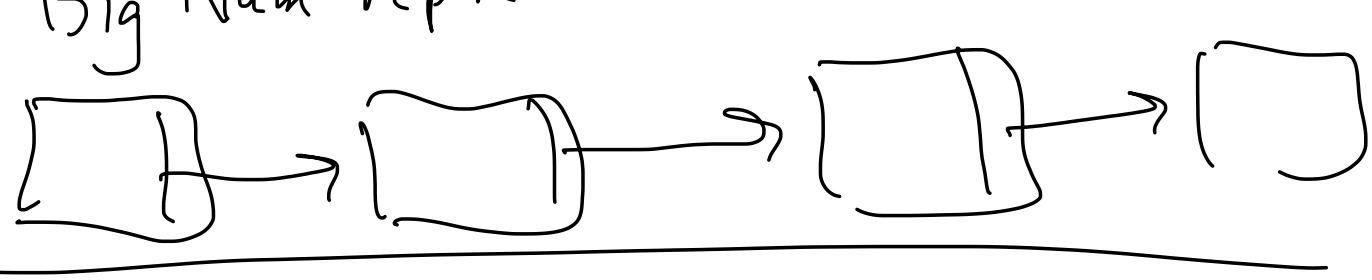
# of mod operations: $\log_2 p$

total # of operations: $\Theta(\log p)$

much better than $\Theta(p)$

New question: How to multiply large integers?

Hint: Big Num representation.



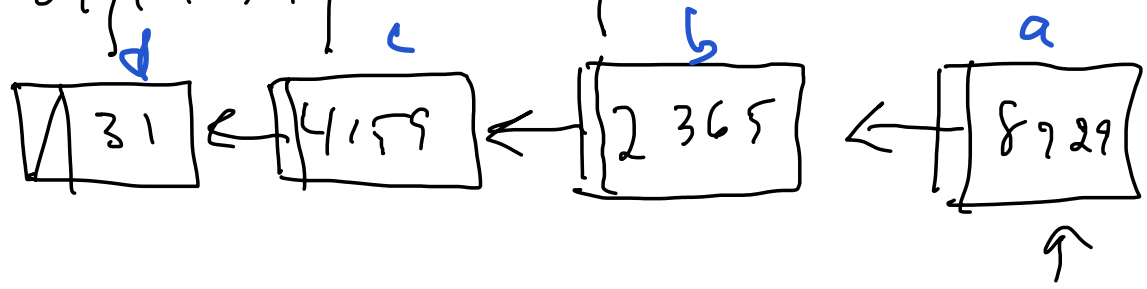
$26^{47} \bmod 17$ $\qquad a^{47} \bmod m$

$47_{10} = 101111_2$

$(((((1^2 \cdot a)^2)^2 a)^2 a)^2 a)^2 a$

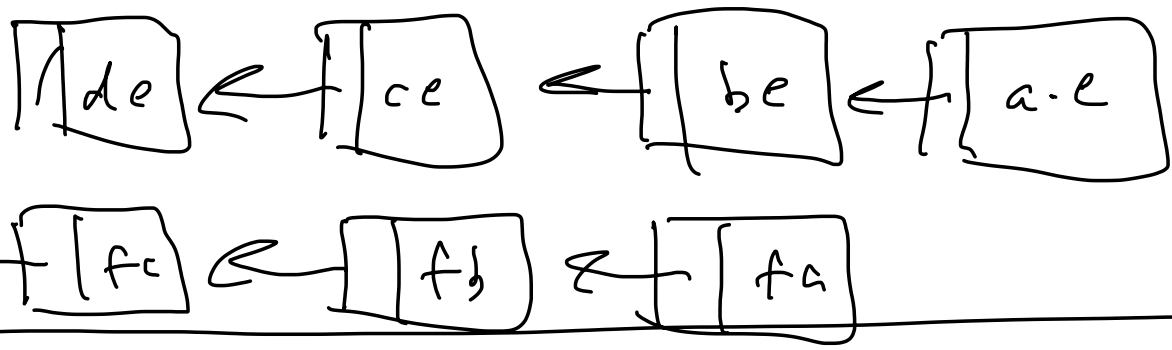
mod m

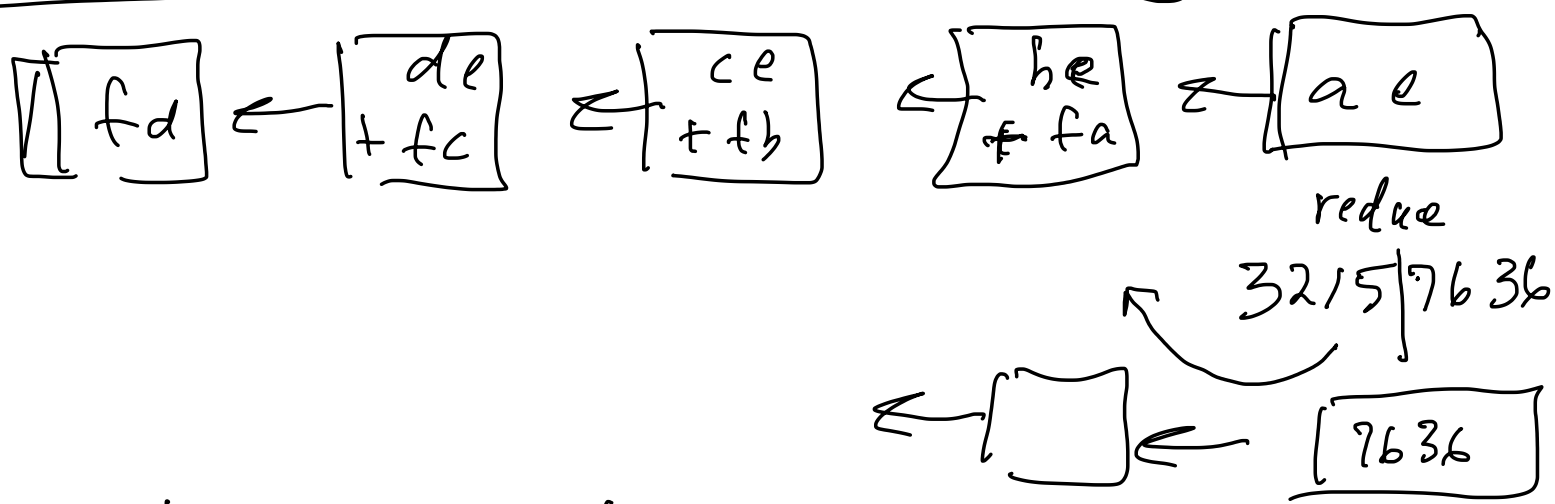
Consider: 31|4159|2365|8729



0..9999

multiply by 42|3684





Complexity: n chunks in both operands

Calculate $n \cdot n$ new chunks (before reducing)

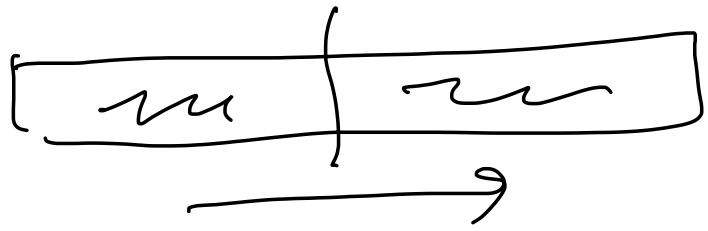
Add about n times, total of $n \cdot n$ chunks.

Reductions: n chunks.

total cost: $\Theta(n^2)$

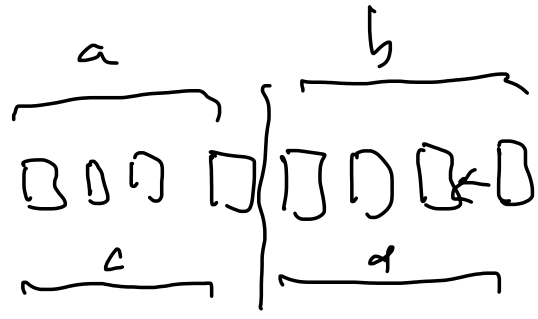
Can we do better?

Anatoly Karatsuba (1962) divide+conquer algorithm.



merge sort

multiply $X \cdot y$



$$X = a \cdot 10^{n/2} + b$$
$$y = c \cdot 10^{n/2} + d$$

n = # of chunks (8 in this example)
10 = limit per chunk (before I used 10000)

$$xy = \underbrace{a \cdot c \cdot 10^{n}}_{\frac{n^2}{4}} + \underbrace{(bc + ad)10^{n/2}}_{\frac{n^2}{4} + \frac{n^2}{4}} + \underbrace{bd}_{\frac{n^2}{4}} \simeq n^2$$

$$C_n \simeq n^1 + 4C_{n/2}$$

$K \quad a \quad b$

$a = 4 \quad b^K = 2$

$\Theta(n^{\log_b a})$

$\Theta(n^{\log_2 4})$

$\Theta(n^2)$

$X = a\,10^{n/2} + b \qquad y = c\,10^{n/2} + d$

$u = ac \qquad w = (a+b)(c+d)$

$v = bd$

$x \cdot y = u\,10^{n} + (w - u - v)10^{n/2} + v$

$$w - u - v \simeq \cancel{ac} + \underline{ad + bc} + \cancel{bd}$$

$$C_n = \underbrace{n}_{k}^{1} + \underbrace{3}_{a} \underbrace{C_{n/2}}_{b}$$

$a \quad b^k$

$3 > 2$

$\Theta(n^{\log_b a})$

$\Theta(n^{\log_2 3})$

$\simeq \Theta(n^{1.585})$

"super linear"

"sub quadratic"

(At $n = 100$ (x, y are 100 chunks long) cost reduces from 10000 ops to 1480 ops.

How big should the chunk limit be?

1) should fit in integer.
2) double size should fit in an integer.
3) reduction fast: mask, shift $\Rightarrow$ power of 2.
4) 64-bit arithmetic

$\Longrightarrow$ 31 bits of a chunk. limit $= 2^{31} - 1 \simeq 2G$.

---

text t: this is the forest primeval...

$|t| = n$

↑ length

pattern p: es

$|p| = m$