Unit-1
Divide and Conquer

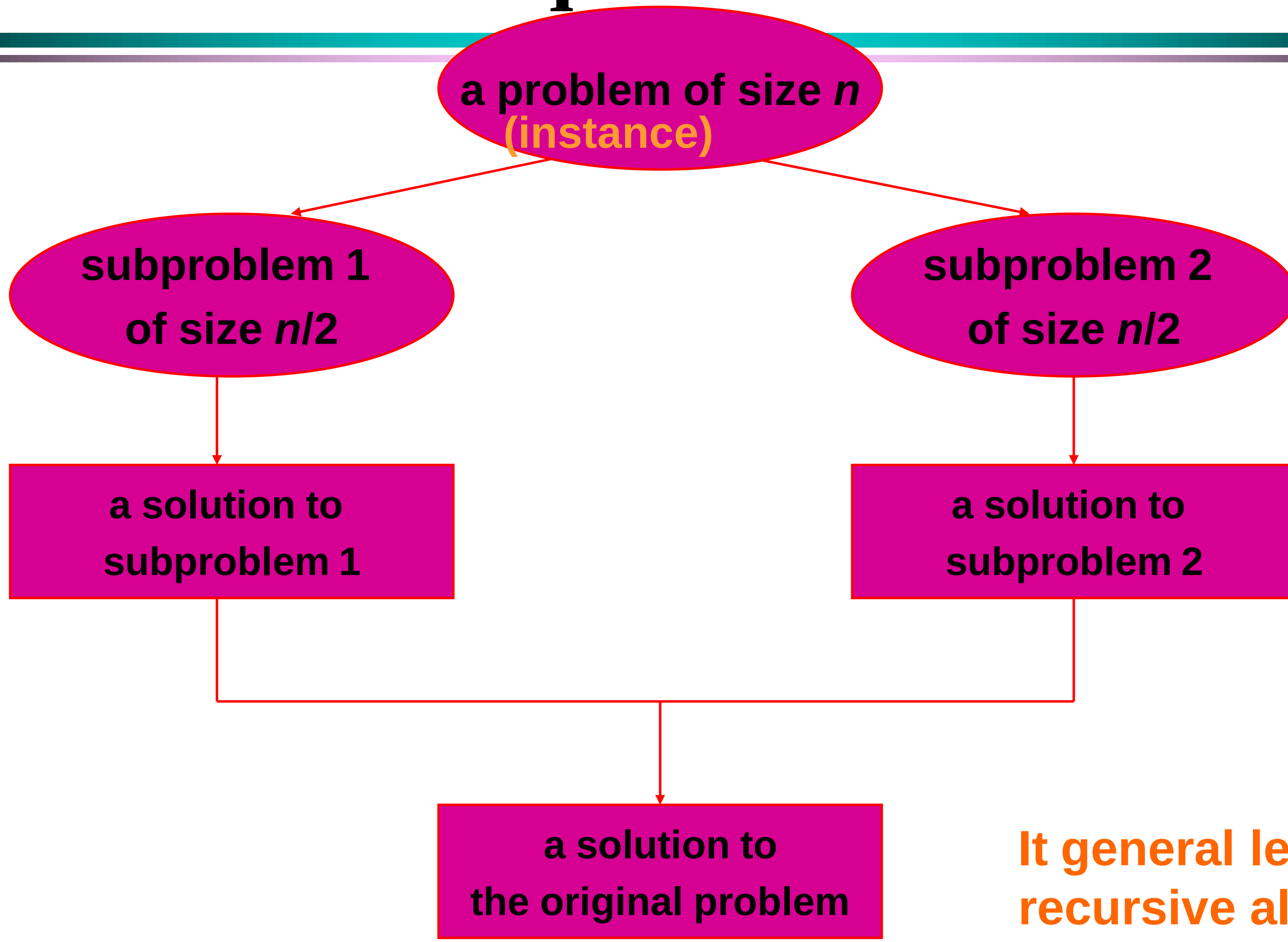
- Dr. K.RAGHAVA RAO
- Professor in CSE
- KL University
- krraocse@gmail.com
- http://mcadaa.blog.com

Divide and Conquer: General Method

Definition:

Divide the problem into a number of subproblems, Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

Divide and Conquer: General Method

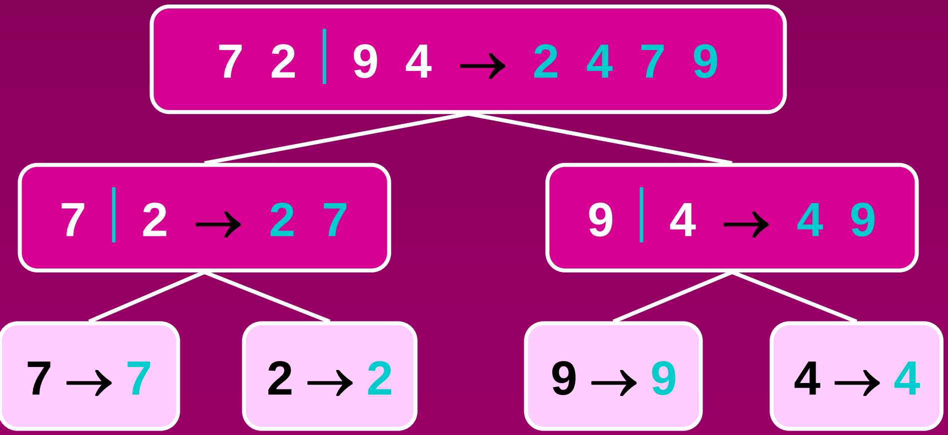


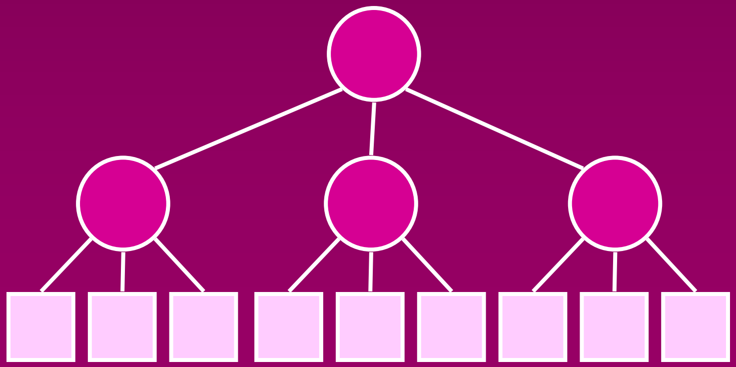
It general leads to a recursive algorithm!

Divide and Conquer: General Method

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two or more disjoint subsets S_1, S_2, …
  - Recur: solve the subproblems recursively
  - Conquer: combine the solutions for S_1, S_2, …, into a solution for S

- The base case for the recursion are subproblems of constant size.Analysis can be done using **recurrence equations**





Divide and Conquer: General Method

```
Algorithm D-and-C(n: input size) {
if n ≤ n0 /* small size problem*/
Solve problem without further sub-division;
Else
  {
    Divide into m sub-problems;
    Conquer the sub-problems by solving them
    independently and recursively; /* D-and-C(n/k) */
   Combine the solutions;
   }
}
```

Advantage: straightforward and running times are often easily Determined

Divide and Conquer: General Method

Divide-and-Conquer Recurrence Relations

Suppose that a recursive algorithm divides a problem of size n into a parts, where each sub-problem is of size n/b. Also, suppose that a total number of g(n) extra operations are needed in the conquer step of the algorithm to combine the solutions of the sub-problems into a solution of the original problem. Let f(n) be the number of operations required to solve the problem of size n. Then f satisfies the recurrence relation

$$f(n)=a\,f(n/b)+g(n)$$

and it is called divide-and-conquer recurrence relat**ion.**

Divide and Conquer: General Method

>-The computing time of Divide and conquer is described by recurrence relation.

>-$T(n) = \{$ $g(n)$ n small

$\{$ $T(n1)+T(n2)+\ldots\ldots\ldots+T(n_k) + f(n)$ other wise

>-T(n) is the time for Divide and Conquer on any input of size n and g(n) is the time to compute the answer directly for small inputs. The function of f(n) is the time for dividing P combining solutions to subproblems.

>-For divide-and-conquer-based algorithms that produce subproblems of the same type as the original problem, then such algorithm described using recursion.

Divide and Conquer: General Method

The complexity of many divide-and-conquer algorithms is given by recurrence of the form.

T(n)= { T(1)   n=1

{a T(n/b) + f(n)   n>1 where a and b are known constants,

and n is a power of b ($n=b^k$).

One of the methods for solving any such recurrence relation is called substitution method.

Divide and Conquer: General Method

Examples:

If a=2 and b=2 . Let T(1)=2 and f(n)=n. Than

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

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

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

$= 4[2T(n/8) + n/4] + 2n$

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

.

.

.

In general , $T(n) = 2^i T(n/2^i) + in$, for any $\log_2 n >= i >= 1$. In Particular ,then $T(n) = 2^{\log_2 n} T(n/2^{\log_2 n}) + n \log_2 n$ corresponding to choice of $i = \log_2 n$. Thus, $T(n) = n T(1) + n \log_2 n = n \log_2 n + 2n$.

Divide and Conquer: General Method

- Exercise for students
- Solve above recurrency relation when 1)$a=1$, $b=2$ and $f(n)=cn$
- 2)$a=5$, $b=4$ and $f(n)=cn^2$
-
- 3)$a=28$ $b=3$ and $f(n) = cn^3$

Divide and Conquer: Min and Max

- The **minimum** of a set of elements:
  - The first order statistic $i = 1$
- The **maximum** of a set of elements:
  - The n-th order statistic $i = n$
- The **median** is the "halfway point" of the set
  - $i = (n+1)/2$, <u>is unique</u> when n is odd
  - $i = \lfloor (n+1)/2 \rfloor$ = n/2 (lower median) and $\lceil (n+1)/2 \rceil$ = n/2+1 (upper median), when n is even

Finding Minimum or Maximum

Alg.: MINIMUM(A, n)

```
min ← A[1]
for i ← 2 to n
    do if min > A[i]
        then min ← A[i]
return min
```

- How many comparisons are needed?
  - $n - 1$: each element, except the minimum, must be compared to a smaller element at least once
  - The same number of comparisons are needed to find the maximum
  - The algorithm is **optimal** with respect to the number of comparisons performed

Simultaneous Min, Max

- Find min and max independently
  - Use $n - 1$ comparisons for each $\Rightarrow$ total of $2n - 2$
- At most $3n/2$ comparisons are needed
  - Process elements in pairs
  - Maintain the minimum and maximum of elements seen so far
  - Don't compare each element to the minimum and maximum separately
  - Compare the elements of a pair to each other
  - Compare the larger element to the maximum so far, and compare the smaller element to the minimum so far
  - This leads to only 3 comparisons for every 2 elements

Analysis of Simultaneous Min, Max

- Setting up initial values: **set both min and max to the first element**
  - n is odd: **compare the first two elements, assign the smallest one to min and the largest one to max**
  - n is even:
- Total number of comparisons:
  - n is odd: we do $3(n-1)/2$ comparisons
  - n is even: we do 1 initial comparison + $3(n-2)/2$ more comparisons = $3n/2 - 2$ comparisons

Example: Simultaneous Min, Max

- n = 5 (odd), array A = {2, 7, 1, 3, 4}

1. Set **min** = **max** = 2

2. Compare elements in pairs:

   – 1 < 7 $\Rightarrow$ compare 1 with **min** and 7 with **max**

     $\Rightarrow$ **min** = 1, **max** = 7

   3 comparisons

   – 3 < 4 $\Rightarrow$ compare 3 with **min** and 4 with **max**

     $\Rightarrow$ **min** = 1, **max** = 7

   3 comparisons

We performed: 3(n-1)/2 = 6 comparisons

Example: Simultaneous Min, Max

- n = 6 (even), array A = {2, 5, 3, 7, 1, 4}

1 comparison

1. Compare 2 with 5: 2 < 5
2. Set **min =** 2, **max** = 5
3. Compare elements in pairs:

3 comparisons

   – 3 < 7 $\Rightarrow$ compare 3 with **min** and 7 with **max**

     $\Rightarrow$ **min** = 2, **max** = 7

3 comparisons

   – 1 < 4 $\Rightarrow$ compare 1 with **min** and 4 with **max**

We performed: 3n/2 - 2 = 7 comparisons

     $\Rightarrow$ **min** = 1, **max** = 7

Divide and Conquer: Binary Search

Binary search method.

The basic idea is to start with an examination of the middle element of the array. This will lead to 3 possible situations:

If this matches the target K, then search can terminate successfully, by printing out the index of the element in the array.

On the other hand, if K<A[middle], then search can be limited to elements to the left of A[middle]. All elements to the right of middle can be ignored.

If it turns out that K >A[middle], then further search is limited to elements to the right of A[middle].

If all elements are exhausted and the target is not found in the array, then the method returns a special value such as –1.

Divide and Conquer: Binary Search

Here is one version of the Binary Search function:

```
int BinarySearch (int A[ ], int n, int K)
{
int L=0, Mid, R= n-1;
while (L<=R)
{
Mid = (L +R)/2;
if ( K= =A[Mid] )
return Mid;
else if ( K > A[Mid] )
L = Mid + 1;
else
R = Mid – 1 ;
}
return –1 ;}
```

Divide and Conquer: Binary Search

Let us now carry out an Analysis of this method to determine its time complexity. Since

there are no “for” loops, we can not use summations to express the total number of

operations. Let us examine the operations for a specific case, where the number of

elements in the array n is 64.

When n= 64 BinarySearch is called to reduce size to n=32

When n= 32 BinarySearch is called to reduce size to n=16

When n= 16 BinarySearch is called to reduce size to n=8

When n= 8 BinarySearch is called to reduce size to n=4

When n= 4 BinarySearch is called to reduce size to n=2

When n= 2 BinarySearch is called to reduce size to n=1

Divide and Conquer: Binary Search

Thus we see that BinarySearch function is called 6 times (6 elements of the array were

examined) for n =64.

Note that 64 = 26

Also we see that the BinarySearch function is called 5 times (5 elements of the array

were examined) for n = 32.

Note that 32 = 25

Let us consider a more general case where n is still a power of 2. Let us say n = 2k .

Divide and Conquer: Binary Search

Following the above argument for 64 elements, it is easily seen that after k searches, the

while loop is executed k times and n reduces to size 1.

Let us assume that each run of the while loop involves at most 5 operations.

Thus total number of operations: 5k.

The value of k can be determined from the expression

$2^k = n$

Taking log of both sides

$\text{Log } 2^k = \log n$

Thus total number of operations = 5 log n.

We conclude that the time complexity of the Binary search method is O(log n), which is much more efficient than the Linear Search method.

Divide and Conquer: Binary Search

Here is second version of the Binary Search function:

Binary-Search (*A; p; q; x)*

1. if *p > q return -1;*

2. *r = b (p + q)=2 c*

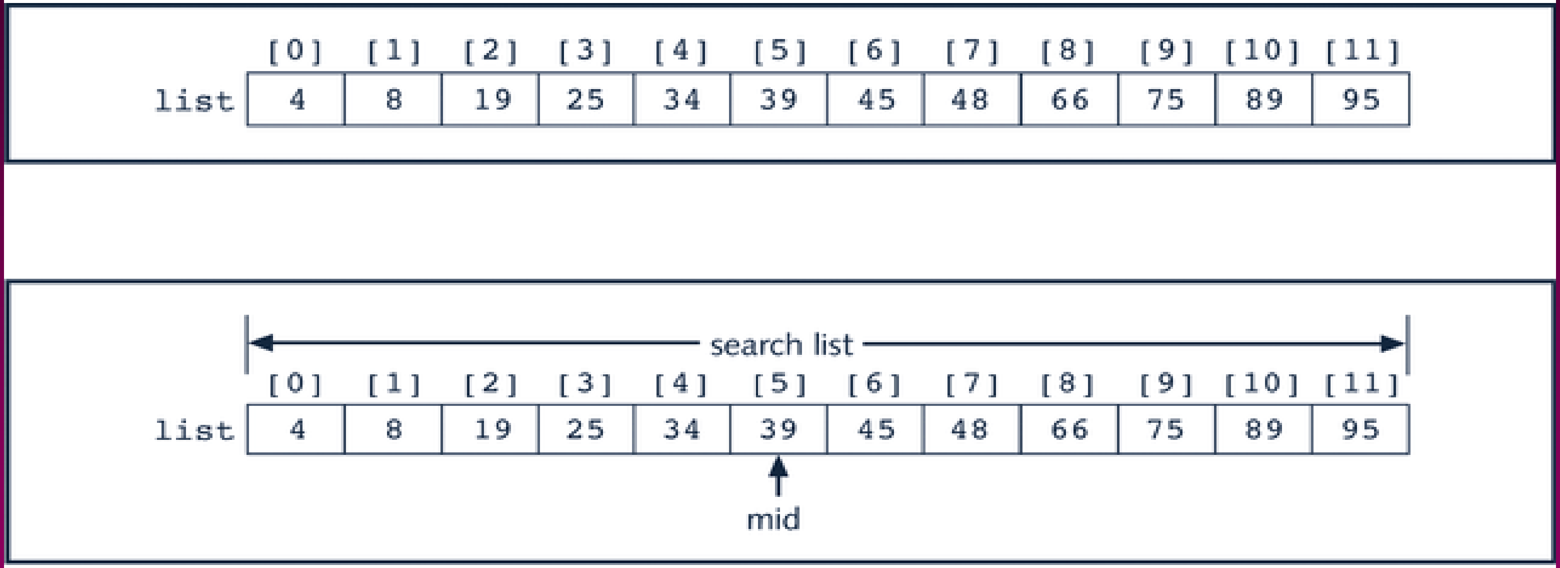
3. if *x = A[r] return r*

4. else if *x < A[r] Binary-Search(A; p; r; x)*

5. else Binary-Search(*A; r + 1; q; x)*

^{2} The initial call is Binary-Search(A; 1; n; x).

Binary Search



Search list, list[0]...list[11]

Binary Search: middle element

$$\texttt{mid} = \frac{\texttt{left + right}}{2}$$

Binary Search: Example

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
list	4	8	19	25	34	39	45	48	66	75	89	95

Figure 9-4 Sorted list for a binary search

Table 9-1 Values of `first`, `last`, and `middle` and the Number of Comparisons for Search Item `89`

Iteration	first	last	mid	list[mid]	Number of Comparisons
1	0	11	5	39	2
2	6	11	8	66	2
3	9	11	10	89	1 (found is true)

Binary Search

[0]	ant
[1]	cat
[2]	chicken
[3]	cow
[4]	deer
[5]	dog
[6]	fish
[7]	goat
[8]	horse
[9]	camel
[10]	snake

Searching for cat

BinarySearch(0, 10)	middle: 5	cat < dog	
BinarySearch(0, 4)	middle: 2	cat < chicken	
BinarySearch(0, 1)	middle: 0	cat > ant	
BinarySearch(1, 1)	middle: 1	cat = cat	**Return: true**

Searching for zebra

BinarySearch(0, 10)	middle: 5	zebra > dog	
BinarySearch(6, 10)	middle: 8	zebra > horse	
BinarySearch(9, 10)	middle: 9	zebra > camel	
BinarySearch(10, 10)	middle: 10	zebra > snake	
BinarySearch(11, 10)		last > first	**Return: false**

Searching for fish

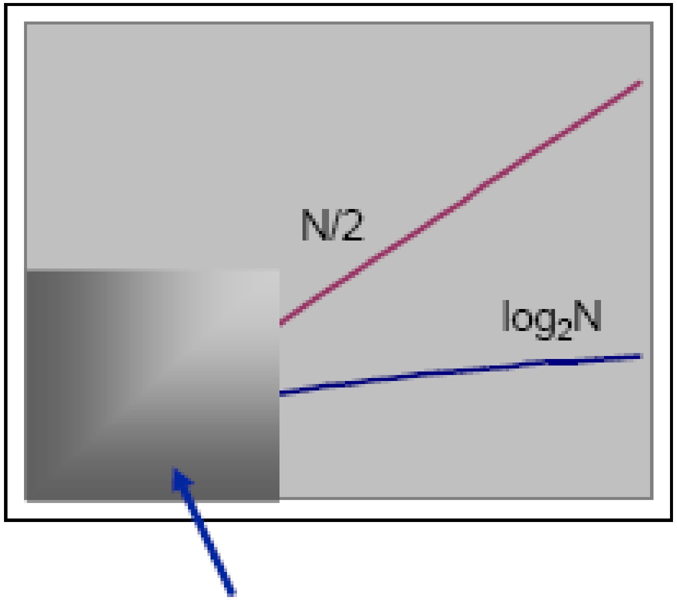
BinarySearch(0, 10)	middle: 5	fish > dog	
BinarySearch(6, 10)	middle: 8	fish < horse	
BinarySearch(6, 7)	middle: 6	fish = fish	**Return: true**

Binary Search

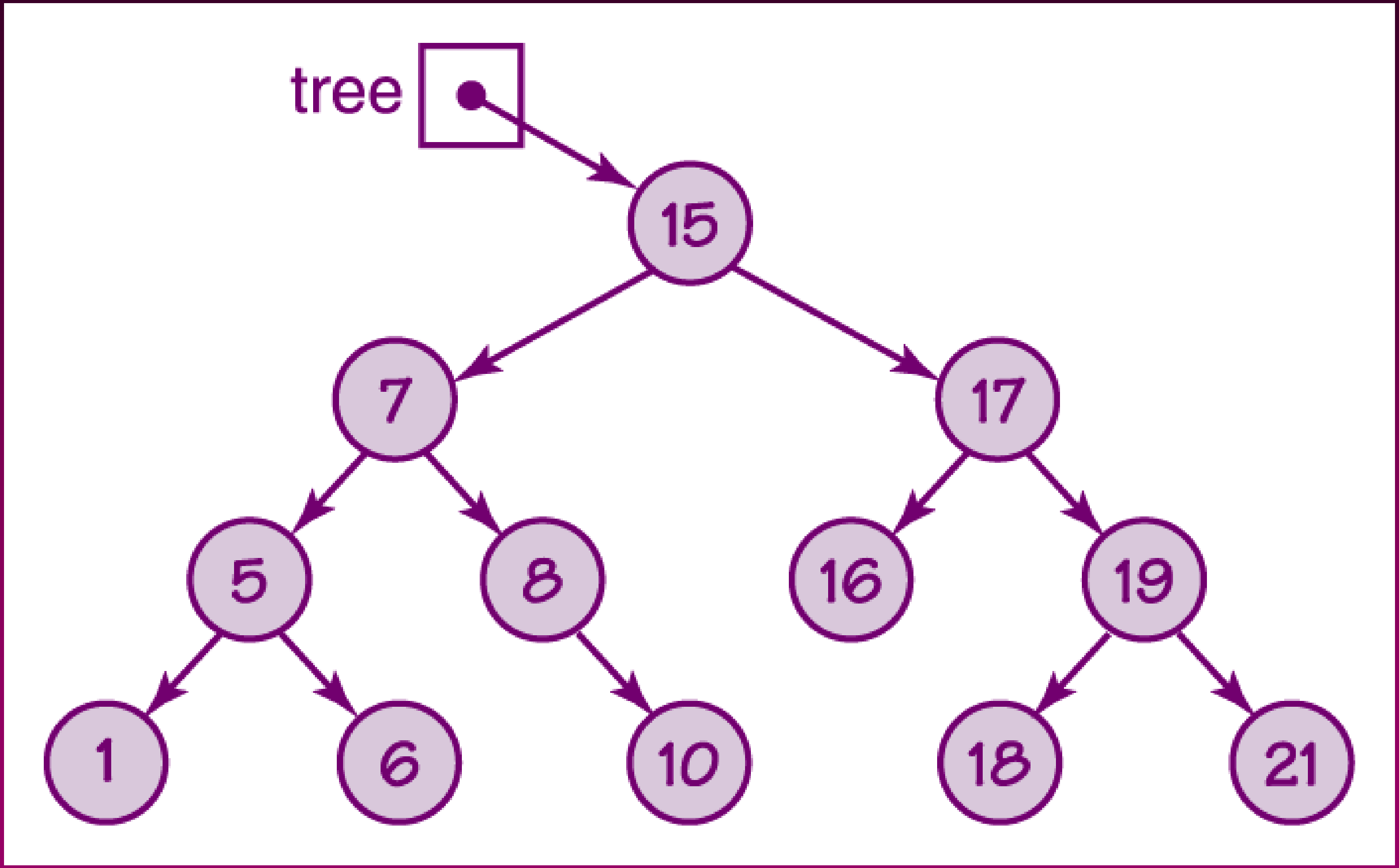
Binary Search Tradeoffs

- Benefit
  - Much more efficient than linear search

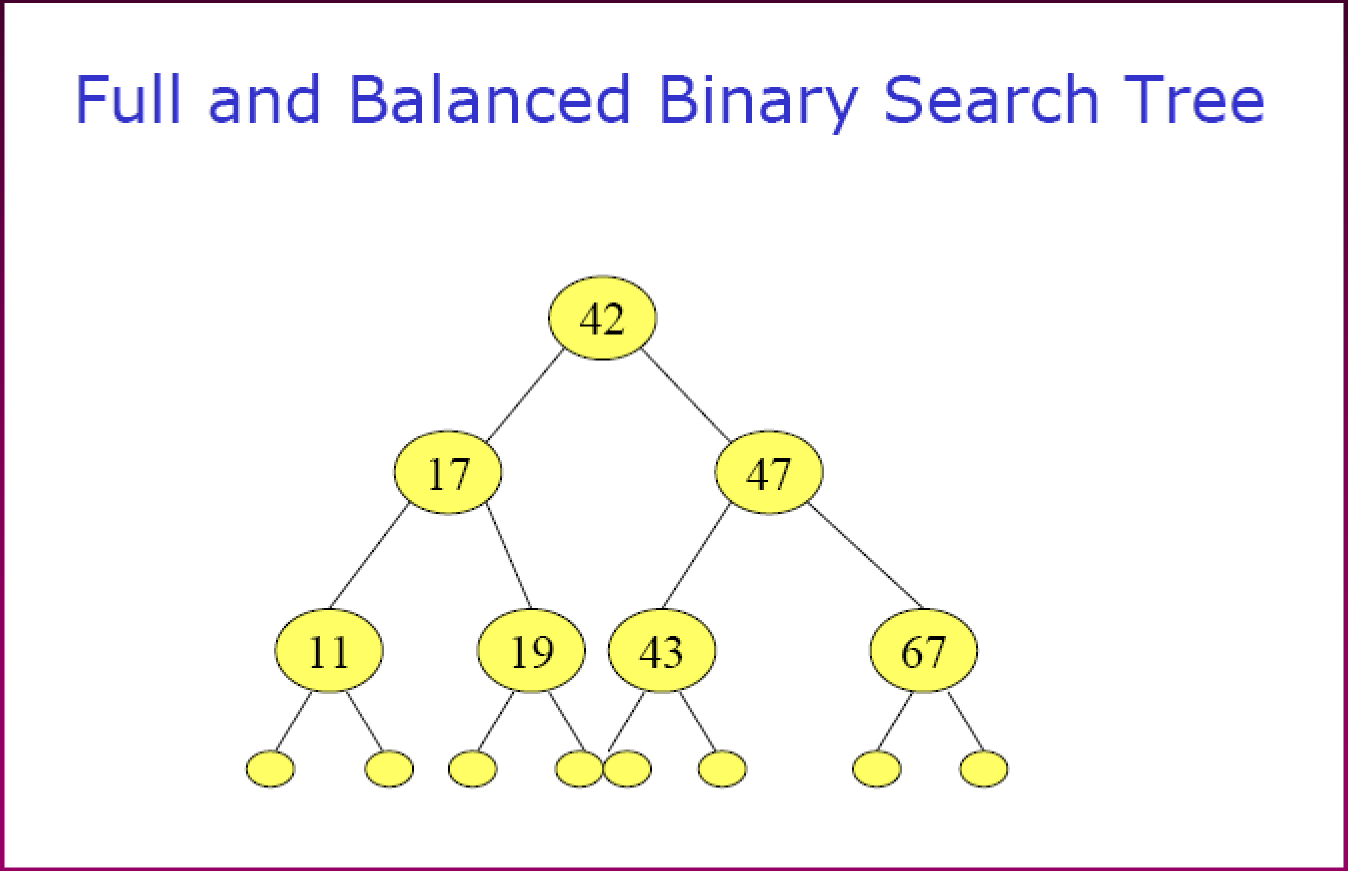
    (For array of **N** elements, performs at most log_2N comparisons)
- Disadvantage
  - Requires that array elements be sorted



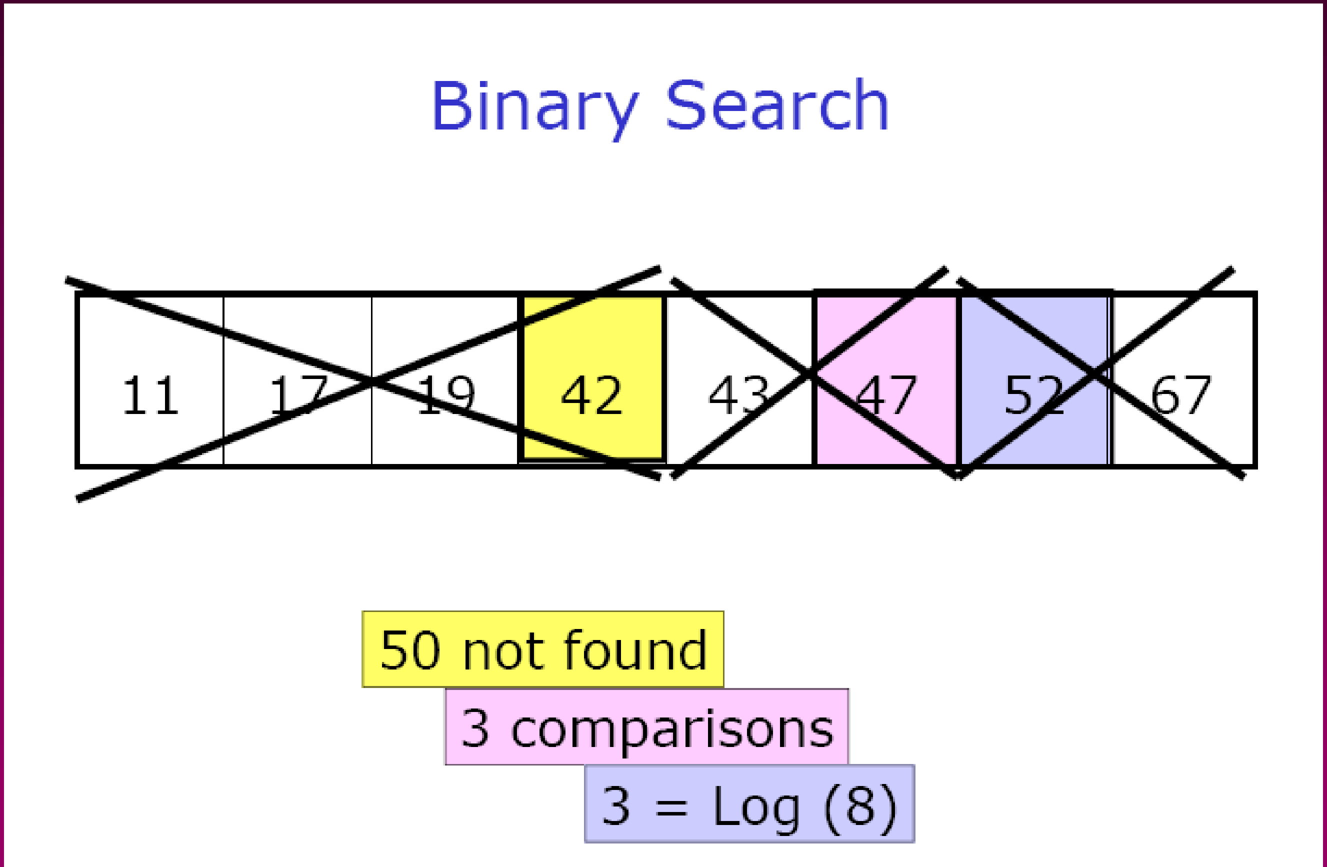
Binary Search Tree



Binary Search Tree



Binary Search Tree



Logarithmic Time Complexity of Binary Search

- P Our analysis shows that binary search can be done in time proportional to the *log* of the number of items in the list
- P This is considered *very fast* when compared to linear or polynomial algorithms
- P The table to the right compares the number of operations that need to be performed for algorithms of various time complexities

The computing time binary search by best, average and worst cases:

Successful searches

$\Theta(1)$ best , $\Theta(\log n)$ average

$\Theta(\log n)$ worst

Unsuccessful searches

$\Theta(\log n)$ for best , average and worst case

Binary Search Tree

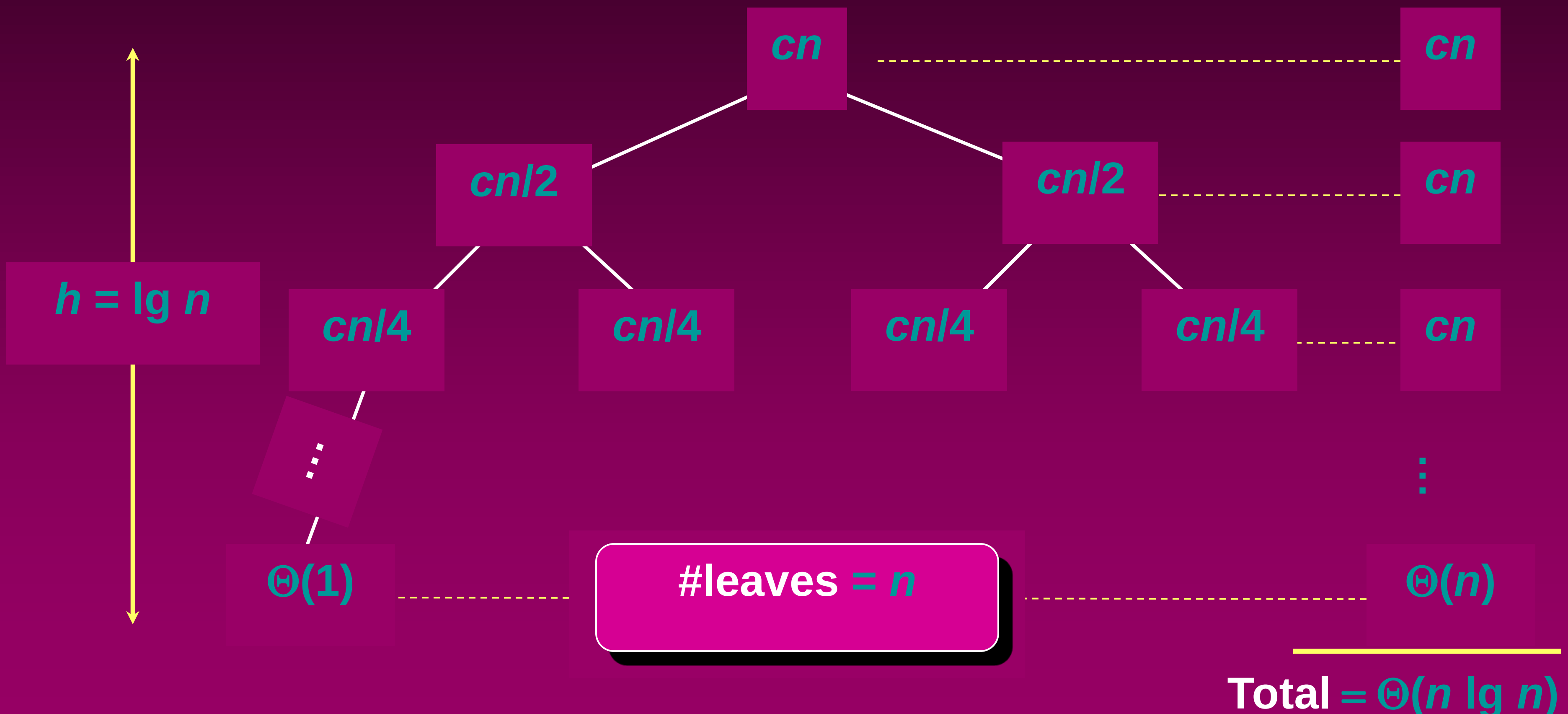
Binary Search

- Can be performed on
  - Sorted arrays
  - Full and balanced BSTs
- Compares and cuts half the work
  - We cut work in ½ each time
  - How many times can we cut in half?

Binary search is **O(Log N)**

Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Big Oh notation

constant
f(n)=16
f(n)=27
f(n)<=16*1 where c=16 and n0=0
F(n)<=27*1 where c=27 and n0=0 so big oh notation as O(1). So f(n)=O(1).

Linear
f(n)=7n+5 find bih oh notation
f(n)=7n+5 for n>=5
7n+5<=7n+n<=8n (c=8 n0=5) , so f(n)=O(n).

Quadratic
f(n)=27n2+16n
f(n)=27n2+16n, for n2>=16n {or 16n<=n2}
27n2+16n<=27n2+n2<=28n2 (c=28, n0=16)
So f(n)= O(n2)

Big Oh notation

If we consider   n<=n2 then

27n2+16n<=27n2 +16n2<=43n2 {c=43, n0=1)

So, f(n)=o(n2)

--

f(n)=27n2+16 for n>=16

27n2+16<=27n2+n

Now, for n<=n2

27n2+n<=27n2+n2<=28n2 {c=28, n0=1}

So, f(n)=O(n2)

Big Oh notation

cubic functions

f(n)=2n3+n2+2n

f(n)=2n3+n2+2n for n2>2n 2n3+n2+2n<=2n3+n2+n2<=2n3+2n2

Now for n3>=2n2

2n3+2n2<=2n3+n3<=3n3 {c=3,n0=2}

So, f(n)=O(n3)

---

f(n)=4n3+2n+3

f(n)=4n3+2n+3 for n>=3

f(n)=4n3+2n+3 <=4n3+2n+n<=4n3+3n for n3<=3n

4n3+3n<=4n3+n3 <=5n3 {c=5, n0=3}

So , f(n)=O(n3).

Big Oh notation

Exponential

f(n)=2pown+6npow2+3n

f(n)= 2pown+6npow2+3n for n2>=3n

2pown+6npow2+3n<=2 pow n +6npow2+npow2<=2pow n +7npow2

for 2 pow n>=n2 (n>=4)

2pown+7npow2+2pow n+7*2pown<=8*2pown {c=8, n0=4}

So f(n)=O(2 pown)

Omega notation

Constant

$f(n)=27$

$f(n)>=26*1$ where $c=26$ and $n0=0$, so $f(n)= \Omega(1)$

Linear

$f(n)=7n+5$

$7n<7n+5$ for all n. {c=7} thus $f(n)= \Omega(n)$

Quadratic

$f(n)=27n^2+16n$

$f(n)=27n^2+16n$

$27n^{2<}27n^{2+}$ $+16$ n, for all n {c=27}

So $f(n)= \Omega(n^{2)}$

Omega notation

cubic function

$f(n) = 2n^3+n^2+2n$

$2n^3 < 2n^3+n^2+2n$, for all n, {c=2}

So, $f(n)= \Omega(n^3)$

$f(n)= 4n^3+2n+3$

$4n^3<4n^3+2n+3$, for all n {c=4}

So $f(n)= \Omega(n^3)$

Exponential

$f(n)=2^n+6n^2+3n$

$4*2^n < 4*2^n +3n$, for all n,{c=4}, $f(n)= \Omega(2^n)$

Theta notation

Constant

$f(n)=1627$

$1626*1 <= f(n) <= 1627$ $c1=1626$, $c2=1627$, and $n_0=0$, so $f(n)= \Theta(1)$

Linear

$f(n)=3n+5$

$3n<3n+5$ for all 'n', $c1=3$

Also

$3n+5 <=4n$ for $n>=5$, $c2=4$,$n_0=5$, thus

$3n<3n+5<=4n$ $c1=3$,$c2=4$, $n_0=5$

So ,$f(n)= \Theta(n)$

Theta notation

Quadratic

$f(n)=27n^2+16n+25$

$27n^2 < 27n^2 + 16n+25$ for all $n>n_0$ $c1=27$

Also

$27n^2+16n+25 <= 28n^2$ $c2=28$, $n>n0=17$, thus

$27n^2 < 27n^2 + 16n+25 <= 28n^2$, $c1=27, c2=28$, $n>=n_0=17$

$f(n)= \Theta(n^2)$

Theta notation

Cubic function

$f(n)=2n^3+n^{2+}2n$

$2n^3 < 2n^3+n^{2+}2n$ for all n>=n0, c1=2

Also

$2n^3+n^{2+}2n<=3n^3$ for all n>=n0=2,c2=3

Thus

$2n^3 <2n^3+n^{2+}2n<=3n^3$

So, $f(n)= \Theta(n^3)$

Exponential

$f(n)=2^n+6n^2+3n$

2pown< 2^n+6n^2+3n for all n>=n0,c1=1

Also 2^n+6n^2+3n for all n>=n0=4, c2=8

thus 2 pown< $2^n+6n^2+3n<8*2$pown for all n>n0=4,c1=1,c2=8

- $F(n) = \Theta(2pown)$