Summary of big O

- Running time of an algorithm is described by some function t(n): from natural numbers to running times (say rational numbers).
- Since we don't know how long basic steps such as array accesses or integer comparisons take, we replace them by constants.
- So we get something like $t(n) = c_1 + c_2 \log_2 n$ for binary search or $t(n) = c_1 n^2 - c_2 n + c_3$ for bubble sort.

Summary of big O continued

- Then we classify algorithms into broad categories according to how fast their running time grows.
- We use the notion of asymptotic upper bound O() to do this. Intuitively, $t(n) \in O(f(n))$ if f(n) grows at least as fast as t(n), it is an upper bound on t(n).
- Formally, $t(n) \in O(f(n))$ if there are constants n_0 and k, such that: for all $n > n_0$, $t(n) \leq k f(n)$.
- For example, $c_1 + c_2 \log_2 n \in O(\log_2 n)$ and $c_1 n^2 - c_2 n + c_3 \in O(n^2)$.

Question from last lecture

- Prove that bubble sort's running time is not in O(n) (its growth rate is greater than linear), in other words,

$c_1 n^2 - c_2 n + c_3 \notin O(n)$

- Assume that c_1 , c_2 and c_3 are positive (c_1 has to be positive for the function to be positive, we already proved $c_1 n^2 + c_2 n + c_3 \notin O(n)$, and proof for negative c_3 will be very similar).

Question from last lecture

- Proof by contradiction. Assume that for some n_0 and k, for all $n > n_0$,

$c_1 n^2 - c_2 n + c_3 \leq k n$.

Then $c_1 n^2 + c_3 \leq k n + c_2 n$. Since c_3 is positive, $c_1 n^2 \leq k n + c_2 n$. Dividing both sides by n, we get: for all $n > n_0$, $c_1 n \leq k + c_2$. Since c_1 , c_2 and k are positive , this is a contradiction. So, $c_1 n^2 - c_2 n + c_3 \notin O(n)$.

Merge sort

- Recursion
- Divide and conquer algorithms
- Merge sort

Recursion

- An algorithm is recursive if it calls itself to do part of its work
- Has to call itself on smaller problems and have a *base case* if it is ever to terminate
- Base case is the case which can be solved without recursive call
- Examples: binary search; reversing a list...

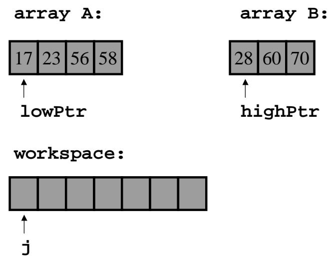
Divide and conquer algorithms

General strategy:

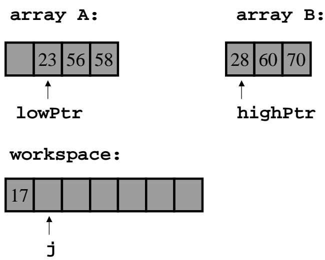
- Split the problem into subproblems
- Solve subproblems
- Combine the results

- Could be implemented recursively or not
- Examples: merge sort in this lecture

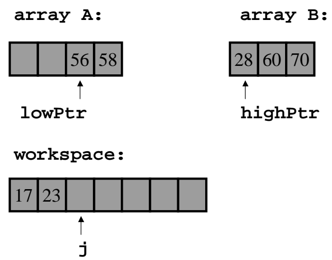
Merging sorted arrays



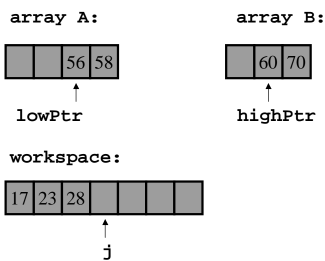
Merging sorted arrays



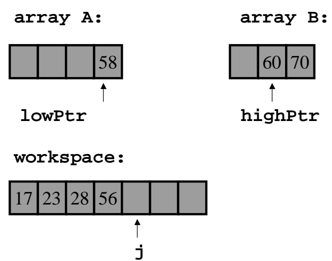
Merging sorted arrays



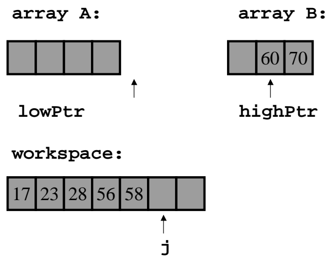
Merging sorted arrays



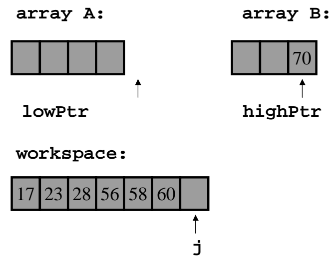
Merging sorted arrays



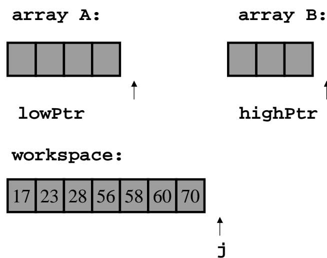
Merging sorted arrays



Merging sorted arrays



Merging sorted arrays



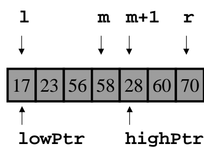
Merge sort

Recursive algorithm:

- split an array in two halves
- call merge sort on each half
- merge sorted halves

Merging halves of an array

Pass boundaries of subarrays to the algorithm instead of new arrays:



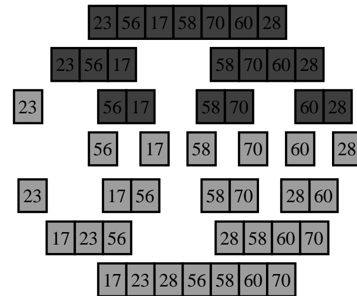
Workspace

- Use workspace to merge sorted halves into.
- Exercise: could you merge the two halves back into the array without using extra space?
- Could you do it in linear time?

Implementation

```
private static void recMergeSort(
int[] arr, int[] workSpace, int l, int r) {
    if(l == r){
        return;
    } else {
        int m = (l+r) / 2;
        recMergeSort(arr, workSpace, l, m);
        recMergeSort(arr, workSpace, m+1, r);
        merge(arr, workSpace, l, m+1, r);
    }
}
```

Example



Complexity

Time complexity:

- levels of recursion: $\log_2 N$
- at each level: merging N items
- time complexity: $O(N \log_2 N)$

Space complexity: O(N)