# Ch 12: Sorting and Selection

## 12.1 Merge-Sort

### Divide-and-Conquer

|  |  |
| --- | --- |
| **Divide-and-conquer** is a general algorithm design paradigm:   * **Divide**: divide input data S in 2 disjoint subsets S1 and S2 * **Recur**: solve subproblems associated with S1 and S2 * **Conquer**: combine solutions for S1 and S2 into a solution for S   The base case for recursion are subproblems of size 0 or 1 | **Merge-sort**: sorting algorithm based on divide-and-conquer paradigm  Like heap-sort:   * it uses a comparator * it has O(n logn) runtime   Unlike heap-sort:   * it doesn’t use auxiliary priority queue * it accesses data in a sequential manner |

### Merge-Sort

|  |  |
| --- | --- |
| Merge-Sort on an input sequence S with n elements consists of 3 steps:   * **Divide**: partition S into 2 sequences S1 and S2 of about n/2 elements each * **Recur**: recursively sort S1 and S2 * **Conquer**: merge S1 and S2 into unique sorted sequence | **Algorithm** mergeSort(S, C)  **Input:** sequence S with n element, comparator C  **Output:** sequence S sorted according to C  **if** S.size() > 1  **then**  (S1, S2) <- partition(S, n/2)  mergeSort(S1, C)  mergeSort(S2, C)  S <- merge(S1, S2) |

Merging Two Sorted Sequences

|  |  |
| --- | --- |
| * **Conquer step**: consists of merging two sorted sequences A and B into a sorted sequence S containing the union of elements of A and B * Merging two sorted sequences, each with n/2 elements and implemented by means of a doubly linked list, takes O(n) time. | **Algorithm** merge(A, B)  **Input:** sorted sequences A and B with  n/2 elements each  **Output :** sorted sequence of A ∪ B  S ← empty sequence  **while** ¬A.isEmpty() ∧ ¬B.isEmpty()  **if** A.first().element() < B.first().element() S.addLast(A.remove(A.first()))  **else**  S.addLast(B.remove(B.first()))  **while** ¬A.isEmpty()  S.addLast(A.remove(A.first()))  **while** ¬B.isEmpty()  S.addLast(B.remove(B.first()))  **return** S |

### Merge-Sort Tree

|  |  |
| --- | --- |
| An execution of merge-sort is depicted by a binary tree   * each node represents a recursive call of merge-sort and stores * unsorted sequence before the execution and its partition * sorted sequence at the end of the execution * the root is the initial call * the leaves are calls on subsequences of size 0 or 1 | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 4.08.34 PM.png |

Example

|  |  |
| --- | --- |
| Partition | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 4.11.22 PM.png |
| Recursive call, partition  (done twice) | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 4.13.15 PM.png |
| Recursive call, base case  (done twice) | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 4.14.33 PM.png |
| Merge | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 4.15.02 PM.png |

### Analysis of Merge-Sort

The height h of the merge-sort tree is O(log n)

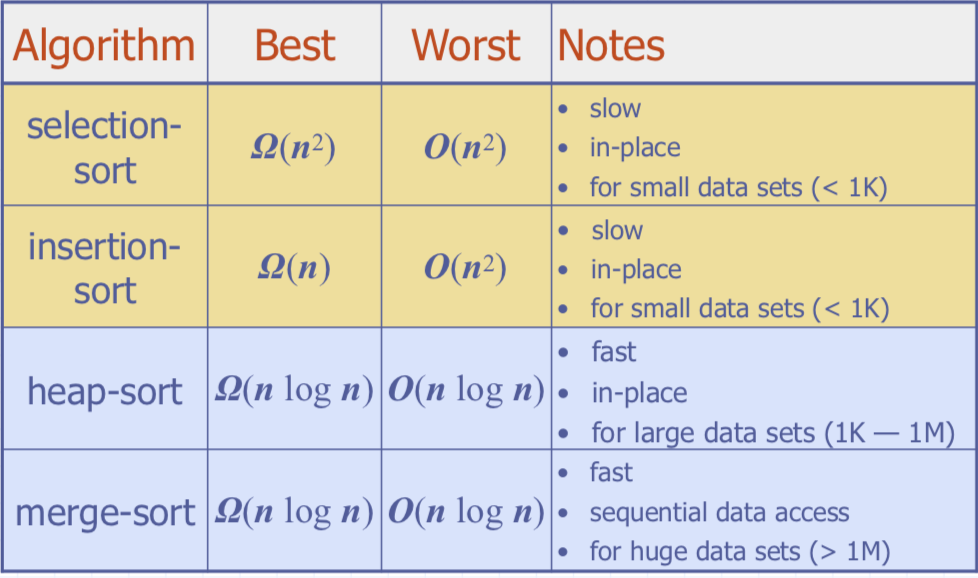
* at each recursive call we divide in half the sequence,

The overall amount or work done at the nodes of depth i is O(n)

* we partition and merge 2i sequences of size n/2i
* we make 2i+1 recursive calls

Thus, the total ***running time of merge-sort*** is **O(n log n)**

### Summary of Sorting Algorithms



### Recurrence Equation Analysis

The conquer step of merge-sort consists of merging two sorted sequences, each with n/2 elements and implemented by means of a doubly linked list, takes at most b·n steps, for some constant b.

Likewise, the base case (n < 2) will take at most b steps.

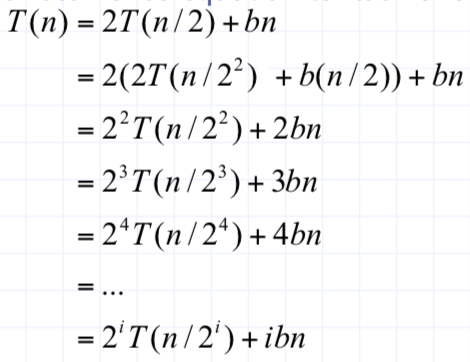
|  |  |
| --- | --- |
| Therefore, if we let T(n) denote the running time of merge-sort | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 4.19.04 PM.png |

We can therefore analyze the running time of merge-sort by finding a ***closed form solution*** to the above equation.

* That is, a solution that has T(n) only on the left-hand side.

### Iterative Substitution

In the iterative substitution, or “plug-and-chug,” technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:



Note that base case, T(n)=b, occurs when 2i=n. That is, i = log n

Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 4.22.32 PM.png

Thus, T(n) is O(n log n).

## 12.2 Quick-Sort

***Quick-sort***: randomized sorting algorithm based on the divide-and-conquer paradigm

|  |  |
| --- | --- |
| **Divide**: pick a random element x (called pivot) and partition S into   * L elements less than x * E elements equal x * G elements greater than x   **Recur**: sort L and G  **Conquer**: join L, E and G | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 4.26.03 PM.pngMacintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 4.27.17 PM.png |

### Partition

|  |  |
| --- | --- |
| We partition an input sequence as follows:   * We remove, in turn, each element y from S and * We insert y into L, E or G, depending on the result of the comparison with the pivot x   Each insertion and removal is at the beginning or at the end of a sequence, and hence takes ***O(1) time***  Thus, the partition step of quick-sort takes ***O(n)******time*** | **Algorithm** **partition**(S, p)  **Input** sequence S, position p of pivot **Output** subsequences L, E, G of the  elements of S less than, equal to, or greater than the pivot, resp.  L, E, G ← empty sequences  x ← S.remove(p)  **while** ¬S.isEmpty()  y ← S.remove(S.first())  **if** y < x  L.addLast(y)  **else** **if** y = x  E.addLast(y)  **else** { y > x }  G.addLast(y)  **return** L, E, G |

### General Quick Sort

|  |  |
| --- | --- |
| * We select the last element of the sequence as the pivot value * We call partition with S and the pivot p. It returns L,E,G * Then, L and G are recursively sorted and afterwards joined (copied) into S * The procedure JoinPartitions is not shown here (see the textbook for the code). It is similar to the algorithm *merge* in *mergeSort* and also takes ***O(n) time*** | **Algorithm** quickSort(S)  **Input** sequence S  **Output** sorted sequence S (modified)  **if** S.size() ≤ 1 **then**  **return**  p ← S.last().element()  (L,E,G) ← partition(S,p)  QuickSort(L) // L is modified QuickSort(G) // G is modified JoinPartitions(S,L,E,G)  // S is modified  **return** |

### Quick-Sort Tree

|  |
| --- |
| An execution of quick-sort is depicted by a binary tree   * Each node represents a recursive call of quick-sort and stores * Unsorted sequence before the execution and its pivot * Sorted sequence at the end of the execution * The root is the initial call * The leaves are calls on subsequences of size 0 or 1 |
| Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 4.33.39 PM.png |

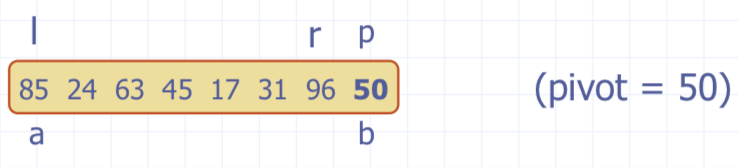
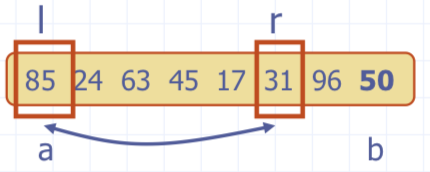
Example

|  |  |
| --- | --- |
| ***Pivot selection (here at random index)*** |  |
| ***Partition, Recursive call, Pivot selection*** |  |
| ***Partition, Recursive call***, ***Base case*** |  |
| ***Recursive call***, … , ***Base case, Join*** |  |
| ***Recursive call, Pivot selection*** | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 4.39.33 PM.png |
| ***Partition, … , Recursive call***, ***Base case*** | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 4.39.42 PM.png |
| ***Join, Join*** | Macintosh HD:Users:noemilemonnier:Desktop:Screen Shot 2017-12-05 at 4.39.52 PM.png |

### In-Place Quick-Sort

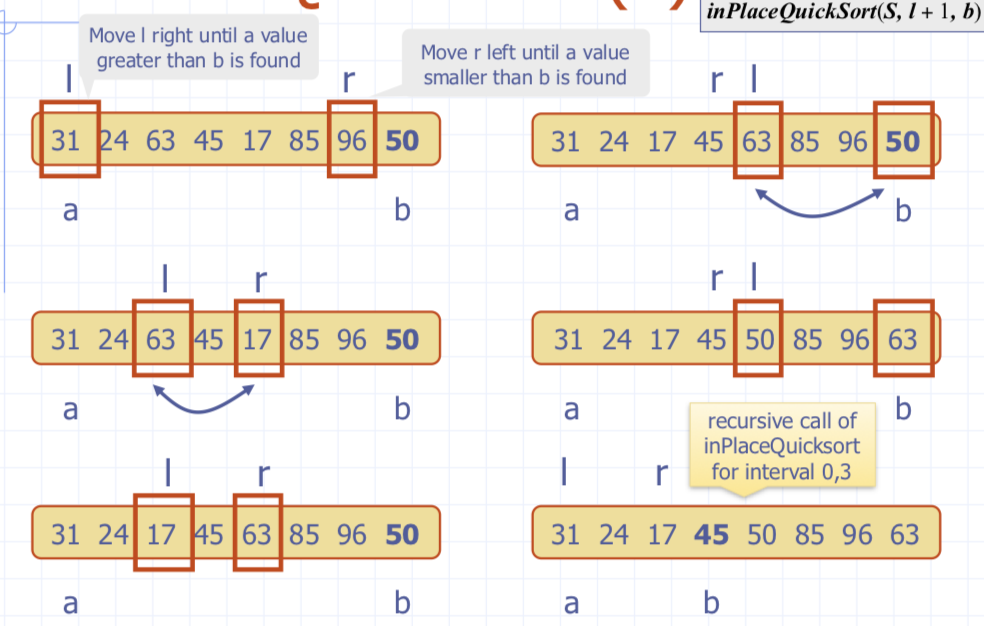
|  |  |
| --- | --- |
| Quick-sort can be implemented to run in-place. This algorithm does not allow duplicate values.  A subsequence is specified by a left most rank l and a right most rank r The divide step is performed by scanning the sequence simultaneously from l forward and r backward   * pairs of elements are swapped if they are in reverse order * when these indices meet, elements to the left of l are less than and the elements to the right of r are greater than the pivot element   The recursive calls sort the two subsequences | **Algorithm** inPlaceQuickSort(S, a, b)  **Input** sequence S (has distinct elements), ranks a and b  **Output**: sequence S with elements of rank between a and b reaaranged in increasing order  **if** a ≥ b **return**  *//pivot is last element*  p ← S.elemAtRank(b)  l ← a  r ← b - 1  **while** l ≤ r **do**  **while** l ≤ r **and** S.elemAtRank(l) ≤ p **do**  l←l+1  **while** r ≥ l **and** S.elemAtRank(r) ≥ p **do**  r ← r - 1  **if** l < r **then**  S.swapElements(S.atRank(l),S.atRank(r))  S.swapElements(S.atRank(l),p)  inPlaceQuickSort(S, a, l − 1)  inPlaceQuickSort(S, l + 1, b) |

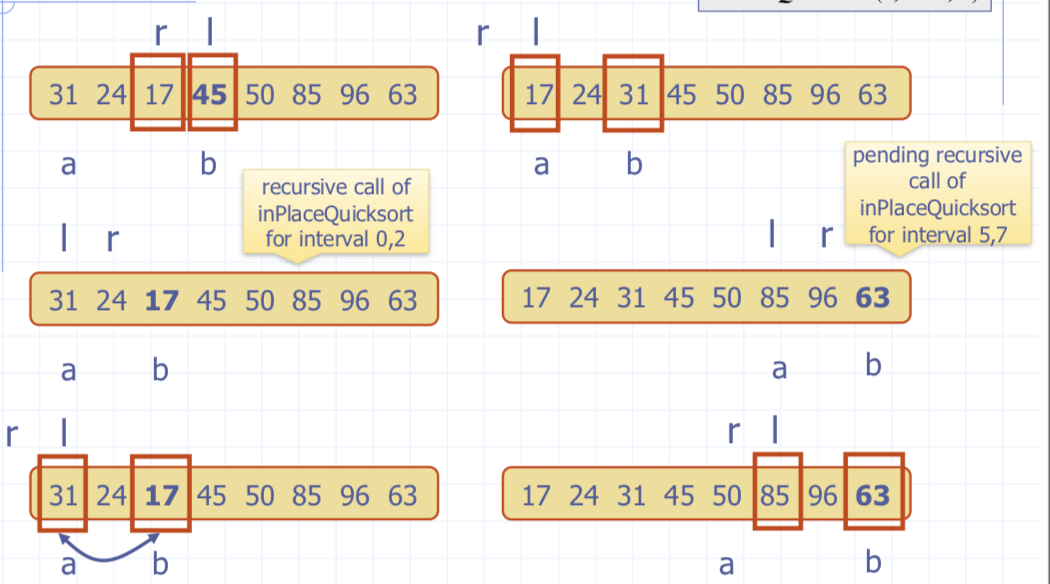
Perform the partition using two indices to split S into L and E⋃G

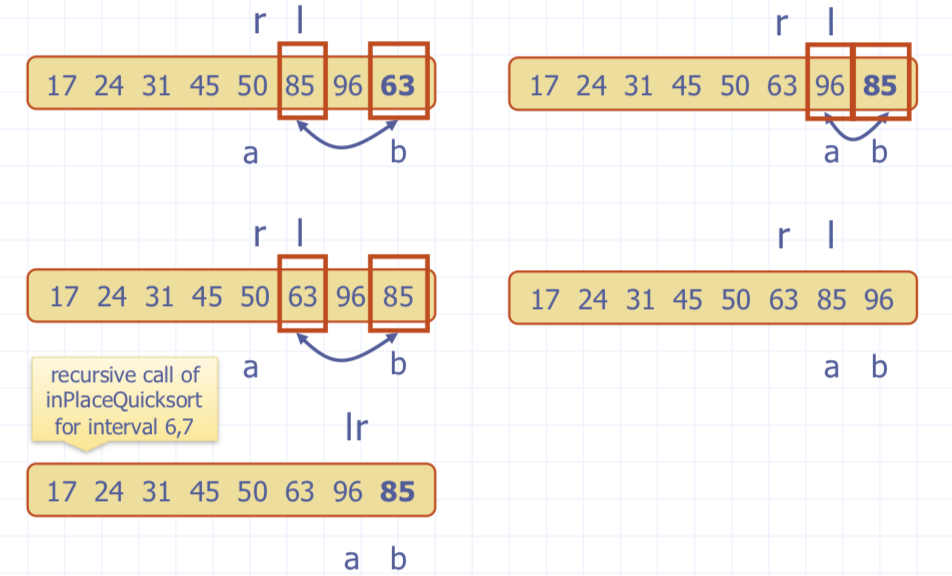


Repeat until l and r cross:

* Scan l to the right until finding an element > p.
* Scan r to the left until finding an element < p.
* Swap elements at indices l and r







### Worst-case Running Time

The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element

One of L and G has size n − 1 and the other has size 0 The running time is proportional to the sum: n + (n − 1) + ... + 2 + 1

Thus, the worst-case running time of quick-sort is O(n2)

## Summary of Sorting Algorithms

