16 - Sorting Algorithms

Joseph Afework
CS 241

Dept. of Computer Science
California Polytechnic State University, Pomona, CA

Agenda

- Intro
- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Heap Sort
- Performance

Reading Assignment

- Read Chapter 28
  - Chapter 28 (Read about: **Sorting Algorithms**)

Sorting Algorithms

A **sorting algorithm** is an algorithm that arranges elements of a list in a specific order.

- Common orders are numerical order and lexicographical order.

Classification

Sorting Algorithms are generally classified by the following properties:

- Computational complexity
- Memory Usage
- Use of Recursion
- Stability
- Adaptability

Stability

Stable sorting algorithms maintain the relative order of records with equal keys.

Ex. (4, 2) **(3, 7)** **(3, 1)** (5, 6)

In this case, two different results are possible, one which maintains the relative order of records with equal keys, and one which does not:

(3, 7) **(3, 1)** (4, 2) (5, 6)
(3, 1) **(3, 7)** (4, 2) (5, 6)

Bubble Sort

Bubble sort: comparison

1. Repeatedly step through the list to be sorted, comparing each pair of adjacent items…
2. Swap if they are in the wrong order…..
3. Repeat until no swaps are needed... (list is sorted)

Bubble Sort Contd.

```
procedure bubbleSort( A : list of sortable items ) defined as:
    do
        swapped := false
        for each i in 0 to length(A) - 1 inclusive do:
            if A[i] > A[i+1] then
                swap( A[i], A[i+1] )
                swapped := true
            end if
        end for
    while swapped
end procedure
```

Runtime: Worst Case O(n^2)
Runtime: Best Case O(n)

Bubble Sort Example

Bubble Sort Example

5	1	4	2	8

1	5	4	2	8

1	4	5	2	8

1	4	2	5	8

1	4	2	5	8

1	4	2	5	8

1	2	4	5	8

1	2	4	5	8

1	2	4	5	8

Selection Sort

Selection Sort: selection

1. Find the minimum value in the list
2. Swap it with the value in the first position
3. Repeat the steps above for the remainder of the list (starting at the second position and advancing each time)

Selection Sort Contd.

```
void selectionSort(int[] a) {
        for (int i = 0; i < a.length - 1; i++) {
                int min = i;
                for (int j = i + 1; j < a.length; j++) {
                        if (a[j] < a[min]) {
                                min = j;
                        }
                }
                if (i != min) {
                        int swap = a[i];
                        a[i] = a[min];
                        a[min] = swap;
                }
        }
}
```

Runtime: Worst Case O(n^2)
Runtime: Best Case O(n^2)

Selection Sort Example

Selection Sort

i				
63	25	12	22	11
i				min
63	25	12	22	11
	i	min		
11	25	12	22	63
		i	min	
11	12	25	22	63
			i, min	
11	12	22	25	63
				i, min
11	12	22	25	63

```
void selectionSort(int[] a) {
  for (int i = 0; i < a.length - 1; i++) {
    int min = i;
    for (int j = i + 1; j < a.length; j++) {
      if (a[j] < a[min]) {
        min = j;
      }
    }
    if (i != min) {
      int swap = a[i];
      a[i] = a[min];
      a[min] = swap;
    }
  }
}
```

Insertion Sort

Insertion Sort: insertion

1. Every iteration of insertion sort removes an element from the input data, inserting it into the correct position in the already-sorted list, until no input elements remain.
2. The choice of which element to remove from the input is arbitrary, and can be made using almost any choice algorithm.
3. Sorting is typically done in-place.

Insertion Sort Contd.

```
insertionSort(array A){
	for i := 1 to length[A] - 1{
		value := A[i];
		j := i - 1;
		while j >= 0 and A[j] > value
		{
			A[j + 1] := A[j];
			j := j - 1;
		}
		A[j + 1] := value;
	}
}
```

Runtime: Worst Case O(n^2)
Runtime: Best Case O(n)

Insertion Sort Contd.

Insertion Sort

63	25	12	22	11

25	63	12	22	11

12	25	63	22	11

12	22	25	63	11

11	12	22	25	63

```
insertionSort(array A)
 begin
  for i := 1 to length[A] - 1 do
  begin
    value := A[ i ];
    j := i - 1;
    while j >= 0 and A[ j ] > value do
    begin
      A[ j + 1] := A[ j ];
      j := j - 1;
    end;
    A[ j + 1] := value;
  end;
 end;
```

Merge Sort

Merge Sort: partitions

1. If the array is of length 0 or 1, then it is already sorted.
2. **Otherwise:**
   a. Divide the array into two arrays of about half the size.
   b. Sort each array recursively by re-applying merge sort **(splitting)**.
   c. Merge the two arrays back into one sorted list **(merge)**.

Merge Sort Contd.

```
Algorithm MergeSort(A, 0, n-1)
{
        MergeSort(A, 0, n/2)
        MergeSort(A, n/2 + 1, n-1)
        MergeTogether(2 arrays above)
}
```

Runtime: Worst Case O(n log n).. how?
Runtime: Best Case O(n log n)... generally

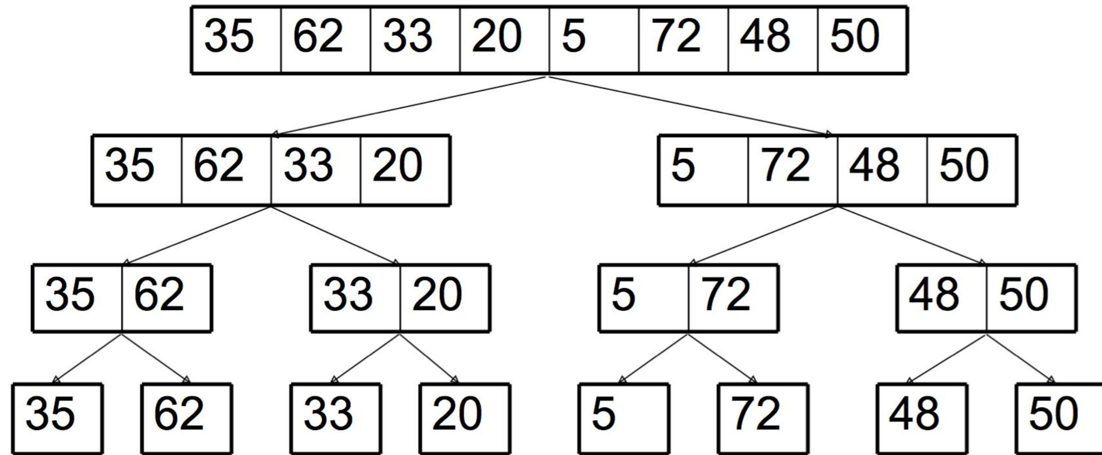
Recurrence Relation

$$
\begin{aligned}
T(n) &= 2\ T(n/2) + n \\
&= 2\ [2\ T(n/4) + n/2] + n \\
&= 4\ T(n/4) + 2n \\
&= 4\ [2\ T(n/8) + n/4] + 2n \\
&= 8\ T(n/8) + 3n \\
&= \textit{(fill in this line)} \\
&= \ldots\ldots \\
&= 2^k\ T(n/2^k) + k\ n
\end{aligned}
$$

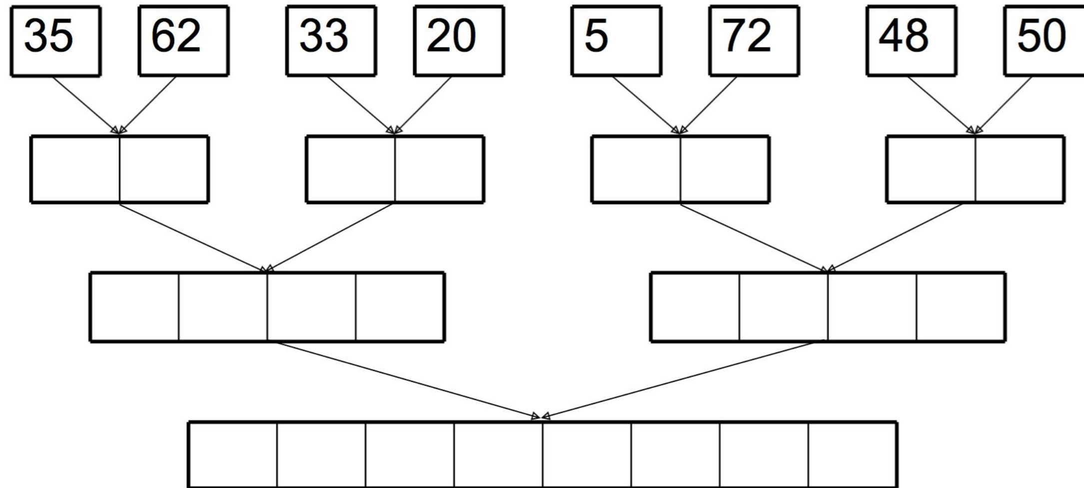
$n/2^k = 1$ *OR* $n = 2^k$ *OR* $\log_2 n = k$

$$
\begin{aligned}
&= 2^k\ T(n/2^k) + k\ n \\
&= 2^{\log_2 n}\ T(1) + (\log_2 n)\ n \\
&= n + n \log_2 n \quad \textit{[remember that } T(1) = 1\textit{]} \\
&= O(n \log n)
\end{aligned}
$$

Merge Sort Example



Merge Sort Example



Quick Sort

Quick Sort: comparison…. divide-conquer

1. Pick an element, called the ***pivot***, from the array.
2. ***(Partition the array):*** Move all elements less than pivot, in front of pivot, move all elements greater than pivot behind in the array.(equal values can go either way).
   a. After this partitioning, the pivot is in its final position (**sorted position**).
3. Recursively sort the two sub-arrays using quicksort. The base case of the recursion are arrays of size zero or one, which are always sorted.

Quick Sort Contd.

```
function quicksort(array)
    var list less, greater
    if length(array) <= 1
        return array
    select and remove a pivot from array
    for each x in array
        if x <= pivot then append x to less
        else append x to greater
    return concatenate(quicksort(less), pivot,
quicksort(greater))
```

Worst Case: $O(n^2)$..
Best Case: O(n log n)...

Picking a good pivot element (to start) is critical.

The closer the pivot is near the median of the array values, the more efficient quicksort is.

Technique: Randomly choose three values from the array and then use the middle of these three values as the pivot element.

Heap Sort

1. Build a max heap out of the data set
2. Remove the largest element from the heap (top element) and place it into the end of the sorted array
3. Restore heap property (reheapify as needed)... just like normal heap value removal
4. Goto Step 2 until no more elements in heap

Performance

Name	Average	Worst	Memory	Stable	Method
Bubble sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Exchanging
Selection sort	$O(n^2)$	$O(n^2)$	$O(1)$	No	Selection
Insertion sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Insertion
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	Merging
Quicksort	$O(n \log n)$	$O(n^2)$	$O(1)$	No	Partitioning
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(1)$	No	Selection