How to improve MergeSort RT

1. Stop recursion at larger value of n. (experimental evidence: n = 11)
2. Use a single temp array for all instances of merge.
3. Don't copy back from temp to A in each call to merge.

```
mergeSort (A)
{ create a tmp array, same size as A
if (mergeSort (A, tmp, 0, A.size - 1) == 1) {
    copy tmp to A }
}

int mergeSort (A, tmp, p, r)                    // Assume that n is a power of 2.
{ n ← r - p + 1                                 // threshold = 11 ?
  if n ≤ threshold then
        use insertion sort to sort A[p..r]
        return 0
  else  q ← (r+p)/2
        t1 ← mergeSort (A, tmp, p, q)
        t2 ← mergeSort (A, tmp, q+1, r)

// t1 = t2 = 0  ⇒ data is in A
// t1 = t2 = 1  ⇒ data is in tmp
  if (t1 == 0) {
        merge (A, tmp, p, q, r)
        return 1
  } else {
        merge (tmp, A, p, q, r)
        return 0
  }
```

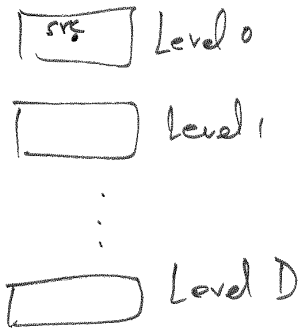
```
merge (A, B, p, q, r)
i ← p    j ← q+1
for k ← p to r do
    if (j > r or (i ≤ q and A[i] ≤ A[j]))
                B[k++] ← A[i]
    else        B[k++] ← A[j].
```

Why does BFS help to find odd cycles?

- Assume that G is connected

BFS(G, src):

Each node at Level i (i>0) has a parent node at Level i-1.



All edges of G: (u,v):

$|u.d - v.d| \leq 1$

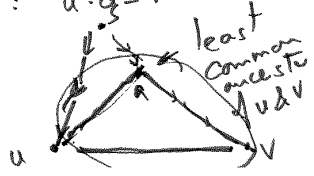
If G has no edges (u,v): $u.d = v.d$

then G is bipartite:

$X = \{u \mid u.d \,\%\, 2 = 0\}$

$Y = \{u \mid u.d \,\%\, 2 = 1\}$

Edge (u,v): $u.d = v.d$




```
ua ← u
while (ua != src) {
    ua ← ua.parent;
}
```

After BFS,

Find cycle (u,v) // $u.d = v.d$

```
{
    ua ← u.parent;
    va ← v.parent;
    while (ua != va) {
        ua ← ua.parent;
        va ← va.parent;
    }
}
```

Output = Union of all nodes ua, va, ... u, v.

Better code:

```
FindCycle (u,v)   // Precondition u.d = v.d after BFS
    List<vertex> l1 = new LinkedList<>(); l1.add(u);
    List<vertex> l2 = new LinkedList<>(); l2.add(v);
Vertex ua ← u.parent ; Vertex va ← v.parent;
    while (ua != va) { l1.add(ua) ; l2.ad (va);
                       ua ← ua.parent; va = va.parent;
                     }
    l1.add (ua);
    return l1 + reverse (l2).
```

Priority Queues

Extension of Queue, where each object has a priority field

Remove function – return object with maximum priority
(several objects with same max priority — any of them can be returned).

Operations:

Traditional name		Java	
Insert (x)	——	add (x)	– Add a new element to queue
DeleteMin ( ) / ExtractMin ( )	——	remove ( )	– Remove and return element with max priority

Note that PQ's don't have find (x) function.
Contains (x)

Min ( )	——	Peek ( )	– return max priority element without removing it

BuildHeap — Given an array, convert it into a binary heap.

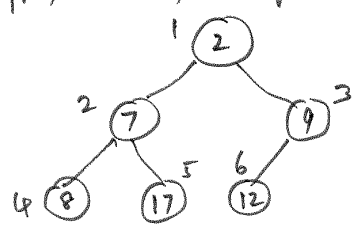
Extended operations

Decreasekey (x) — Priority of x has been decreased
Update priority queue to reflect this change.
— not implemented in Java.

Binary Heap implementation

Binary tree: (a) complete binary tree — structure constraint
(b) Heap order property: a node's priority is higher than the priority of its children
↳ true at all nodes.

Mapping binary heaps to arrays:



Mapping #1: root → 1
Node at index i: children → 2i, 2i+1

Node at index i: parent: $\lfloor i/2 \rfloor$
children: 2i, 2i+1

·	2	7	9	8	17	12