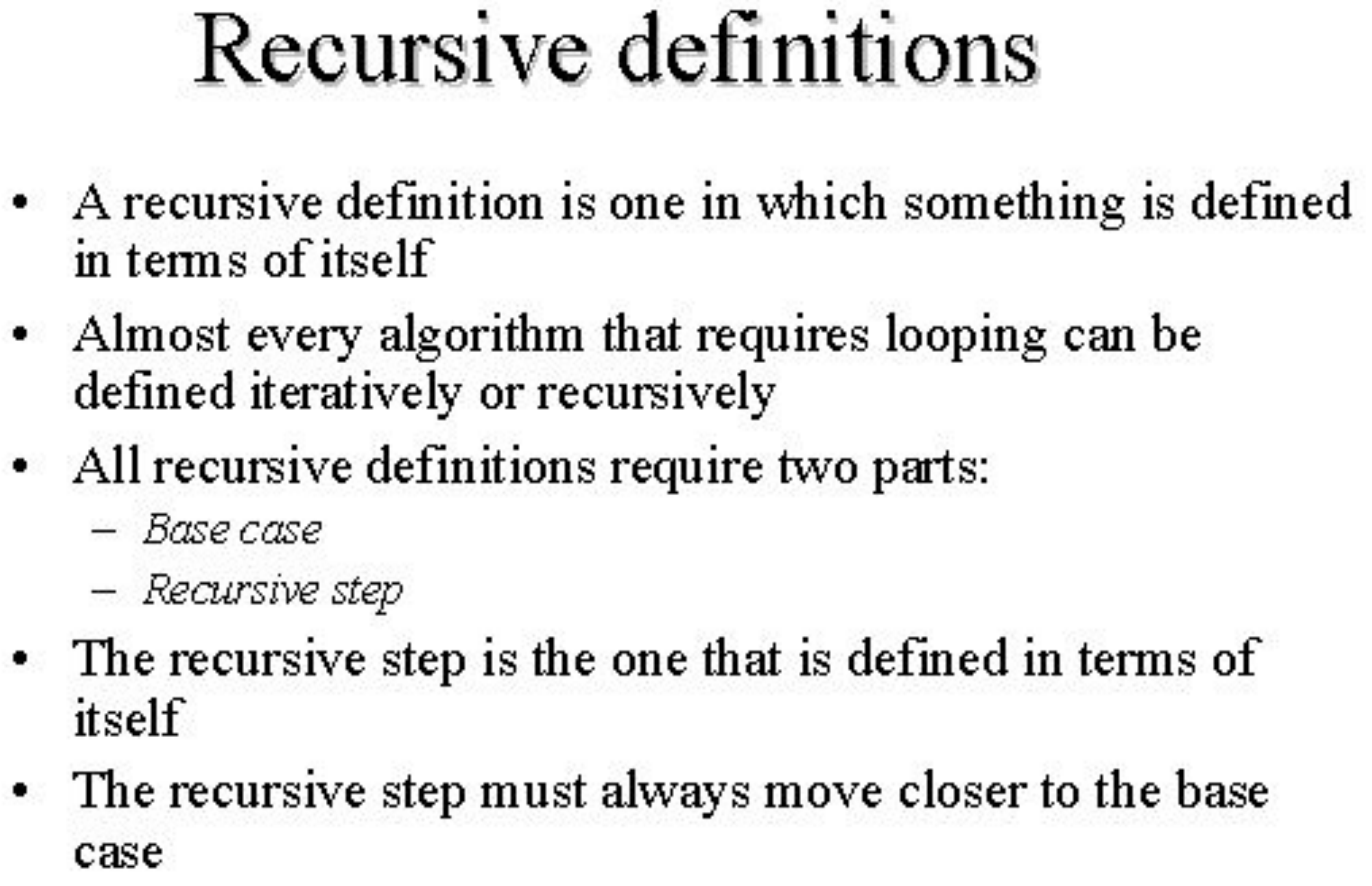
Chapter 1

Introduction

Recursive and Non recursive algorithms



Recursive definitions

- A recursive definition is one in which something is defined in terms of itself
- Almost every algorithm that requires looping can be defined iteratively or recursively
- All recursive definitions require two parts:
  - *Base case*
  - *Recursive step*
- The recursive step is the one that is defined in terms of itself
- The recursive step must always move closer to the base case

- Factorial
  - $n! = n * (n-1)!$

```
int fact(int n) {
  if (n <= 1) return 1;
  else return n * fact(n-1);
} // fact
```

- In general, we can define the factorial function in the following way:

$$\text{Factorial}(n) = \begin{bmatrix} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 & \text{if } n > 0 \end{bmatrix}$$

$$\begin{aligned} 1! &= 1 \\ 2! &= 1 \times 2 = 2 \\ 3! &= 1 \times 2 \times 3 = 6 \\ 4! &= 1 \times 2 \times 3 \times 4 = 24 \\ 5! &= 1 \times 2 \times 3 \times 4 \times 5 = 120 \end{aligned}$$

Iterative Definition

- This is an ***iterative*** definition of the factorial function.
- It is iterative because the definition only contains the algorithm parameters and not the algorithm itself.
- This will be easier to see after defining the recursive implementation.

Recursive Definition

- We can also define the factorial function in the following way:

$$\text{Factorial}(n) = \left[\begin{array}{ll} 1 & \text{if } n = 0 \\ n \times (\text{Factorial}(n-1)) & \text{if } n > 0 \end{array}\right]$$

Iterative vs. Recursive

- **Iterative**

factorial(n) = $\begin{cases} 1 & \text{if} \\ n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 & \text{if } n>0 \end{cases}$

Function does NOT calls itself

- **Recursive**

Function calls itself

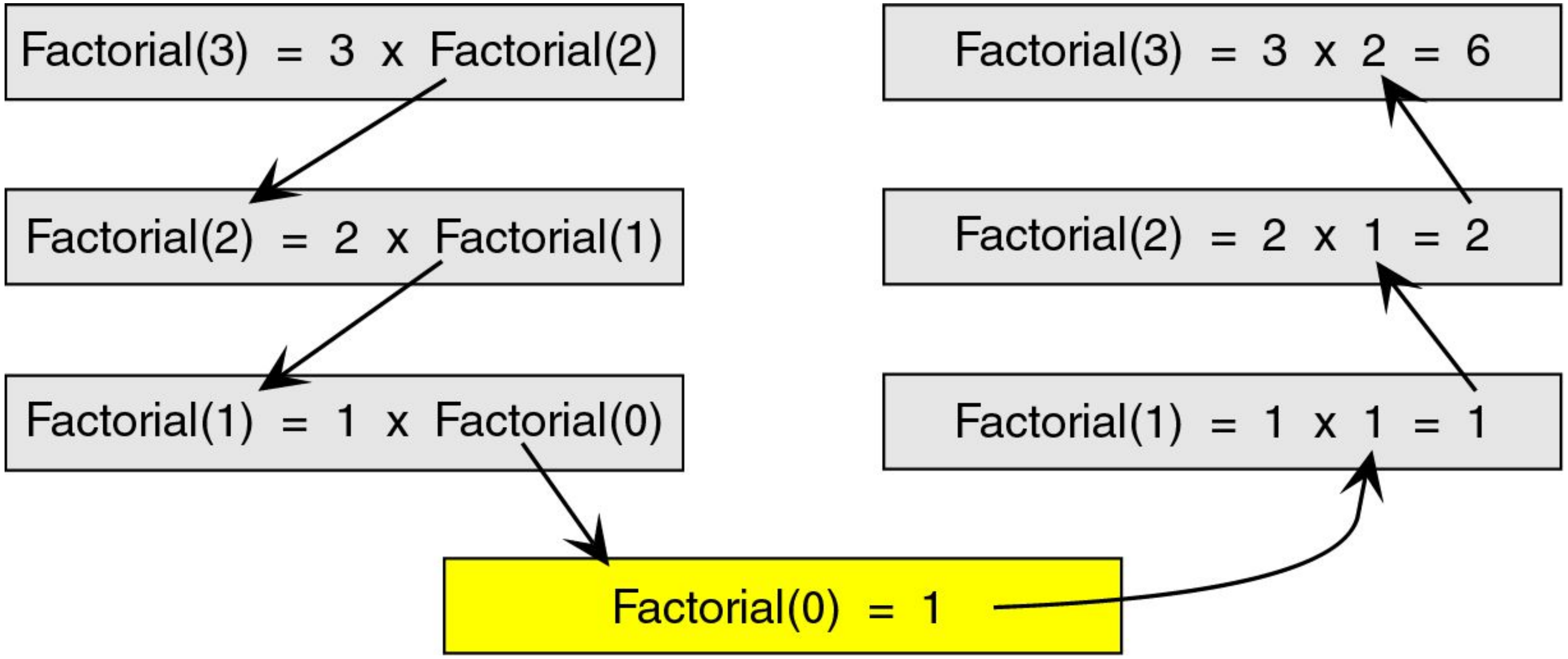
factorial(n) = $\begin{cases} 1 & \text{if } n=0 \\ n \times \text{factorial}(n-1) & \text{if } n>0 \end{cases}$

Iteration vs. recursion

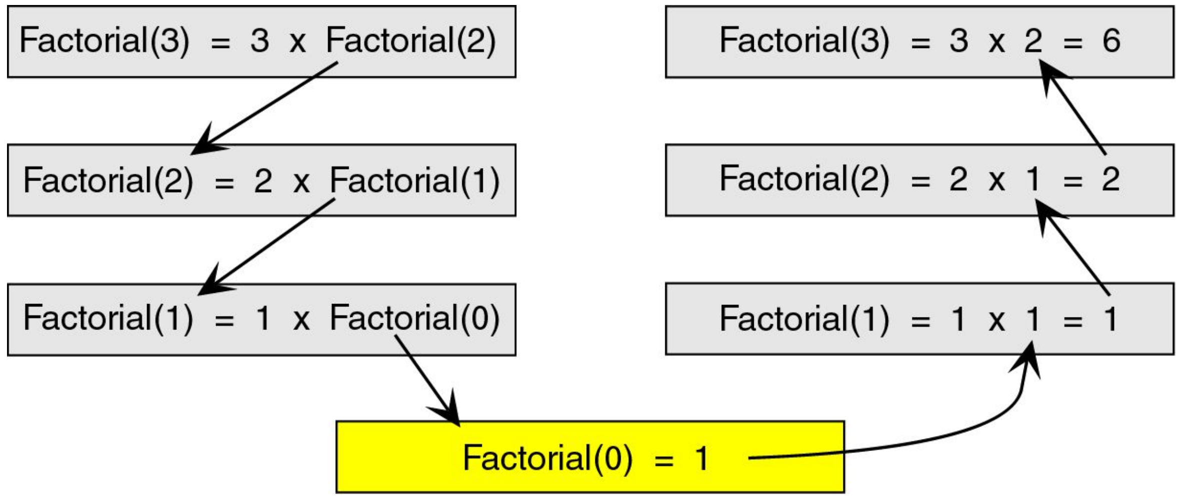
- Some things (e.g. reading from a file) are easier to implement iteratively
- Other things (e.g. mergesort) are easier to implement recursively
- Others are just as easy both ways
- When there is no real benefit to the programmer to choose recursion, iteration is the more efficient choice
- It can be proved that two methods performing the same task, one implementing an iteration algorithm and one implementing a recursive version, are equivalent

Recursion

- To see how the recursion works, let’s break down the factorial function to solve factorial(3)


Factorial(3) = 3 x Factorial(2)
Factorial(2) = 2 x Factorial(1)
Factorial(1) = 1 x Factorial(0)
Factorial(0) = 1
Factorial(1) = 1 x 1 = 1
Factorial(2) = 2 x 1 = 2
Factorial(3) = 3 x 2 = 6

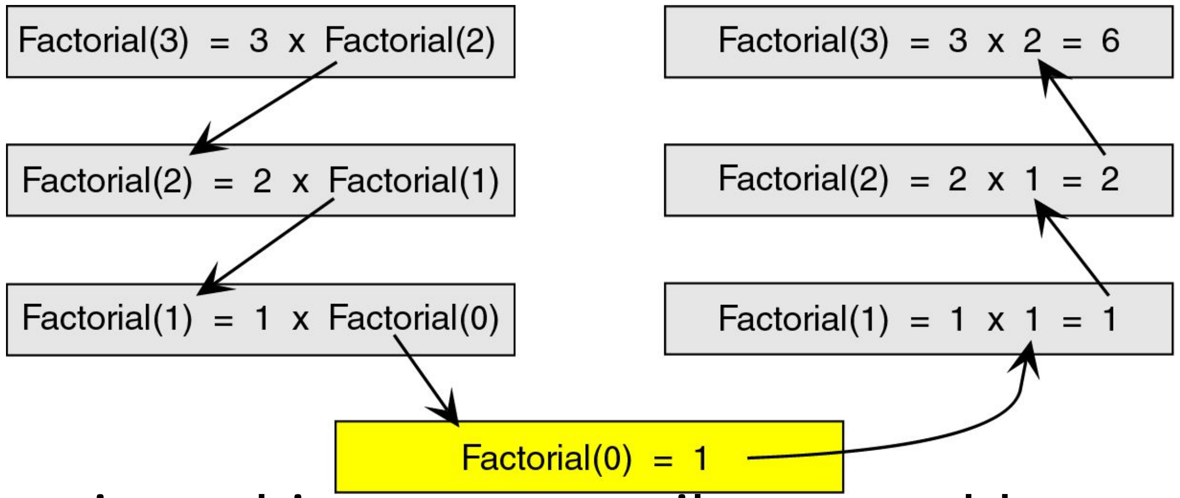
Breakdown



Factorial(3) = 3 x Factorial(2)
Factorial(2) = 2 x Factorial(1)
Factorial(1) = 1 x Factorial(0)
Factorial(0) = 1
Factorial(1) = 1 x 1 = 1
Factorial(2) = 2 x 1 = 2
Factorial(3) = 3 x 2 = 6

- Here, we see that we start at the top level, factorial(3), and simplify the problem into 3 x factorial(2).
- Now, we have a slightly less complicated problem in factorial(2), and we simplify this problem into 2 x factorial(1).

Breakdown


Factorial(3) = 3 x Factorial(2)
Factorial(2) = 2 x Factorial(1)
Factorial(1) = 1 x Factorial(0)
Factorial(0) = 1
Factorial(1) = 1 x 1 = 1
Factorial(2) = 2 x 1 = 2
Factorial(3) = 3 x 2 = 6

- We continue this process until we are able to reach a problem that has a known solution.
- In this case, that known solution is factorial(0) = 1.
- The functions then return in reverse order to complete the solution.

Breakdown

- This known solution is called the ***base case***.
- Every recursive algorithm must have a base case to simplify to.
- Otherwise, the algorithm would run forever (or until the computer ran out of memory).

Breakdown

- The other parts of the algorithm, excluding the base case, are known as the general case.
- For example:

  3 x factorial(2) general case

  2 x factorial(1) general case

  etc ...

Iteration vs. Recursion

- Now that we know the difference between an iterative algorithm and a recursive algorithm, we will develop both an iterative and a recursive algorithm to calculate the factorial of a number.
- We will then compare the 2 algorithms.

Iterative Algorithm

```
factorial(n) {
    i = 1
    factN = 1
    loop (i <= n)
        factN = factN * i
        i = i + 1
    end loop
    return factN
}
```

The iterative solution is very straightforward. We simply loop through all the integers between 1 and n and multiply them together.

Recursive Algorithm

```
factorial(n) {
    if (n = 0)
    return 1
    else
        return n*factorial(n-1)
    end if
}
```

Note how much simpler the code for the recursive version of the algorithm is as compared with the iterative version ☐

we have eliminated the loop and implemented the algorithm with 1 'if' statement.

Another example

- Describe a palindrome recursively (a palindrome is a word has the same spelling both forward and backwards e.g. bob)
  - Base case:

    An empty string is a palindrome

    A string of one character is a palindrome
  - Recursive step

    If S is a palindrome, then aSb is also a palindrome if a=b

How recursion works

- To truly understand how recursion works we need to first explore how any function call works.
- When a program calls a subroutine (function) the current function must suspend its processing.
- The called function then takes over control of the program. When the function is finished, it needs to return to the function that called it.
- The calling function then ‛wakes up’ and continues processing.
- One important point in this interaction is that, unless changed through call-by-reference, all local data in the calling module must remain unchanged.

- Therefore, when a function is called, some information needs to be saved in order to return the calling module back to its original state (i.e., the state it was in before the call).
- We need to save information such as the local variables and the spot in the code to return to after the called function is finished
- To do this we use a stack.
- Before a function is called, all relevant data is stored in a ***stackframe***.
- This stackframe is then pushed onto the system stack.
- After the called function is finished, it simply pops the system stack to return to the original state.

- By using a stack, we can have functions call other functions which can call other functions, etc.
- Because the stack is a first-in, last-out data structure, as the stackframes are popped, the data comes out in the correct order.

- Two criteria are used to judge algorithms:
  (i) time complexity (ii) space complexity.
- Space Complexity of an algorithm is the amount of memory it needs to run to completion.
- Time Complexity of an algorithm is the amount of CPU time it needs to run to completion.
- Space Complexity:
- Memory space S(P) needed by a program P, consists of two components:
  - A fixed part: needed for instruction space (byte code), simple variable space, constants space etc. □ c
  - A variable part: dependent on a particular instance of input and output data. □ S_p(instance)
- S(P) = c + S_p(instance characteristics)

Space complexity example:

```
Algorithm abc (a, b, c)
{
  return a+b+b*c+(a+b-c)/(a+b)+4.0;
}
```

- For every instance 3 computer words required to store variables: a, b, and c. Therefore
- $S_p()$= 3. S(P) = 3.

```
Algorithm Sum(a[ ], n)
{
    s:= 0.0;
    for i = 1 to n do
        s := s + a[i];
    return s;
}
```

- Every instance needs to store array `a[]` & `n`.
  - Space needed to store `n` = 1 word.
  - Space needed to store `a[ ]` = n floating point words (or at least n words)
  - Space needed to store `i` and `s` = 2 words
- $S_p(n) = (n + 3)$. Hence $S(P) = (n + 3)$.

Space Complexity

- Space complexity is a measure of the amount of working storage an algorithm needs. That means how much memory, in the worst case, is needed at any point in the algorithm.

```
int sum(int x, int y, int z)
{ int r = x + y + z;
return r; }.
```

requires 3 units of space for the parameters and 1 for the local variable, and this never changes, so this is O(1).

Time Complexity

- Time required T(P) to run a program P also consists of two components:
  - A fixed part: compile time which is independent of the problem instance c.
  - A variable part: run time which depends on the problem instance t_p(instance)
- T(P) = c + t_p(instance characteristic)
- How to measure T(P)?
  - Measure experimentally, using a “stop watch”
     T(P) obtained in secs, msecs.
  - Count program steps T(P) obtained as a step count.
- Fixed part is usually ignored; only the variable part t_p() is measured.

- What is a <u>program step</u>?
  - a+b+b*c+(a+b)/(a-b) one step;
  - comments zero steps;
  - `while (<expr>) do` step count equal to the number of times <expr> is executed.
  - `for i=<expr> to <expr1> do` step count equal to number of times <expr1> is checked.

	Statements	S/E	Freq.	Total
1	`Algorithm Sum(a[ ],n)`	0	–	0
2	`{`	0	–	0
3	`    S = 0.0;`	1	1	1
4	`    for i=1 to n do`	1	n+1	n+1
5	`        s = s+a[i];`	1	n	n
6	`    return s;`	1	1	1
7	`}`	0	–	0

$\boxed{2n+3}$

	Statements	S/E	Freq.	Total
1	Algorithm Sum(a[ ],n,m)	0	–	0
2	{	0	–	0
3	for i=1 to n do;	1	n+1	n+1
4	for j=1 to m do	1	n(m+1)	n(m+1)
5	s = s+a[i][j];	1	nm	nm
6	return s;	1	1	1
7	}	0	–	0

$\boxed{2nm+2n+2}$

- Example: $f(n) = 10n^2+4n+2$
  is $O(n^2)$ because $10n^2+4n+2 <= 11n^2$
  for all $n >=5$.
- Example: $f(n) = 6*2^n+n^2$ is $O(2^n)$
  because $6*2^n+n^2 <=7*2^n$ for all $n>=4$.
- Algorithms can be: $O(1)$ □ constant;
- $O(\log n)$ □ logarithmic;
- $O(n\log n)$; $O(n)$□ linear;
- $O(n^2)$ □ quadratic;
- $O(n^3)$ □ cubic;
- $O(2^n)$ □ exponential.

Some results

Sum of two functions: If $f(n) = f_1(n) + f_2(n)$, and $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f(n) = O(max(|g_1(n)|, |g_2(n)|))$.

Product of two functions: If $f(n) = f_1(n) * f_2(n)$, and $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f(n) = O(g_1(n) * g_2(n))$.

```
int sum(int a[], int n)
 { int r = 0;
for (int i = 0; i < n; ++i)
{ r += a[i]; }
return r; }
```

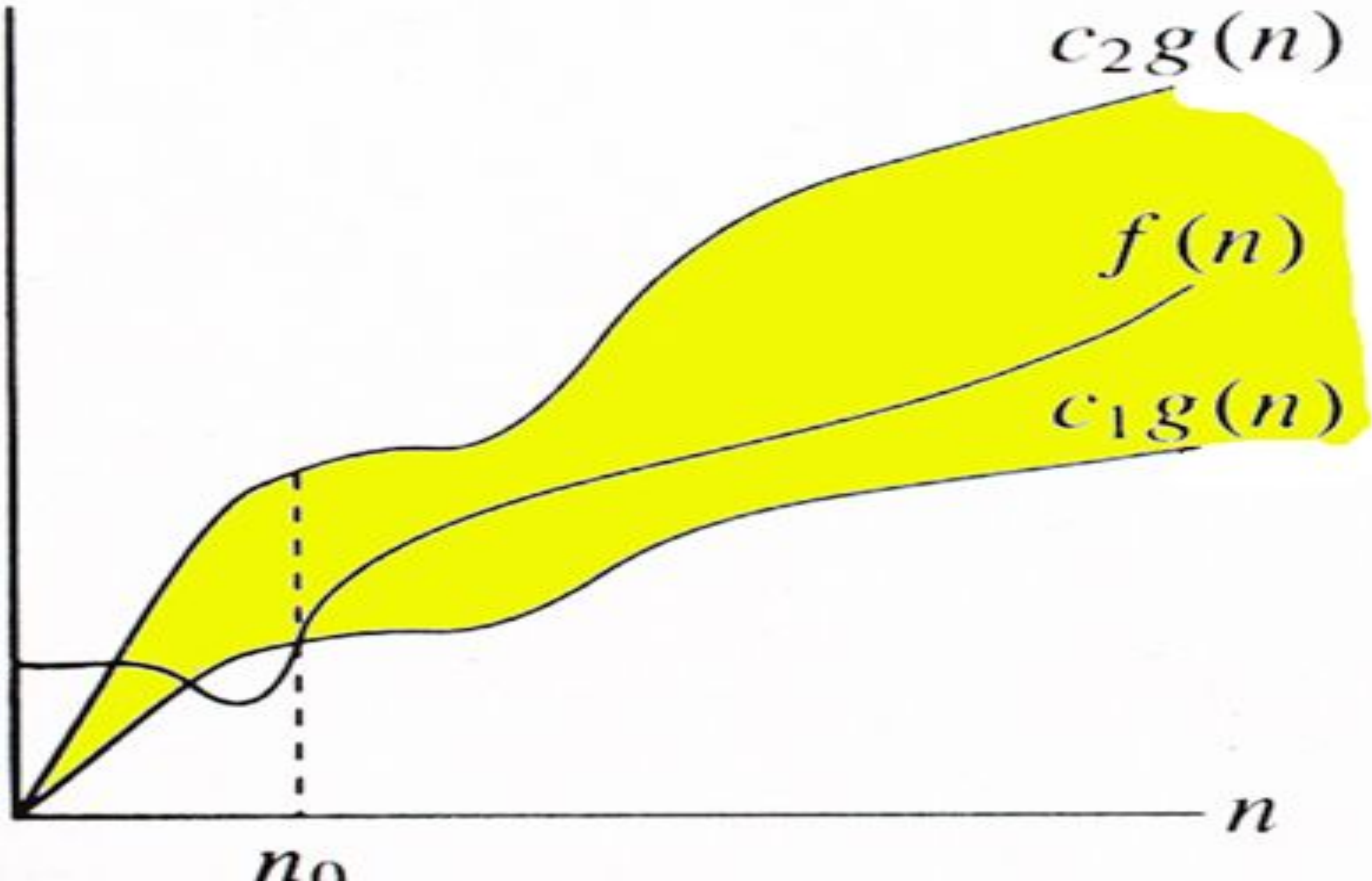
requires N units for a, plus space for n, r and i, so it's O(N).

Asymptotic Notation

- Asymptotic analysis is to have a measure of efficiency of algorithms that doesn’t depend on machine specific constants, and doesn’t require algorithms to be implemented and time taken by programs to be compared.
- Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis.
- The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

Θ Notation:

- The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.
  A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants.
- For example, consider the following expression.
  $3n^3 + 6n^2 + 6000 = \Theta(n^3)$
  Dropping lower order terms is always fine because there will always be a n0 after which $\Theta(n^3)$ beats $\Theta(n^2)$ irrespective of the constants involved.
  For a given function g(n), we denote Θ(g(n)) is following set of functions.



$$f(n) = \Theta(g(n))$$

- Θ(g(n)) = { f(n): there exist positive constants c1, c2 and n_0
  such that 0 <= c1*g(n) <= f(n) <= c2*g(n)
  for all n >= n0
  }
- The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >=n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n_0.

Big O Notation:

- The Big O notation defines an upper bound of an algorithm, it bounds a function only from above.
- Big O of a function gives us 'rate of growth' of the step count function *f(n)*, in terms of a simple function *g(n)*, which is easy to compare.
- For example : Consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is $O(n^2)$. Note that $O(n^2)$ also covers linear time.
- If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:
  1. The worst case time complexity of Insertion Sort is $\Theta(n^2)$.
  2. The best case time complexity of Insertion Sort is $\Theta(n)$.
- The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

- O(g(n)) = { f(n): there exist positive constants c and n_0 such that 0 <= f(n) <= c*g(n)
  for all n >= n0
  }
- Example: f(n) = 3n+2 is O(n) because 3n+2 <= 4n for all n >= 2. c = 4, n_0 = 2. Here g(n) = n.
- F(n) = 3n + 3 = O (n) as 3n+3 <= 4n for all n>=3.
- F(n) = 100n + 6 = O(n) as 100n + 6 <=101n for all n>=6.
- F(n) = 10n^2+4n+2 <=11n^2

- Suppose f(n) = 5n and g(n) = n. • To show that f = O(g), we have to show the existence of a constant C as given earlier. Clearly 5 is such a constant so f(n) = 5 * g(n). • We could choose a larger C such as 6, because the definition states that f(n) must be less than or equal to CS 4407, Algorithms , University College Cork,, Gregory M. Provan C * g(n), but we usually try and find the smallest one. Therefore, a constant C exists (we only need one) and f = O(g).
- In the previous timing analysis, we ended up with T(n) = 4n + 5, and we concluded intuitively that T(n) = O(n) because the running time grows linearly as n grows. Now, however, we can prove it mathematically: To show that f(n) = 4n + 5 = O(n), we need to produce a constant C such that: f(n) <= C * n for all n. If we try C = 4, this doesn't work because 4n + 5 is not less than 4n. We need C to be at least 9 to cover all n. If n = 1, C has to be 9, but C can be smaller CS 4407, Algorithms , University College Cork,, Gregory M. Provan for greater values of n (if n = 100, C can be 5). Since the chosen C must work for all n, we must use 9: 4n + 5 <= 4n + 5n = 9n Since we have produced a constant C that works for all n, we can conclude: T(4n + 5) = O(n)

Say f(n) = n2: We will prove that f(n) ≠ O(n). • To do this, we must show that there cannot exist a constant C that satisfies the big-Oh definition. We will prove this by contradiction. Suppose there is a constant C that works; then, by the definition of big-Oh: n2 ≤ C * n for all n. • Suppose n is any positive real number greater than C, CS 4407, Algorithms , University College Cork,, Gregory M. Provan then: n * n > C * n, or n2 > C * n. So there exists a real number n such that n2 > C * n. This contradicts the supposition, so the supposition is false. There is no C that can work for all n: f(n) ≠ O(n) when f(n) = n2

Suppose f(n) = n2 + 3n - 1. We want to show that f(n) = O(n2). f(n) = n2 + 3n - 1 < n2 + 3n (subtraction makes things smaller so drop it) <= n2 + 3n2 (since n <= n2 for all integers n) = 4n2 Therefore, if C = 4, we have shown that f(n) = O(n2). Notice that all we are doing is finding a simple function that is an upper bound on the original function. Because of this, we could also say that CS 4407, Algorithms , University College Cork,, Gregory M. Provan This would be a much weaker description, but it is still valid. f(n) = O(n3) since (n3) is an upper bound on n2

- **Ω Notation:** Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.
- Ω Notation : can be useful when we have lower bound on time complexity of an algorithm.
- The best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.
- For a given function g(n), we denote by Ω(g(n)) the set of functions.

- Ω (g(n)) = { f(n) : there exist positive constants c and n_0 such that 0 <= c*g(n) <= f(n)

  for all n >= n0

  }.

- **Definition (Theta)** : Consider a function *f*(*n*) which is non-negative for all integers . We say that ``*f*(*n*) is theta *g*(*n*),'' which we write , if and only if *f*(*n*) is *O*(*g*(*n*)) *and f*(*n*) is .
- **Definition (Little Oh)** : Consider a function *f*(*n*) which is non-negative for all integers. We say that ``*f*(*n*) is little oh *g*(*n*),'' which we write *f*(*n*)=*o*(*g*(*n*)), if and only if *f*(*n*) is *O*(*g*(*n*)) but *f*(*n*) is *not*.
- Little oh notation represents a kind of *loose asymptotic bound* in the sense that if we are given that *f*(*n*)=*o*(*g*(*n*)), then we know that *g*(*n*) is an asymptotic upper bound since *f*(*n*)=*O*(*g*(*n*)), but *g*(*n*) is *not* an asymptotic lower bound since *f*(*n*)=*O*(*g*(*n*)) and implies that.
- For example, consider the function *f*(*n*)=*n*+1.
- what *c* we choose, for large enough *n*, . Thus, we may write .

- Suppose f(n) = 5n and g(n) = n. • To show that f = O(g), we have to show the existence of a constant C as given earlier. Clearly 5 is such a constant so f(n) = 5 * g(n). • We could choose a larger C such as 6, because the definition states that f(n) must be less than or equal to C * g(n), but we usually try and find the smallest one. Therefore, a constant C exists (we only need one) and f = O(g).

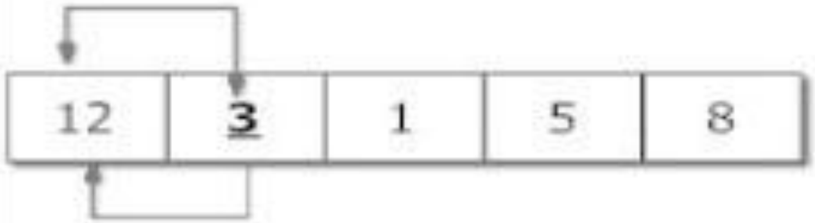
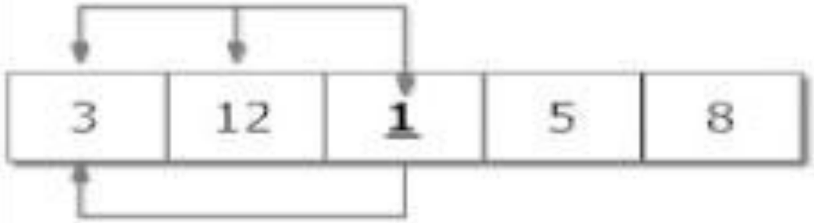
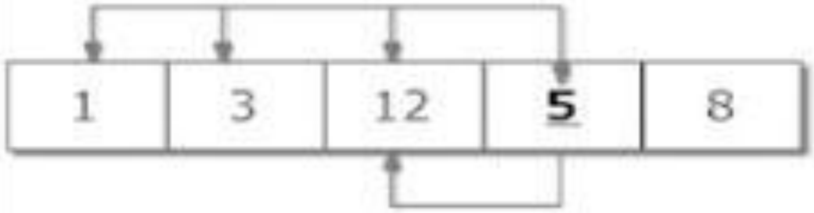
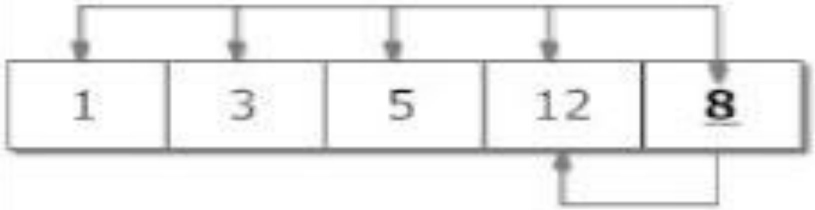
Step 1		**Checking second element of array with element before it and inserting it in proper position. In this case, 3 is inserted in position of 12.**
Step 2		**Checking third element of array with elements before it and inserting it in proper position. In this case, 1 is inserted in position of 3.**
Step 3		**Checking fourth element of array with elements before it and inserting it in proper position. In this case, 5 is inserted in position of 12.**
Step 4		**Checking fifth element of array with elements before it and inserting it in proper position. In this case, 8 is inserted in position of 12.**
	0 | 1 | 3 | 8 | 12	**Sorted Array in Ascending Order**

Figure: Sorting Array in Ascending Order Using Insertion Sort Algorithm

Insertion Sort

```
for(i=1; i<n ; i++)
{   temp = data[i];
    j=i-1;
    while(temp<data[j] && j>=0)
    {data[j+1] = data[j]; --j; }
    data[j+1]=temp; }
    printf("In ascending order: ");
    for(i=0; i<n; i++)
    printf("%d\t",data[i]);
    return 0; }
```

```
**Algorithm** InsertionSort($a, n$)
// Sort the array $a[1:n]$ into nondecreasing order, $n \geq 1$.
{
    **for** $j := 2$ **to** $n$ **do**
    {
        // $a[1:j-1]$ is already sorted.
        $item := a[j]; i := j - 1;$
        **while** $((i \geq 1)$ **and** $(item < a[i]))$ **do**
        {
            $a[i+1] := a[i]; i := i - 1;$
        }
        $a[i+1] := item;$
    }
}
```

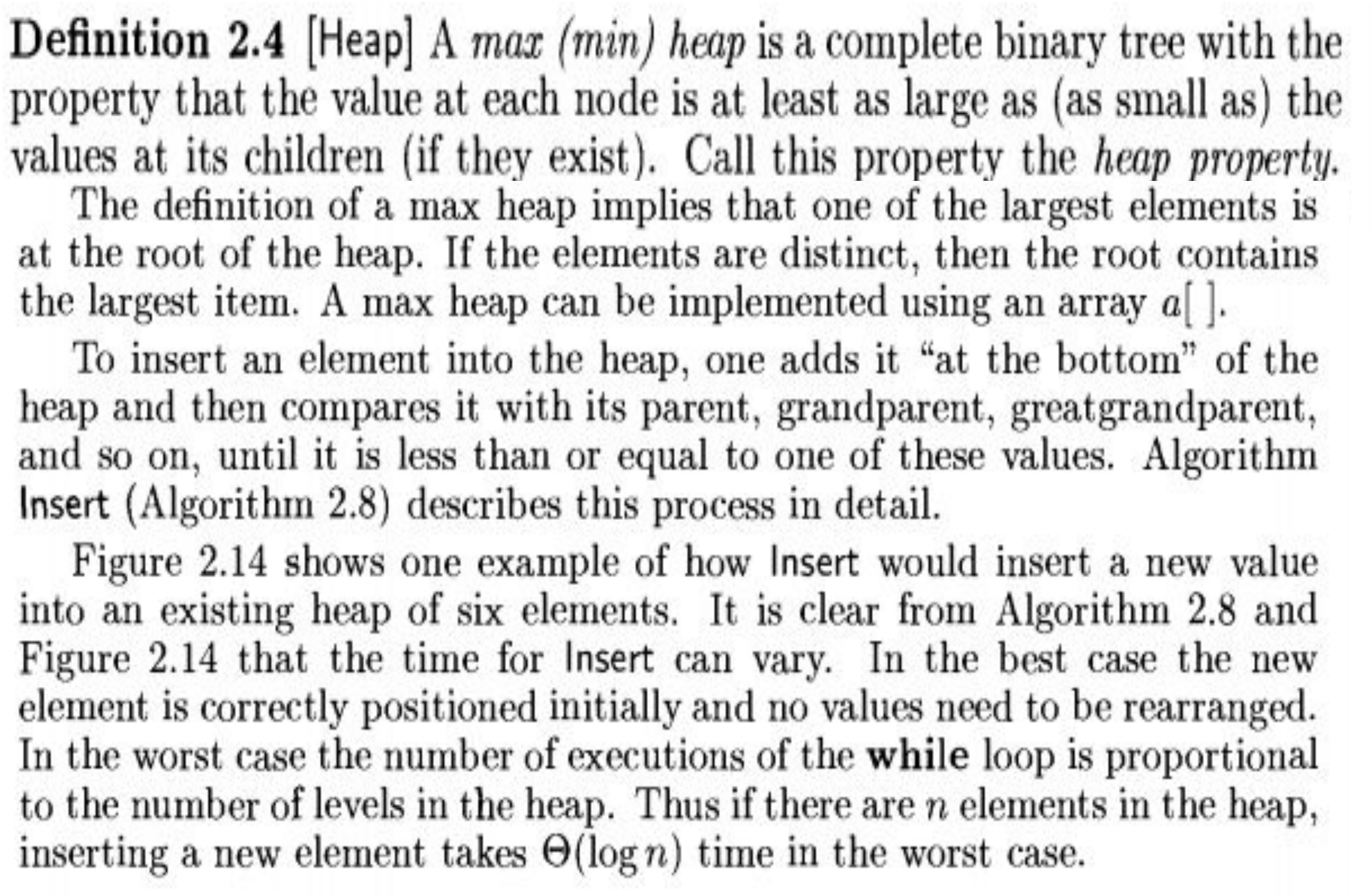
Time Complexity of insertion sort :

If we take a closer look at the insertion sort code, we can notice that every iteration of while loop reduces one inversion. The while loop executes only if $i > j$ and $arr[i] < arr[j]$. Therefore total number of while loop iterations (For all values of i) is same as number of inversions. Therefore overall time complexity of the insertion sort is $O(n + f(n))$ where $f(n)$ is inversion count. If the inversion count is $O(n)$, then the time complexity of insertion sort is $O(n)$. In worst case, there can be $n*(n-1)/2$ inversions. The worst case occurs when the array is sorted in reverse order. So the worst case time complexity of insertion sort is **$O(n^2)$.**

```
Algorithm Insert($a, n$)
{
    // Inserts $a[n]$ into the heap which is stored in $a[1 : n-1]$.
    $i := n$; $item := a[n]$;
    while (($i > 1$) and ($a[\lfloor i/2 \rfloor] < item$)) do
    {
        $a[i] := a[\lfloor i/2 \rfloor]$; $i := \lfloor i/2 \rfloor$;
    }
    $a[i] := item$; return true;
}
```

Algorithm 2.8 Insertion into a heap

Heaps & Heap Sort



Definition 2.4 [Heap] A *max (min) heap* is a complete binary tree with the property that the value at each node is at least as large as (as small as) the values at its children (if they exist). Call this property the *heap property*.

The definition of a max heap implies that one of the largest elements is at the root of the heap. If the elements are distinct, then the root contains the largest item. A max heap can be implemented using an array $a[\,]$.

To insert an element into the heap, one adds it "at the bottom" of the heap and then compares it with its parent, grandparent, greatgrandparent, and so on, until it is less than or equal to one of these values. Algorithm Insert (Algorithm 2.8) describes this process in detail.

Figure 2.14 shows one example of how Insert would insert a new value into an existing heap of six elements. It is clear from Algorithm 2.8 and Figure 2.14 that the time for Insert can vary. In the best case the new element is correctly positioned initially and no values need to be rearranged. In the worst case the number of executions of the **while** loop is proportional to the number of levels in the heap. Thus if there are n elements in the heap, inserting a new element takes $\Theta(\log n)$ time in the worst case.

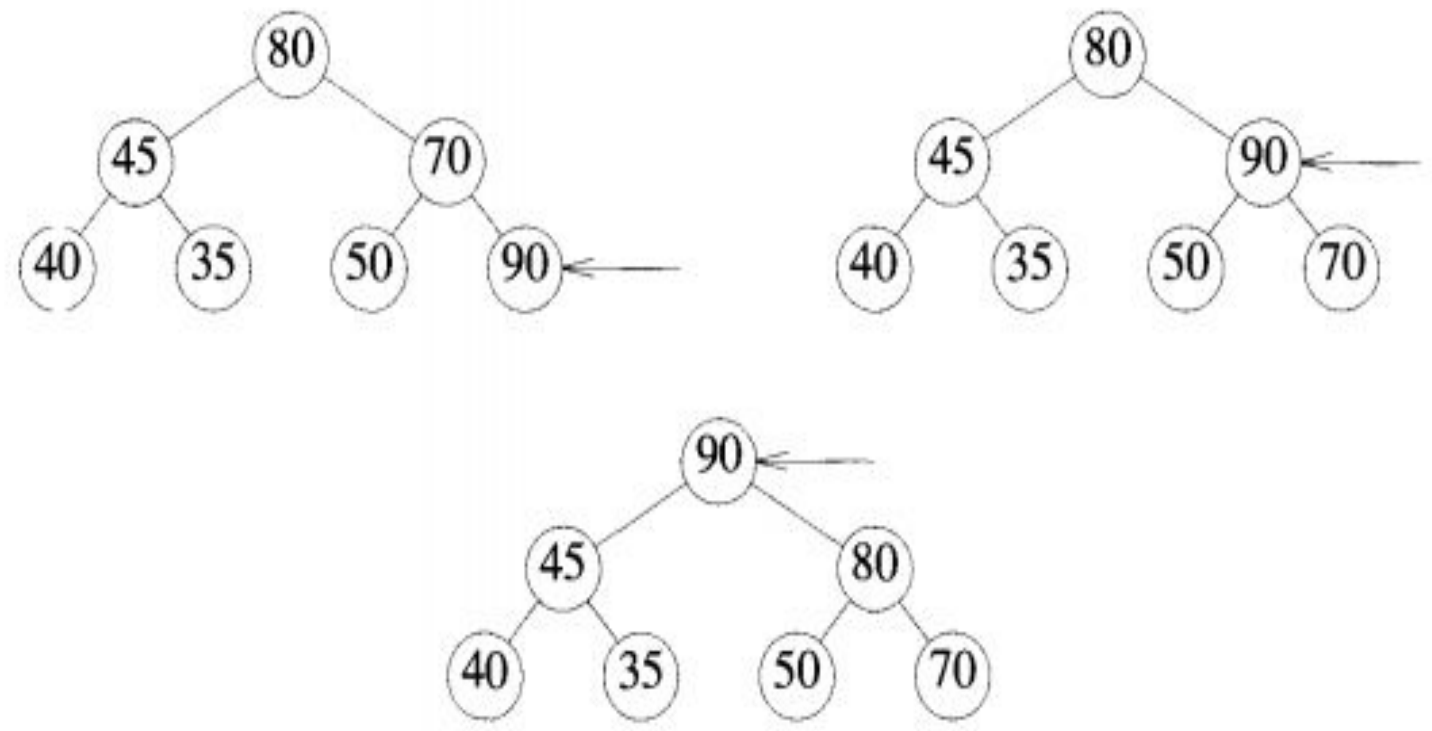
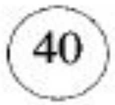


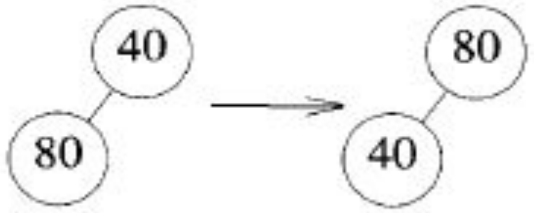
Figure 2.14 Action of Insert inserting 90 into an existing heap

To delete the maximum key from the max heap, we use an algorithm called Adjust. Adjust takes as input the array $a[\,]$ and the integers i and n. It regards $a[1:n]$ as a complete binary tree. If the subtrees rooted at $2i$ and $2i+1$ are already max heaps, then Adjust will rearrange elements of $a[\,]$ such that the tree rooted at i is also a max heap. The maximum element from the max heap $a[1:n]$ can be deleted by deleting the root of the corresponding complete binary tree. The last element of the array, that is, $a[n]$, is copied to the root, and finally we call Adjust$(a, 1, n-1)$. Both Adjust and DelMax are described in Algorithm 2.9.

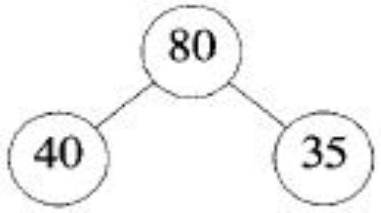
```
Algorithm Adjust($a, i, n$)
// The complete binary trees with roots $2i$ and $2i+1$ are
// combined with node $i$ to form a heap rooted at $i$. No
// node has an address greater than $n$ or less than 1.
{
    $j := 2i$; $item := a[i]$;
    while ($j \leq n$) do
    {
        if (($j < n$) and ($a[j] < a[j+1]$)) then $j := j+1$;
            // Compare left and right child
            // and let $j$ be the larger child.
        if ($item \geq a[j]$) then break;
            // A position for $item$ is found.
        $a[\lfloor j/2 \rfloor] := a[j]$; $j := 2j$;
    }
    $a[\lfloor j/2 \rfloor] := item$;
}
```


40

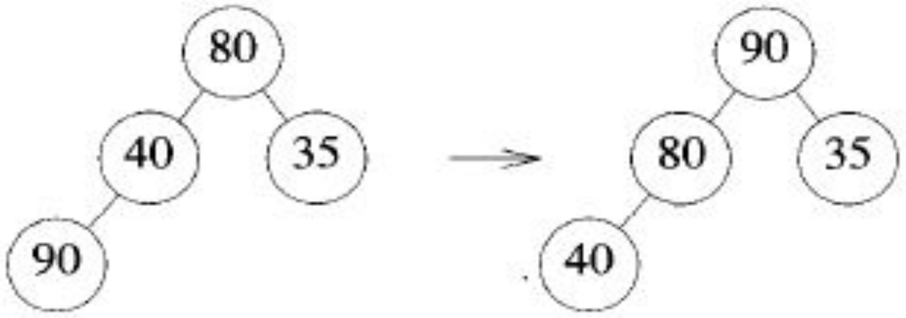
(a)


40
80
80
40

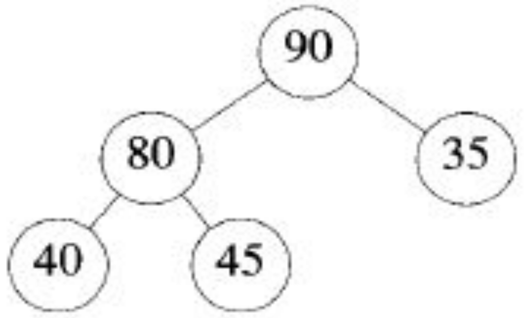
(b)


80
40
35

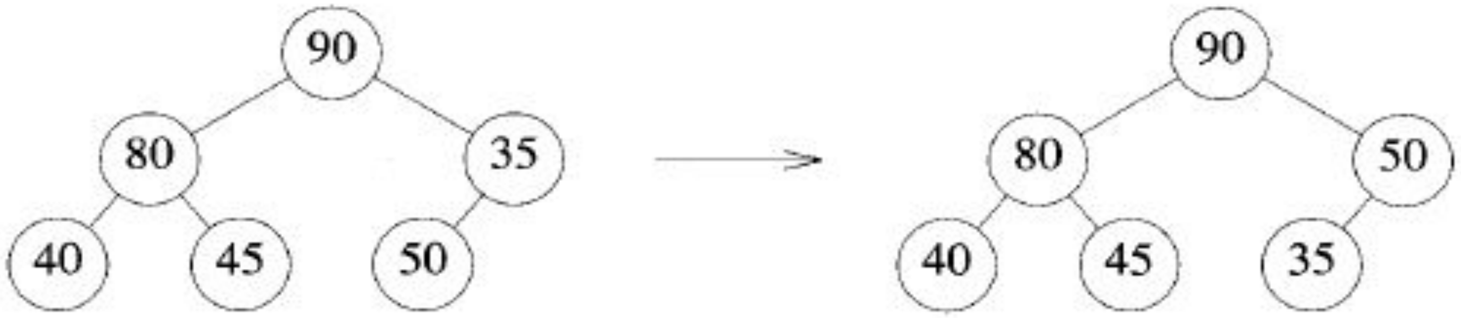
(c)


80
40
35
90
90
80
35
40

(d)


90
80
35
40
45

(e)


90
80
35
40
45
50
90
80
50
40
45
35

(f)

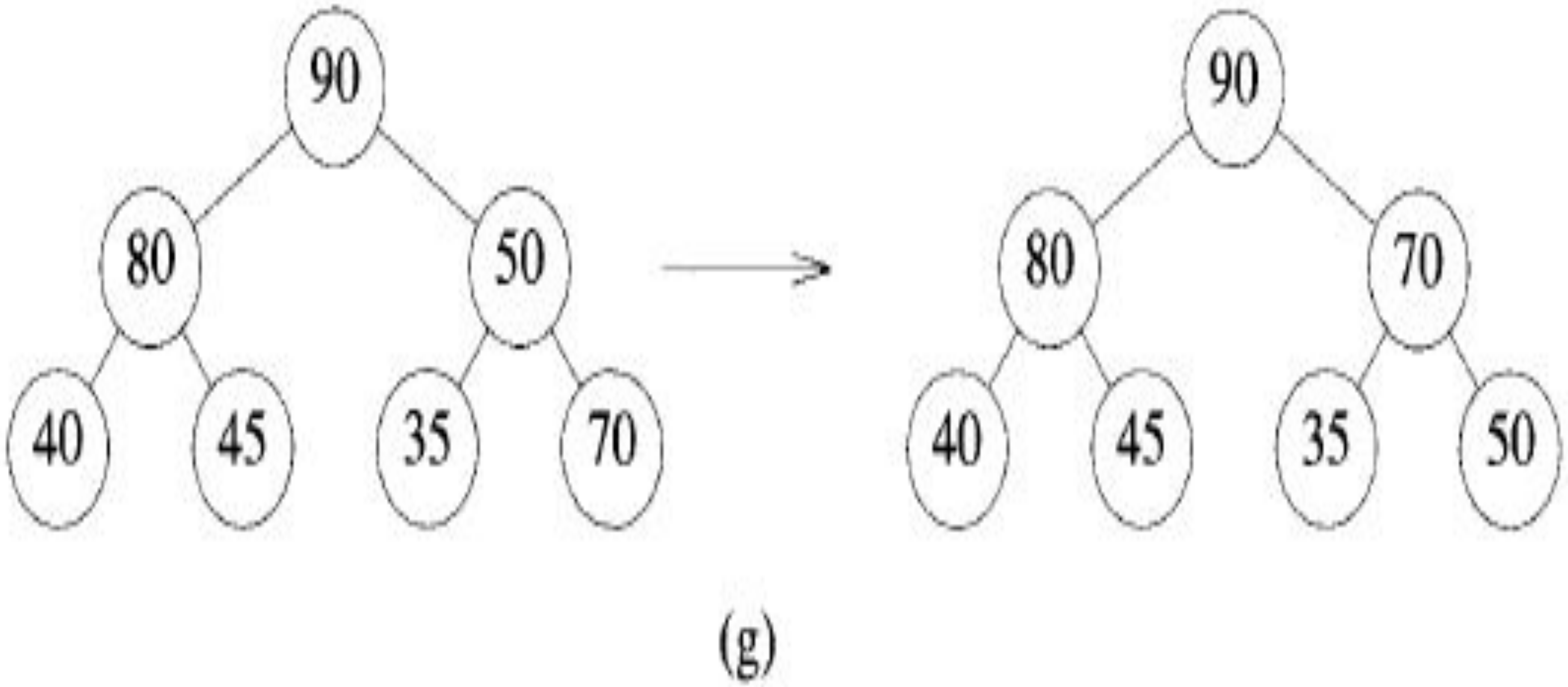


Figure 2.15 Forming a heap from the set $\{40, 80, 35, 90, 45, 50, 70\}$

```
Algorithm DelMax($a, n, x$)
// Delete the maximum from the heap $a[1:n]$ and store it in $x$.
{
    if ($n = 0$) then
    {
        write ("heap is empty"); return false;
    }
    $x := a[1]$; $a[1] := a[n]$;
    Adjust($a, 1, n - 1$); return true;
}
```

Note that the worst-case run time of Adjust is also proportional to the height of the tree. Therefore, if there are n elements in a heap, deleting the maximum can be done in $O(\log n)$ time.

To sort n elements, it suffices to make n insertions followed by n deletions from a heap. Algorithm 2.10 has the details. Since insertion and deletion take $O(\log n)$ time each in the worst case, this sorting algorithm has a time complexity of $O(n \log n)$.

```
Algorithm Sort(a, n)
// Sort the elements a[1 : n].
{
    for i := 1 to n do Insert(a, i);
    for i := n to 1 step -1 do
    {
        DelMax(a, i, x); a[i] := x;
    }
}
```

Algorithm 2.10 A sorting algorithm