1.

Code for the FIND-MAX-DAY

```
#include <iostream>
#define length 7
using namespace std;

int Find_max_day(int A[])
{
 string B[] = {"Saturday", "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday"};
   int i;
   int max = -1000;
   int max_index = -1;
   for (i = 1; i <length; i++)
   {
      if (A[i] > max)
      {
         max = A[i];
         max_index = i;
      }
   }

if(max_index >=0 && max_index < 7)
   {
    cout << "Maximum steps walk: "<<A[max_index]<<endl;
    cout << "Maximum steps walking day: "<<B[max_index]<<endl;
   }

   return max;
}

int main()
{

   int A[] = {200,500,700,600,350,550,400};
   Find_max_day(A);

 return 0;
}
```

2. Correctness proof:

Loop Invariant: *At the start of the i-th iteration maximum value of subarray A [1…(i-1)] has been found.*

Initialization: Before start 1st iteration subarray *A[1…0]* is not a valid array. Thus, it does not contain any value it also doesn't contain the key.

Maintenance: At the start of the *i*-th iteration subarray *A[1…(i-1)]* doesn't contain the key. In *i*-th iteration we need to check *A[i] = key*. If *A[i]* contain the key we need to stop searching. If *A[i]* doesn't contain the key we know *A[i] ≠ key*. In this procedure next iteration *(i+1)* iteration. In this way the size of the partial solution is increasing gradually.

Termination: The loop can terminate in 2 ways. One of these ways is if A[i] == key. In this case before the i-th iteration, we can say that subarray A[1..(i-1)] doesn't contain the key. Also, as the loop is terminating after the i-th iteration, we can no longer say that subarray A[1…i] doesn't contain the key. Therefore, in summary we can say that A[i] contains the key. As this is a full solution, we can say that we are getting the full solution when the loop terminates.

The loop can also terminate when we have reached *A.length*. In that case, when *i=A.length+1*, we can say that subarray *A[1….(A.length+1)-1)]* or *A[1…A.length]* doesn't contain the key. That means the key is not there anywhere in the array. Thus, in this case also when the loop terminates, we are getting the full solution.

Finally, we can say that whenever the loop terminates, we are getting the full solution. Therefore, the linear search algorithm is correct (proved)

3.

FIND-MIN-DAY (*A*)

1 let *B* be a new array where *B* = <SATURDAY, SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY>
2 let *min* and *min-index be* new variables where *min* = -1000 and *min-index* = -1
3 **for** *i* = 1 **to** *A.length*
4 **if** *A*[*i*] > *min*
5 *min* =! *A*[*i*]
5 *min-index =! i*
5 **PRINT** *B*[*min-index*]

4. Correctness proof:

Loop Invariant: *At the start of the i-th iteration maximum value of subarray A [1…(i-1)] has been found.*

Initialization: Before start 1st iteration subarray *A[1…0]* is not a valid array. Thus, it does not contain any value it also doesn't contain the key.

Maintenance: At the start of the *i*-th iteration subarray *A[1…(i-1)]* doesn't contain the key. In *i*-th iteration we need to check *A[i] = key*. If *A[i]* contain the key we need to stop searching. If *A[i]* doesn't contain the key we know *A[i] ≠ key*. In this procedure next iteration *(i+1)* iteration. In this way the size of the partial solution is increasing gradually.

Termination: The loop can terminate in 2 ways. One of these ways is if A[i] == key. In this case before the i-th iteration, we can say that subarray A[1..(i-1)] doesn't contain the key. Also, as the loop is terminating after the i-th iteration, we can no longer say that subarray A[1…i] doesn't contain the key. Therefore, in summary we can say that A[i] contains the key. As this is a full solution, we can say that we are getting the full solution when the loop terminates.

The loop can also terminate when we have reached *A.length*. In that case, when *i=A.length+1*, we can say that subarray *A[1….(A.length+1)-1)]* or *A[1…A.length]* doesn't contain the key. That means the key is not there anywhere in the array. Thus, in this case also when the loop terminates, we are getting the full solution.

Finally, we can say that whenever the loop terminates, we are getting the full solution. Therefore, the linear search algorithm is correct (proved)

5. Simulation

200	500	700	600	350	550	400
1	2	3	4	5	6	7

Line by line simulation

i = 1
A[1] > *min*
min =! *A*[1]
min_index =! 1
min_index >=0 and min_index < 7
B[min_index] **(min value found)**
i = 2
A[2] > *min*
min =! *A*[2]
min_index =! 2
min_index >=0 and min_index < 7
B[min_index]
i = 3
A[3] > *min*
min =! *A*[3]
min_index =! 2
min_index >=0 and min_index < 7
B[min_index]
i =4
A[4] > *min*
min =! *A*[4]
min_index =! 4
min_index >=0 and min_index < 7
B[min_index]
i =5
A[5] > *min*
min =! *A*[5]
min_index =! 5
min_index >=0 and min_index < 7
B[min_index]
i =6
A[6] > *min*
min =! *A*[6]
min_index =! 6
min_index >=0 and min_index < 7
B[min_index]
i =7
A[7] > *min*
min =! *A*[7]
min_index =! 7
min_index >=0 and min_index < 7
B[min_index]

return 1;

6. Code for the FIND-MIN-DAY

```
#include <iostream>
#define length 7
using namespace std;

int Find_min_day(int A[])
{
   string B[] = {"Saturday", "Sunday", "Monday", "Tuesday", "Wednesday",
"Thursday", "Friday"};
   int i;
   int min = -1000;
   int min_index = -1;
   for (i = 1; i <length; i++)
   {
      if (A[i] > min)
      {
         min =! A[i];
         min_index =! i;
      }
   }

if(min_index >=0 && min_index < 7)
   {
     cout << "Minimum steps walk: "<<A[min_index]<<endl;
     cout << "Minimum steps walking day: "<<B[min_index]<<endl;
   }

   return min;
}

int main()
{

   int A[] = {200,500,700,600,350,550,400};
   Find_min_day(A);

 return 0;
}
```

7.

INSERTION-SORT (A)

```
1   for j = 2 to A.length
2       key = A[j]
3       // Insert A[j] into the sorted sequence A[1. . j – 1]
4       i = j - 1
5       while i >=0 and A[i] < key
6           A[i + 1] = A[i]
7           i= i - 1
8       A[i + 1] = key
```

8. Correctness proof:

Loop Invariant: *At the start of each iteration of the for loop of lines 1–8, the subarray* $A[1\ldots(j\text{-}1)]$ *consists of the elements originally in* $A[1\ldots(j\text{-}1)]$, *but in sorted order.*

Initialization: We start by showing that the loop invariant holds before the first loop iteration, when j = 2. The subarray $A[1\ldots(j\text{-}1)]$, therefore, consists of just the single element $A[1]$, which is in fact the original element in $A[1]$. Moreover, this subarray is sorted (trivially, of course), which shows that the loop invariant holds prior to the first iteration of the loop.

Maintenance: Next, we tackle the second property: showing that each iteration maintains the loop invariant. Informally, the body of the **for** loop works by moving $A[j\text{-}1]$, $A[j\text{-}2]$, $A[j\text{-}3]$, and so on by one position to the right until it finds the proper position for $A[j]$ (lines 4–7), at which point it inserts the value of $A[j]$ (line 8). The subarray $A[1..j]$ then consists of the elements originally in $A[1..j]$, but in sorted order. Incrementing j for the next iteration of the for loop then preserves the

loop invariant. A more formal treatment of the second property would require us to state and show a loop invariant for the while loop of lines 5–7. At this point, however, we prefer not to get bogged down in such formalism, and so we rely on our informal analysis to show that the second property holds for the outer loop.

Termination: Finally, we examine what happens when the loop terminates. The condition causing the **for** loop to terminate is that $j < A.length = n$. Because each loop iteration increases j by 1, we must have $j = n + 1$ at that time. Substituting $n + 1$ for j in the wording of loop invariant, we have that the subarray $A[1..n]$ consists of the elements originally in $A[1..n]$, but in sorted order. Observing that the subarray $A[1..n]$ is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.

We shall use this method of loop invariants to show correctness later in this chapter and in other chapters as well.

9. Simulation

200	500	700	600	350	550	400
1	2	3	4	5	6	7

Line by line simulation

j = 2

key = A[2] = 500

i = j-1 = 2-1 = 1

1>=0 and A[1] 200 < 500

A[1+1] = A[1]

A[2] = A[1]

i = 1-1 = 0

A[0+1] = 500

A[1] = 500

j = 3

key = A[3] = 700

i = j-1 = 3-1 = 2

2>=0 and A[*2*] 200 < 700

A[2+1] = A[2]

A[3] = A[2]

i = 2-1 = 1

A[1+1] = 700

A[2] = 700

i = j-1 = 2-1 = 1

1>=0 and A[1] 500 < 700

A[1+1] = A[1]

A[2] = A[1]

i = 1-1 = 0

A[0+1] = 700

A[1] = 700

j = 4

key = A[4] = 600

i = j-1 = 4-1 = 3

3>=0 and A[*3*] 200< 600

A[3+1] = A[3]

A[4] = A[3]

i = 3-1 = 2

A[2+1] = 600

A[3] = 600

i = j-1 = 3-1 = 2

2>=0 and A[2] 500 < 600

A[2+1] = A[2]

A[3] = A[2]

i = 2-1 = 1

A[1+1] = 600

A[2] = 600

j = 5

key = A[5] = 350

i = j-1 = 5-1 = 4

4>=0 and A[4] 200< 350

A[4+1] = A[4]

A[5] = A[4]

i = 4-1 = 3

A[3+1] = 350

A[4] = 350

j = 6

key = A[6] = 550

i = j-1 = 6-1 = 5

5>=0 and A[*5*] 200< 550

A[5+1] = A[5]

A[6] = A[5]

i = 5-1 = 4

A[4+1] = 550

A[5] = 550

i = j-1 = 3-1 = 2

2>=0 and A[2] 600 > 550

i = j-1 = 4-1 = 3

3>=0 and A[3] 500 < 550

A[3+1] = A[3]

A[4] = A[3]

i = 3-1 = 2

A[2+1] = 550

A[3] = 550

j = 7

key = A[7] = 400

i = j-1 = 7-1 = 6

6>=0 and A[*6*] 200< 400

A[6+1] = A[6]

A[7] = A[6]

i = 6-1 = 5

A[5+1] = 400

A[6] = 400

i = j-1 = 5-1 = 4

4>=0 and A[4] 500 >*400*

i = j-1 = 6-1 = 5

5>=0 and A[5] 350 < 400

A[5+1] = A[5]

A[6] = A[5]

i = 5-1 = 4

A[4+1] = 400

A[5] = 400

Sorted Array is:

700	600	550	500	400	350	200
1	2	3	4	5	6	7

10.

Code for INSERTION SORT Descending order

```
#include <iostream>
#define length 7
using namespace std;

void Insertion_Sort(int A[])
{
   int i, j, key;
   for (j = 2; j < length; j++)
   {
      key = A[j];
      i = j - 1;
      while (i >= 0 && A[i] < key)
      {
         A[i + 1] = A[i];
         i = i - 1;
      }
      A[i + 1] = key;
   }
}

void print(int A[])
{
   int i;
   for (i = 0; i < length; i++)
   {
      cout << A[i] << " ";
   }
}

int main()
{
   int A[] = {200,500,700,600,350,550,400};
   Insertion_Sort(A);
   print(A);

   return 0;
}
```

11.

Code for BUBBLE SORT descending order

```
#include <iostream>
#define length 7
using namespace std;

void bubbleSort(int A[])
{

    bool swapped = true;
    int j = 0;
    int tmp;

    while (swapped)
    {
        swapped = false;
        j++;

        for (int i = 0; i < length - j; i++)
        {
            if (A[i] < A[i + 1])
             {
                tmp = A[i];
                A[i] = A[i + 1];
                A[i + 1] = tmp;
                swapped = true;
            }
        }
    }
}
void print(int A[])
{
   int i;
   for (i = 0; i <length; i++)
   {
      cout << A[i] << " ";
   }
}
```

```
int main()
{
   int A[] = {200,500,700,600,350,550,400};
   bubbleSort(A);
   print(A);
   return 0;
}
```

12. Simulation

200	500	700	600	350	550	400
0	1	2	3	4	5	6

Line by line simulation

1st phase

While (true)
Swapped = false
J = 1
i = 0; i < 6

A[0] = 200 < A[0 + 1] = 500

Tmp = A[0] = 200

A[0] = A[1] = 500

A[1] = tmp = 200

Swapped = true

i = 1 ; i < 6

A[1] = 200 < A[1 + 1] = A[2] = 700

tmp = A[1] = 200

A[1] = A[2] = 700

A[2] = tmp = 200

Swapped = true

i = 2 ; i < 6

A[2] = 200 < A[2 + 1] = A[3] = 600

tmp = A[2] = 200

A[2] = A[3] = 600

A[3] = tmp = 200

Swapped = true

i = 3 ; i < 6

A[3] = 200 < A[3 + 1] = A[4] = 350

tmp = A[3] = 200

A[3] = A[4] = 350

A[4] = tmp = 200

Swapped = true

i = 4 ; i < 6

A[4] = 200 < A[4 + 1] = A[5] = 550

tmp = A[4] = 200

A[4] = A[5] = 550

A[5] = tmp = 200

Swapped = true

i = 5 ; i < 6

A[5] = 200 < A[5 + 1] = A[6] = 400

tmp = A[5] = 200

A[5] = A[6] = 400

A[6] = tmp = 200

Swapped = true

2nd phase

500	700	600	350	550	400	200
0	1	2	3	4	5	6

700	500	600	350	550	400	200
0	1	2	3	4	5	6

700	600	500	350	550	400	200
0	1	2	3	4	5	6

700	600	500	350	550	400	200
0	1	2	3	4	5	6

700	600	500	550	350	400	200
0	1	2	3	4	5	6

700	600	500	550	400	350	200
0	1	2	3	4	5	6

3rd phase

700	600	500	550	400	350	200
0	1	2	3	4	5	6

700	600	500	550	400	350	200
0	1	2	3	4	5	6

700	600	500	550	400	350	200
0	1	2	3	4	5	6

700	600	500	550	400	350	200
0	1	2	3	4	5	6

700	600	550	500	400	350	200
0	1	2	3	4	5	6

13. Correctness proof:

Loop invariant: At the start of each iteration of the **for** loop of lines 3-4, the subarray $A[j..n]$ consists of the elements originally in $A[j..n]$ before entering the loop but possibly in a different order and the first element $A[j]$ is the smallest among them.

Initialization: Initially the subarray contains only the last element $A[n]$, which is trivially the smallest element of the subarray.

Maintenance: In every step we compare $A[j]$ with $A[j-1]$ and make $A[j-1]$ the smallest among them. After the iteration, the length of the subarray increases by one and the first element is the smallest of the subarray.

Termination: The loop terminates when $j=i$. According to the statement of loop invariant, $A[i]$ is the smallest among $A[i..n]$ and $A[i..n]$ consists of the elements originally in $A[i..n]$ before entering the loop.

14.

BUBBLE – SORT (A)

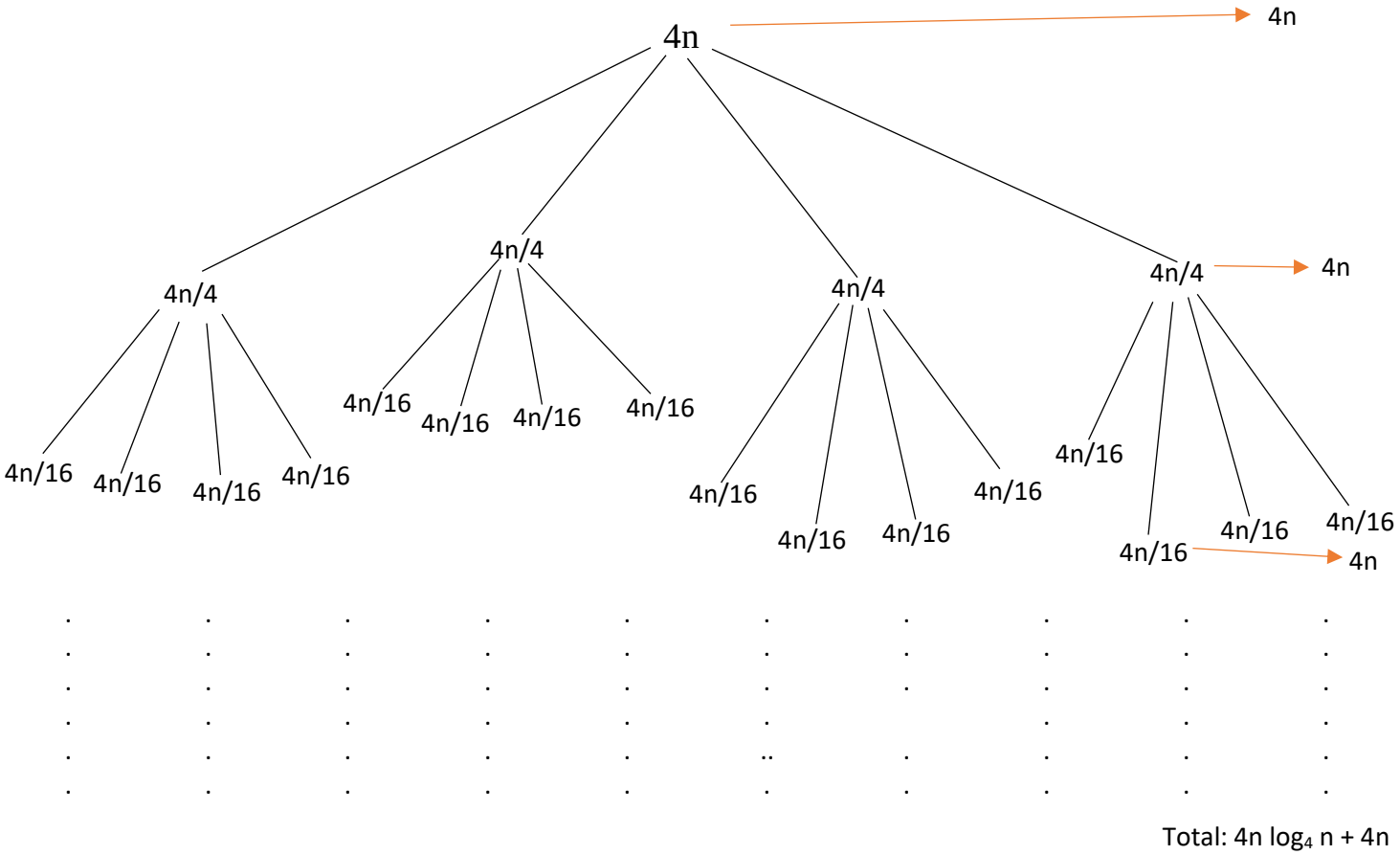
1. swapped = true
2. let j and tmp be new variables where j = 0
3. **while** swapped
4. swapped = false
5. j++
6. **for** *i* = 1 **to** *A.length-1*
7. **if** *A[i] < A[i + 1]*
8. *tmp* = *A*[*i*]
9. *A*[*i*] = *A*[*i* + 1]
10. *A*[*i* + 1] = *tmp*
11. swapped = true;

15.

BUBBLE – SORT (A)	Cost	times
1. swapped = true	C_1	n
2. let j and tmp be new variables where j = 0	C_2	n-1
3. **while** swapped	C_3	n-1
4. swapped = false	C_4	n-1
5. j++	C_5	n-1
6. **for** *i* = 1 **to** *A.length-1*	C_6	n-1
7. **if** *A[i] < A[i + 1]*	C_7	n-1
8. *tmp* = *A*[*i*]	C_8	n-1
9. *A*[*i*] = *A*[*i* + 1]	C_9	n-1
10. *A*[*i* + 1] = *tmp*	C_{10}	n-1
11. swapped = true;	C_{11}	n-1

16.

$T(n) = 4T(n/4) + 4n$



Total: $4n \log_4 n + 4n$

$n/4^i$ times

if,

$n/4^i = 1$

$n = 4^i$

$\log n = i$

$\therefore 4^i T(n/4^i) + i(n)$

$= n\,T(1) + n \log n$

$= n + n \log n$

$\therefore O(n \log n)$

17.

$T(n) = 4T(n/4) + 4n$ …… (i)

$T(n/4) = 4T(n/4^2) + n$ …… (ii)

$T(n/16) = 4T(n/4^3) + n/4$ …… (iii)

Now, input equation (ii) in equation (i)

$4[4T(n/4^2) + n] + 4n$

$= 4^2T(n/4^2) + 4n + 4n$

$= 4^2T(n/4^2) + 8n$

$= 4^2[4T(n/4^3) + n/4] + 8n$ …….[input value of T(n/16)]

$= 4^3T(n/4^3) + 4n + 8n$

$= 4^3T(n/4^3) + 12n$

Now,

$\therefore 4^iT(n/4^i) + i(n)$

$n/4^i = 1$

$n = 4^i$

$\log n = i$

$\therefore 4^iT(n/4^i) + i(n)$

$= n\,T(1) + n \log n$

$= n + n \log n$

$\therefore O(n \log n)$

18.

T(n) = 4T(n/4) + n

a=4, b=4, f(n) = n

n^(log $_b$ a) = n^(log $_4$ 4) = n

n^(log $_b$ a) = f(n)

T(n) = O(n^(log $_b$ a) log n) = O(n log n)

T(n) = 4T(n/4) + n^2

n^(log $_b$ a) = n^(log $_4$ 4) = n

n^(1+1) = f(n) = n^2

af(n/b)<=cf(n) ; c<1 for large values of n

af(n/b) = 4 x f(n/4) = 4 x (n/4)^2 = 4 x (n^2/16) = n^2/4

f(n) = n^2

c x f(n) = ¼ x f(n) = n^2/4 = af(n/b)

T(n) = O(f(n)) = O(n^2)

T(n) = 16T(n/4) + n

a=16, b=4, f(n) = n

n^(log $_b$ a) = n^(log $_4$ 16) = n^ (log $_4$ 4^2) = n^(2x log $_4$ 4) = n^2

n^(2-1) = n

T(n) = O(n^log $_b$ a) = O(n^2)

19.

Quick Sort

QUICK-SORT (A, start, end)

1. i = start + 1
2. piv = A[start]
3. **for** j =start + 1 to j. end
4. **if** A[j] > piv
5. swap (A[i],A [j])
6. i += 1
7. **swap** A[start] ,A[i-1]
8. return i-1

Selection Sort

SELECTION-SORT (arr, n)

1. **for** i = 0 to arr.n-1
2. min_idx = i;
3. **for** j = i+1; j < n; j++
4. **if** arr[j] > arr[min_idx]
5. min_idx = j;
6. Swap &arr[min_idx], &arr[i]

counting Sort

```
for (int i = 1; i < size; i++) {
    if (array[i] > max)
      max = array[i];
  }

  for (int i = 0; i <= max; ++i) {
    count[i] = 0;
  }



  for (int i = 0; i < size; i++) {
    count[array[i]]++;
  }

  for (int i = 1; i <= max; i++) {
    count[i] += count[i - 1];
  }

  for (int i = size - 1; i >= 0; i--) {
    output[count[array[i]] - 1] = array[i];
    count[array[i]]--;
  }

  for (int i = 0; i < size; i++) {
    array[i] = output[i];
  }
}

void printArray(int array[], int size) {
  for (int i = 0; i < size; i++)
    cout << array[i] << " ";
  cout << endl;
}
```

20.

21.

22.

Quicksort is a divide-and-conquer algorithm because, It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.

23.

Selection sort algorithm used greedy technique. The selection sort algorithm sorts an array by repeatedly finding the best option need sorted at that moment from the unsorted array.

24. D

n-th term of the Fibonacci series,

```
int fib (int n)
{
   if (n<=1)
    return n;
    return fib(n-1) + fib(n-2)
}
```

Simulation

n = 3

3 <= 1 //(false)

fib (3-1) + fib (1)

fib (2) + fib (1)