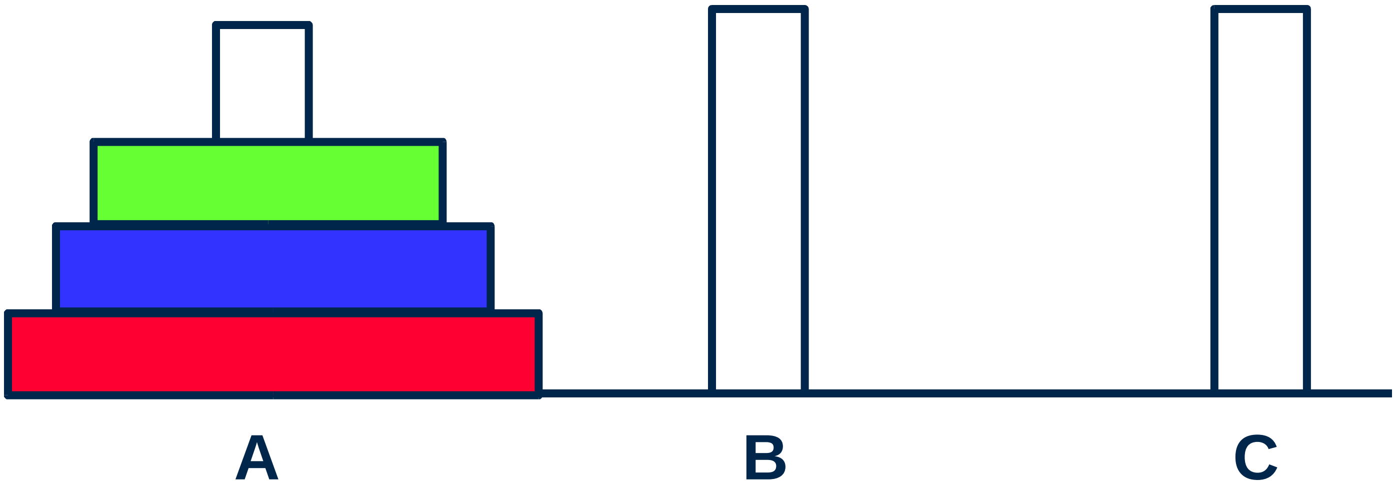
The Towers of Hanoi



Goal: Move stack of rings to another peg

- **Rule 1: May move only 1 ring at a time**
- **Rule 2: May never have larger ring on top of smaller ring**

Recursive Towers of Hanoi

```
public void toh(int n, char from, char inter,
  char to) {
  if (n == 1)
    System.out.println(“disk 1 from “ +
      from + ” to “ + to);
  else {
    toh(n-1, from, to, inter); // from->inter

    System.out.println(“disk “ + n + “
      from “ + from + “ to “ + to);

    toh(n-1, inter, from, to); // inter->to
  }
}
```

Towers of Hanoi - Complexity

- For 1 rings we have 1 operation.
- For 2 rings we have 3 operations.
- For 3 rings we have 7 operations.
- For 4 rings we have 15 operations.
- In general, the cost is $2^n - 1 = O(2^n)$
- Each time we increment **n**, we double the amount of work.
- This grows incredibly fast!

The Wise Peasant's Pay

Day(n)	**Pieces of Grain**
1	**2**
2	**4**
3	**8**
4	**16**
...	
63	**9,223,000,000,000,000,000**
64	**18,450,000,000,000,000,000**

2^N

How Bad is 2^n?

- Imagine being able to grow a billion (1,000,000,000) pieces of grain a second…

- It would take
  - **585 years** to grow enough grain just for the 64th day
  - Over a thousand years to fulfill the peasant's request**!**

It can get worse

- Ackerman's function
  - public long ack(m,n) {
    if (m==0) return n+1;
    if (n==0) return ack(m-1,1);
    return ack(m-1, ack(m, n-1)); }
  - ack(3, 65553) = 2^65553-3
  - ack(4,2) is greater than the number of particles in the universe

Divide-and-Conquer

- Recall the recursive binary search.
- It divides a big problem into two smaller parts and solves each one separately.
- Subdivision keeps going in each half until solution is reached.
- Divide-and-Conquer is a prime candidate for a recursive method using two recursive calls.

Mergesort

- Yet another sorting algorithm!
- More efficient than any of the sorting algorithms seen so far.
- The idea is to divide and conquer: divide the array in half, sort each independently, then merge the two already sorted arrays.
- All the work is in merging.

Recursive Mergesort

```
public void mergesort(long[] result, int
  lower, int upper) {
  if (lower == upper) return;
  else {
    int mid = (lower+upper)/2;
    // sort lower half
    mergesort(result, lower, mid)
    // sort upper half
    mergesort(result, mid+1, upper);
    // merge
    merge(result, lower, mid+1, upper);
}
```

Merge

```
public void merge(long[] a, int low,
                  int high, int highend) {
  int i=0, lowstart=low,lowend=high-1;
  int mid=high-1;
  while (low<=lowend && high<=highend)
    if theArray[low] < theArray[high]
      a[i++] = theArray[low++];
    else a[i++] = theArray[high++];
  while (low<=mid) // if high ended
    a[i++] = theArray[low++];
  while (high <= highend) // if low ended
    a[i++] = theArray[high++];
  for(i=0; i<highend-lowstart+1; i++)
    theArray[lowstart+i] = a[i];
}
```

Mergesort Efficiency

- Number of copies
  - There are log (n) number of sorting levels
  - At each level there are n copies
  - Total nlog(n)
- Number of comparisons for each merge
  - worse case: 1 less than the number of copies
  - best case: half of the number of copies
- Mergesort is nlog(n)