SIT221: Data Structures and Algorithms

Lecture 2: Sorting and Searching Algorithms

Updates

- Consultation:
  - Saturday 4:00pm – 5:00pm via Bb

Lecture 1 – Recap

- Computational complexity analysis of algorithms
- Running time T(n) - number of operations to reach a solution
- Big-O, Big-Ω, Big-Θ

Revisit - Why algorithm analysis?

- Machines do not really dominate the efficiency
  - Example
    - Sort an array of 1 million elements.
    - Two algorithms
      - A => complexity = 2 * (number of elements)^2,
      - B => complexity = 50* number of elements * log(number of elements)
    - Two machines
      - C1 (1 billion instructions per second), runs algorithm A
      - C2 (10 millions instructions per second), runs algorithm B
    - Time to sort 1 million elements on C1 = 2 * $(10^6)^2 / 10^9$) = 2000 seconds
    - Time to sort 1 million elements on C2 = 50 * 10^6 * $\log_2$ (10^6) / 10^7 = 100 seconds !!!
- How efficient is my algorithm? How to improve?
- Given two algorithms A, B; which one is more efficient?

Properties of Big-O, Big-Ω, Big-Θ

- Scaling
  - If $f(n) = O(g(n))$, then $\forall c, cf(n) = O(g(n))$

Properties of Big-O, Big-Ω, Big-Θ (cont.)

- Transitivity
  - If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$

Properties of Big-O, Big-Ω, Big-Θ (cont.)

- Rule of sums
  - If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\})$

Properties of Big-O, Big-Ω, Big-Θ (cont.)

- Rule of products
  - If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n)f_2(n) = O(g_1(n)g_2(n))$

Properties of Big-O, Big-Ω, Big-Θ (cont.)

- Limit
  - If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = l$ exits, then

$$\begin{cases} l = 0, & f(n) = \boxed{?}O(g(n)) \\ 0 < l < \infty, & f(n) = \quad (g(n)) \\ l = \infty, & f(n) = \Omega(g(n)) \end{cases}$$

Exercise 1

- Find Big-Θ of

a) $50n$

b) $8n^2 + 3n - 15$

c) $4\log_2 n$

d) $2nlog_2 n + n^2 + 2n + 10000$

e) $40(2^n + 108n^{10})$

f) $2e^n n^4 + n^8 \log_2 n$

g) $1^k + 2^k + \cdots + n^k$

Exercise 2

- Given

$f(n)$	$g(n)$
$100n + \log_2 n$	$n + (\log_2 n)^2$
$\log_2 n$	$\log_2 n^2$
$\frac{n^2}{\log_2 n}$	$n(\log_2 n)^2$
$\sqrt{n}$	$(\log_2 n)^5$
$n2^n$	3^n

- Question: $f(n) = O(g(n))$ or $g(n) = O(f(n))$ or both?

Sorting & Searching

- Sorting
  - Bubble sort
  - Insertion sort
  - Selection sort
  - Merge sort
  - Quick sort
- Searching
  - Linear Search
  - Binary Search

Sorting

- Where did you use sorting function before?
  - Playing cards
  - Excel sheets
  - Database
  - Sorted arrays, makes search easier, duplicates, median, etc.
  - Comparing different lists, or sets

Let’s take an example – Intersection operation

- Imagine you have two Vectors A, B of type int and similar length
- Can you check if both lists have the same elements?
- How your code will look like? Two nested loops?
- Let’s run a demo?
- What’s the T(n) for this solution?
- What does it mean for bigger datasets?

A	B
2	9
1	8
3	7
4	1
7	2
9	3
6	6
8	5

List Intersection – after sorting

A
1
2
3
4
6
7
8
9

B
1
2
3
5
6
7
8
9

- Now, both lists are sorted
- Is it easier to find the intersection now?
- Let's demo?
- Can you tell the running time?

Back to sorting…

- Stable sorting algorithm:
  - A sorting algorithm is stable if every two elements that have the same value have the same relative order after sorting as before.
  - E.g. We want to sort students by marks and if two students have the same mark, the two students should be ordered alphabetically. Which one would work?
    - Sort by mark, then by name
    - Sort by name, then by mark

Key Criteria for Sorting Algorithms

- Stable: Equal keys aren't reordered.
- Operates in place, e.g. requiring O(1) extra space.
- Worst-case key comparisons.
- Adaptive: Speeds up when data is nearly sorted or when there are few unique keys.

Sorting Algorithms … Who can sort this list?

- Bubble Sort ---- Bubble up the largest number to the end
- Insertion Sort
- Selection Sort
- Merge Sort
- Quick Sort
- …

A
2
1
3
4
7
9
6
8

Check this website for visual sorting:
http://visualgo.net/sorting

Comparison-based Sorting

Bubble Sort

```
for (int i = 0; i < n - 1; i++)
{
   for (int j = 0; j < n - i - 1; j++)
   {
     if (a[j] > a[j+1])
     {
          swap(a[j], a[j+1]);
     }
  }
}
```

- Is this stable sorting algorithm ?
- What's the running time?
- What's the best/worst/average case?
- Big-O?
- Let's demo sorting of 100, 1000 elements

Bubble Sort - Example

```
for (int i = 0; i < n - 1; i++)
{
   for (int j = 0; j < n - i - 1; j++)
   {
     if (a[j] > a[j+1])
     {
          swap(a[j], a[j+1]);
     }
   }
}
```

Input	**5**	**1**	**4**	**6**	**2**
i=0	5	1	4	6	2
	1	**5**	4	6	2
	1	4	**5**	6	2
	1	4	5	2	**6**
i=1	1	4	5	2	**6**
	1	4	2	**5**	**6**
i=2	1	4	2	**5**	**6**
	1	2	**4**	**5**	**6**
i=3	1	2	**4**	**5**	**6**
	1	**2**	**4**	**5**	**6**

Insertion Sort

```
for (int i = 1; i < n; i++)
{
    j = i;
   while (j > 0 && a[j-1] > a[j])
   {
         swap(a[j-1], a[j]);
         j--;
   }
}
```

- Is this stable sorting algorithm ?
- What's the running time?
- What's the best/worst/average case?
- Big-O?
- Let's demo sorting of 100, 1000 elements

Insertion Sort - Example

```
for (int i = 1; i < n; i++)
{
    j = i;
   while (j > 0 && a[j-1] > a[j])
   {
         swap(a[j-1], a[j]);
         j--;
   }
}
```

Input	**5**	**1**	**4**	**6**	**2**
i=1	**1**	5	4	6	2
i=2	1	**4**	5	6	2
i=3	1	4	5	**6**	2
i=4	1	4	5	**2**	6
	1	4	**2**	5	6
	1	**2**	4	5	6

Selection Sort

- Find the smallest element, and bring it to its correct position - first place in the array, then repeat the same proces for the second index, third index, until last index.

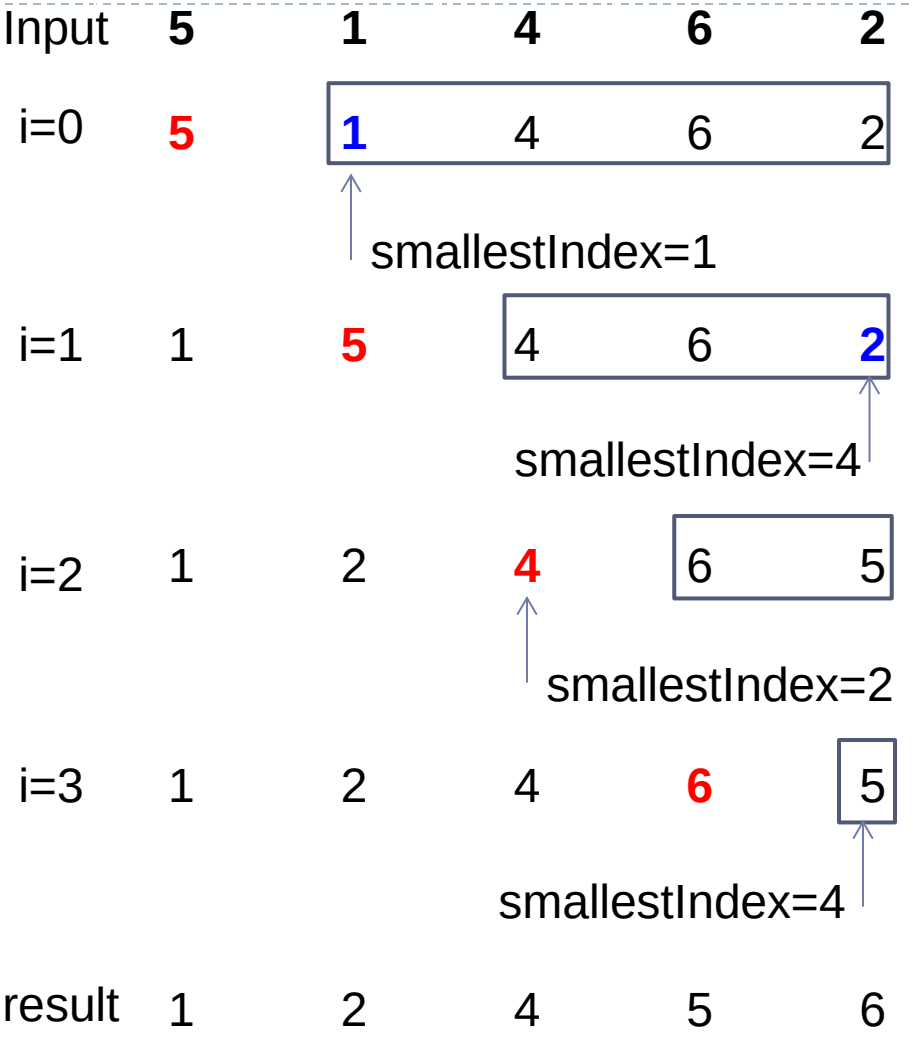
Selection Sort

```
for (int i = 0; i < n - 1; i++)
{
   int smallestIndex = i;
   for (int j = i + 1; j < n; j++)
    {
          if (a[j] < a[smallestIndex])
             smallestIndex = j;
    }
    swap(a[i], a[smallestIndex]);
}
```

- Is this stable sorting algorithm ?
- What's the running time?
- What's the best/worst/average case?
- Big-O?

Selection Sort - Example

```
for (int i = 0; i < n - 1; i++)
{
   int smallestIndex = i;
   for (int j = i + 1; j < n; j++)
    {
          if (a[j] < a[smallestIndex])
             smallestIndex = j;
    }
    swap(a[i], a[smallestIndex]);
}
```



Merge Sort - Let's divide and conquer

- How it works?
  - Merge Sort(list)
    - Divide the list into two sub-lists (sublist1, sublist2)
    - Merge Sort (sublist1)
    - Merge Sort (sublist2)
  - Two types of merge sort: top-down & bottom-up
    - **Implementation of top-down approach is given in Week 3 prac's solution**

Merge Sort - Can you tell what is the Big-O?

```
// top-down
public void MergeSort(listToSort, int leftIndex, int rightIndex) {
    int midIndex = (leftIndex + rightIndex) / 2;
    if (leftIndex < rightIndex) {
        MergeSort(listToSort, leftIndex, midIndex);
        MergeSort(listToSort, midIndex + 1, rightIndex);
        PartitionAndMerge(listToSort, leftIndex, midIndex, rightIndex);
    }
}
```

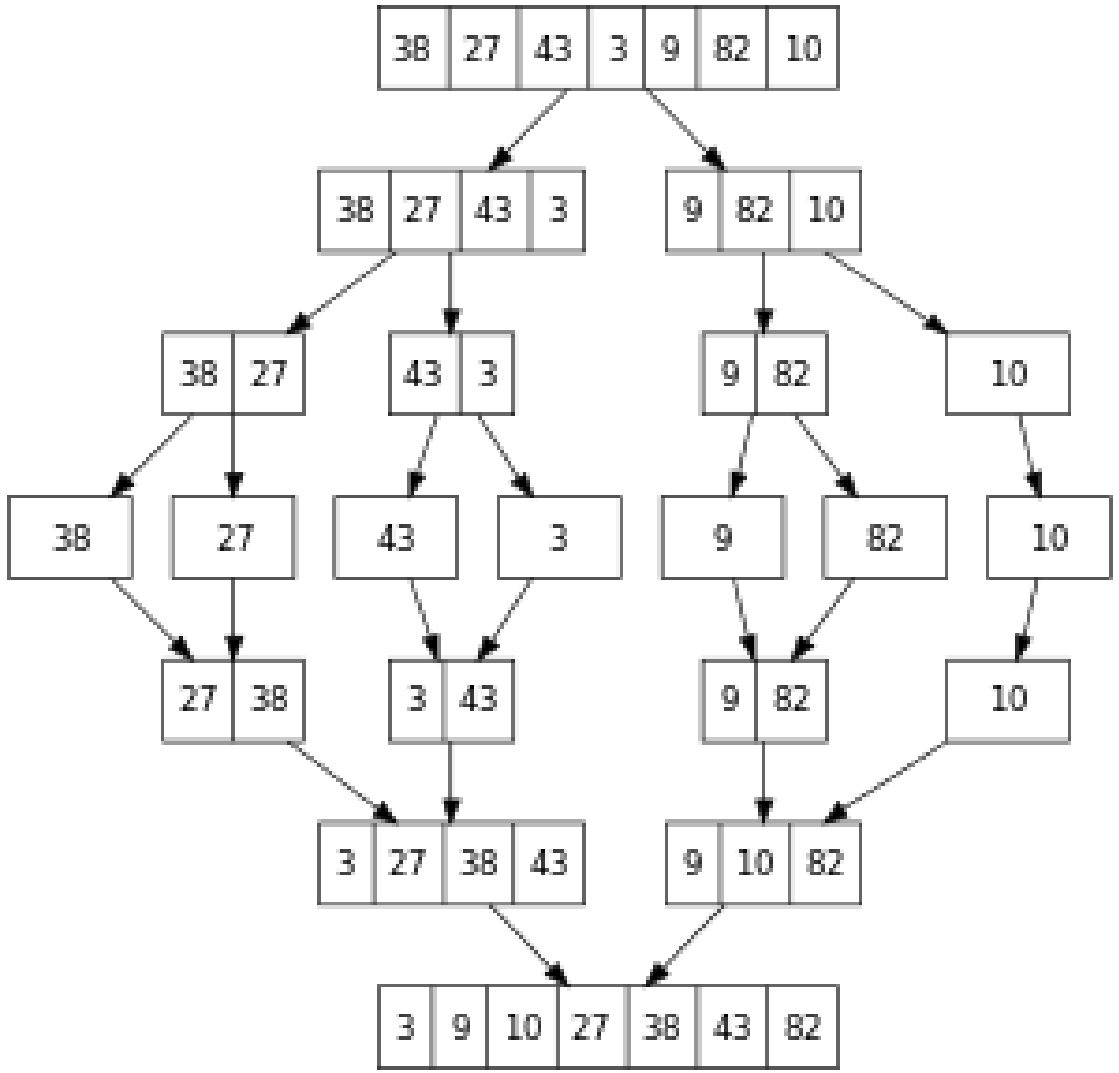
```
public void Merge(listToSort, int left, int mid, intright) {
    …
    while ( left < mid) && mid <= right ) {
        if (listToSort[left] <= listToSort [mid])
            temp[tmp_pos++] =  listToSort [left++];
        else
            temp[tmp_pos++] = listToSort [mid++];
    }

    while (left <= left_end)
        temp[tmp_pos++] = listToSort [left++];

    while (mid <= right)
        temp[tmp_pos++] = listToSort[mid++];

    for (i = 0; i < num_elements; i++)  {
        listToSort [right] = temp[right];
        right--;
    }
}
```

Merge Sort - Example



Quick Sort

1.Pick a *pivot* element.

2.Reorder array elements such that all elements less than the pivot come before the pivot, while all elements greater than the pivot come after it (equal values can go either way).

3.Repeat the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

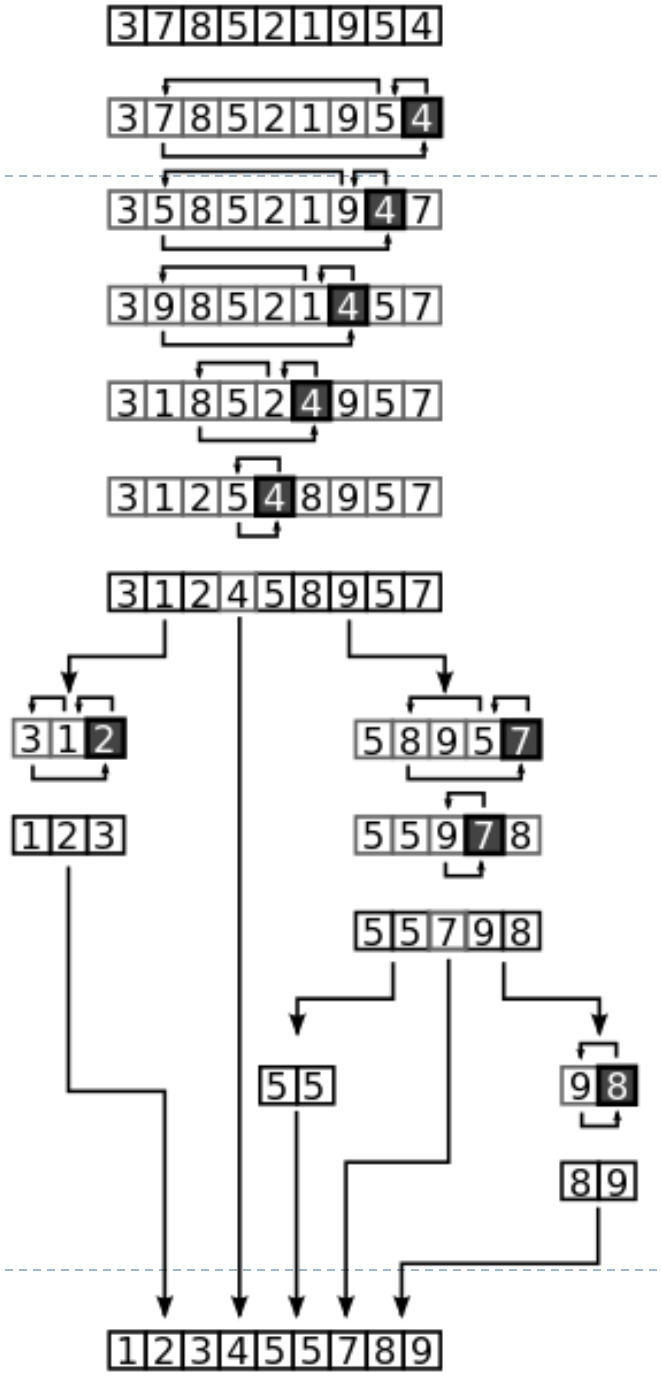
Implementation is given in Week 3 prac's solution

Quick Sort

```
Public void quicksort(int[] A, int lo, int hi) {
    if (lo < hi)  {
        int pivot = partition(A, lo, hi);
        quicksort(A, lo, pivot – 1);
        quicksort(A, pivot + 1, hi) ;
    }
}
```

```
Public int partition(int[] A,  int lo,  int hi) {
    int pivot := A[hi];
    int i := lo-1;  // i is the wall between low &
                    // high element
    for (int j := lo; j <= hi – 1; j++) {
        if (A[j] ≤ pivot) {
            i++;
            swap(A[i], A[j]);
        }
    }
    if (A[hi] < A[i+1)
        swap (A[i+1], A[hi]);
    return i+1;
}
```

Quick Sort - Example



Sorting Algorithms Time & Space complexity

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Timsort	O(n)	O(n log(n))	O(n log(n))	O(n)
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Tree Sort	O(n log(n))	O(n log(n))	O(n^2)	O(n)
Shell Sort	O(n log(n))	O(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(k)
Cubesort	O(n)	O(n log(n))	O(n log(n))	O(n)

http://bigocheatsheet.com/

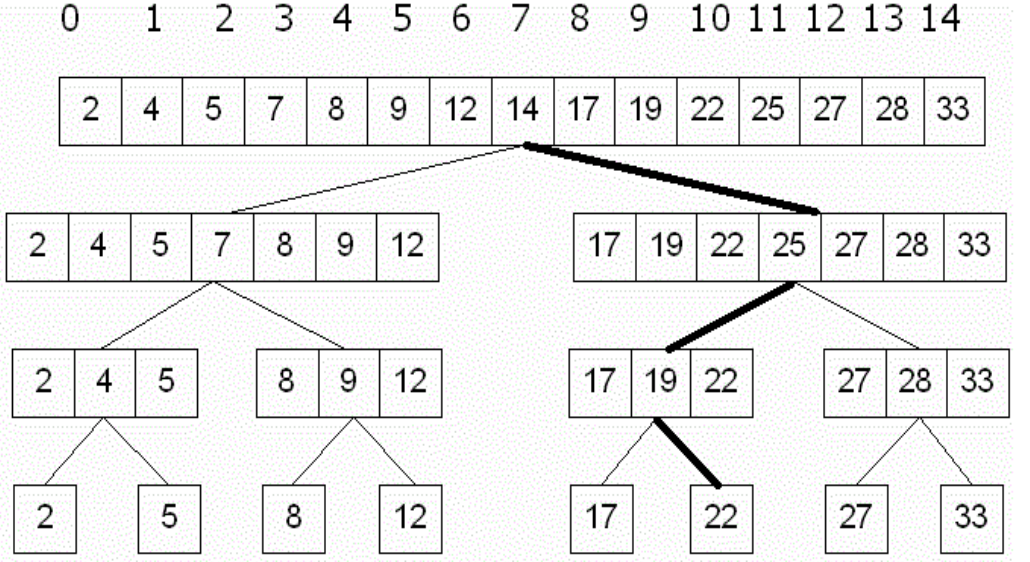
Linear Search

- How many of you know this one? Can you tell the Big-O?
- What are the key conditions we have
  - Have we reached the end of the array?
  - Is the currently checked element is the element we looking for?
  - Can we improve? Yes
    - Sentinel Search – Put it in
    - Probability Search – Learn from your previous search – shuffle elements around

Binary Search Big-O?

- Have you every searched for a contact number in the telephone book? Why is it fast?

```
public int BinarySearch(int[] source, int value, int left, int right) {
    if (left <= right) {
        int middle = (left + right) / 2;
        if (source[middle] == value)
            return middle;
        if (source[middle] > value)
            return BinarySearch(source, value, left, middle - 1);
        if (source[middle] < value)
            return BinarySearch(source, value, middle + 1, right);
    }
    return -1;
}
```



Practical 2

- Let’s add sorting & searching capabilities to our Vector class.
- Let’s compare data using IComparer & IComparable interfaces