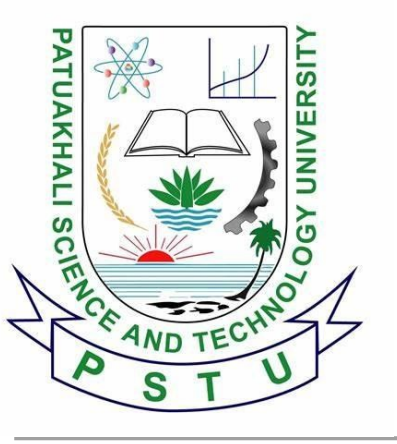


PATUAKHALI SCIENCE AND TECHNOLOGY UNIVERSITY

COURSE CODE CIT-121

SUBMITTED TO:

Prof. Dr. M. A. Masud
Department of Computer and Information Technology
Faculty of Computer Science and Engineering

SUBMITTED BY:

Md. Sharafat Karim
ID: **2102024**,
Registration No: **10151**
Faculty of Computer Science and Engineering

Second semester final examination
Topic: Induction and Recursion
Date:

Recursion

Recursion is the process where we can use the same object inside the same object. It makes it easy to call a function from itself. It can be used to define sequences, sets and functions. For example, we can consider the following function,

$$\sum_{k=0}^{n} a_k.$$

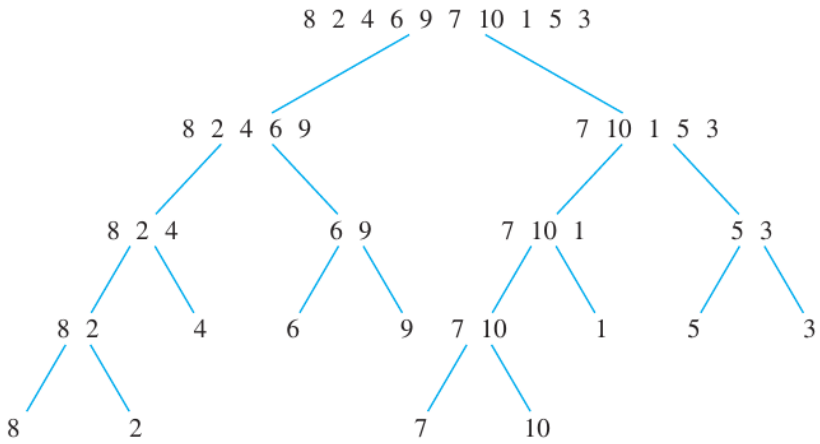
Here, we can expend it like,

$$\sum_{k=0}^{n+1} a_k = \left(\sum_{k=0}^{n} a_k\right) + a_{n+1}.$$

In this way we can call the same function from itself to find the result of the sequence. With recursion we can write graph and binary tree algorithms more efficiently. It is also vastly used in dynamic programming.

The Merge Sort

Merge sort is an bottom to top approach of sorting an array with support of recursion. In a merge sort, at first we try to split the list into 2 smaller parts. In this way we try to make things even more smaller. Finally we compare and merge them together. And from the bottom to top we try to merge the list. And this is why it is known as merge sort. So we have to merge items. We can draw it like a binary tree in this way.

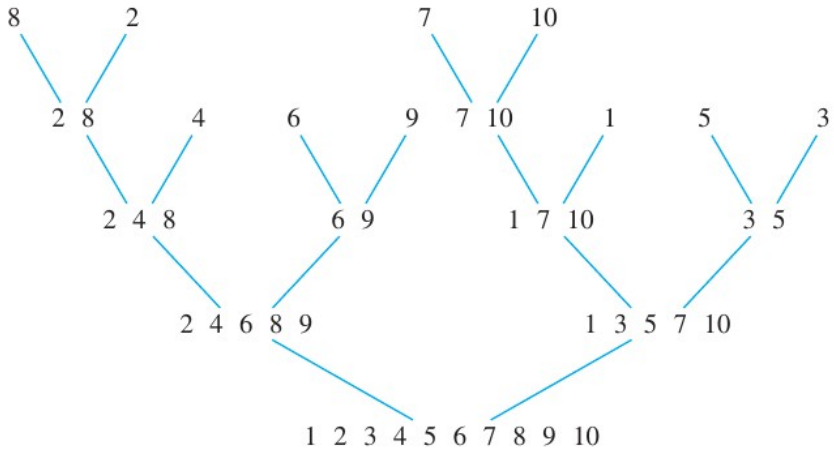


At first we will work with the bottom numbers. Here we will sort 8 and 2 into a new list. Thus it will become 2, 8. Later we will compare it with 4. So we need to functions. One function is to split the array into two smaller parts. And then we will use an another function to merge them back altogether. Here's the algorithm for merging two sorted array,

Merging two lists

procedure merge(L1, L2 : sorted lists)
L := empty list
while L1 and L2 are both nonempty
    remove smaller of first elements of L1 and L2 from its list; put it at the right end of L
    if this removal makes one list empty then remove all elements from the other list
and append them to L
return L {L is the merged list with elements in increasing order}

After merging two lists, we will move to complete the algorithm with a recursive function where we will call the same function again to divide our array into two smaller parts for our calculation until we have only one element left in our hand.



The algorithm's sudo-code is given below,

A Recursive Merge Sort.

procedure mergesort(L = a1 , ... , an)
if n > 1 then
    m := ⌊n/2⌋
    L1 := a1 , a2, ... , am
    L2 := am+1 , am+2, ... , an
    L := merge(mergesort(L1), mergesort(L2))
{L is now sorted into elements in nondecreasing order}

On the above example if we run the merge sort after finding the sorted sub arrays we are backtracking to the parent node. Then we are using the merge again and again to achieve result. So in the worst case scenario we have to iterate through we have to check through all of the values.

Time complexity
Time complexity of merge sort is n log(n). Here we are splitting the array into two parts like binary search. So the number of comparison is **O(n log(n))**.

Here's a c++ program to show the above merge sort algorithm,

```
#include <bits/stdc++.h>
using namespace std;

void print(int ar[], int n) {
    for (int i=0; i <n; i++) {
        cout << ar[i] << " ";
    } cout << "\n";
}
```

```
int* merge(int *ar_1, int i, int *ar_2, int j) {
    int p=0, p_1=0, p_2=0;
    int* ar = (int *) malloc((i+j)*sizeof(int));
    while (p_1 < i && p_2 < j) {
        if (ar_1[p_1] < ar_2[p_2]) {
            ar[p++] = ar_1[p_1++];
        } else {
            ar[p++] = ar_2[p_2++];
        }
    }
    while (p_1 < i) {
        ar[p++] = ar_1[p_1++];
    } while (p_2 < j) {
        ar[p++] = ar_2[p_2++];
    }
    return ar;
}

int* mergeSort(int *ar, int n) {
    if (n == 1) return ar;
    int i = n / 2, j = n - i;
    int ar_1[i], ar_2[j];
    for (int k=0; k < i; k++) {
        ar_1[k] = ar[k];
    } for (int k=i, l=0; k<n; k++) {
        ar_2[l++] = ar[k];
    }
    return merge(mergeSort(ar_1, i), i, mergeSort(ar_2, j), j);
}

int main() {
    int ar[] = {8, 2, 4, 6, 9, 7, 10, 1, 5, 3};
    int* sorted = mergeSort(ar, sizeof(ar)/ sizeof(int));
    print(sorted, sizeof(ar)/ sizeof(int));
}
```

Here the ar is a one dimensional array data structure. We use the merge sort algorithm on this array. We are using pointers to pass as a reference and then using an another array to copy it’s contents. And after sorting we are using malloc for dynamic programming to create a permanent memory space and returning it’s location which is the final result.

Induction
Induction is the process of proving theorems of sequence summation of iterative patterns.

For example, let's consider, P(n) is true for

$$n = b, b + 1, b + 2, \ldots ,$$

where b is an integer other than 1. We can use mathematical induction to accomplish this, as long as we change the basis step by replacing P(1) with P(b).
In other words, to use mathematical induction to show that P(n) is true for

$$n = b, b + 1, b + 2, \ldots ,$$

where b is an integer other than 1, we show that P(b) is true in the basis step. In the inductive step, we show that the conditional statement **P(k)** → **P(k + 1)** is true for k = b, b + 1, b + 2, … In this way we can prove that a certain equation is correct.

<u>Here's an example,</u>
let's consider a equation $1 + 2 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$.

Solution:
Let P(n) be the proposition that $1 + 2 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$ for the integer n.
P(0) is true because $2^0 = 1 = 2^1 - 1$.
This completes the basis step.

For the inductive hypothesis,
we assume that P(k) is true for an arbitrary nonnegative integer k.
That is, we assume that

$$1 + 2 + 2^2 + \ldots + 2^k = 2^{k+1} - 1.$$

To carry out the inductive step using this assumption,
we must show that when we assume that P(k) is true,
then P(k + 1) is also true.
That is, we must show that

$$1 + 2 + 2^2 + \ldots + 2^k + 2^{k+1} = 2^{(k+1)+1} - 1 = 2^{k+2} - 1$$

assuming the inductive hypothesis P(k).
Under the assumption of P(k),
we see that

$$\begin{aligned} 1 + 2 + 2^2 + \ldots + 2^k + 2^{k+1} &= (1 + 2 + 2^2 + \ldots + 2^k) + 2^{k+1} \\ &= (2^{k+1} - 1) + 2^{k+1} \\ &= 2 \cdot 2^{k+1} - 1 \\ &= 2^{k+2} - 1. \end{aligned}$$

So these two terms are same. So we can say that our induction process is working perfectly.