CSEN3001:
DESIGN AND ANALYSIS OF ALGORITHMS

UNIT-I: INTRODUCTION TO ALGORITHMS
And Binary Search

Divide and Conquer(DandC)

- **The general method, Control Abstraction**
- **Binary search**
- Finding maximum and minimum
- Merge sort
- Quick sort selection
- Strassen's matrix multiplication.

Divide and Conquer(DAndC): Introduction

- Split the inputs into k distinct subsets, $1<k\leq n$, yielding k sub problems.
- If the sub-problems are still large, apply the divide-and-conquer strategy again. If the sub-problems are still large, apply divide and conquer until it reduces to small enough to carryout the task.
- The sub-Problems must be solved & then a method must be found to combine sub-solutions into a solution of the whole.
- The sub-problems resulting from a divide and conquer are of the same type as the original Problem. The reapplication of the divide and conquer principle is expressed by a recursive algorithm.

Control Abstraction of Divide and Conquer

```
1. Algorithm DAndC(P)
2. {
3.    if small(P) then return  S(P);
4.    else
5.     {
6.       Divide P into small instances P1,  P2, P3,…………Pk  k>1;
7.       Apply DAndC to each of these sub Problems
8.       return combine (DAndC(P1), DAndC(P2), DAndC(P3),…DAndC(Pk));
9.        }  // end of if
10.   }   // end of algorithm DAndC
```

Computing Time:

$$T(n) = \begin{cases} g(n) & n \text{ is small} \\ T(n1) + T(n2) + \cdots + T(nk) + f(n) & \text{otherwise} \end{cases}$$

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT\left(\frac{n}{b}\right) + f(n) & n > 1 \end{cases}$$

Explanation:

- . The time complexity of a divide and conquer algorithm can often be analyzed using a recurrence relation. This relation captures how the problem is divided into subproblems and how these subproblems are combined to form the solution. A typical form of the recurrence relation for a divide and conquer algorithm is:

$$\boxed{\mathrm{T}(n) = a\mathrm{T}\left(\frac{n}{b}\right) + f(n)}$$

Where:

- T(n) is the time complexity of the problem of size n.
- **a** is the number of subproblems the problem is divided into.
- n/b is the size of each subproblem.
- f(n) is the cost of dividing the problem and combining the results of the subproblems

Explanation:

- **Number of subproblems (a):** This term represents how many subproblems the original problem is divided into. For example, in the merge sort algorithm, each problem is divided into two subproblems (a=2).
- **Size of each subproblem (n/b):** This term shows how the size of the problem decreases with each recursive step. For instance, in merge sort, each subproblem is half the size of the original problem (b=2).
- **Cost of dividing and combining (f(n)):** This is the time required to divide the problem into subproblems and then combine the solutions of the subproblems. In merge sort, the merging process takes O(n) time.

Explanation for the recurrence relation

The recurrence form arises from the fundamental structure of divide-and-conquer algorithms. These algorithms solve a problem by breaking it into smaller subproblems, solving each subproblem independently, and then combining their solutions. Let's break down the components of this recurrence to understand its reasoning:

$$\boxed{T(n) = aT\left(\frac{n}{b}\right) + f(n)}$$

1. Dividing the Problem:
   - Number of Subproblems (a): The problem of size n is divided into '***a***' smaller subproblems.
   - Size of Each Subproblem (n\b): Each subproblem is of size n\b..

2. Solving the Subproblems:
   - The time complexity to solve each subproblem of size n\b is T(n\b).
   - Since there are '***a***' subproblems, the total time to solve all subproblems is aT(n\b).

Example:

3. Combining the Solutions:

- Cost of Dividing and Combining (f(n)): After solving the subproblems, we must combine their solutions to get the final result. The time complexity for this combination step is f(n).
- f(n)can vary depending on the specific algorithm. For example, in merge sort, f(n) is the time taken to merge two sorted arrays, which is O(n).

Example:

Derivation of the Recurrence

Let's consider a generic divide-and-conquer algorithm:

- 1. Divide: The algorithm divides the problem of size ***n*** into ‘***a***’ smaller subproblems, each of size n\b.
- 2. Conquer: The algorithm recursively solves each of these subproblems. The time complexity for this step is aT(n\b).
- 3. Combine: The algorithm combines the results of the subproblems to get the final solution. The time complexity for this step is f(n).

Therefore, the total time complexity T(n) is the sum of the time complexities of these three steps:

$$\boxed{\mathrm{T}(n) = a\mathrm{T}\left(\frac{n}{b}\right) + f(n)}$$

Example:

This recurrence relation captures the essence of the divide-and-conquer approach:

- Recursive Division: The problem is recursively divided into smaller parts.
- Parallel Solving: Each part is solved independently (conceptually and in parallel).
- Merging Results: The partial results are merged to form the final solution.

Example:

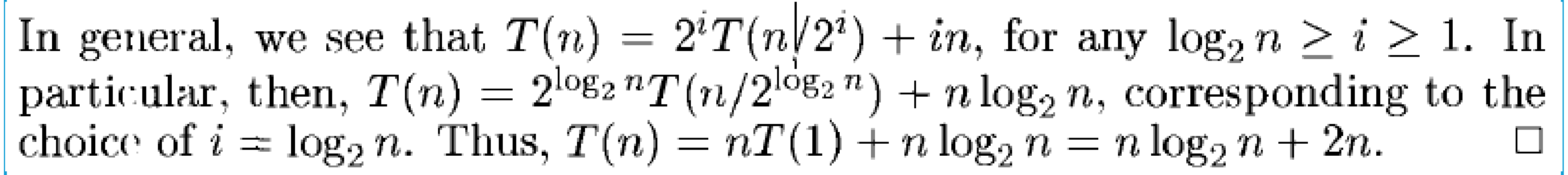
Substitution method for solving the recurrence relation:

Consider the case in which a=2 and b=2

Let T(1)=2 and f(n)=n .

we have..

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2[2T(n/4) + n/2] + n \\
&= 4T(n/4) + 2n \\
&= 4[\,2T(n/8) + n/4] + 2n \\
&= 8T(n/8) + 3n
\end{aligned}
$$



In general, we see that $T(n) = 2^i T(n/2^i) + in$, for any $\log_2 n \geq i \geq 1$. In particular, then, $T(n) = 2^{\log_2 n} T(n/2^{\log_2 n}) + n \log_2 n$, corresponding to the choice of $i = \log_2 n$. Thus, $T(n) = nT(1) + n \log_2 n = n \log_2 n + 2n$. □

Search

- Linear Search
  - The data need not be in sorted form
- Binary Search
  - The data should be in sorted form on the search key

- Ascending order / Increasing order
- Descending order / Decreasing order
- Non-decreasing order
- Non-increasing order

Binary Search

- Let a[ ] $1 \leq i \leq n$, be a list of elements that are sorted in non-decreasing order.
- Problem statement: determine whether a given element x is present in the list.
  - If x is present, determine a value j such that a[j] = x.
  - If x is not in the list, then j is to be set to zero.
- Let $P = (n, a_i, a_{i+1}, a_{i+2}, \ldots, a_l, x)$ denote an arbitrary instance of this search problem.
- n is the number of elements in the list
- a[i : l] is the list of elements
- x is the element to be searched

Binary Search

- Let Small(P)be true if n = 1.
- In this case, S(P)will take the value ***i***, if x = a[i], otherwise it will take the value 0.
- Then g(1) = Θ(1).
- If P has more than one element, it can be divided (or reduced) into a new sub-problem.
- Pick an index q (in the range [i, l])and compare x with a[q].

1. x = a[q]: In this case, the problem P is immediately solved.
2. X < a[q]: In this case x has to be searched for only in the sub list a[i], a[i+1],....a[q-1].
   Therefore, P reduces to (q-l, a[i], a[i+1], ...,a[q-1], x).
3. X > a[q]: In this case, the sub list to be searched is a[q+1], a[q+2], ..., a[l].

   Therefore, P reduces to (l-q, a[q+1], a[q+2], ... a[l], x).

Algorithm: Binary Search -- Recursive Method

```
1. Algorithm BinSrch(a, i, l, x)
2. // Given an array a[i:l] of elements in non-decreasing order, 1<=i<=l,
3. //determine whether x is present and if so, return j such that x=a[j]; else return 0
4. {  if (i==l) then // If Small(P)
5.      { if x==a[i] then return I;
6.        else return 0; }
7.    else
8.        { // Reduce P into smaller subproblem
9.           mid = $\lfloor (i+l)/2 \rfloor$
10.          if (x = a[mid] then return mid;
11.          else if (x <a[mid]) then return BinSrch(a, i, mid-1, x)
12.                  else return BinSrch(a, mid+1,l,x)
13.          }  // end of if(i==l)
14. } // end of Algorithm
```

Algorithm Binary Search: Iterative Method

```
Algorithm BinSearch(a, n, x)
// Given an array a[1:n] of elements in non decreasing order, n>0, determine
//whether x is present, and if so, return j such that x=a[j]; else return 0.
 {
        low = 1; high = n;
        while (low<=high)
        {    $mid = ceil(i+1)/2$
                if (x<a[mid] then high = mid -1;
                else if (x>a[mid] then low = mid +1;
                else return mid;
        } // end of while loop
    return 0;
} // end of Algorithm
```

Binary Search : Example

1	2	3	4	5	6	7	8	9	10	11	12	13	14	Index
-15	-6	0	7	9	23	54	82	101	112	125	131	142	151	Value

Example 1: X = 151

Low	high	mid
1	14	7
8	14	11
12	14	13
14	14	14
		found

Example 2: x = -14

Low	high	mid
1	14	7
1	6	3
1	2	1
2	2	2
2	1	1
		not found

Example 3: x = 9

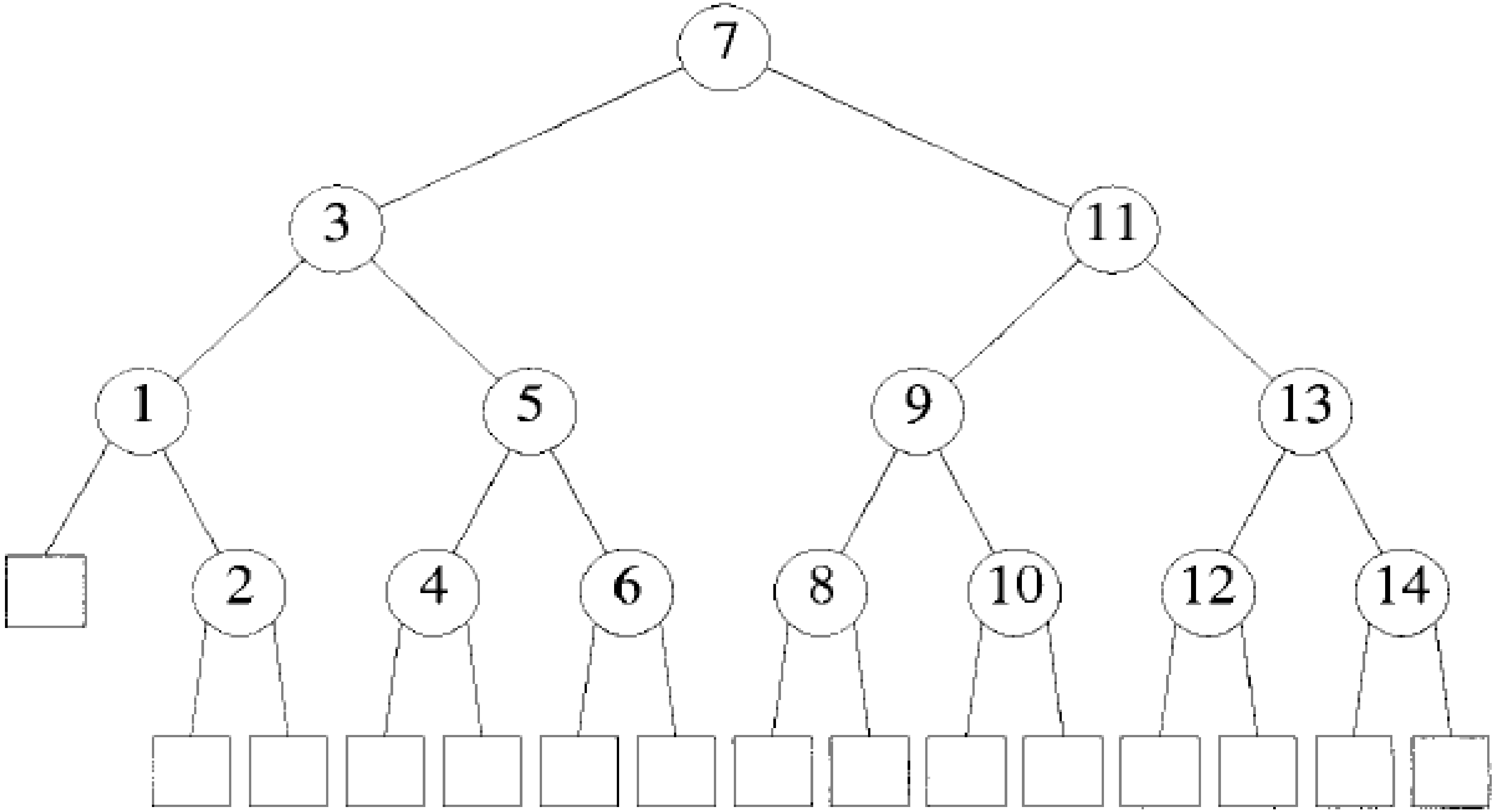
Low	high	mid
1	14	7
1	6	3
4	6	5
		found

Algorithm: Binary Search1

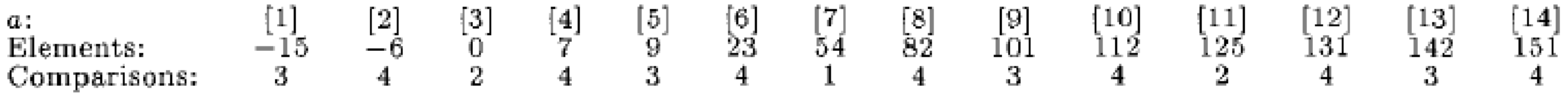
```
Algorithm BinSearch1(a,n,x)
// Same specification as BinSearch except n>0
{
      low =1; high = n+1;
     // high is one more than possible
     while(low<(high-1) do
     {   $mid = ceil(i + 1)/2$
       if (x<a[mid] then high = mid;
                // only one comparison in the loop
        else low = mid;
        }
      if (x = a[low]) then return low;
     else return 0;
} // end of algorithm
```

Binary Decision Tree for Binary Search, n=14

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
Elements:	−15	−6	0	7	9	23	54	82	101	112	125	131	142	151
Comparisons:	3	4	2	4	3	4	1	4	3	4	2	4	3	4



Binary Decision Tree for Binary Search, n=14



a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
Elements:	−15	−6	0	7	9	23	54	82	101	112	125	131	142	151
Comparisons:	3	4	2	4	3	4	1	4	3	4	2	4	3	4

- NO element requires more than 4 comparisons to be found.
- Average for successful : (3+4+2+4+3+4+1+4+3+4+2+4+3+4)/14 = 45/14 = 3.21~
- Average for unsuccessful : (3 + 14 * 4)/15 = 59 /15 = 3.93~

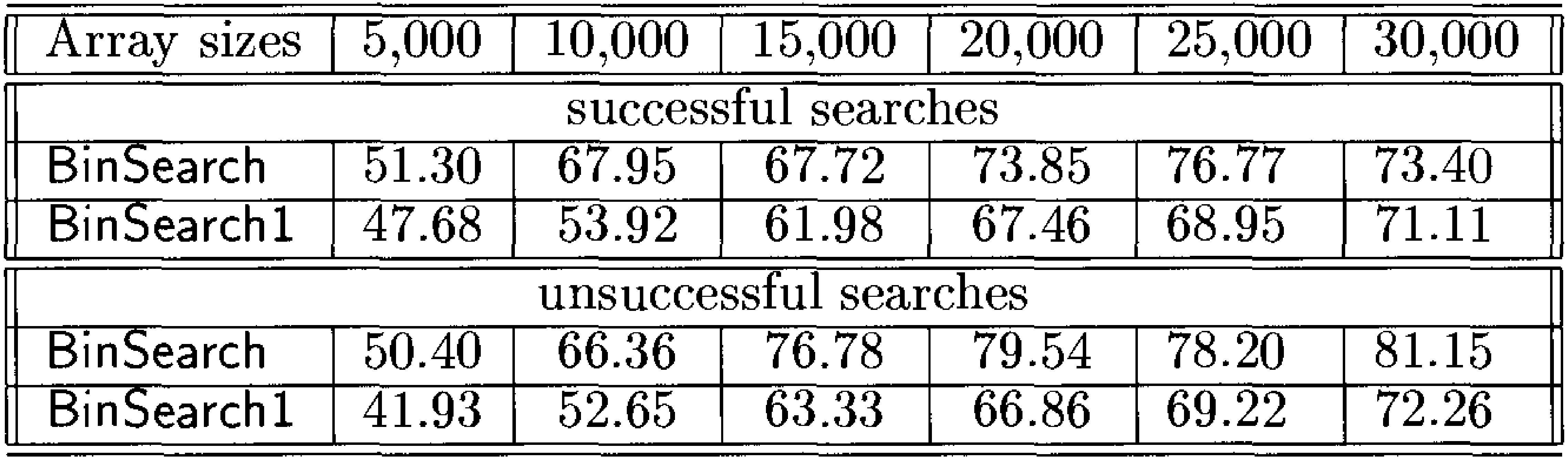
Time Complexity of Binary Search

Successful searches
Unsuccessful searches

Best	Average	Worst
θ(1)	**θ(log n)**	**θ(log n)**

- If n is in the range $[2^{k-1}, 2^k)$ the BinSearch makes at most k element comparisons for a successful search and either k-1 or k comparisons for an unsuccessful search.
- The time for successful search is O(log n) and for unsuccessful search is θ(log n)

Computing time of two binary search algorithms



Array sizes	5,000	10,000	15,000	20,000	25,000	30,000
successful searches						
BinSearch	51.30	67.95	67.72	73.85	76.77	73.40
BinSearch1	47.68	53.92	61.98	67.46	68.95	71.11
unsuccessful searches						
BinSearch	50.40	66.36	76.78	79.54	78.20	81.15
BinSearch1	41.93	52.65	63.33	66.86	69.22	72.26