Internal Sorting by Comparison

Problem Specification

Consider the collection of data related to the students of a particular class. Each data consists of

- Roll Number: `char rollNo[9]`
- Name: `char name[50]`
- cgpa: `double cgpa`

It is necessary to prepare the merit list of the students.

Roll No.	Name	CGPA
02ZO2001	V. Bansal	7.50
02ZO2002	P. K. Singh	8.00
02ZO2003	Imtiaz Ali	8.50
02ZO2004	S. P. Sengupta	8.25
02ZO2005	P. Baluchandran	9.25
02ZO2006	V. K. R. V. Rao	9.00
02ZO2007	L. P. Yadav	6.50
02ZO2008	A. Maria Watson	8.00
02ZO2009	S. V. Reddy	7.00
02ZO2010	D. K. Sarlekar	7.50

Sorting

The merit list should be sorted on cgpa of students in descending order.

Roll No.	Name	CGPA
02ZO2005	P. Baluchandran	9.25
02ZO2006	V. K. R. V. Rao	9.00
02ZO2003	Imtiaz Ali	8.50
02ZO2004	S. P. Sengupta	8.25
02ZO2002	P. K. Singh	8.00
02ZO2008	A. Maria Watson	8.00
02ZO2001	V. Bansal	7.50
02ZO2010	D. K. Sarlekar	7.50
02ZO2009	S. V. Reddy	7.00
02ZO2007	L. P. Yadav	6.50

Problem Abstraction

We only consider the cgpa field for discussion of sorting algorithms.

Unsorted Data									
7.5	8.0	8.5	8.25	9.25	9.0	6.5	8.0	7.0	7.5

Sorted Data									
9.25	9.0	8.5	8.25	8.0	8.0	7.5	7.5	7.0	6.5

Sorting by Comparison

We shall consider sorting of data by comparison. There are other sorting techniques. We also assume that the whole data set is available in the main memory.

Simple Sorting Algorithms

- Selection Sort
- Insertion Sort
- Bubble Sort

Selection Sort

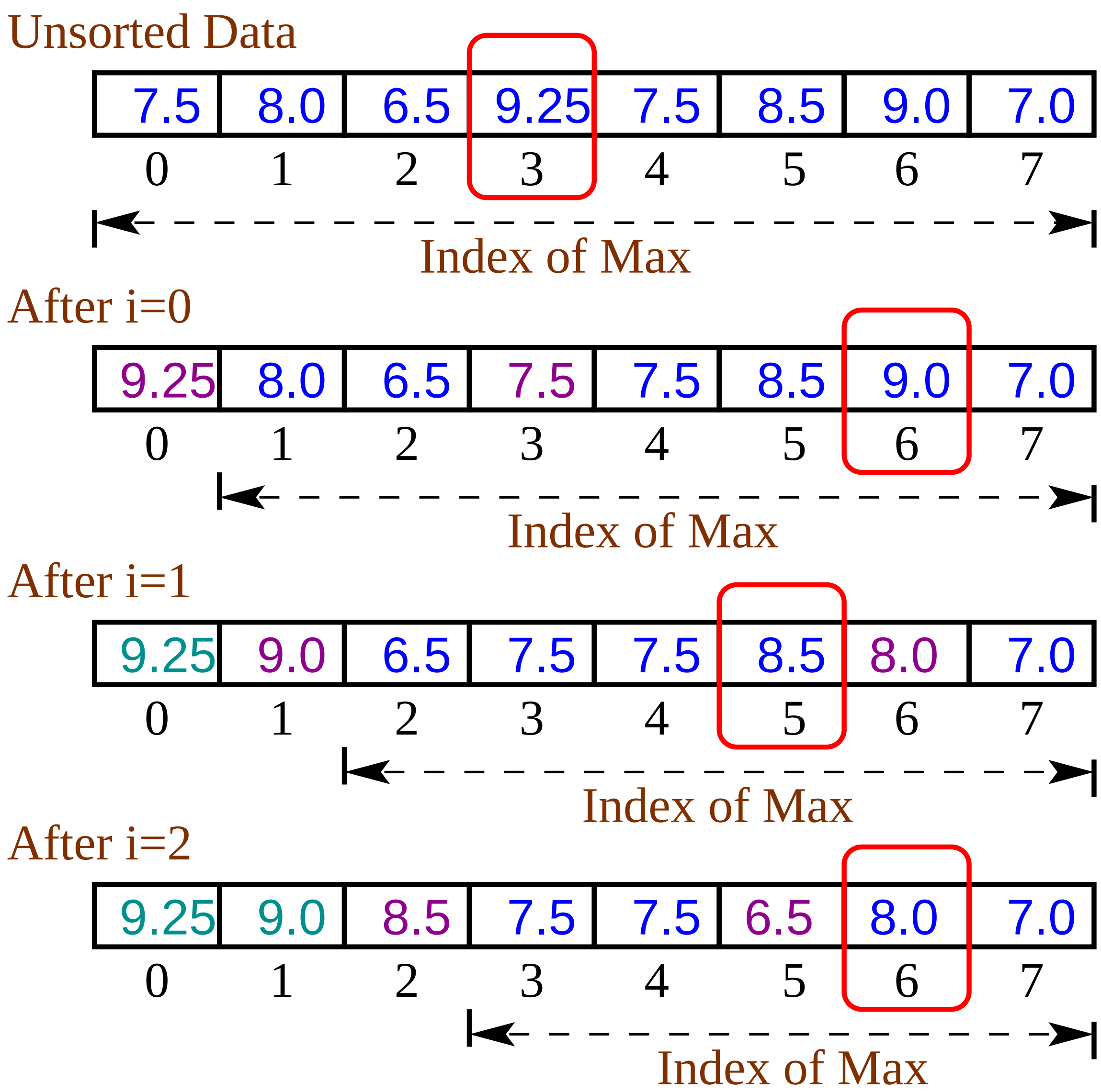
The data is stored in an 1-D array and we sort them in non-ascending order. Let the number of data be n

for i $\leftarrow 0$ to $n-2$ do

  maxIndex $\leftarrow$ indexOfMax({a[i], $\cdots$, a[n-1]})

  a[i] $\leftrightarrow$ a[maxIndex] #Exchange

endFor


Unsorted Data
7.5 8.0 6.5 9.25 7.5 8.5 9.0 7.0
0 1 2 3 4 5 6 7
Index of Max
After i=0
9.25 8.0 6.5 7.5 7.5 8.5 9.0 7.0
0 1 2 3 4 5 6 7
Index of Max
After i=1
9.25 9.0 6.5 7.5 7.5 8.5 8.0 7.0
0 1 2 3 4 5 6 7
Index of Max
After i=2
9.25 9.0 8.5 7.5 7.5 6.5 8.0 7.0
0 1 2 3 4 5 6 7
Index of Max

C Program

```
int indexOfMax(double cgpa[],int low,int high) {
    double max = cgpa[low];
    int indMax = low, i;
    for(i=low+1; i<=high; ++i)
        if(cgpa[i] > max) {
            max = cgpa[i];
            indMax = i;
        }
    return indMax;
} // selSort.c
#define EXCH(X,Y,Z) ((Z)=(X), (X)=(Y), (Y)=(Z))
void selectionSort(double cgpa[], int noOfStdnt) {
     int i ;
```

```
    for(i = 0; i < noOfStdnt - 1; ++i) {
        int max = indexOfMax(cgpa, i, noOfStdnt-1);
        double temp ;
        EXCH(cgpa[i], cgpa[max], temp);
    }
} // selSort.c
```

Measure of Goodness of an Algorithm

- Correctness of the algorithm.
- Increase of execution time with the increase in the size of input.
- Increase of the requirement of extra space (other than the space required by the input data) with the increase in the size of input.
- Difficulty in coding the algorithm, $\cdots$

Execution Time

The execution time of a program (algorithm) depends on many factors e.g. the machine parameters (clock speed, instruction set, memory access time etc.), the code generated by the compiler, other processes sharing time on the OS, data set, data structure and encoding of the algorithm etc.

Execution Time Abstraction

It is necessary to get an abstract view of the execution time, to compare different algorithms, that essentially depends on the algorithm and the data structure.

Execution of `selectionSort()`

If there are n data, the for-loop in the function `selectionSort()`, is executed $(n-1)$ times $([i : 0 \cdots (n-2)])$, so the number of assignments, array acess, comparison and call to `indexOfMax()` are all approximately proportional to the data count, n[a].

[a] It is difficult to get the exact count of these operations from the high-level coding of the algorithm.

Execution of `indexOfMax()`

For each value of i in the for-loop of selectionSort() there is a call to `indexOfMax()` (`low` $\leftarrow$ `i`)

- Corresponding to each call the for-loop of `indexOfMax()` is executed `high - low - 1` $= (n-1) - i - 1 = n - i - 2$ times.
- The total number of comparisons for each i inside `indexOfMax()`, are $2(n-i-2)+1 = 2n-2i-3$.

- The number of assignments are $3n - 3i - 6 + 3 = 3n - 3i - 3$.
- And the number of array access are $2n - 2i - 2 + 1 = 2n - 2i - 1$.

Execution Time

- The total number of comparisons both in `selectionSort()` and `indexOfMax()` is $n + \sum_{i=0}^{n-2}(2n - 2i - 3) = n + 2n(n-1) - (n-1)(n-2) - 3(n-1) = n^2 - n + 1$.
- Similarly we can calculate total number of assignments and array access.

Execution Time

Different operations have different costs, that makes the execution time a complex function of n. But for a large value of n (data count), the number of each operation is approximately proportional to n^2.

Execution Time

If we assume identical costs for each of these operations (abstraction), the running time of selection sort is approximately proportional to n^2[a].

This roughly means that the running time of selection sort algorithm will be four times if the data count is doubled.

[a] n is the number of data to be sorted.

Time Complexity

We say that the running time or the time complexity of selection sort is of order n^2, $\Theta(n^2)$. We shall define this notion precisely.

Space Complexity

The extra space requirement for selection sort does not depend on n for this implementation. What ever be the value of n we are using only half a dozen of extra variables e.g. `cgpa[]`, `low`, `high`, `max`, ...

Selection Sort with Recursion

```
int indexOfMax(double cgpa[],int low,int high) {
    int max ;

    if(low == high) return low;
    max = indexOfMax(cgpa,low+1,high);
    if(cgpa[low] > cgpa[max]) return low ;
    return max;
} // selSort.c
#define EXCH(X,Y,Z) ((Z)=(X), (X)=(Y), (Y)=(Z))
void selectionSort(double cgpa[], int noOfStdnt) {
     int i ;

     for(i = 0; i < noOfStdnt - 1; ++i) {
         int max = indexOfMax(cgpa, i, noOfStdnt-1);
```

```
        double temp ;
        EXCH(cgpa[i], cgpa[max], temp);
    }
} // selSort2.c
```

Space Complexity

In the recursive version, the volume of extra space depends on the number of data elements due to recursive calls. It is roughly proportional to n.

Selection Sort: a Single Function

```
#define EXCH(X,Y,Z) ((Z)=(X), (X)=(Y), (Y)=(Z))
void selectionSort(double cgpa[], int noOfStdnt) {
     int i ;

     for(i = 0; i < noOfStdnt - 1; ++i) {
         int max, j ;
         double temp ;

         temp = cgpa[i] ;
         max = i ;
         for(j = i+1; j < noOfStdnt; ++j)
         if(cgpa[j] > temp) {
            temp = cgpa[j] ;
            max = j ;
```

```
        }
        EXCH(cgpa[i], cgpa[max], temp);
    }
} // selSort1.c
```

Note

We shall introduce the notation of upper bound (O), lower bound (Ω) and order (Θ) of non-decreasing positive real-valued functions[a]. These notations are useful to express the running time and space usages of algorithms.

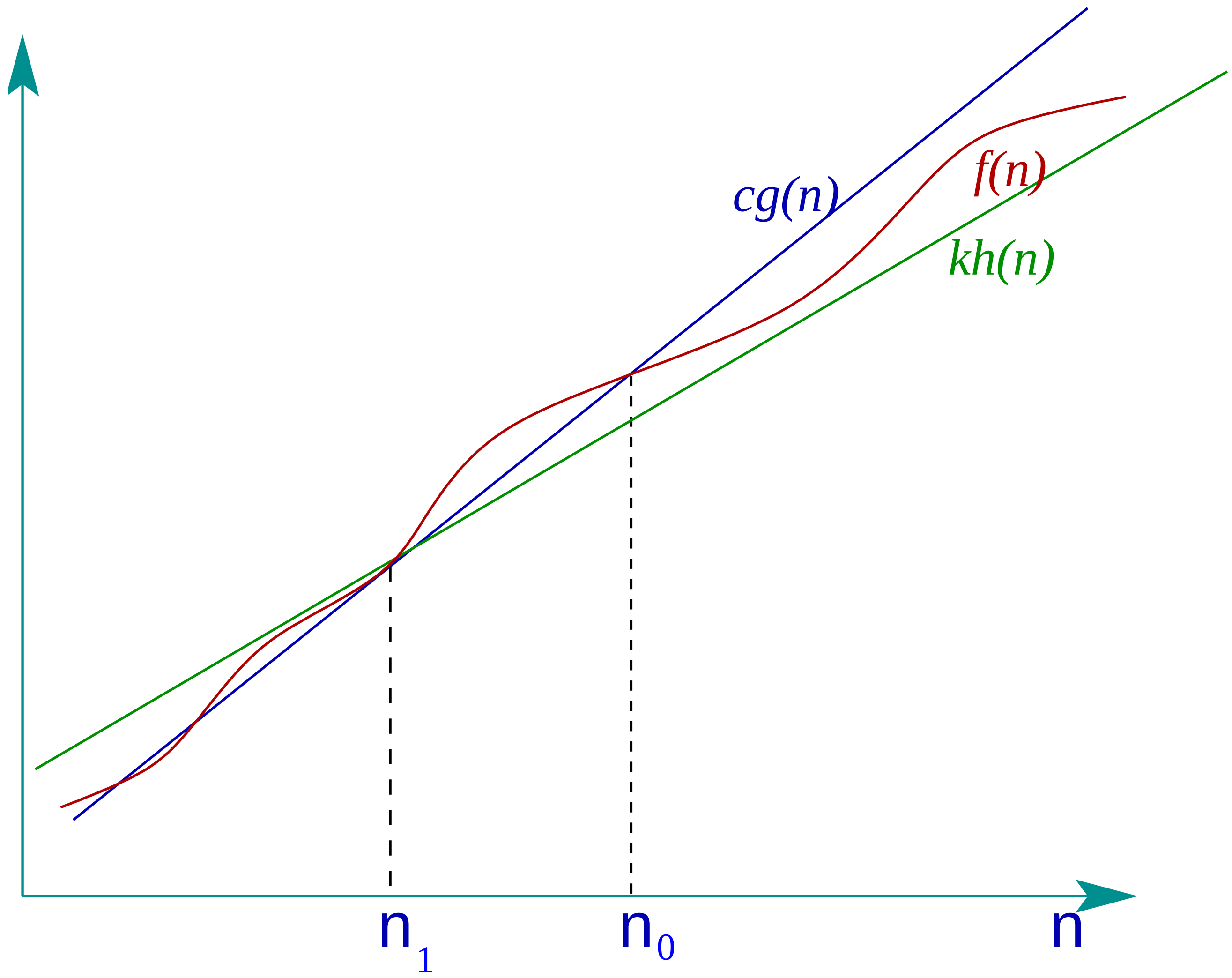
[a] Our actual domain is $\mathbb{N}$, but we shall take it as positive reals while drawing the graph.

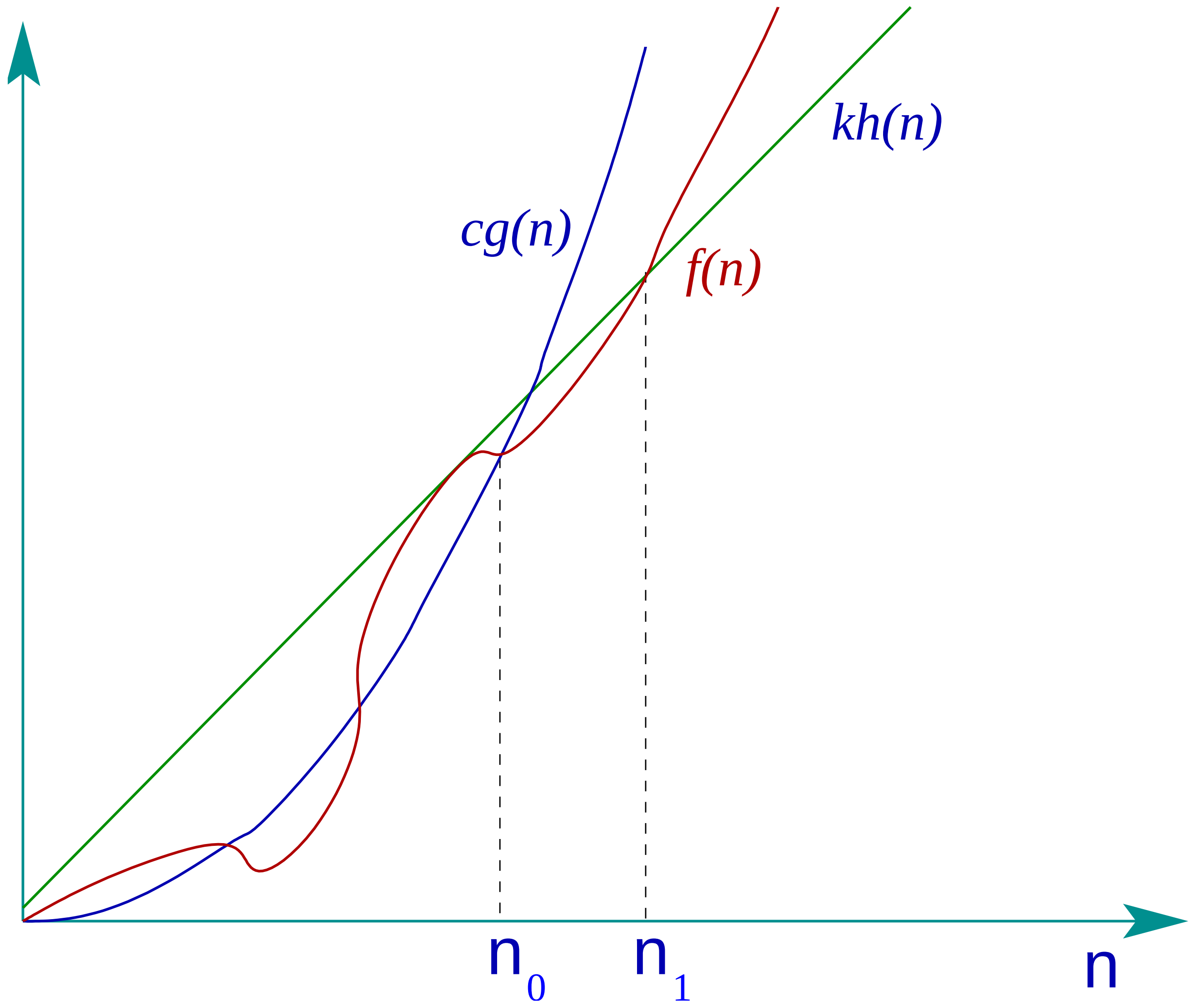
Big O: Asymptotic Upper Bound

Consider two functions $f, g : \mathbb{N} \longrightarrow \mathbb{R}^{+}$. We say that $f(n)$ is $O(g(n))$ or $f(n) \in O(g(n))$ or $f(n) = O(g(n))$, if there are two positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$, for all $n \geq n_0$.
$g(n)$ is called an upper bound of $f(n)$.

Ω: Asymptotic Lower Bound

Consider two functions $f, h : \mathbb{N} \longrightarrow \mathbb{R}^+$. We say that $f(n)$ is $\Omega(h(n))$ ($f(n) \in \Omega(h(n))$ or $f(n) = \Omega(h(n))$), if there are two positive constants c and n_0 such that $0 \leq ch(n) \leq f(n)$, for all $n \geq n_0$.
$h(n)$ is called a lower bound of $f(n)$.


cg(n)
f(n)
kh(n)
n1
n0
n


kh(n)
cg(n)
f(n)
n0
n1
n

Examples

- $n^2 + n + 5 = O(n^2)$: It is easy to verify that $2n^2 \geq n^2 + n + 5$ for all $n \geq 3$ i.e. $c = 2$ and $n_0 = 3$.
- $n^2 + n + 5 \neq O(n)$ and
- $n^2 + n + 5 = O(n^3), O(n^4), \cdots$.

Examples

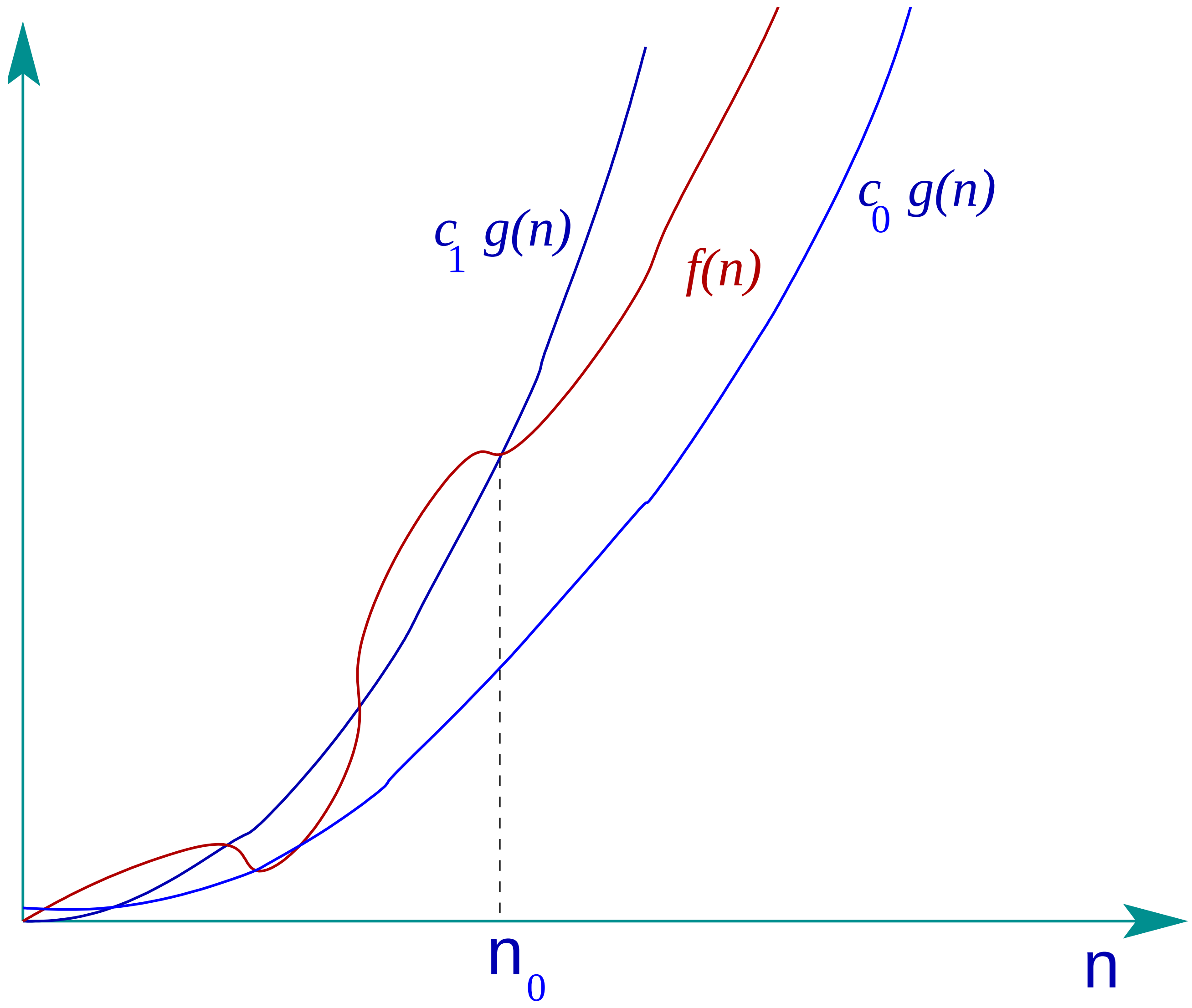
- $n^2 + n + 5 = \Omega(n^2)$: It is easy to verify that $\frac{n^2}{2} < n^2 + n + 5$ for all n i.e. $c = 0.5$ and $n_0 = 0$.
- $n^2 + n + 5 = \Omega(n), \Omega(n \log n), \Omega(\log n)$ and
- $n^2 + n + 5 \neq \Omega(n^3), \Omega(n^4), \cdots$.

Big Θ: Asymptotically Tight Bound

Consider two functions $f, g : \mathbb{N} \longrightarrow \mathbb{R}^+$. We say $f(n) = \Theta(g(n))$, if there are three positive constants c_1, c_2 and n_0 such that

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n),$$

for all $n \geq n_0$.
$g(n)$ is an asymptotically tight bound of $f(n)$ or $g(n)$ is of order $f(n)$.
$f(n) = \Theta(g(n))$ is equivalent to $f(n) = O(g(n))$ and $g(n) = O(f(n))$.


$c_1\ g(n)$
$f(n)$
$c_0\ g(n)$
n_0
n

Examples

- $g(n) = n^2 + n + 5 = \Theta(n^2)$, take $c_1 = \frac{1}{3}$, $c_2 = 1$ and $n_0 = 2$.

n	0	1	2	3	$\cdots$
$\frac{1}{3}g(n)$	$\frac{5}{3}$	$\frac{7}{3}$	$\frac{11}{3}$	$\frac{17}{3}$	$\cdots$
n^2	0	1	4	9	$\cdots$

- But $n^2 + n + 5 \neq \Theta(n^3), \Theta(n), \cdots$.

Selection Sort

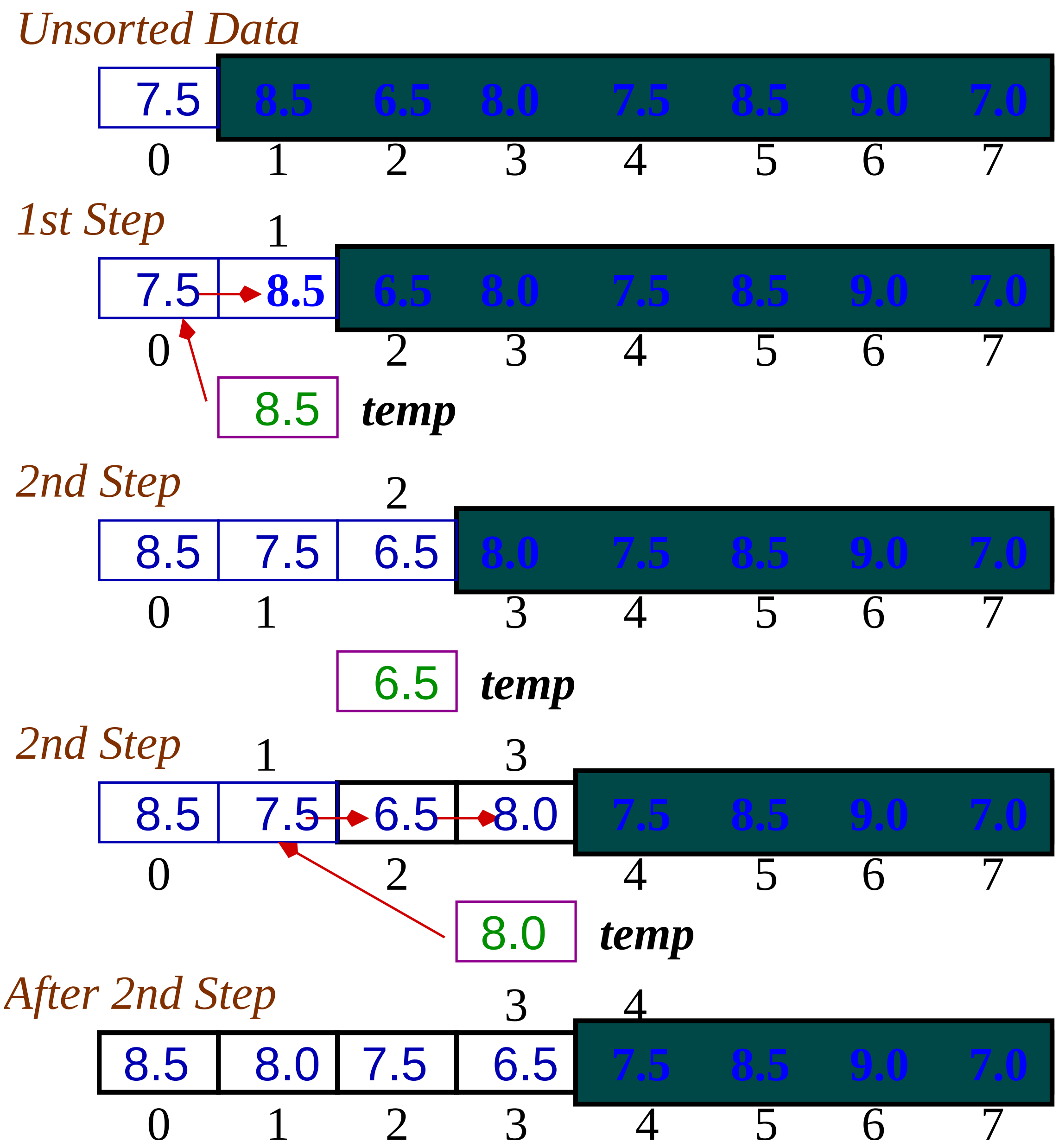
Running time of selection sort is $\Theta(n^2)$ and the space requirement is $\Theta(1)$ (no-recursive), where n is the number of data to sort.

Note

Let n be the size of the input. The worst case running time of an algorithm is

- $\Theta(n)$ implies that it takes almost double the time if the input size is doubled;
- $\Theta(n^2)$ implies that it takes almost four times the time if the input is doubled;
- $\Theta(\log n)$ implies that it takes a constant amount of extra time if the input is doubled;

Insertion Sort


Unsorted Data
7.5 8.5 6.5 8.0 7.5 8.5 9.0 7.0
0 1 2 3 4 5 6 7
1st Step
1
7.5 8.5 6.5 8.0 7.5 8.5 9.0 7.0
0 2 3 4 5 6 7
8.5 temp
2nd Step
2
8.5 7.5 6.5 8.0 7.5 8.5 9.0 7.0
0 1 3 4 5 6 7
6.5 temp
2nd Step
1 3
8.5 7.5 6.5 8.0 7.5 8.5 9.0 7.0
0 2 4 5 6 7
8.0 temp
After 2nd Step
3 4
8.5 8.0 7.5 6.5 7.5 8.5 9.0 7.0
0 1 2 3 4 5 6 7

Insertion Sort Algorithm

```
for i ← 1 to noOfStdnt −1 do
    temp ← cgpa[i]
    for j ← i-1 downto 0 do
        if cgpa[j] < temp
            cgpa[j+1] ← cgpa[j]
        else go out of loop
    endFor
    cgpa[j+1] ← temp
endFor
```

C Program

```
void insertionSort(double cgpa[], int noOfStdnt){
     int i, j ;

     for(i=1; i < noOfStdnt; ++i) {
          double temp = cgpa[i] ;
          for(j = i-1; j >= 0; --j) {
               if(cgpa[j]<temp) cgpa[j+1]=cgpa[j];
               else break ;
          }
          cgpa[j+1] = temp ;
     }
} // insertionSort.c
```

Execution Time

Let n be the number of data. The outer for-loop will always be executed $n-1$ times. The number of times the inner for-loop is executed depends on data. It is entered at least once but the maximum number of execution may be i.

Execution Time

If for most of the values of i, $0 \leq i < n$, the inner loop is executed near the minimum value (for an almost sorted data), the execution time will be almost proportional to n i.e. linear in n.

Worst Case Execution Time

But in the worst case, The inner for-loop will be executed

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)$$

times. So the running time of insertion sort is $O(n^2)$, the worst case running is $\Theta(n^2)$, the best case running time is $\Theta(n)$.

Extra Space for Computation

The extra space required for the computation of insertion sort does not depend on number of data. It is $\Theta(1)$ (so it is also $O(1)$ and $\Omega(1)$).

Bubble Sort

i = 0

	0	1	2	3	4	5	6	7
j = 7	53	22	49	15	21	82	37	16
							no−exchange	
j = 6	53	22	49	15	21	82	37	16
						no−exchange		
j = 5	53	22	49	15	21	82	37	16
					exchange			
j = 4	53	22	49	15	82	21	37	16
				exchange				
j = 3	53	22	49	82	15	21	37	16
			exchange					
j = 2	53	22	82	49	15	21	37	16
		exchange						
j = 1	53	82	22	49	15	21	37	16
	exchange							
	82	53	22	49	15	21	37	16

Bubble Sort Algorithm

```
for i ← 0 to noOfStdnt −2 do
    exchange = NO
    for j ← noOfStdnt −1 downto i +1 do
        if (cgpa[j-1] < cgpa[j])
            cgpa[j-1] ↔ cgpa[j] # Exchange
            exchange = YES
    endFor
    if (exchange == NO) break
endFor
```

C Program

```
#define EXCHANGE 0
#define NOEXCHANGE 1
#define EXCH(X,Y,Z) ((Z)=(X), (X)=(Y), (Y)=(Z))
void bubbleSort(double cgpa[], int noOfStdnt) {
     int i, j, exchange, temp ;

     for(i=0; i < noOfStdnt - 1; ++i) {
         exchange = NOEXCHANGE ;
         for(j = noOfStdnt - 1; j > i; --j)
             if(cgpa[j-1] < cgpa[j]) {
                EXCH(cgpa[j-1], cgpa[j], temp);
                exchange = EXCHANGE ;
             }
             if(exchange) break ;
```

```
        }
} // bubbleSort.c
```

Execution Time

The number of times the outer for-loop is executed depends on the input data, as there is a conditional `break`. If the data is sorted in the desired order, there is no exchange, and in the best case the outer loop is executed only once. This makes the best running time of bubble sort approximately proportional to n.

Execution Time and Space

But in the worst case the outer loop is executed $n - 1$ times. The inner loop is executed $(n - 1) - i$ times for every value of i. So in the worst case, the total number of times the inner loop is executed is

$$\sum_{i=0}^{n-1}(n-1)-i = \frac{n(n-1)}{2} = \Theta(n^2)$$

times.

Worst Case Complexity

- The running time of bubble sort (worst case time complexity) is $O(n^2)$ (quadratic in n).
- The extra storage requirement does not depend on the size of data and the space complexity is $\Theta(1)$.