CS1100 – Introduction to Programming

Instructor:

Shweta Agrawal (shweta.a@cse.iitm.ac.in)

Lecture 26

Recursive Thinking: Largest Element in an Array

(Better) recursive thinking: Find the largest of the first half, then in the second half, and then return the largest of the two.

Recursive Thinking: Largest Element in an Array

(Better) recursive thinking: Find the largest of the first half, then in the second half, and then return the largest of the two.

```
int largest(int i, int j)
{
  if (i == j) return arr[i];

  int l1,l2;
  l1 = largest(i,(i+j)/2);
  l2 = largest((i+j)/2+1,j);
  if (l1 > l2)
       return l1;
  else return l2;
}
```

Why is this algorithm better?

- We divide the array into two equal halves, recursively call `largest` on each half, and perform one comparison after they return.

Why is this algorithm better?

- We divide the array into two equal halves, recursively call `largest` on each half, and perform one comparison after they return.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.

Why is this algorithm better?

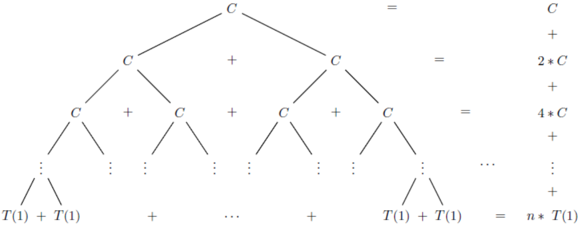
- We divide the array into two equal halves, recursively call `largest` on each half, and perform one comparison after they return.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.
- Then we have: $T(n) = 2T(n/2) + C$.

Why is this algorithm better?

- We divide the array into two equal halves, recursively call `largest` on each half, and perform one comparison after they return.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.
- Then we have: $T(n) = 2T(n/2) + C$.
- Depth of the *recursion tree* is $\log n$, and total time taken is $\approx n \times K$ for some constant K.

Why is this algorithm better?

- We divide the array into two equal halves, recursively call `largest` on each half, and perform one comparison after they return.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.
- Then we have: $T(n) = 2T(n/2) + C$.
- Depth of the *recursion tree* is $\log n$, and total time taken is $\approx n \times K$ for some constant K.



What happens in the older algorithm?

- We divide the array into two *unequal* parts of size 1 and $n-1$, recursively call `largest` on the sub-array, and perform one comparison after they return.

What happens in the older algorithm?

- We divide the array into two *unequal* parts of size 1 and $n-1$, recursively call `largest` on the sub-array, and perform one comparison after they return.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.

What happens in the older algorithm?

- We divide the array into two *unequal* parts of size 1 and $n - 1$, recursively call `largest` on the sub-array, and perform one comparison after they return.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.
- Then we have: $T(n) = T(n-1) + C$.

What happens in the older algorithm?

- We divide the array into two *unequal* parts of size 1 and $n-1$, recursively call `largest` on the sub-array, and perform one comparison after they return.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.
- Then we have: $T(n) = T(n-1) + C$.
- Depth of the *recursion tree* is n, and total time taken is again $\approx n \times K$ for some constant K.

What happens in the older algorithm?

- We divide the array into two *unequal* parts of size 1 and $n - 1$, recursively call `largest` on the sub-array, and perform one comparison after they return.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.
- Then we have: $T(n) = T(n-1) + C$.
- Depth of the *recursion tree* is n, and total time taken is again $\approx n \times K$ for some constant K.
- So total time is roughly the same. But depth of recursion tree is larger in one case.

What happens in the older algorithm?

- We divide the array into two *unequal* parts of size 1 and $n-1$, recursively call `largest` on the sub-array, and perform one comparison after they return.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.
- Then we have: $T(n) = T(n-1) + C$.
- Depth of the *recursion tree* is n, and total time taken is again $\approx n \times K$ for some constant K.
- So total time is roughly the same. But depth of recursion tree is larger in one case.
- The *space* complexity of recursive algorithm is proportinal to maximum depth of recursion tree generated. So this is how the second algorithm is better than the first.

Recall: Coding Binary Search

```
#include <stdio.h>
#define SIZE 1000000
int deepmax(int arr[], int start, int end) {
    if (start == end) return arr[start];
    else {
         int l = deepmax(arr, start+1, end);
         if (arr[start] > l) return arr[start];
         else return l;
    }
}
int shallowmax(int arr[], int start, int end) {
    if (start == end) return arr[start];
    else  {
         int mid = (start+end)/2;
         int l1 = shallowmax(arr, start, mid);
         int l2 = shallowmax(arr, mid+1, end);
         if (l1 > l2) return l1;
         else return l2;
    }
}
main() {
    int arr[SIZE];
    for (int i=0; i<SIZE; i++) {
        arr[i] = i;
    }
    int max1 = shallowmax(arr, 0, SIZE-1);
    printf("shallowmax answer = %d\n", max1);
    int max2 = deepmax(arr, 0, SIZE-1);
    printf("deepmax answer = %d\n", max2);
}
```

Analyzing Binary Search

Given an array A that is sorted, search for a given element `key` in the array.

Analyzing Binary Search

Given an array A that is sorted, search for a given element `key` in the array.

- Divide the array into two halves, check if the key element is less than the middle element - then search in the left half of the array, else search in the right half of the array.

Analyzing Binary Search

Given an array A that is sorted, search for a given element `key` in the array.

- Divide the array into two halves, check if the key element is less than the middle element - then search in the left half of the array, else search in the right half of the array.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.

Analyzing Binary Search

Given an array A that is sorted, search for a given element `key` in the array.

- Divide the array into two halves, check if the key element is less than the middle element - then search in the left half of the array, else search in the right half of the array.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.
- Then we have: $T(n) = T(n/2) + C$.

Analyzing Binary Search

Given an array A that is sorted, search for a given element `key` in the array.

- Divide the array into two halves, check if the key element is less than the middle element - then search in the left half of the array, else search in the right half of the array.
- Let us say C is the time taken for a single comparison and $T(n)$ is the time taken for the program on input size n.
- Then we have: $T(n) = T(n/2) + C$.
- This gives us $T(n) \approx \log_2(n)$.

The relations with $T(n)$ in LHS are called *recurrence relations*

Recall: Coding Binary Search

```
#include <stdio.h>

int binarySearch(int array[], int x, int low, int high) {
  if (high >= low) {
    int mid = low + (high - low) / 2;

    // If found at mid, then return it
    if (array[mid] == x)
      return mid;

    // Search the left half
    if (array[mid] > x)
      return binarySearch(array, x, low, mid - 1);

    // Search the right half
    return binarySearch(array, x, mid + 1, high);
  }
  return -1;
}

int main(void) {
  int array[] = {3, 4, 5, 6, 7, 8, 9};
  int n = sizeof(array) / sizeof(array[0]);
  int x = 4;
  int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)
    printf("Not found");
  else
    printf("Element is found at index %d", result);
}
```

Sorting an array in decreasing order

Task : Given array of n ($n \leq 1000$) numbers. Sort them in decreasing order of numbers.

15	8	3	12	30	7	9	17	32	19

Sorting an array in decreasing order

Task : Given array of n ($n \leq 1000$) numbers. Sort them in decreasing order of numbers.

15	8	3	12	30	7	9	17	32	19

One possible way: An **algorithm**

- Find max, place it at first location.
- Sort the array from second location to end.

Called **Selection Sort**.

Selection sort

15	8	3	12	30	7	9	17	32	19

Selection sort

15	8	3	12	30	7	9	17	32	19
32	8	3	12	30	7	9	17	15	19

Selection sort

15	8	3	12	30	7	9	17	32	19
32	8	3	12	30	7	9	17	15	19
32	30	3	12	8	7	9	17	15	19

Selection sort

15	8	3	12	30	7	9	17	32	19
32	8	3	12	30	7	9	17	15	19
32	30	3	12	8	7	9	17	15	19
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
32	30	19	17	15	12	9	8	7	3

Pseudo-code :

- while $(i \leq n)$
  - maxindex $=$ index of the max element in the part of the array indexed from i to n. Find maxindex.

Selection sort

15	8	3	12	30	7	9	17	32	19
32	8	3	12	30	7	9	17	15	19
32	30	3	12	8	7	9	17	15	19
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
32	30	19	17	15	12	9	8	7	3

Pseudo-code :

- while $(i \leq n)$
  - maxindex $=$ index of the max element in the part of the array indexed from i to n. Find maxindex. (We have solved this !!)

Selection sort

15	8	3	12	30	7	9	17	32	19
32	8	3	12	30	7	9	17	15	19
32	30	3	12	8	7	9	17	15	19
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
32	30	19	17	15	12	9	8	7	3

Pseudo-code :

- while $(i \leq n)$
  - maxindex = index of the max element in the part of the array indexed from i to n. Find maxindex. (We have solved this !!)
  - swap elements array[i] and array[maxindex];

Selection sort

15	8	3	12	30	7	9	17	32	19
32	8	3	12	30	7	9	17	15	19
32	30	3	12	8	7	9	17	15	19
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
32	30	19	17	15	12	9	8	7	3

Pseudo-code :

- while ($i \leq n$)
  - maxindex = index of the max element in the part of the array indexed from i to n. Find maxindex. (We have solved this !!)
  - swap elements array[i] and array[maxindex]; (We have solved this too !!)

Selection sort - from the pseudocode to the program

Pseudo-code :

for i ranging from 1 to n

- maxindex = the index of the max element in the part of the array indexed from i to n. Find maxindex.
- swap elements array[i] and array[maxindex];

Selection sort - from the pseudocode to the program

Pseudo-code :

for *i* ranging from 1 to *n*

- maxindex = the index of the max element in the part of the array indexed from *i* to *n*. Find maxindex.
- swap elements array[i] and array[maxindex];

Program Segment:

```
for (i=0; i<n; i++)
{
   max = i;
   for (j=i+1; j<n; j++)
   {
     if (a[j] > a[max])
        max = j;
   }
   temp = a[i];
   a[i] = a[max];
   a[max] = temp;
}
```

Selection sort – number of comparisons

- Which input do we consider?

Selection sort – number of comparisons

- Which input do we consider?
- Do number of comparisons depend on the particular array values?

Selection sort – number of comparisons

- Which input do we consider?
- Do number of comparisons depend on the particular array values?
- How does the method perform when the array is nearly sorted?

Selection sort – number of comparisons

- Which input do we consider?
- Do number of comparisons depend on the particular array values?
- How does the method perform when the array is nearly sorted?

- Consider a "worst-case" input.

Selection sort – number of comparisons

- Which input do we consider?
- Do number of comparisons depend on the particular array values?
- How does the method perform when the array is nearly sorted?

- Consider a "worst-case" input.
- Irrespective of whether the array is sorted or not, the method always needs $\frac{n(n-1)}{2}$ comparisons.

From Modular Perspective: Selection Sort

Selection Sort: Sort n numbers in descending order

Pseudo-code :

for i ranging from 1 to n

- maxindex = the index of the max element in the part of the array indexed from i to n. Find maxindex.
- swap elements array[i] and array[maxindex];

From Modular Perspective: Selection Sort

Selection Sort: Sort n numbers in descending order

Pseudo-code :

for i ranging from 1 to n

- maxindex = the index of the max element in the part of the array indexed from i to n. Find maxindex.
- swap elements array[i] and array[maxindex];

Subtasks identified:

`findmax(i,n)` : find the index of maxelement in the subarray from i to n.

`swap(i,j)` : swap i^{th} and j^{th} elements of A.

From Modular Perspective: Selection Sort

Selection Sort: Sort n numbers in descending order

Pseudo-code :

for i ranging from 1 to n

- maxindex = the index of the max element in the part of the array indexed from i to n. Find maxindex.
- swap elements array[i] and array[maxindex];

Subtasks identified:

`findmax(i,n)` : find the index of maxelement in the subarray from i to n.

`swap(i,j)` : swap i^{th} and j^{th} elements of A.

Once this is done (and solved), here is the remaining code.

```
for i = 1 to n
{
  max = findmax(i,n);
  swap(i,max);
}
```

Insertion Sort

Insertion Sort

15	8	3	12	30	7	9	17	32	19

Insertion Sort

15	8	3	12	30	7	9	17	32	19
15	8	3	12	30	7	9	17	32	19

Insertion Sort

15	8	3	12	30	7	9	17	32	19
15	8	3	12	30	7	9	17	32	19
15	12	8	3	30	7	9	17	15	19

Insertion Sort

15	8	3	12	30	7	9	17	32	19
15	8	3	12	30	7	9	17	32	19
15	12	8	3	30	7	9	17	15	19
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
32	30	19	17	15	12	9	8	7	3

Insertion Sort

15	8	3	12	30	7	9	17	32	19
15	8	3	12	30	7	9	17	32	19
15	12	8	3	30	7	9	17	15	19
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
32	30	19	17	15	12	9	8	7	3

- Although final result is the same, intermediate steps are different from selection sort.

Insertion Sort

15	8	3	12	30	7	9	17	32	19
15	8	3	12	30	7	9	17	32	19

Insertion Sort

15	8	3	12	30	7	9	17	32	19
15	8	3	12	30	7	9	17	32	19
15	8	12	3	30	7	9	17	32	19

Insertion Sort

15	8	3	12	30	7	9	17	32	19
15	8	3	12	30	7	9	17	32	19
15	8	12	3	30	7	9	17	32	19
15	12	8	3	30	7	9	17	32	19

Insertion Sort

15	8	3	12	30	7	9	17	32	19
15	8	3	12	30	7	9	17	32	19
15	8	12	3	30	7	9	17	32	19
15	12	8	3	30	7	9	17	32	19
15	12	8	3	30	7	9	17	32	19

Insertion Sort

15	8	3	12	30	7	9	17	32	19
15	8	3	12	30	7	9	17	32	19
15	8	12	3	30	7	9	17	32	19
15	12	8	3	30	7	9	17	32	19
15	12	8	3	30	7	9	17	32	19

```
// Array index starts from 0.

for (i=1; i<= n-1; i++) {
    j = i;
    if (A[j] <= A[j-1])
        continue;
    // Now A[j] > A[j-1]

    // swap A[j] with A[j-1] till ...?
}
```

Insertion Sort

15	8	3	12	30	7	9	17	32	19
15	8	3	12	30	7	9	17	32	19
15	8	12	3	30	7	9	17	32	19
15	12	8	3	30	7	9	17	32	19
15	12	8	3	30	7	9	17	32	19

```
// Array index starts from 0.
for (i=1; i<= n-1; i++) {
    j = i;
    if (A[j] <= A[j-1])
        continue;
    // Now A[j] > A[j-1]
    while (A[j] > A[j-1]) {
        // Swap A[j] and A[j-1]
        j = j-1;
    }
}
```

Insertion Sort

15	8	3	12	30	7	9	17	32	19
15	8	3	12	30	7	9	17	32	19
15	8	12	3	30	7	9	17	32	19
15	12	8	3	30	7	9	17	32	19
15	12	8	3	30	7	9	17	32	19

```
// Array index starts from 0.
for (i=1; i<= n-1; i++) {
    j = i;
    if (A[j] <= A[j-1])
        continue;
    // Now A[j] > A[j-1]
    while (A[j] > A[j-1]) {
        // Swap A[j] and A[j-1]
        j = j-1;
    }
}
```

What happens for j=4?

Insertion Sort

15	8	3	12	30	7	9	17	32	19
15	8	3	12	30	7	9	17	32	19
15	8	12	3	30	7	9	17	32	19
15	12	8	3	30	7	9	17	32	19
15	12	8	3	30	7	9	17	32	19

```
// Array index starts from 0.

for (i=1; i<= n-1; i++) {
    j = i;
    while (A[j]>A[j-1] && j>0) {
        // Swap A[j] and A[j-1]
        j = j-1;
    }
}
```

- Note $j > 0$
- Is it correct?

Insertion Sort

15	8	3	12	30	7	9	17	32	19
15	8	3	12	30	7	9	17	32	19
15	8	12	3	30	7	9	17	32	19
15	12	8	3	30	7	9	17	32	19
15	12	8	3	30	7	9	17	32	19

```
// Array index starts from 0.

for (i=1; i<= n-1; i++) {
    j = i;
    while (A[j]>A[j-1] && j>0) {
        // Swap A[j] and A[j-1]
        j = j-1;
    }
}
```

- Note $j > 0$
- Is it correct? No!

Insertion Sort

```
// Array index starts from 0.

for (i=1; i<= n-1; i++) {
    j = i;
    while (j>0 && A[j]>A[j-1]) {
        // Swap A[j] and A[j-1]
        j = j-1;
    }
}
```

- Need to check ($j > 0$) before checking A[j] $>$ A[j-1]!
- Note the use of short-circuiting.

Insertion Sort – another way

15	8	3	12	30	7	9	17	32	19
15	8	3	12	30	7	9	17	32	19

Insertion Sort – another way

15	8	3	12	30	7	9	17	32	19
15	8	3	12	30	7	9	17	32	19
15	8	3	3	30	7	9	17	32	19

Insertion Sort – another way

15	8	3	12	30	7	9	17	32	19
15	8	3	12	30	7	9	17	32	19
15	8	3	3	30	7	9	17	32	19
15	8	8	3	30	7	9	17	32	19

Insertion Sort – another way

15	8	3	12	30	7	9	17	32	19
15	8	3	12	30	7	9	17	32	19
15	8	3	3	30	7	9	17	32	19
15	8	8	3	30	7	9	17	32	19
15	12	8	3	30	7	9	17	32	19

```
// Array index starts from 0.

for (i=1; i<=n-1; i++) {
    x = A[i];
    j = i-1;
    while (--------) {
        A[j+1] = A[j];
        j = j-1;
    }
}
```

Insertion Sort – another way

15	8	3	12	30	7	9	17	32	19
15	8	3	12	30	7	9	17	32	19
15	8	3	3	30	7	9	17	32	19
15	8	8	3	30	7	9	17	32	19
15	12	8	3	30	7	9	17	32	19

```
// Array index starts from 0.

for (i=1; i<=n-1; i++) {
    x = A[i];
    j = i-1;
    while ( j >= 0 && x > A[j]) {
        A[j+1] = A[j];
        j = j-1;
    }
    A[j+1] = x;
}
```

Insertion Sort – another way

15	8	3	12	30	7	9	17	32	19
15	8	3	12	30	7	9	17	32	19
15	8	3	3	30	7	9	17	32	19
15	8	8	3	30	7	9	17	32	19
15	12	8	3	30	7	9	17	32	19

```
// Array index starts from 0.

for (i=1; i<=n-1; i++) {
    x = A[i];
    j = i-1;
    while ( j >= 0 && x > A[j]) {
        A[j+1] = A[j];
        j = j-1;
    }
    A[j+1] = x;
}
```

- Note $j \geq 0$ (line 6).

Insertion Sort – another way

15	8	3	12	30	7	9	17	32	19
15	8	3	12	30	7	9	17	32	19
15	8	3	3	30	7	9	17	32	19
15	8	8	3	30	7	9	17	32	19
15	12	8	3	30	7	9	17	32	19

```
// Array index starts from 0.

for (i=1; i<=n-1; i++) {
    x = A[i];
    j = i-1;
    while ( j >= 0 && x > A[j]) {
        A[j+1] = A[j];
        j = j-1;
    }
    A[j+1] = x;
}
```

- Note j $\geq$ 0 (line 6).
- See line 10.