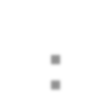
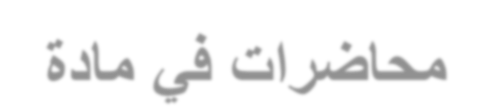
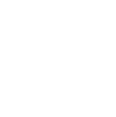
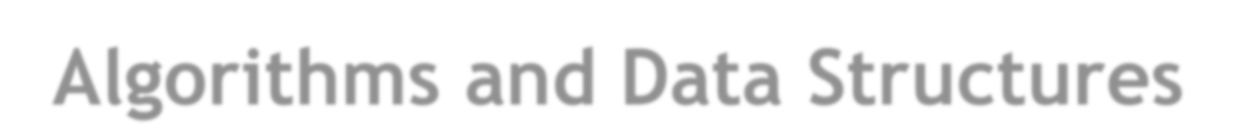
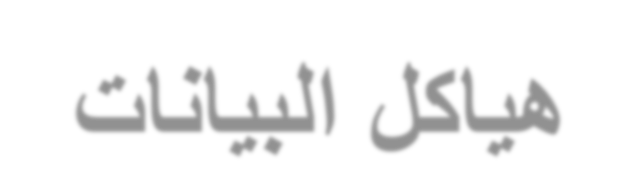
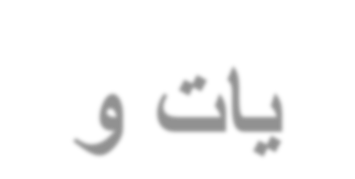
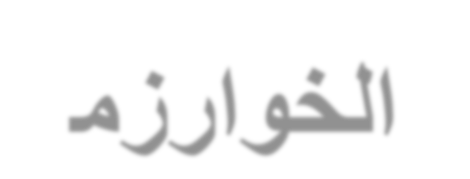
258



**Lecture # 11**

Sorting

Outline

• **Complicated sorting algorithms:**

 Merge Sort.

 Quick Sort.

(4) Merge Sort

• ***Merge Sort*** orders a list of values by recursively dividing the list in half until each sub-list has one element, then recombining.

• **Merge sort algorithm:**

 Divide the list into two roughly equal parts.

 Recursively divide each part in half, continuing until a part contains only one element.

 Merge the two parts into one sorted list.

 Continue to merge parts as the recursion unfolds.

• This is a "**divide-and-conquer**" strategy.

**Merge Sort – Divide-and-Conquer Strategy**

261

Divide-and-Conquer strategy

An algorithm design technique:

• Divide a problem of size ***n*** into sub-problems.

• Solve all sub-problems by applying the same approach.

• Merge/Combine the sub-solutions.

 This can result in VERY substantial improvements.

**Merge Sort**

Merge Sort (cont…)

262

• **Merge sort idea:**

 Divide the array into two halves.

 Recursively sort the two halves (using ***mergeSort***).

 Use ***merge*** to combine the two arrays.

mergeSort(0, mid) mergeSort(mid+1, n-1)

sort sort merge(0, mid, mid+1, n-1)

Merge sort example 1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 13 | 6 | 21 | 18 | 9 | 4 | 8 | 20 |

0 7

|  |  |  |  |
| --- | --- | --- | --- |
| 9 | 4 | 8 | 20 |

|  |  |  |  |
| --- | --- | --- | --- |
| 6 | 13 | 18 | 21 |

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 8 | 9 | 20 |

|  |  |  |  |
| --- | --- | --- | --- |
| 13 | 6 | 21 | 18 |

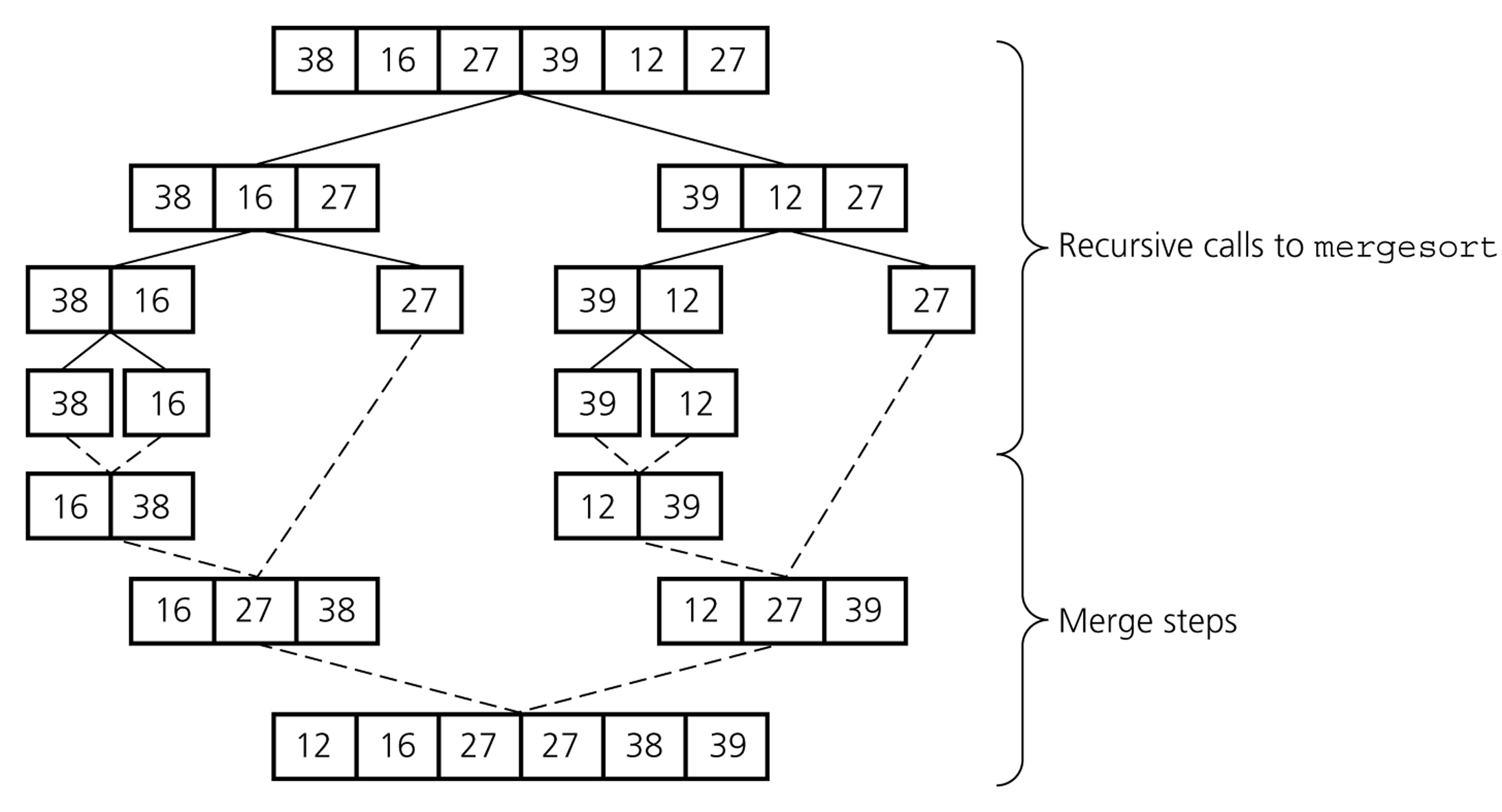
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | 0 |  |  |  |  |  | 3 |  |  |  | 4 |  |  |  |  |  | 7 |  | |
| 13 |  |  | 6 |  |  | 21 |  | 18 |  |  |  | 9 |  | 4 |  |  | 8 |  |  | 20 |
| 0 |  |  | 1 |  |  | 2 |  | 3 |  |  |  | 4 |  | 5 |  |  | 6 |  |  | 7 |
| 13 |  |  | 6 |  |  | 21 |  | 18 |  |  | 9 |  |  | 4 |  |  | 8 |  |  | 20 |
| 0 |  |  | 1 |  |  | 2 |  | 3 |  |  | 4 |  |  | 5 |  |  | 6 |  |  | 7 |
| 6 |  |  | 13 |  |  | 18 |  | 21 |  |  |  | 4 |  | 9 |  |  | 8 |  |  | 20 |
| 0 |  |  | 1 |  |  | 2 |  | 3 |  |  |  | 4 |  | 5 |  |  | 6 |  |  | 7 |

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 3 | 4 | 7 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 4 | 6 | 8 | 9 | 13 | 18 | 20 | 21 |

0 7

Merge sort example 2



Merging two sorted arrays

• ***merge* operation:**

 Given two sorted arrays, ***merge*** operation produces a sorted array with all the elements of the two arrays.

A B

|  |  |  |  |
| --- | --- | --- | --- |
| 6 | 13 | 18 | 21 |

|  |  |  |  |
| --- | --- | --- | --- |
| 4 | 8 | 9 | 20 |

C

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 4 | 6 | 8 | 9 | 13 | 18 | 20 | 21 |

 Running time of ***merge***: **O(*n*)**, where ***n*** is the number of elements in the merged array.

 When merging two sorted parts of the same array, we'll need a

*temporary array* to store the merged whole.

MergeSort function code

**void MergeSort(int a[], int first, int last)**

**{**

**if (first < last)**

**{**

**int mid = (first + last)/2;**

**MergeSort(a, first, mid); MergeSort(a, mid+1, last);**

**Merge(a, first, mid, mid+1, last);**

**}**

**}**

Merge function code

**void Merge(int a[], int leftFirst, int leftLast,**

**int rightFirst, int rightLast)**

**{**

**int temp[size];**

**int i = leftFirst;**

**int saveFirst = leftFirst;**

**while((leftFirst <= leftLast) && (rightFirst <= rightLast))**

**{**

**if(a[leftFirst] < a[rightFirst])**

**{**

**temp[i] = a[leftFirst];**

**leftFirst++;**

**}**

**else {**

**temp[i] = a[rightFirst];**

**rightFirst++;**

**}**

**i++;**

**}**

Merge operation code (cont…)

**// Copy remaining items from left half**

**while(leftFirst <= leftLast)**

**{**

**temp[i] = a[leftFirst];**

**leftFirst++;**

**i++;**

**}**

**// Copy remaining items from right half while(rightFirst <= rightLast)**

**{**

**temp[i] = a[rightFirst];**

**rightFirst++;**

**i++;**

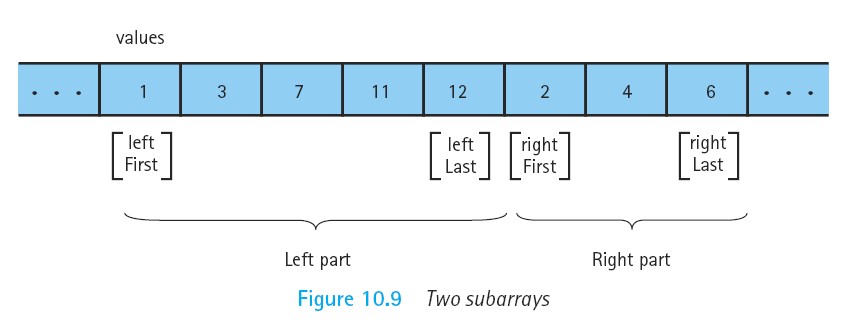
**}**

**for(i = saveFirst; i <= rightLast; i++)**

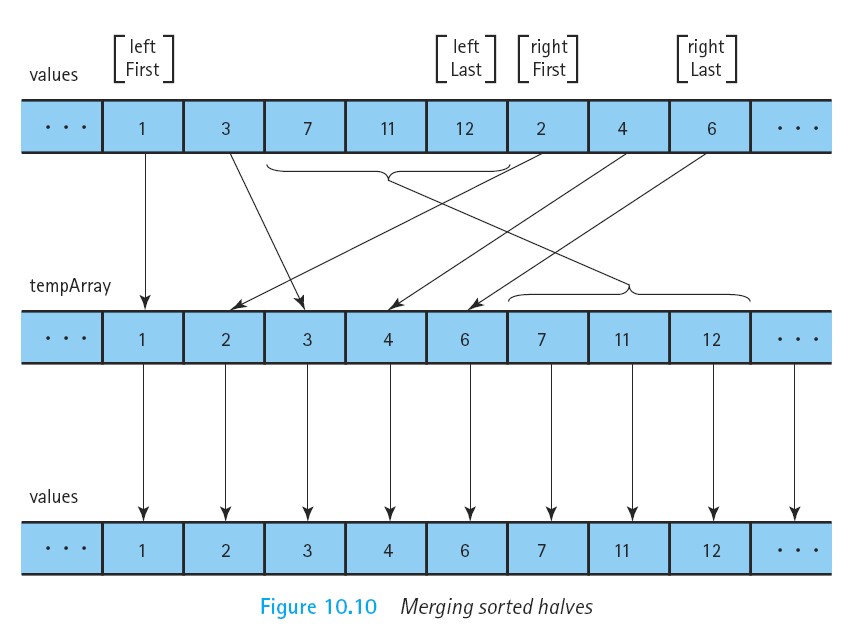
**a[i] = temp[i];**

**}**

**Dr. Gasmelseed Ibrahim, International University of Africa, Faculty of Computer Studies** **Algorithms and Data Structures: Lecture (11)**



**Dr. Gasmelseed Ibrahim, International University of Africa, Faculty of Computer Studies** **Algorithms and Data Structures: Lecture (11)**



Merge Sort Runtime

• Merging lists with a total of ***n*** elements takes:

 at most ***n-*1** compares,

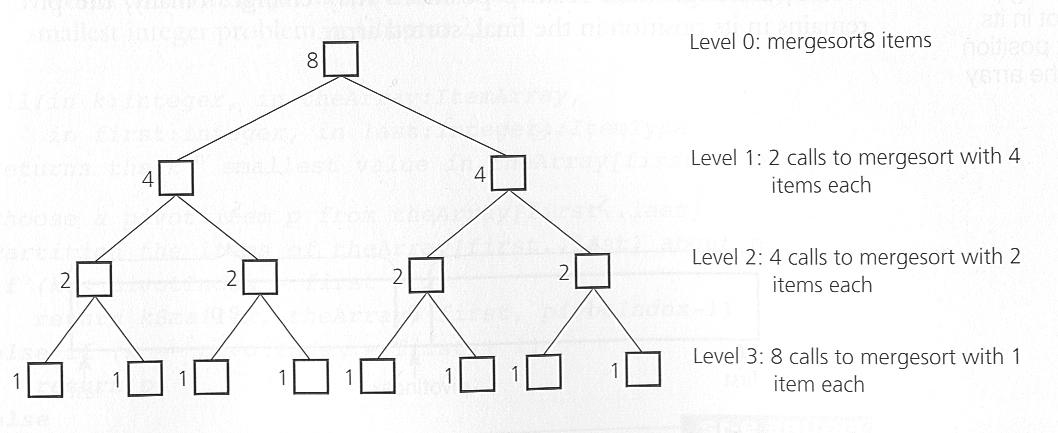
 ***n*** moves to temp array,

 and ***n*** moves back to original array.

• So, takes **3*\*n-*1** operations for each ***merge level***.

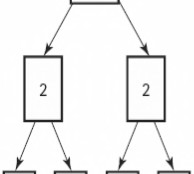
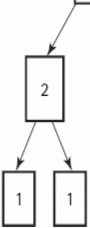
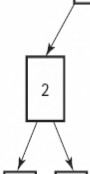
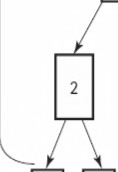
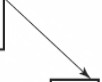
• **How many merge levels are there?**

Merge Sort Runtime (cont…)



Merge Sort Runtime (cont…)

16



8 8

log216 = 4 levels of merging

4 4 4 **4**

2 2 2

**Figure 10.11** *Analysis of the function tiergesort:* ***with N*** *= 16*

Merge Sort Runtime (cont…)

• The recursion goes **log**

**2**

***n*** layers deep,

 resulting in the same number of merge levels.

• Each layer performs **3\**n*-1**  **O(*n*)** operations.

• Have **log**

**2**

***n*** merge levels, with each level taking **O(*n*)** steps; results in:

 **O(*n* log**

**2**

***n*)** overall.

• Thus, the whole runtime of the Merge Sort Algorithm is: **O(*n* log**

**2**

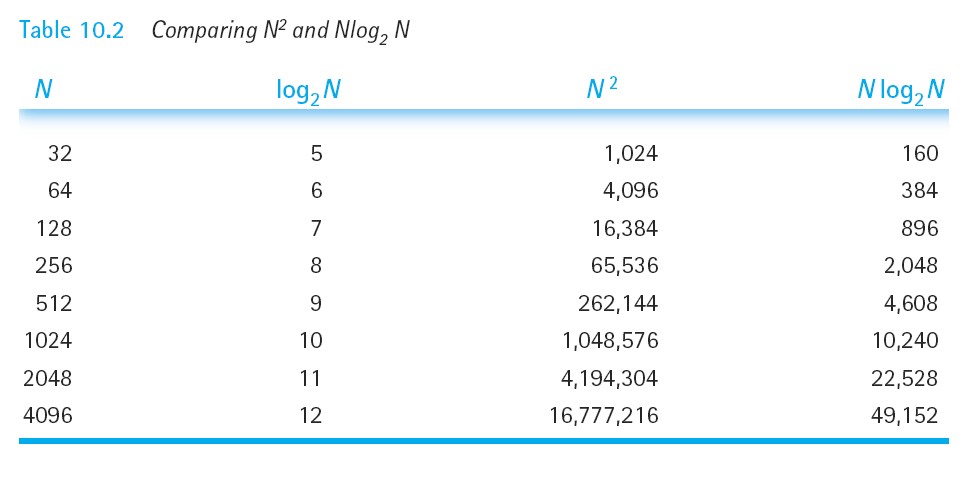
***n*)**.

• **Drawback:** Requires a second temporary array.

Merge Sort Runtime (cont…)

• Table 10.2 below illustrates that:

 for large values of ***n***, **O(*n* log**



**2**

***n*)** is a big improvement over **O(*n*2)**.

Merge Sort summary

**Average case: O(*n* log *n*)**

•

**2**

**Worst case: O(*n* log *n*)**

•

**2**

• **Storage Requirements:**

 It requires an additional array, so that it need **O(*n*)** storage.

(5) Quick Sort

• ***Quick Sort*** orders a list of values by partitioning the list around one element called a ***pivot***, then sorting each partition.

• **Quick sort algorithm:**

 Choose one element in the list to be the ***pivot*** (= partition element).

 Organize the elements so that all elements less than the ***pivot***

are to its left and all greater are to its right.

 Apply the quick sort algorithm (recursively) to both partitions.

• This also a "**divide-and-conquer**" strategy.

Quick sort (cont…)

• For correctness, it's okay to choose any ***pivot***.

• For efficiency, one of following is best case, the other worst case:

 ***pivot*** partitions the list roughly in half.

 ***pivot*** is greatest or least element in list.

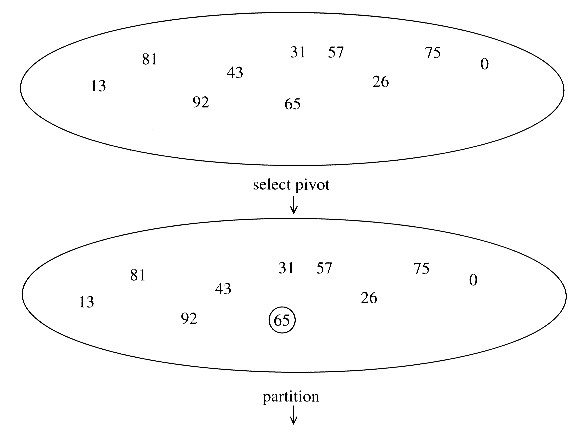
• Which case above is best?

• We will divide the work into two methods:

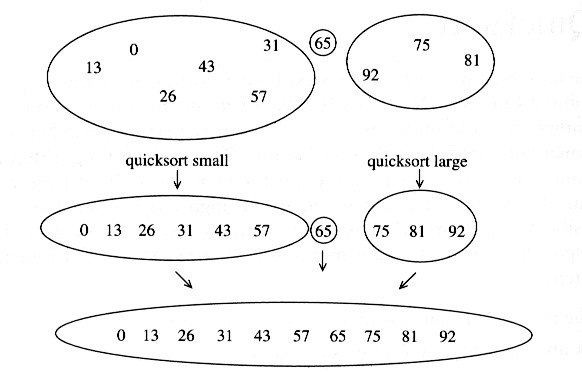
 **QuickSort** – performs the recursive algorithm.

 **Partition** – rearranges the elements into two partitions.

Example: QuickSort



Example: QuickSort



How to choose a pivot

• **First element:**

 bad if input is sorted or in reverse sorted order.

 bad if input is nearly sorted.

 variation: particular element (e.g. middle element).

• **Random element:**

 even a malicious agent cannot arrange a bad input.

• **Median of three elements:**

 choose the median of the left, right, and center elements.

Partitioning Algorithm

The basic idea:

1. Move the ***pivot*** to the leftmost/rightmost position.

2. Starting from the left, find an element **> *pivot***. Call the position ***l***.

3. Starting from the right, find an element  ***pivot***. Call the position ***r***.

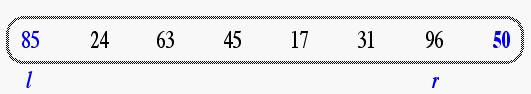
4. Swap **a[*l*]** and **a[*r*]**.

5. Repeat steps 2, 3, 4 until (***l*** > ***r***).

6. Swap **a[*pivot*]** and **a[*r*]**.

Quick sort Example 1

• Divide step: ***l*** scans the sequence from the left, and ***r*** from the right.

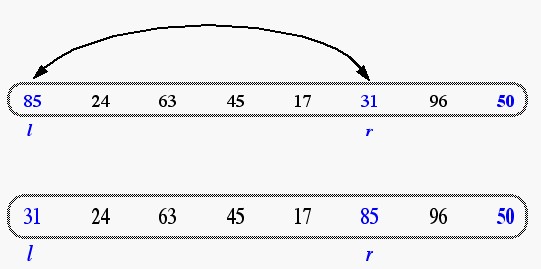


• A swap is performed when ***l*** is at an element larger than the ***pivot*** and ***r***

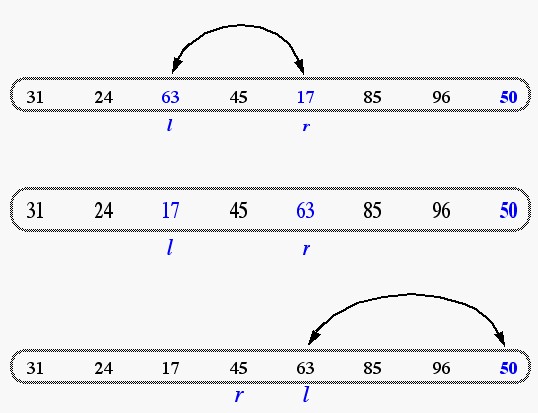
is at one smaller than the ***pivot***.

• In this example ***pivot*** = 50.

Quick sort Example 1 (cont…)

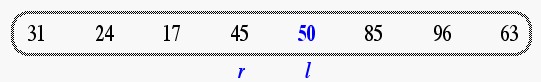


Quick sort Example 1 (cont…)



Quick sort Example 1 (cont…)

• A final swap with the ***pivot*** completes the divide step.



Quick sort Example 2

0 1 2 3 4 5 6 7 8

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 44 | 75 | 23 | 43 | 55 | 12 | 64 | 77 | 33 |

Left Right

Quick sort Example 2 (cont…)

• Define the value in position Left to be the ***pivot***.

• Define ***l*** to start from Left and ***r*** to start from Right.

***pivot* = 44**

0 1 2 3 4 5 6 7 8

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 44 | 75 | 23 | 43 | 55 | 12 | 64 | 77 | 33 |

Left Right

***l r***

Quick sort Example 2 (cont…)

• Move ***l*** to the first value ***> pivot***.

• Move ***r*** to the first value ***<= pivot***.

***pivot* = 44**

0 1 2 3 4 5 6 7 8

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 44 | 75 | 23 | 43 | 55 | 12 | 64 | 77 | 33 |

Quick sort Example 2 (cont…)

• Exchange these values.

***pivot* = 44**

0 1 2 3 4 5 6 7 8

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 44 | 33 | 23 | 43 | 55 | 12 | 64 | 77 | 75 |

Quick sort Example 2 (cont…)

• Move ***l*** to the first value ***> pivot***.

• Move ***r*** to the first value ***<= pivot***.

***pivot* = 44**

0 1 2 3 4 5 6 7 8

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 44 | 33 | 23 | 43 | 55 | 12 | 64 | 77 | 75 |

Quick sort Example 2 (cont…)

• Exchange these values.

***pivot* = 44**

0 1 2 3 4 5 6 7 8

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 44 | 33 | 23 | 43 | 12 | 55 | 64 | 77 | 75 |

Quick sort Example 2 (cont…)

• Move ***l*** to the first value ***> pivot***.

• Move ***r*** to the first value ***<= pivot***.

***pivot* = 44**

0 1 2 3 4 5 6 7 8

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 44 | 33 | 23 | 43 | 12 | 55 | 64 | 77 | 75 |

Quick sort Example 2 (cont…)

• ***l*** and ***r*** have passed each other (that is ***l > r***),

• So, exchange the ***pivot*** value and the value in ***r***.

***pivot* = 44**

0 1 2 3 4 5 6 7 8

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 44 | 33 | 23 | 43 | 12 | 55 | 64 | 77 | 75 |

Quick sort Example 2 (cont…)

• ***l*** and ***r*** have passed each other (that is ***l > r***),

• So, exchange the ***pivot*** value and the value in ***r*** .

***pivot* = 44**

0 1 2 3 4 5 6 7 8

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 33 | 23 | 43 | 44 | 55 | 64 | 77 | 75 |

Quick sort Example 2 (cont…)

• Now, all values before ***pivot index*** are ***<= pivot***.

• And all values after ***pivot index*** are ***> pivot***.

***pivot* = 44**

0 1 2 3 4 5 6 7 8

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 33 | 23 | 43 | 44 | 55 | 64 | 77 | 75 |

Quick sort Example 2 (cont…)

• This gives us two new sub arrays to partition.

***pivot* = 44**

0 1 2 3 4 5 6 7 8

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 12 | 33 | 23 | 43 | 44 | 55 | 64 | 77 | 75 |

Left first

Left last

***pivot index***

Right first

Right

last

QuickSort function code

**void QuickSort(int a[], int left, int right)**

**{**

**if (left < right)**

**{**

**int S = Partition(a, left, right); QuickSort(a, left, S-1); QuickSort(a, S+1, right);**

**}**

**}**

Partition function code

**int Partition(int a[], int left, int right)**

**{**

**int l, r;**

**l = left;**

**r = right;**

**if(a[left] > a[right]) swap(a[left],a[right]);**

|  |  |  |  |
| --- | --- | --- | --- |
| **//in this** | **program the pivot value is the** | **first** | **element** |
| **int pivot** | **= a[left];** |  |  |
| **do {** |  |  |  |
| **do {** | **l++; } while (a[l] < pivot);** |  |  |
| **do {** | **r--; } while (a[r] > pivot);** |  |  |

**if(l < r) swap(a[l],a[r]);**

**} while(l < r);**

**swap(pivot,a[r]);**

**return r;**

**}**

Quick Sort Runtime

• **Best case:** pivot is the median.

 The runtime is **O(*n* log**

**2**

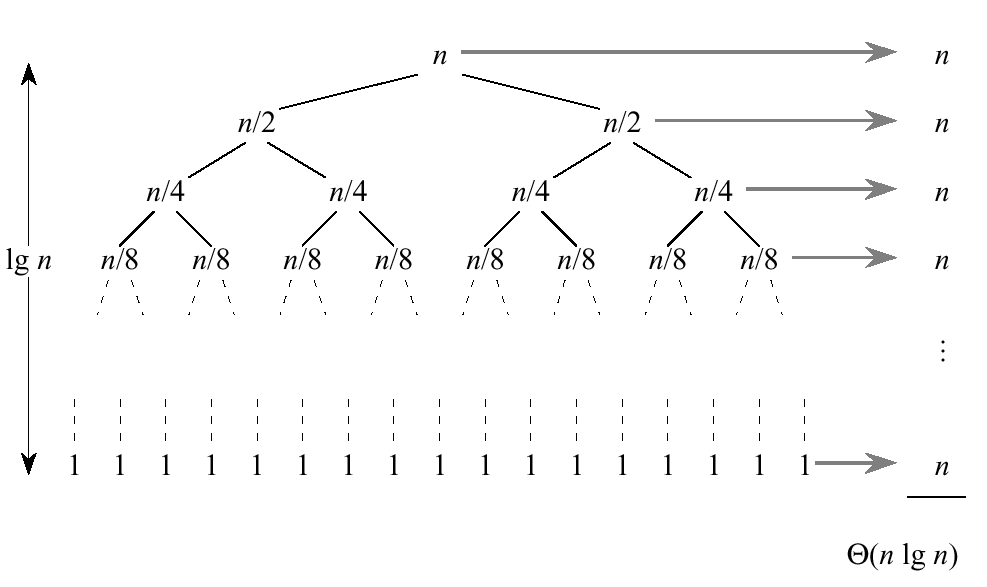
***n*)**.

• **Worst case:** pivot is the smallest (or largest) element all the time.

 The runtime is **O(*n2*)**.

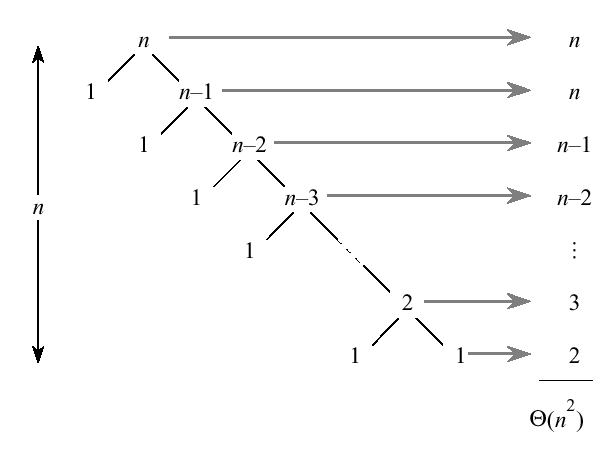
Quick Sort Runtime (cont…)

• **Best Case:** Partition splits the array evenly.



Quick Sort Runtime (cont…)

• **Worst Case:** One side of the parition has only one element.



Quick Sort summary

**Average case: O(*n* log *n*)**

•

**2**

• **Worst case: O(*n2*)**

• **Storage Requirements:**

 Sorts almost in place, i.e., does not require an additional array.

 like insertion sort, unlike merge sort.

Comparing Complicated Sorts

• We've seen “complicated" sorting algorithms, such as:

 Merge sort, and Quick sort.

|  |  |  |
| --- | --- | --- |
|  | | **Comparisons** |
| **Merge** |  | O(n log n)  2 |
| **Quick** | Worst | O(n2) |
| Average | O(n log n)  2 |

 In fact, Quick sort is the currently fastest known sorting algorithm and

is often the best practical choice for sorting, as its average expected

running time is **O(n log**

**2**

**(n)**.