Sorting

Fall 2020

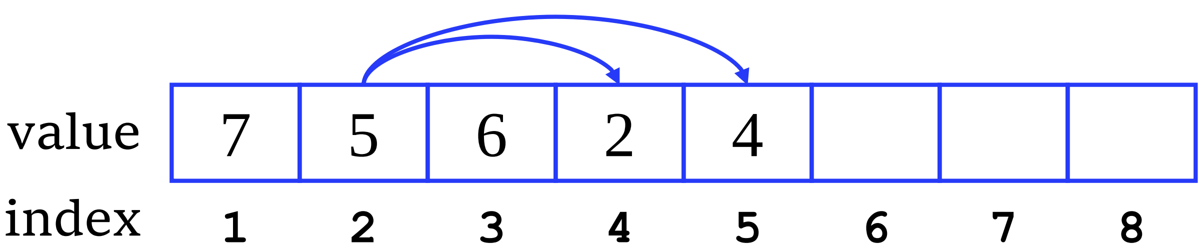
School of Software Engineering

South China University of Technology

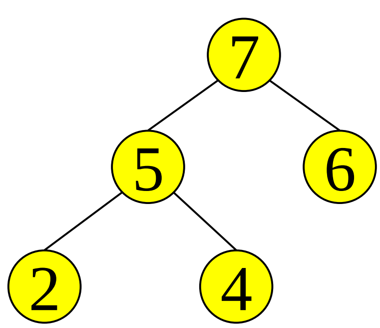
O(NlogN) Sorting Algorithms

Heap Sort

- We use a Max-Heap
- Root node = A[1]
- Children of A[i] = A[2i], A[2i+1]
- Keep track of current size N (number of nodes)

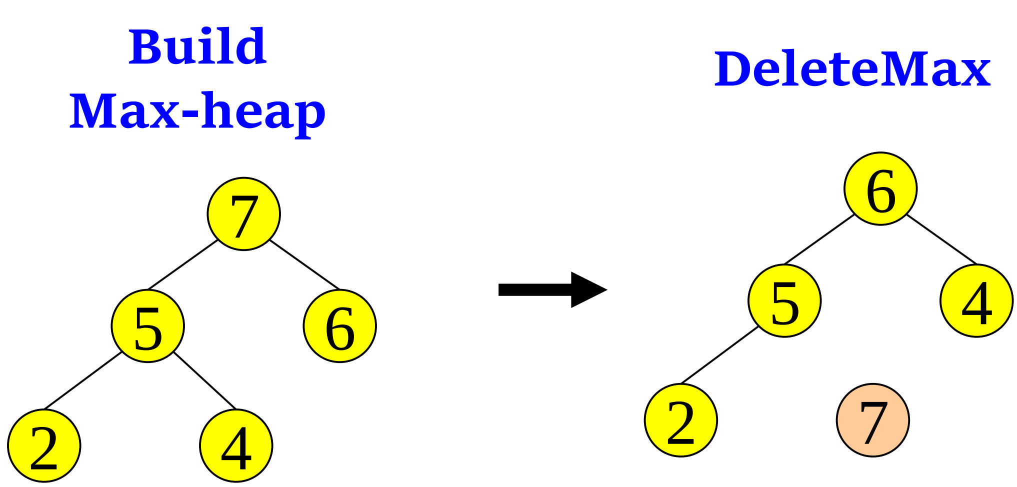


N = 5



Using Binary Heaps for Sorting

- Build a max-heap
- Do N DeleteMax operations and store each Max element as it comes out of the heap
- Data comes out in largest to smallest order
- Where can we put the elements as they are removed from the heap?

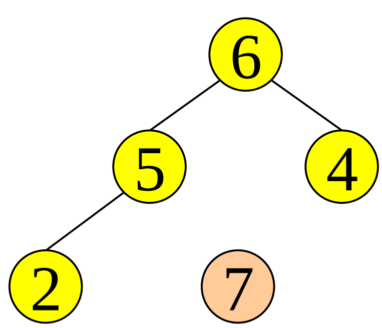


1 Removal = 1 Addition

- Every time we do a DeleteMax, the heap gets smaller by one node, and we have one more node to store
  - Store the data at the end of the heap array
  - Not "in the heap" but it is in the heap array

value	6	5	4	2	7			
index	1	2	3	4	5	6	7	8

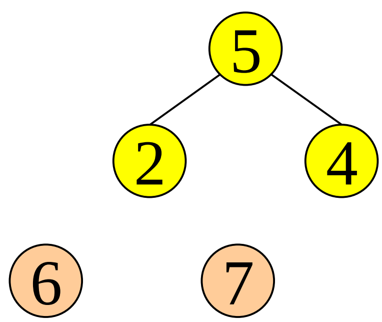
N = 4



Repeated DeleteMax

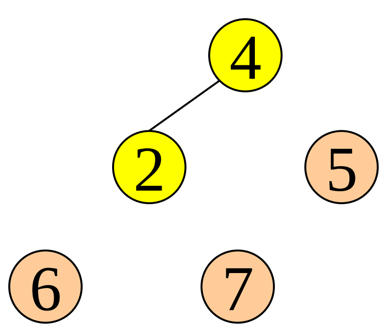
5	2	4	6	7			
1	2	3	4	5	6	7	8

N = 3



4	2	5	6	7			
1	2	3	4	5	6	7	8

N = 2

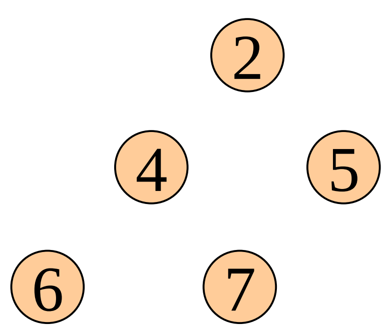


Heap Sort is In-place

- After all the DeleteMaxs, the heap is gone but the array is full and is in sorted order

value	2	4	5	6	7			
index	1	2	3	4	5	6	7	8

N = 0



Heapsort: Analysis

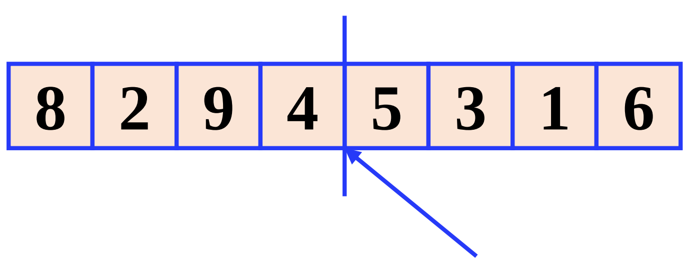
- Running time
  - time to build max-heap is $O(N)$
  - time for N DeleteMax operations is N $O(\log N)$
  - total time is **$O(N \log N)$**
- Can also show that running time is $\Omega(N \log N)$ for some inputs,
  - so *worst case* is **$\Theta(N \log N)$**
  - *Average case* running time is also $O(N \log N)$
- Heapsort is in-place but not stable (why?)

"Divide and Conquer"

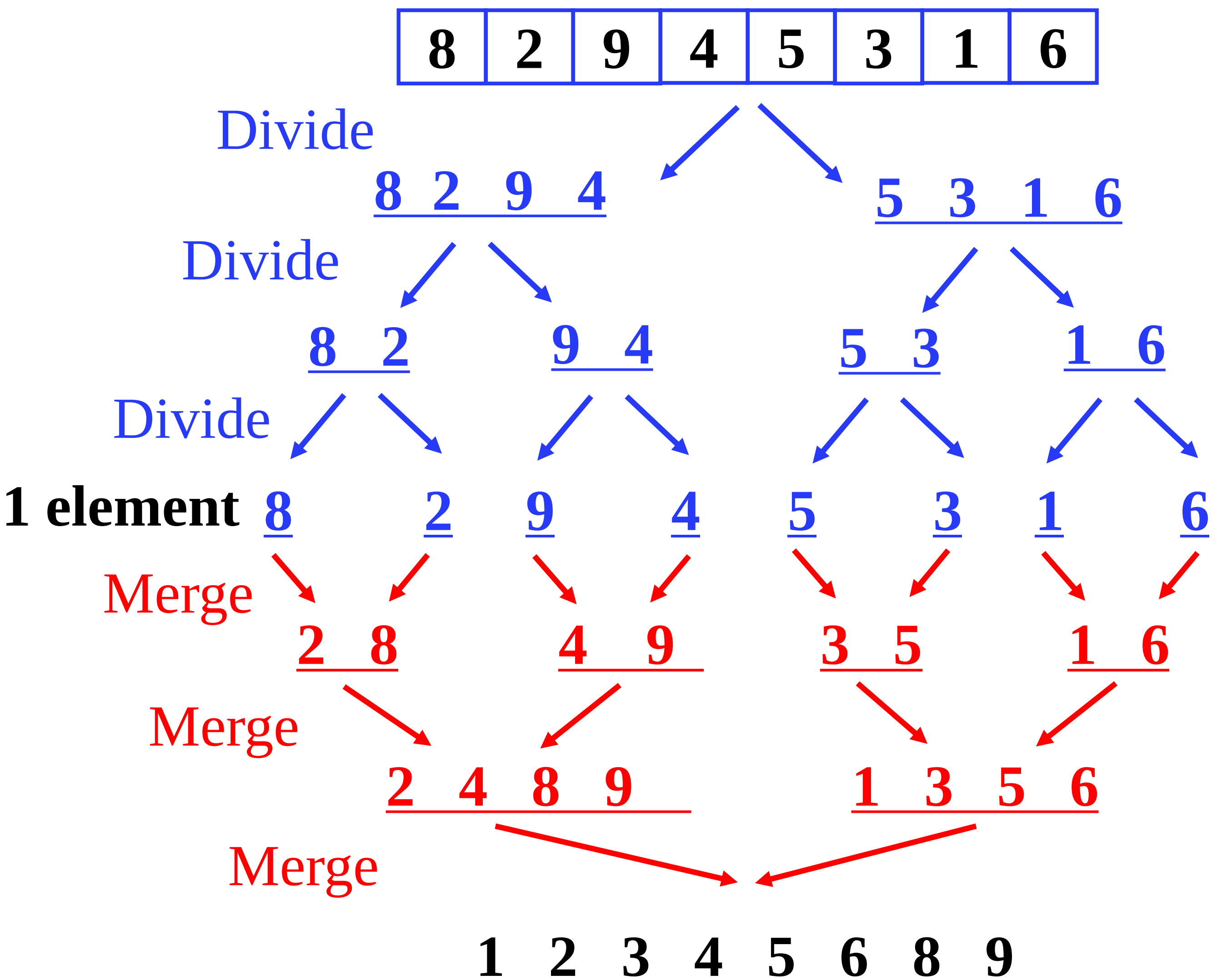
- Very important strategy in computer science:
  - Divide problem into smaller parts
  - Independently solve the parts
  - Combine these solutions to get overall solution
- **Idea 1**: Divide array into two halves, *recursively* sort left and right halves, then *merge* two halves → Mergesort
- **Idea 2 :** Partition array into items that are "small" and items that are "large", then recursively sort the two sets → Quicksort

MergeSort

- Divide it in two at the midpoint
- Conquer each side in turn (by recursively sorting)
- Merge two halves together

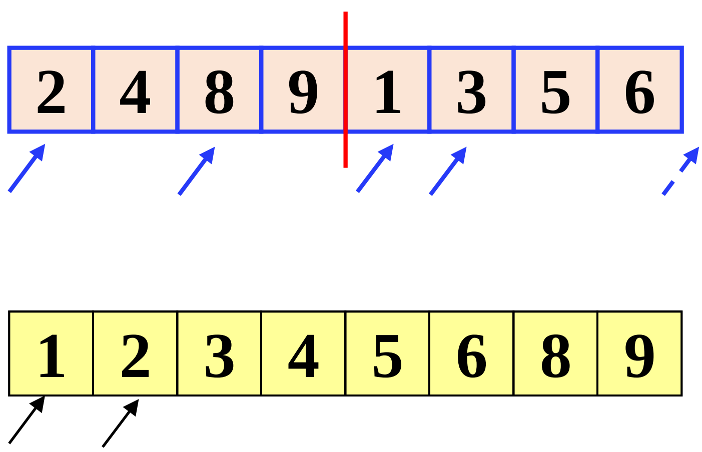


Mergesort Example



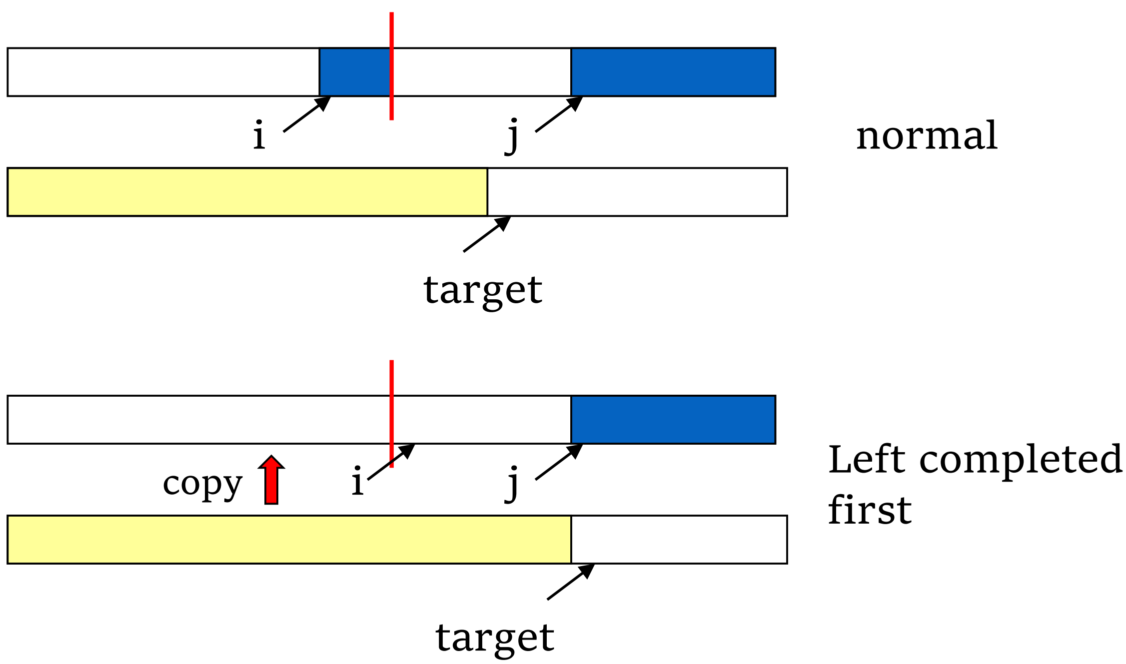
Auxiliary Array

- The merging requires an auxiliary array.

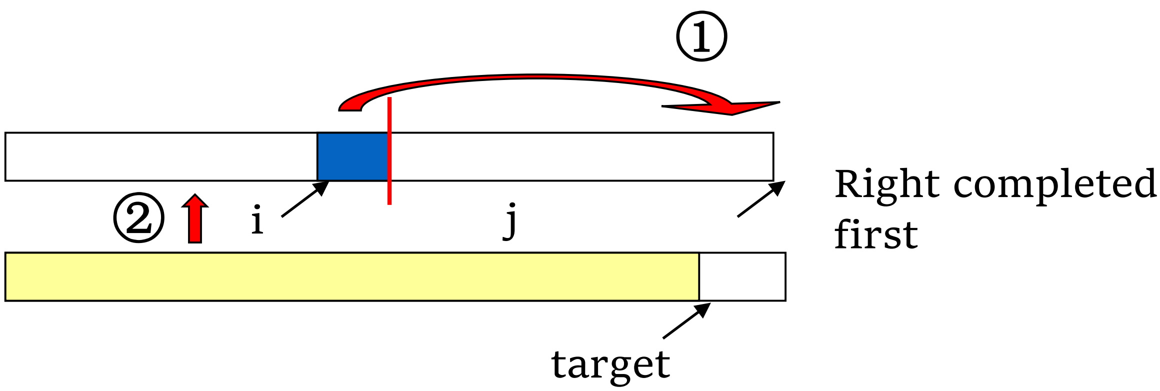


Auxiliary array

Merging



Merging



Merging Algorithm

```
/**
* pseudo code for Merging Algorithm in
MergeSort
**/
Merge(A[], T[] : integer array, left, right : integer)
{
 mid, i, j, k, l, target : integer;
 mid := (right + left)/2;
 i := left; j := mid + 1; target := left;
 while i ≤ mid and j ≤ right do
  if A[i] ≤ A[j] then T[target] := A[i] ; i:= i + 1;
   else T[target] := A[j]; j := j + 1;
  target := target + 1;
 if i > mid then //left completed//
  for k := left to target-1 do A[k] := T[k];
 if j > right then //right completed//
  k : = mid; l := right;
  while k ≥ i do A[l] := A[k]; k := k-1; l := l-1;
  for k := left to target-1 do A[k] := T[k];
}
```

Recursive Mergesort

```
/**
* Recursive implementation for Mergesort
**/
Mergesort(A[], T[] : integer array, left, right : integer) :
{
 if left < right then
   mid := (left + right)/2;
   Mergesort(A,T,left,mid);
   Mergesort(A,T,mid+1,right);
   Merge(A,T,left,right);
}

//Driver
MainMergesort(A[1..n]: integer array, n : integer) : {
 T[1..n]: integer array;
 Mergesort[A,T,1,n];
}
```

Another Impl for Mergesort

- Make use of Insertion Sort to sort small arrays.
- No test is needed to check for when one of the two subarrays becomes empty

Another Impl for Mergesort

```
template <typename E, typename Comp>
void mergesort(E A[], E temp[], int left, int right) {
  if ((right-left) <= THRESHOLD) { //small list
    insertionsort<E,Comp>(&A[left], right-left+1);
```

	0	1	2	3	4	5	6
Temp	A[0]	A[1]	A[2]	A[3]	A[6]	A[5]	A[4]

```
  mergesort<E, Comp>(A, temp, left, mid);
  mergesort<E, Comp>(A, temp, mid+1, right);
  //Do the merge operation. First copy two halves to
temp
  for (i=mid; i>=left; i--) temp[i] = A[i];
  for (j=1; j<=right-mid; j++) temp[right-j+1] =
A[j+mid];
  //Merge sublists back to A
  for (i=left, j=right, k=left; k<=right; k++)
    if (Comp::prior(temp[i], temp[j])) A[k] = temp[i++];
    else A[k] = temp[j--];
}
```

Another Impl for Mergesort

```
template <typename E, typename Comp>
void mergesort(E A[], E temp[], int left, int right) {
  if ((right-left) <= THRESHOLD) { //small list
    insertionsort<E,Comp>(&A[left], right-left+1);
    return;
  }
  int i, j, k, mid = (left+right)/2;
  if (left == right) return;
  mergesort<E, Comp>(A, temp, left, mid);
  mergesort<E, Comp>(A, temp, mid+1, right);
  //Do the merge operation. First copy two halves to
temp
  for (i=mid; i>=left; i--) temp[i] = A[i];
  for (j=1; j<=right-mid; j++) temp[right-j+1] =
A[j+mid];
  //Merge sublists back to A
  for (i=left, j=right, k=left; k<=right; k++)
    if (Comp::prior(temp[i], temp[j])) A[k] = temp[i++];
    else A[k] = temp[j--];
}
```

Mergesort Analysis

- Let T(N) be the running time for an array of N elements
- Mergesort divides array in half and calls itself on the two halves. After returning, it merges both halves using a temporary array
- Each recursive call takes T(N/2) and merging takes O(N)

Mergesort Recurrence Relation

- The recurrence relation for T(N) is:
  - $T(1) \leq a$
    - base case: 1 element array → constant time
  - $T(N) \leq 2T(N/2) + bN$
    - Sorting N elements takes
      - the time to sort the left half
      - plus the time to sort the right half
      - plus an O(N) time to merge the two halves
- How to calculate T(N) ?

Mergesort Analysis

- One calculation
  - Divide the T(N) = 2T(N/2)+bN through by N

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + b$$

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + b$$

$$\frac{T(N/4)}{N/4} = \frac{T(N/8)}{N/8} + b$$

…

$$\frac{T(2)}{2} = \frac{T(1)}{1} + b$$

  - add up all the equations,then

$$\frac{T(N)}{N} = \frac{T(1)}{1} + b\log N = a + b\log N$$

$$T(N) = aN + bN\log N$$

- So, T(N) = O(n log n)

Properties of Mergesort

- Not in-place
  - Requires an auxiliary array (O(n) extra space)
- Stable
  - Make sure that left is sent to target on equal values.

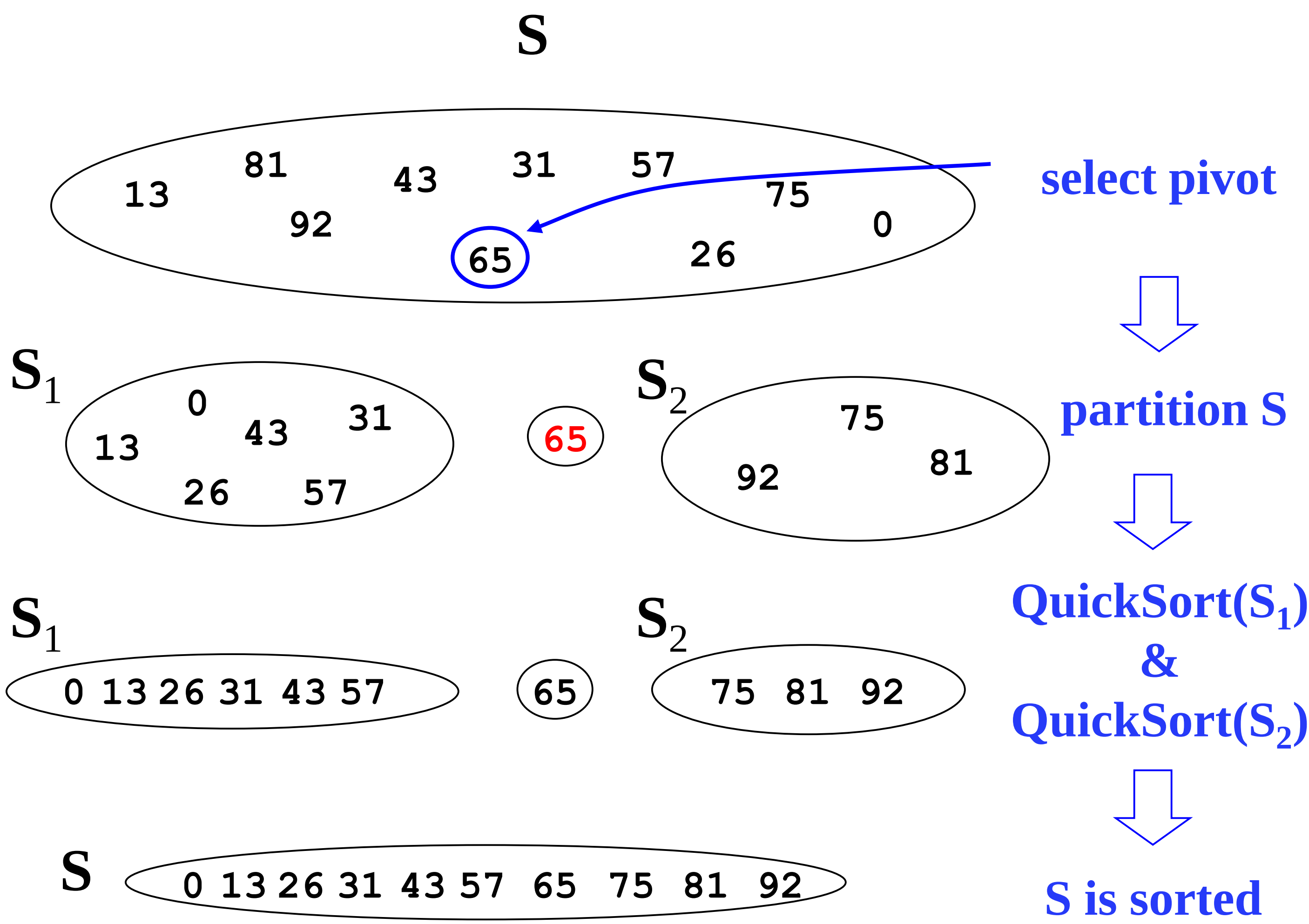
Quicksort

- Quicksort uses a divide and conquer strategy, but does not require the O(N) extra space that MergeSort does
  - Partition array into left and right sub-arrays
    - Choose an element of the array, called **pivot**
    - the elements in left sub-array are all less than pivot
    - elements in right sub-array are all greater than pivot
  - Recursively sort left and right sub-arrays
  - Concatenate left and right sub-arrays in O(1) time

“Four easy steps”

- To sort an array **S**
  1. If the number of elements in **S** is 0 or 1, then return. The array is sorted.
  2. Pick an element ***v*** in **S**. This is the ***pivot*** value.
  3. Partition **S**-{***v***} into two disjoint subsets, $\mathbf{S}_1$ = {all values $x \le v$}, and $\mathbf{S}_2$ = {all values $x \ge v$}.
  4. Return QuickSort($\mathbf{S}_1$), v, QuickSort($\mathbf{S}_2$)

The steps of QuickSort



Partitioning

- Picking the pivot
  - want a value that will cause $|S_1|$ and $|S_2|$ to be non-zero, and close to equal in size if possible
- Partitioning Strategy
  - How do the elements get to the correct partition

Select pivot

- The find pivot function
  - Use the value in the first position?
    - poor partitioning if the input is sorted or reverse sorted.
  - Pick a value at random position?
    - using a random number generator is expensive
  - Select the value in the middle position in the array
    - $\lfloor (left + right)/2 \rfloor$
  - The midian of the array
    - Median-of-Three
      - Select the median of the left, right, and center elements.

Partitioning Strategy

- Several partitioning strategies used in practice
  - Digging and filling
  - 3-way partitioning
  - ……
  - 2-way partitioning (we discuss)

2-way partitioning

- Set pointers i and j to start and end of array
- Increment i until you hit element A[i] > pivot
- Decrement j until you hit element A[j] < pivot
- Swap A[i] and A[j]
- Repeat until i and j cross
- Swap pivot (at A[N-1]) with A[i]

Example

0	1	2	3	4	5	6	7	8	9
8	1	4	9	6	3	5	2	7	0

Choose the pivot using Median-of-Three

8	1	4	9	0	3	5	2	7	6
l								r	

Swap the pivot with last element A[N-1].

Example

8	1	4	9	0	3	5	2	7	6
l								r	

Increment i until you hit element A[i] > pivot

8	1	4	9	0	3	5	2	7	6
l								r	

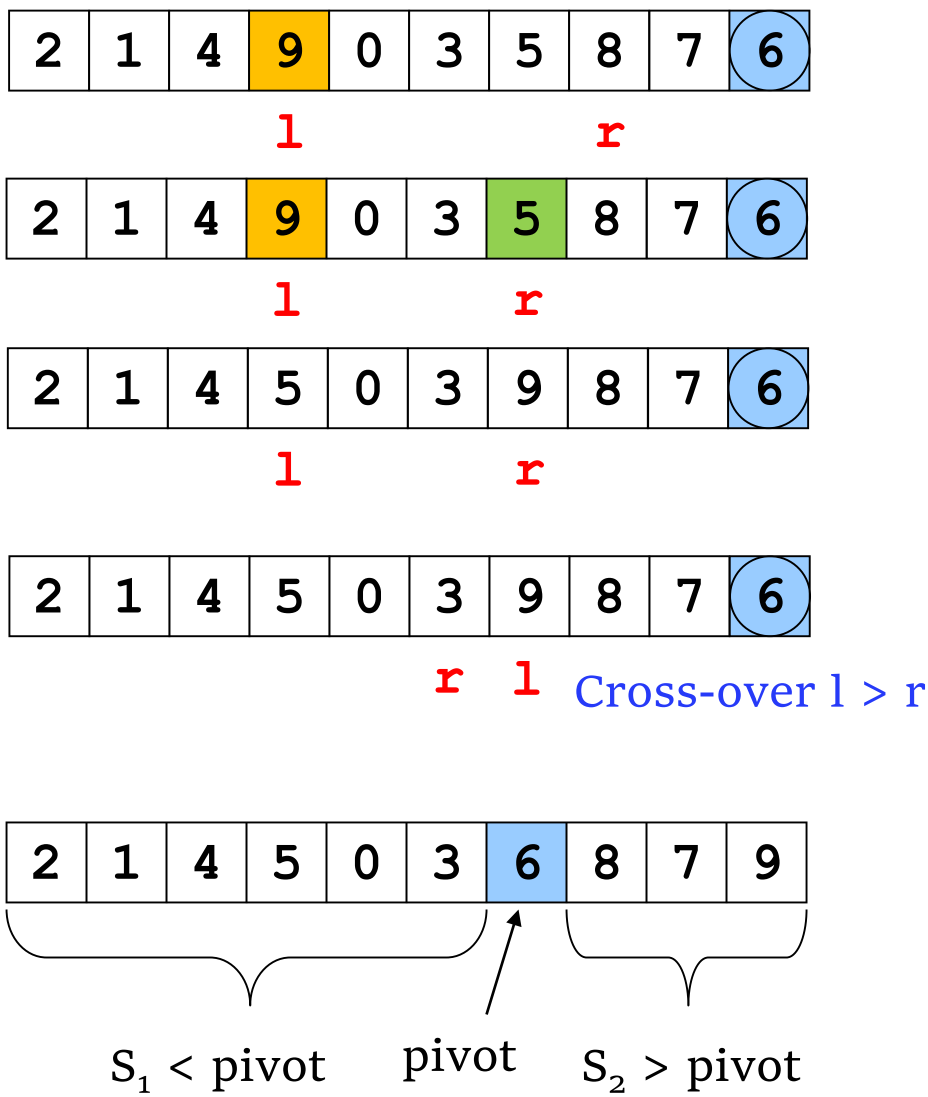
Decrement j until you hit element A[j] < pivot

8	1	4	9	0	3	5	2	7	6
l							r		

Swap A[i] and A[j]

2	1	4	9	0	3	5	8	7	6
l							r		

Example


2 1 4 9 0 3 5 8 7 6
l r
2 1 4 9 0 3 5 8 7 6
l r
2 1 4 5 0 3 9 8 7 6
l r
2 1 4 5 0 3 9 8 7 6
r l
Cross-over l > r
2 1 4 5 0 3 6 8 7 9
S1 < pivot
pivot
S2 > pivot

Quicksort

```
//QuickSort
template <typename E, typename Comp>
void quickSort(E A[], int i, int j) {
   if (j <= i) return; //List with 0 or 1 element

   int pivotindex = findpivot(A, i, j); //get pivot
   swap(A, pivotindex, j);  //Put pivot at end

   //k will be the 1st position on right side
   int k = partition<E,Comp>(A, i-1, j, A[j]);
   swap(A, k, j);   // Put pivot in place

   quickSort<E,Comp>(A, i, k-1); //Recursively
   quickSort<E,Comp>(A, k+1, j);
}
```

Quicksort

```
template <typename E, typename Comp>
int partition(E A[], int l, int r, E& pivot) {
  do {
    //Move the bounds inward until they meet
    //Move l right and r left
    while (Comp::prior(A[++l], pivot));
    while ((l<r) && Comp::prior(A[--r],pivot));
    // Swap out-of-place values
    swap(A, l, r);
  } while (l < r); // Stop when they cross
  swap(A, l, r);   // Reverse last swap
  return l; //Return 1st position in right part
}
```

Optimizations

- Quicksort has more overhead for small arrays.
  - A good cutoff range is 10.(Empirical threshold)
  - Using insertion sort for small arrays.

- Reduce recursive calls

Analysis of Quicksort

- Assuming a random pivot and no cutoff for small arrays.
  - $T(0) = T(1) = O(1)$
    - constant time if 0 or 1 element
  - For $N > 1$, 2 recursive calls plus linear time for partitioning
  - $T(N) = T(i) + T(N - i - 1) + cN$

- Best Case
  - Algorithm always chooses best pivot and splits sub-arrays in half at each recursion
  - $T(N) = 2T(N/2) + O(N)$
    - Same recurrence relation as Mergesort
  - $T(N) =$ <u>$O(N \log N)$</u>

Analysis of Quicksort

- Worst case
  - Algorithm always chooses the worst pivot – one sub-array is empty at each recursion
  - $T(N) = T(N-1) + bN$
  - $\quad = T(N-2) + b(N-1) + bN$
  - $\quad = T(1) + b(1 + \ldots (N-1) + N)$
  - $\quad = O(1) + O(N^2)$
  - $T(N) = O(N^2)$

- Fortunately, average case performance is O(N log N) (see text for proof)

Properties of Quicksort

- Not stable because of long distance swapping.
- Pure quicksort not good for small arrays.
- “In-place”, but uses auxiliary storage because of recursive call (O(logN) space).
- O(N logN) average case performance, but $O(N^2)$ worst case performance.

Folklore

- “Quicksort is the best in-memory sorting algorithm.”
- Truth
  - Quicksort uses very few comparisons on average.
  - Quicksort does have good performance in the memory hierarchy.
    - Small footprint
    - Good locality

Homework 5-2

- To Implement all sorting algorithms discussed. (The mission is not homework.)
- Textbook exercises 7.11, 7.12, 7.15, 7.17, 7.19, 7.20, 7.28(not required)
- Deadline: to be confirmed.